Contactless Voltage and Current Estimation Using Signal Processing and Machine Learning

by

Alan Casallas

B.S., Massachusetts Institute of Technology (2019)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2019

Author .....
Department of Electrical Engineering and Computer Science
August 23, 2019

Certified by .....
Jeffrey Lang
Vitesse Professor of Electrical Engineering
Thesis Supervisor

Accepted by .....
Katrina LaCurts
Chair, Master of Engineering Thesis Committee

Contactless Voltage and Current Estimation Using Signal Processing and Machine Learning

by

Alan Casallas

Submitted to the Department of Electrical Engineering and Computer Science
on August 23, 2019, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

This thesis describes a contactless sensor developed to estimate the line currents and line-to-line voltages of a multi-phase cable in the presence of significant external disturbances. The current estimates are derived from an array of point magnetic-field measurements processed by a linear least-square-error estimator. The gains in the estimator are chosen using a probabilistic model of measurement errors created by external magnetic field sources. Test bed validation of the estimates demonstrates estimation errors below 1% even in the presence of nearby cables carrying comparable currents, metal plates that could support eddy currents, and large magnetizable cores. The voltage estimates are derived using actively-guarded electrodes that capacitively couple to the cable conductors. Knowing the coupling capacitance, test bed validation of the estimates again demonstrates estimation errors below 1% even in the presence of nearby cables carrying comparable voltages, and metal plates. A method involving capacitively coupling signals onto the cables is also proposed and demonstrated to determine the coupling capacitance without operator intervention.

Thesis Supervisor: Jeffrey Lang
Title: Vitesse Professor of Electrical Engineering

Acknowledgments

The completion of this thesis would not have been possible without the help of several people and organizations. I would like to thank my thesis supervisor, Professor Jeffrey Lang, for his guidance in developing, testing, and documenting the work in this thesis. Professor Lang was always available to assist me and his vast experience in the field of Electrical Engineering was essential for the success of this research.

I would also like to thank HARTING Corporation for their sponsorship of this research. I would like to thank Vivek Dave for his role as a representative of HARTING, for facilitating the use of their resources, and for envisioning the future use of the detector produced by this research. I would like to thank Felix Loske, Lutz Troeger, Maren Knop, and Oliver Beyer for maintaining an active interest in the research and for welcoming me during my trip to HARTING facilities in Germany, where the detector was tested on power systems in their production facilities.

I would like to thank Texas Instruments, who were also sponsors of this research.

I would like to thank Nevan Clancy Hanumara. Nevan helped us build the light bulb box demo that was presented in the HARTING Business Conference and used extensively for test bed validation.

I would also like to thank the staff at The Cypress Engineering Design Studio at MIT. Dave Lewis and Anthony Pennes were always ready to guide me when assembling the PCB board components and 3D printing the yoke.

I would also like to thank Dave Otten for being available when I needed assistance with laboratory equipment and with finding the right hardware to use during experiments.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivations

Monitoring cable currents and voltages is a critical task in many commercial and industrial environments, since it can be used to examine the quality and use of electrical power, and track machine health and process performance. However, installing equipment to make contact measurements of current and voltage can be time-consuming and require a lengthy shut down of equipment. For this reason, we have developed a contactless voltage and current sensing system that can be clipped around a set of cables to estimate the current and voltage waveforms within those cables. These estimates are processed by a computer, which can then perform further digital analysis using the processed waveforms and potentially transmit the data to other computers in a network.

One use for monitoring cable voltage and current is to automatically detect when a machine is failing or encountering trouble. For example, a CNC milling machine executing an automated script can encounter problems if a drill bit cracks or if the script was incorrectly designed and the drill runs into material it cannot cut through. In this case, the machine may start to draw more current as it applies more torque in an effort to complete its task. This increased current draw would be detected by the system we have developed, and since the data can be transmitted via computer networks, an alert can be raised by a central system to inform the operators of the

machine that a problem has occurred. It is even possible to automatically monitor a large number of machines from a central control point.

Another use for monitoring voltage and current is for inferring the state of a machine. For example, a research team at MIT developed a method of estimating the rotor velocity and rotor position of a Permanent-Magnet Synchronous Motor by measuring the current and voltage drawn by the motor by using an observer state space model. [18] Similarly, the state of other machines can be inferred from current and voltage waveforms by using observer state space methods and lumped parameter models. Furthermore, this state information can be used as the feedback component in a control system, effectively enabling sensorless control of machines, such as controlling the position of a motor without the need to install a physical encoder.

Voltage and current information can also be used to diagnose the power quality in a set of a cables. Utility companies will often include terms in their contracts with commercial and industrial customers that penalize injection of harmonics into power lines and require power quality to stay above a certain level. [4] This is because non-linear loads owned by a customer can cause harmonics to appear in power systems, which then propagate back to a utility company's power lines and create energy losses and extra stress on infrastructure, such as electrical transformers [19]. To deal with this issue, a customer can use the system we have developed to analyze the Fourier components in their cabling and identify when their loads are injecting harmonics into a power system.

1.2 System Overview

The system we developed consists of several components, the implementations of which evolved during the research process:

- magnetic field sensors to form current estimates;
- electric field sensors to form voltage estimates;

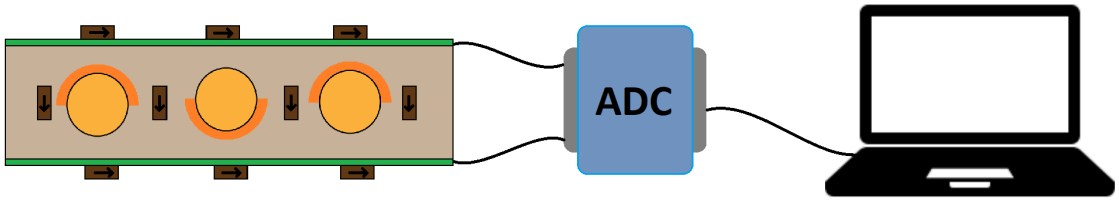

Figure 1-1: The system enclosing the three cables consists of a 3D printed yoke, magnetic field sensors (shown in brown), voltage sensors (shown in orange), and PCB boards (shown in green). The sensor signals are collected by an Analog-to-Digital converter (ADC) and then transmitted to a laptop for digital processing. The arrows inside the magnetic field sensors indicate their axis of sensitivity.

- PCB boards to house the electronics and provide electrical connections;
- a 3D printed yoke made of two halves that clip around the set of cables;
- an Analog-to-Digital Converter (ADC) to convert the analog output of the sensors to digital readings that can be processed by a computer;
- a computer to digitally process the readings.

The methods we developed could be used to estimate current and voltage in any number of cables. The system we built is designed to be used with a set of three cables, since a set of balanced three phase cables is commonly found in industrial environments. After digitally processing the sensor readings, the system outputs three current waveforms and two voltage waveforms. The estimated currents are independent of each other; they are not necessarily assumed to be balanced three phase cables. The two estimated voltages are the line-to-line voltages between each pair of cables. In some cases, these waveforms were collected for a one-second or ten-second period and then saved into CSV files. In other cases, they were streamed continuously and displayed in a live graphical user interface. Figure 1-1 shows a sketch of the entire system.

1.3 Thesis Goals

The goal of the thesis was to create a system that could estimate current and voltage waveforms in a set of balanced three-phase cables to within 1% error of the true current and voltage waveforms. These tolerances were chosen because they are comparable to results achieved by commercially available products and the most recent state of the art research. Furthermore, we wished to calculate instantaneous power, power quality, and power harmonics from these measurements. We aimed to estimate current and voltage up to a frequency of 3 kHz, since this is the maximum frequency for which American industrial power customers are usually penalized for harmonic injection. This required a sensing bandwidth of at least 6 kHz. However, we wanted the methods we used to be easily portable to systems requiring much higher bandwidths in the future.

Producing such accurate estimates is a challenging task because of the abundance of nearby interference in industrial environments. Magnetic fields generated from sources other than the cables inside the yoke, such as nearby cables and ferromagnetic materials, can introduce error in a current estimate. Capacitive pickup of external electric fields, coming from sources like nearby electronics and cables, can affect a voltage estimate. Designing a system to reject these disturbances to a tolerance of 1% was a principal novelty of our work.

To deal with external interference in our estimates, we considered two different approaches:

- using multiple sensors at different locations and using spatial filtering algorithms to separate the external interference from the internal fields;
- using hardware shielding to block out the external fields from reaching our sensors.

In this thesis, we opted to use the first option for current estimation and the latter option for voltage estimation. One reason for this was that the magnetic field shields would be much larger and costlier than the required voltage sensor shields. Another

reason was that this configuration provided the best opportunity to research novel approaches to current and voltage estimation that had not been attempted before and provided promising room for improvement over previous results. Thus, our approach to current estimation involved developing algorithms and software to spatially filter external magnetic fields, while our approach to voltage estimation involved creating the sensor hardware and the shielding mechanism to block external electric fields from reaching the sensors.

1.4 Thesis Contributions

In this thesis we improved on the latest research into current and voltage estimation and we also developed a method to calibrate the capacitance of the electrode used in voltage estimation without operator intervention.

Accurate current estimation of three cable currents using an array of magnetic field sensors in the presence of external magnetic fields has been the topic of several research papers. [12] [17] Many papers examine the use of an Ordinary Least Squares estimator. However, this estimator is not effective in rejecting nearby external interference. A more complex estimator was published by researchers at Politecnico di Milano. [24] The research team modelled external magnetic fields as an infinite sum of harmonics and developed a linear estimator that performed better than the Ordinary Least Squares estimator in the presence of a single external cable. The team did not publish results regarding the performance of the estimator in the presence of more challenging interference, such as external plates.

In this thesis we will present a current estimator that offers superior performance to any estimators currently published. The estimator produces estimates with an error of less than 1% in the presence of many different forms of nearby interference. This performance was achieved both by placing sensors in locations that had not previously been considered, as well as by using a probabilistic model of external interference to generate a Best Linear Unbiased estimate. This current estimator, as well as other estimators that we experimented with, is presented in more detail in

Chapter 4.

Accurate voltage estimation in the presence of electric field interference has also been the topic of several research papers. A team at MIT developed a method of using two sensing electrodes at slightly different distances from a cable to increase sensitivity to close electric fields and reduce sensitivity to far away fields. [6] A different research team at La Plata National University developed a voltage sensor that uses a physical shield driven to ground voltage to protect against external interference. [21] However, this shield is only briefly mentioned in the paper and no error percentages are published with regards to its performance.

In this thesis we present a voltage estimator protected by an active shield that is driven to the same voltage as the sensing electrode. We present experiments in which we introduce various forms of interference and demonstrate that the shield enables estimates with less than 1% error in the presence of interference, whereas the error would be significantly greater without the shield. We will present voltage estimation in Chapter 5.

To form accurate voltages measurements without operator intervention, it is also necessary to calculate the capacitance between the cable and the sensing electrode through a calibration scheme. The most successful calibration scheme has been developed by the team at La Plata National University. They report voltage estimates with less than 1% error by use of their calibration method. [20] However, the method requires connecting the detector to the ground of the system being measured, and assumes no significant load between the voltage being measured and the ground of the system.

In this thesis we present a method to calibrate the sensing electrode capacitance without requiring a connection to the ground of the system being measured. In experiments using this method, we estimated the electrode capacitance value with a 30-90% error. However, this is due to limitations of the hardware we used in the detector prototype. A more carefully manufactured detector in which all necessary hardware exists in the form of PCB board components will yield better results using the calibration method we have developed.

Chapter 2

Previous Work

There has been much research into contactless current detection using magnetic field sensor arrays, voltage detection using capacitive sensors, and signal separation techniques using machine learning, which can be useful for separating magnetic fields generated by currents from those generated by external sources. We will now present the latest results in these three fields of research.

2.1 Current Estimation

Contactless current estimation is performed by measuring the magnetic fields produced by a current. According to Ampere's Law, an infinite current-carrying cable will produce a circular magnetic field where the magnitude $|B|$ at a certain point in space is

$$|B| = \frac{\mu_0 I}{2\pi r} \tag{2.1}$$

where r is the distance between the point in space and the center of the cable, I is the current magnitude, and μ_0 is the permeability of free space. The magnetic field will have a direction perpendicular to the vector formed from the center of the cable to the point being measured.

One of the most common contactless current sensors today is the Hall Effect current sensor. [19] A Hall effect sensor detects a magnetic field by running an

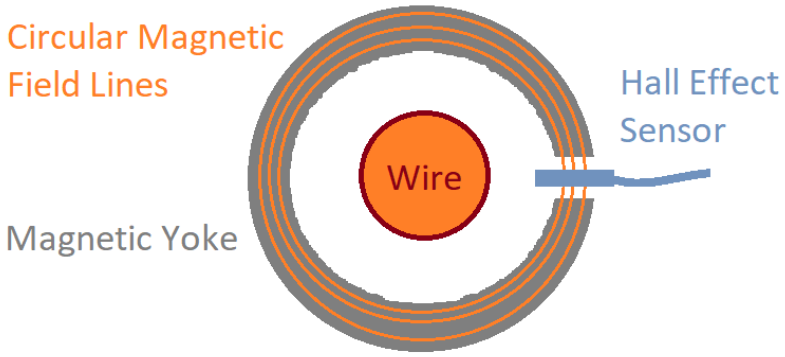

Figure 2-1: Current sensors using the Hall effect commonly include a circular magnetic core, along with an air gap that contains the Hall effect sensor to detect the focused magnetic field.

electric current through a semiconductor in such a way that opposing current carriers will be pulled towards opposing sides of the semiconductor by the magnetic Lorentz force. This pulling will create a voltage difference across the width of the conductor, and this voltage difference is measured to determine the magnetic field strength.

Many Hall effect current sensors will include a circular magnetic yoke, used to focus the current-produced magnetic field, with an air gap where the focused magnetic field can be measured by the Hall effect sensor, as shown in Figure 2-1. The magnetic yoke also serves to shield the Hall effect sensor from external magnetic field interference, although this shielding is not perfect and the air gap can be susceptible to interference. [24]

The Hall effect sensor suffers from several shortcomings. Hall effect sensors have a strong dependence on temperature. One study found that the Hall effect voltage changed by as much as 3 mV over a range of temperatures from $-40\ ^{\circ}C$ to $125\ ^{\circ}C$ when the magnetic field being measured was kept constant. [26] In fact, although the HARTING HCM open-loop current sensor offers an error of 1% at $25\ ^{\circ}C$, the error rises to 5% at high temperatures. However, HARTING does offer a closed-loop Hall effect sensor that uses a feedback control loop to offer 1% error at higher temperatures, although this sensor draws more power. [14]

Hall effect sensors also tend to be large and expensive. The magnetic core used

Figure 2-2: A photo of three HARTING Hall effect current sensors installed around three cables. The installation of these sensors required an almost full-day shutdown of the electrical system pictured. A single 200 A HARTING Hall effect sensor can cost $90. Photo courtesy of HARTING corporation.

to shield the sensor from external magnetic fields can be large and bulky, such as the one belonging to the HARTING Hall effect sensor pictured in Figure 2-2. A single 900 A Harting Hall effect sensor costs around $90.

Another limitation of the Hall effect sensor is that the properties of the magnetic yoke can limit the frequency of current that can be measured. This can be a limitation if a user is seeking to examine harmonics that exist beyond that range. For example, the open-loop HARTING HCM sensor can detect up to 25 KHz and the closed-loop sensor can detect up to 100 KHz.

There are other types of magnetic field sensors. Magnetoresistive sensors such as AMR and GMR contain materials whose resistance changes with magnetic field. Fluxgate sensors measure magnetic field by using two coils, a drive coil and a sense coil, wound around a ferromagnetic material. A current is run through the drive coil producing an internal magnetic field and driving the material in and out of saturation. The sense coil then detects how much additional magnetic field exists in

the ferromagnetic material from the external surrounding magnetic field. Fluxgate sensors have the advantage of being more accurate, less noisy, and less sensitive to temperature than both Hall effect and magnetoresistive sensors. [16]

Considerable research has been conducted on estimating cable currents without the use of a magnetic yoke core to shield the cable from external fields, due to the high cost and large size of magnetic cores. Such an estimation system involves placing several sensors around a cable to obtain an array of magnetic field point measurements. As long as the sensor array is stationary in space, the relationship between magnetic field and current will be linear. This is because the spatial terms in (2.1) will stay constant, so the magnetic field will simply be linearly proportional to the cable current.

One common setup is to design the sensor array such that each sensor is equally distant from the center of the cable and to average the magnetic field values detected by each sensor. [13] Although the accuracy of the current estimate will be affected by errors in the placement of the sensors, the error is not very large. A study into the effect that sensor misplacement had on current estimates found that when 4 sensors were used to estimate current in one cable, a 10 mm lateral offset was needed to introduce a 0.5% error in the estimate and a 25 $^\circ$ rotational offset was needed to introduce a 1% error. The error was even lower when using a larger number of sensors. [13]

Another setup that has been studied extensively is the use of a magnetic field sensor array to estimate the currents in three cables simultaneously. Although the gains between each current and each sensor will be different, they will yield a system of equations that can be used to estimate the currents. If N magnetic field sensors are used to detect the magnetic fields produced by P currents, a system of N equations with P unknown variables can be solved to estimate the P currents. This system of equations can be represented as a matrix multiplication $AI = b$, where I is a vector of currents, b is a vector of magnetic fields, and A is a gain matrix. Each component in the matrix A is equal to the term $\frac{\mu_0 cos(\theta)}{2\pi r}$, where θ is the angle between the axis of sensitivity of the sensor and the vector pointing from the center of the cable to the

point being measured. As long as the number of measurements is equal to or greater than the number of unknowns, a current estimate can be derived. Most commonly, Ordinary Least Squares is used to generate a current estimate, given by the formula $\hat{I} = (A^T A)^{-1} A^T \hat{B}$. [11]

Some research projects have focused on the problem of calibrating the gain matrix A. Although the values of the gain matrix can be theoretically computed, in practice those values may not be known precisely due to sensor or cable misplacement. A power meter developed by a team at MIT attempts to solve this by providing the user with reference loads that can be connected to the cables being measured. [17] These loads draw a predetermined amount of 3 KHz PWM current. This PWM signal can be observed by the sensors and can be used to solve for the gains of the gain matrix. The researchers also developed an algorithm to calibrate the gain matrix without requiring the customer to attach reference loads by instead having the customers power devices already connected to the cables on and off. The algorithm presented in [7] can then calibrate the gain matrix up to a scaling factor. The scaling factor can then be solved for using readings from contact energy meters.

A team at the University of Alberta explored a method to calibrate the gain matrix by solving for the locations of the magnetic field sensors from the magnetic field readings. [12] If the locations of the magnetic field sensors and cables are treated as unknowns, then a system of equations can be formed using the relationship between current and magnetic fields that contains two unknowns for every sensor and cable, specifically, the x and y location for each element. This model assumes the cables are parallel so that two unknowns are sufficient to describe the location of each element. When enough magnetic field readings are collected from an AC current waveform, there will be more equations than unknowns in the system of equations. Since the system of equations is non-linear, the researchers used a Non-Linear Least Squares (NLLS) solver to solve the equations. An advantage of this method was that since no assumptions are made about the geometry of the cables, it can be used to estimate currents when the three cables are bundled inside a single insulator. Using this method, the team was able to estimate current with an error of 4.63%.

Many research papers have also focused on the challenge of estimating a set of three currents in the presence of external magnetic field interference. In many real-life environments magnetic field sensors will detect not only the magnetic fields created by the currents, but also magnetic fields from other sources, including Earth's magnetic field, fields from nearby cables, and fields from eddy currents induced in nearby metallic plates. This presents a major challenge that several research teams have made attempts to characterize and solve.

Researchers at Politecnico di Milano approached this problem by characterizing all possible external magnetic fields by a set of linear equations. [24] To do so, they observed that the cross-sectional area containing the three cables would not contain the sources of the magnetic fields that must be filtered out. The sources, such as other cables, are located outside of this area. They then observed that in an area free of magnetic field sources, the magnetic scalar potential at any point will obey Laplace's Equation, $\nabla^2\Phi(r,\phi,t) = 0$, and that the solution to this equation can be expressed as an infinite series known as the circular harmonics of Laplace's equation. Once the derivative of the magnetic scalar potential is taken, the external magnetic fields can be represented as

$$\begin{aligned} H(r,\phi,t) = &-\sum_{m=1}^{M} mr^{m-1}(a_m(t)cos(m\phi) + b_m(t)sin(m\phi))\hat{r} \\ &+\sum_{m=1}^{M} mr^{m-1}(b_m(t)cos(m\phi) - a_m(t)sin(m\phi))\hat{\theta} \end{aligned} \tag{2.2}$$

where r and ϕ represent the location of each sensor in polar coordinates, and M is the number of harmonics chosen to represent the external fields. Thus, the external magnetic fields are represented as a series of linear equations where the components $a_m(t)$ and $b_m(t)$ are unknown. The total magnetic field detected by each sensor can be modeled as the sum of the three current-created magnetic fields and the harmonic series above. This leads to a system of equations in which there are $3+2M$ unknowns. As long as the number of sensors N is equal to or greater than $3 + 2M$, the system of equations can be solved.

The main drawback of this approach is that to fully characterize Laplace's Equation, an infinite number of linear terms are required. Thus, using a finite number M of components will lead to error in the estimate. However, the researchers conducted simulations that found that this method was effective in reducing the current estimation error created by external interference. For example, when an external cable carrying the same current magnitude as the three internal cables was located about 10 cm above the sensor array, the error when using an Ordinary Least Squares estimate was around 50%. When using four harmonics the error reduced to around 20% and when using eight harmonics the error reduced to 5%. Note that the results are presented in the form of graphical heatmaps and the values we present are estimates based on the heatmaps presented in the paper.

Several other papers have covered the topic of estimating currents in the presence of interference using an array of magnetic field readings, and have attempted methods similar to the ones described in this section. These papers can be found in [25], [22], [9], [10], [5], and [23].

2.2 Voltage Estimation

Contactless voltage detection can be performed by placing electrodes near cables so that the electrodes capacitively couple with the cable voltages. Since voltage is a differential measurement, in our thesis we focused on measuring line-to-line voltage by using two electrodes to capacitively couple with two adjacent cables.

One of the first contactless voltage detectors was described in 1928. [28] It involved vibrating a plate near a conductor until there was no current running through the plate, which indicated that the plate was vibrating at the same frequency as the voltage in the conductor. The author of this method claims it could measure voltage to 1/1000 volts, although it took several seconds to reach the correct vibrating frequency.

More recent voltage detection methods involve electrodes that do not vibrate. A goal of these methods is to measure cable voltage while minimizing the pickup of external electric fields unrelated to the cable voltage. An example of such a system is

one developed by a team at MIT. In this system, two electrodes are placed at slightly different distances from the cable. [15] The two electrodes capacitively couple with the cable voltage with capacitances C_{P1} and C_{P2}, and a known resistor and capacitor are placed between each electrode and ground in parallel. The transfer function between the cable voltage and each electrode is approximately $H(j\omega) \approx \omega R C_{P1}$ and $H(j\omega) \approx \omega R C_{P2}$. The capacitances C_{P1} and C_{P2} are inversely proportional to the distance between the cable and each electrode, such as $C_{P1} \propto \frac{1}{d}$. However, rather than considering the voltage of a single plate, the system considers the differential voltage between the two plate electrodes. The paper shows that since the distance between the two electrodes is much smaller than the distance between the electrodes and the cable, the effective capacitance of the detector, $C_{P1} - C_{P2}$, can be approximated to be inversely proportional to the square of the distance, $C_{P1} - C_{P2} \propto \frac{1}{d^2}$. Thus, the electrode measurement is significantly more sensitive to the cable voltage compared to external disturbances located at far distances.

However, it was not a goal of this MIT research to accurately estimate the magnitude of the cable voltage. Thus, no attempt was made develop a technique that could calibrate the electrode capacitance C_P, which can vary depending on cable insulation material. Rather, the designers of the system focused on examining the spectral components of the voltage and were more interested in the relative magnitudes between frequencies. The resulting system exhibited an error of up to 11.2%, but produced digital waveform estimates that allowed for the relative analysis of the voltage Fourier spectrum. Furthermore, the use of two plate electrodes was effective in mitigating the effect of external electric fields. In one experiment, a fan was suddenly turned on 30 cm from the voltage detection system. Although the estimated voltage initially spiked due to the strong inductive effect of the fan being turned on, the effects of the fan on the voltage estimate dissipated after two line cycles.

Other voltage detection systems have been developed with the goal of accurately estimating cable voltage magnitudes. A system developed at Prince of Songkla University uses a copper film wrapped around a cable to serve as an electrode to capacitively couple with the cable voltage. [27] The researchers manually measured the

electrode capacitance using an LCR meter and found it to be 7.55 pF. They then used a lumped parameter model of their system to estimate the cable voltage and achieved a 2.5% estimation error.

However, the researchers recognized that a major drawback of their system is that the capacitance between the electrode and cable are unknown for different types of cable insulation. Thus, the team has proposed attaching a second electrode to the cable and applying a known voltage to the second electrode. This known voltage will create a capacitively coupled voltage in the cable, which will then be detected by the sensing electrode. If the two electrodes have the same capacitance, the voltage in the sensing electrode can be used to solve for the capacitance of both electrodes. [15] However, this method was only briefly proposed as something that the team would explore and as of this writing has not yet been developed.

Another contactless voltage measurement system has been developed at La Plata National University in Argentina. In this system, a sensing electrode is also used but is driven by a reference voltage. Since the cable voltage and the sensing electrode are at different voltages, a current will flow out of the sensing electrode. This current is processed by an op-amp and a small analog circuit before being digitally processed and used to estimate the cable voltage. [21]

The reference voltage is used to calibrate the system by determining the capacitance C_X between the cable conductor and the sensing electrode. The researchers developed a lumped parameter model of their voltage detection circuit which not only includes the capacitance C_X, but also the capacitance between the op-amp input and ground, C_{IN}. According to their model, the output voltage of the detection system is related to the cable voltage V_X and reference voltage V_{REF} by the transfer function

$$V_O(s) = -[V_X(s) - V_{REF}(s)]sC_XR + V_{REF}sC_{IN}R \tag{2.3}$$

where R is the value of a known resistor placed between the sensing electrode and ground.

To solve for V_X from V_O, the values of C_X and C_{IN} have to be calibrated. To

perform this calibration, the researchers developed a two step process. In the first step, the detector was disconnected from the power cable so that the output voltage would only be related to the reference voltage by $V_O(s) = V_{REF} s C_{IN} R$. This allows for the value of C_{IN} to be determined. In the second step, the detection system is attached to the power cable, and the reference voltage is run at a different frequency than the cable voltage. Thus, the cable voltage and the reference voltage contributions to the output voltage are distinguishable, and the value of C_X can be solved.

The latest validation tests run by the researchers achieved an error of 0.7%, which is an improvement over a previous publication of this system which reported errors greater than 2%. Also of note is that this system uses a shield around the electrode driven by the reference voltage to protect against pickup from external electric field sources, although no experimental data is presented to demonstrate the effectiveness of the shield.

A potential concern with this system is that the calibration scheme requires a connection to the ground of the system being measured. The lumped parameter model assumes that the reference voltage is applied with respect to the system ground. However, in some cases it may not be possible to access the ground of the system being measured or there may be a significant and unknown impedance between the cable voltage and the ground.

2.3 Neural Network Methods

The use of neural networks to separate interference from time signals has also been demonstrated. A team at the University of Surrey trained a neural network to separate vocal audio from song tracks consisting of a mixture of vocal and instrumental audio. [2] Separating human vocal sounds from background noise is referred to as the cocktail party problem.

The research team trained a fully connected neural network of 20500x20500x20500 units. The training samples consisted of 20 second-long segments of music recorded at 44.1 kHz. Each training sample contained the mixed audio consisting of both

vocals and instrumentals, as well as the corresponding recording of vocal-only audio. Sigmoid activation functions were used. The neural network was trained over 100 epochs of training data.

The neural network was successful in separating vocal audio from song tracks. The research team compared the performance of the neural network to Non-negative Matrix Fatorization (NMF), a traditional linear method used for audio signal separation. The neural network performed better than NMF. To quantify their results, the team compared the signal-to-artifact (SAR) ratio to the signal-to-interference (SIR) ratio of the processed tracks. These are two commonly used metrics when evaluating signal separation algorithms. When mean SAR was plotted as a function of mean SIR, the tracks separated using neural networks achieved a score 2.5 dB better than tracks separated using NMF. The team credits the non-linear nature of the neural network for its superior performance compared to NMF, since it allowed the network to identify non-linear relationships between the vocal audio and the mixed audio.

2.4 Summary

The research presented in this section constitutes the most advanced current and voltage estimation techniques published before the research that we will present in the following chapters. In our research, we perform current estimation by using an array of fluxgate magnetic field sensors. We chose fluxgate sensors due to their insensitivity to temperature change, their low cost, the accuracy of their readings, and the low noise level of their readings. We also do not use a magnetic core or any type of magnetic shielding in our detector so as to avoid the weight, size, and cost of such shielding. The focus of our research in current estimation was accuracy in the presence of external interference. The team at Politecnico di Milano has also focused on this task, and approached the problem by orienting all the magnetic field sensors in their array horizontally and by using a harmonic expansion of a physical model of external interference to perform their estimates. [24] In contrast, we not only place magnetic field sensors above and below the cables being estimated, but

also in between and to the sides of the cables, which are locations we did not see considered in any of the papers we reviewed. Additionally, we position the sensors in two different perpendicular orientations, which is a technique we also did not see in any published paper. Furthermore, although all the papers previously discussed used deterministic physical models to estimate currents in the presence of external interference, we use a probabilistic model of external magnetic fields to derive a novel linear least squares current estimator which we will present in Chapter 4.

We perform voltage estimation by using a set of electrodes that capacitively couple with the voltages in the cables. While the papers we presented in this section attempt to estimate the absolute voltage of a cable with respect to the ground of the system being estimated, in our research we are concerned with the line-to-line differential voltage of two adjacent cables. In fact, our voltage detection system is isolated from the ground of the system we are estimating. We also use an active shield to protect the sensing electrodes from external interference. Although the team at La Plata National University uses a shield driven to ground in their detector, they do not present experimental data regarding the performance of their shield. [20] Since rejecting interference was an important aspect of our work, we collected data regarding its performance and will present it in Chapter 5.

We have also designed a method of calibrating the capacitance between the cable and sensing electrode of the voltage detection system. A calibration system is presented by the team at La Plata National University. [21] However, this calibration scheme requires a connection to the ground of the system being estimated. Furthermore, while a system similar to the one we have designed is proposed by the team at Prince of Songkla University, no paper has been published presenting data related to that system. [27]. We have implemented our calibration system and will present the challenges we faced and the estimates we obtained in Chapter 5.

Chapter 3

Hardware and Software

The current and voltage detector consists of several modular components: magnetic field sensors, voltage sensors, PCB boards to house the electronics, a plastic yoke that clips around the cables, analog-to-digital converters to capture the data, anti-aliasing hardware filters, and a laptop to process the readings. The magnetic field sensors and voltage sensors will be described in Chapters 4 and 5. In this chapter, we will describe the other mentioned components.

3.1 Yoke and PCB Board

Over the course of the thesis, we used three different hardware designs for the detector. The first detector, shown in Figure 3-1, used a PCB board that was designed by a group of undergraduate researchers that worked on this project before the start of the thesis. This board contained 8 magnetic field sensors and did not contain any voltage detection hardware. Initially, the detector was envisioned as a clip-on that would attach to the HARTING Han-C connector, an industrial connector that attaches to 8 AWG cables. To prototype the system, we glued the board to the connector as shown in Figure 3-2.

Later we decided the detector should be independent of the HARTING connector and instead designed a 3D-printed ABS plastic yoke that could be clipped around a set of three 8 AWG cables. We also redesigned the PCB board to hold 12 magnetic

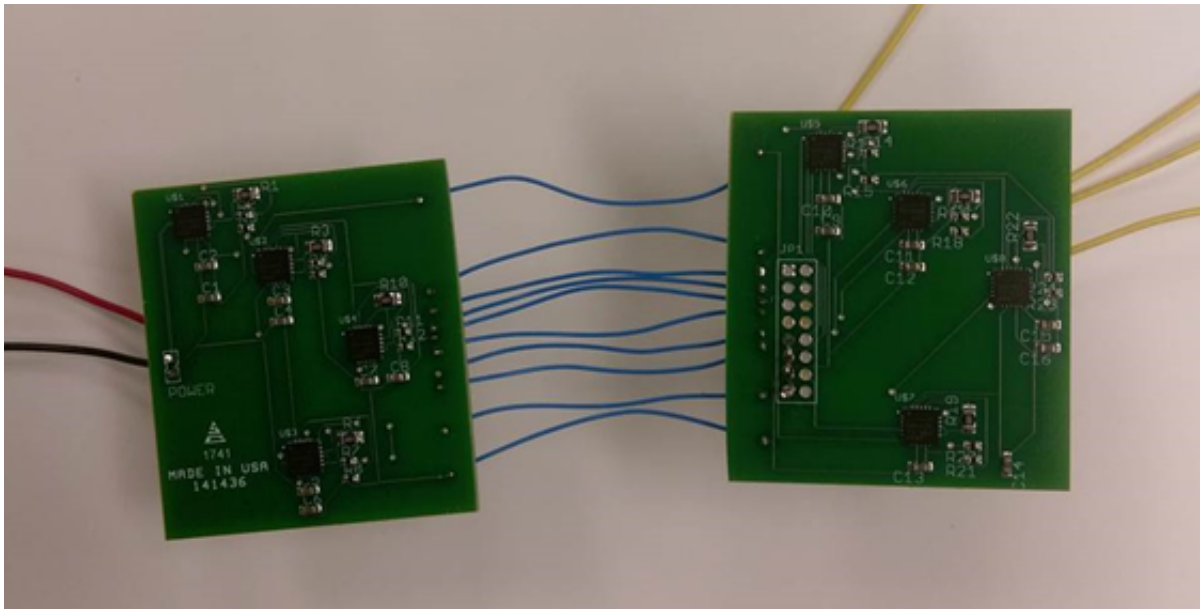

Figure 3-1: The first board, which contained eight magnetic field sensors and no voltage detection hardware.

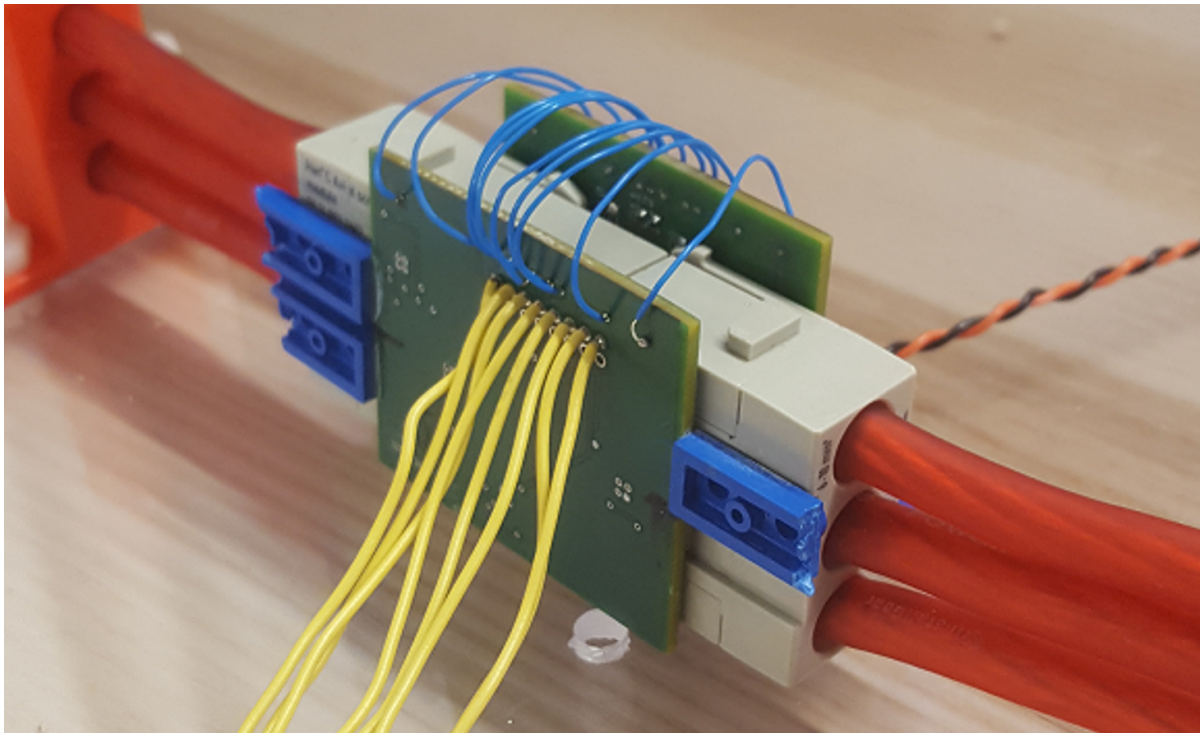

Figure 3-2: The board glued to the HARTING Han-C Connector, which attached to three 8 AWG cables.

field sensors and to house the op-amps used with the voltage detection system. We also included holes to join the board and yoke using nylon screws. An image of this

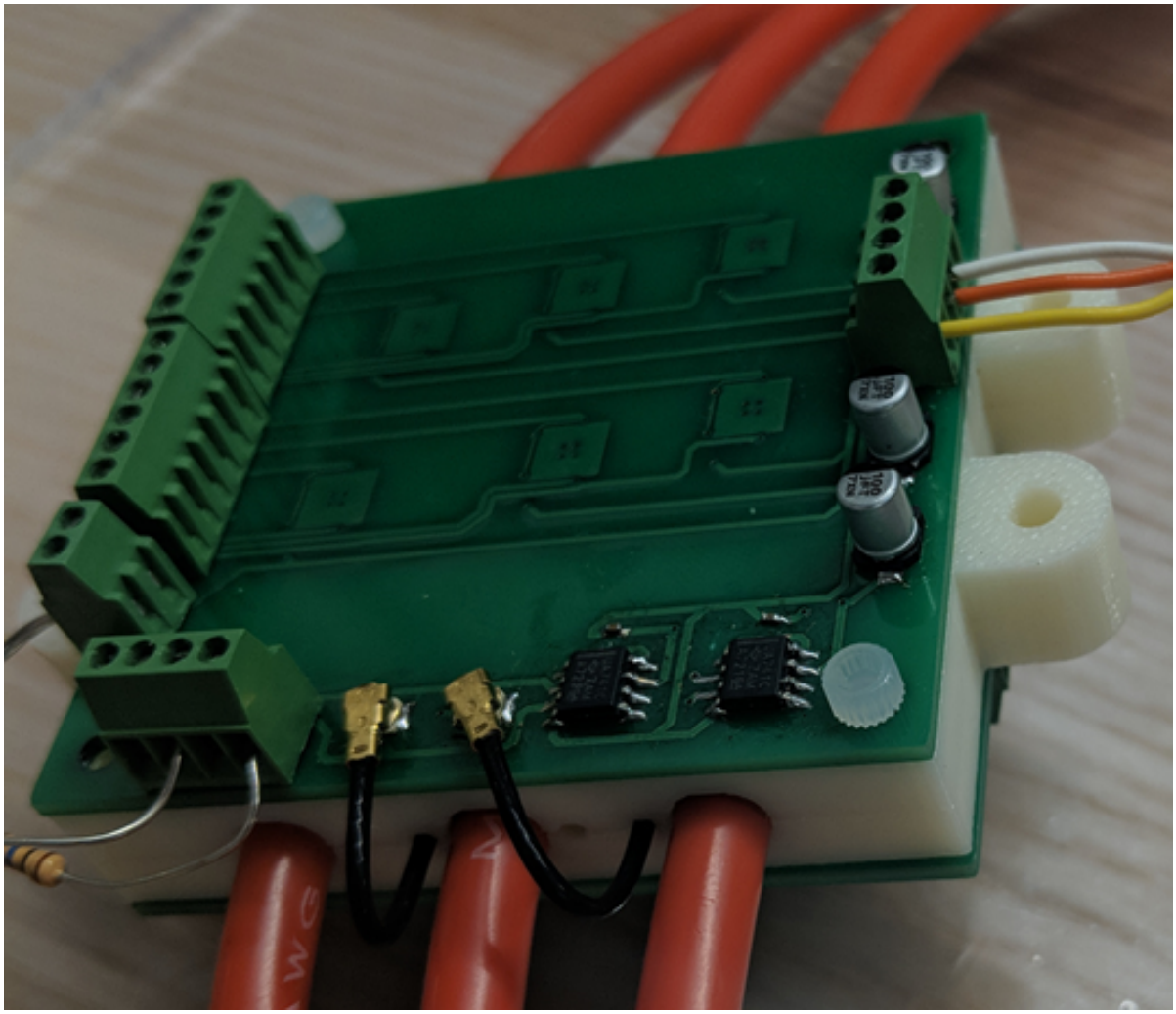

Figure 3-3: The second version of the detector, which contained 12 magnetic field sensors and voltage detection hardware.

board is seen in Figure 3-3.

Lastly, we designed a third version of the detector, shown in figure 3-4. In addition to six horizontally-oriented magnetic field sensors, this board also contained four vertically-oriented magnetic field sensors to achieve more accurate current estimates, using 90-degree angle pins to electrically connect the magnetic field sensors of different orientations. Furthermore, the voltage detection circuitry was re-arranged to reduce parasitic capacitance.

The 3D-printed plastic yoke was designed using OnShape. CAD drawings of yoke are shown in Figures 3-5 and 3-6. The PCB boards were designed using the Eagle

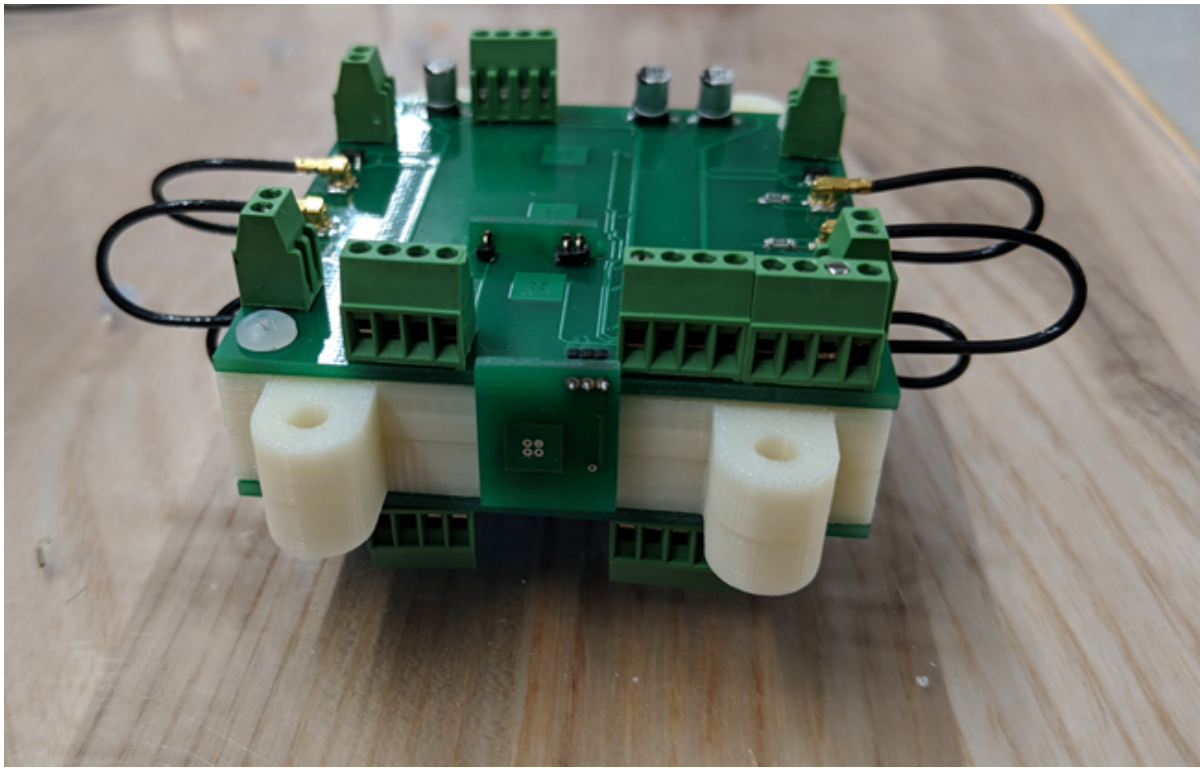

Figure 3-4: The third version of the detector, which contains 10 magnetic field sensors, including 4 vertically oriented sensors, and voltage hardware.

software. The electrical schematic of both main PCB boards is shown in Figure 3-7. The board schematics of the two PCB boards were different and are shown in Figures 3-8 and 3-9. The electrical schematic of the small PCB boards use to house the vertical sensors is shown in Figure 3-10. The board schematic of the small PCB boards is shown in Figure 3-11.

3.2 Analog-to-Digital Converter

To record voltages and process them into digital form, we used a pair of Analog-to-Digital Converter (ADC) devices by Measurement Computing, the USB-205 and the USB-231. [3] Each device was capable of reading 8 analog inputs from -10 volts to +10 volts.

The USB-205 was a 12-bit device and could represent 2^{12} values between -10 and +10 volts. By applying a range of voltages to the USB-205 device, we confirmed all

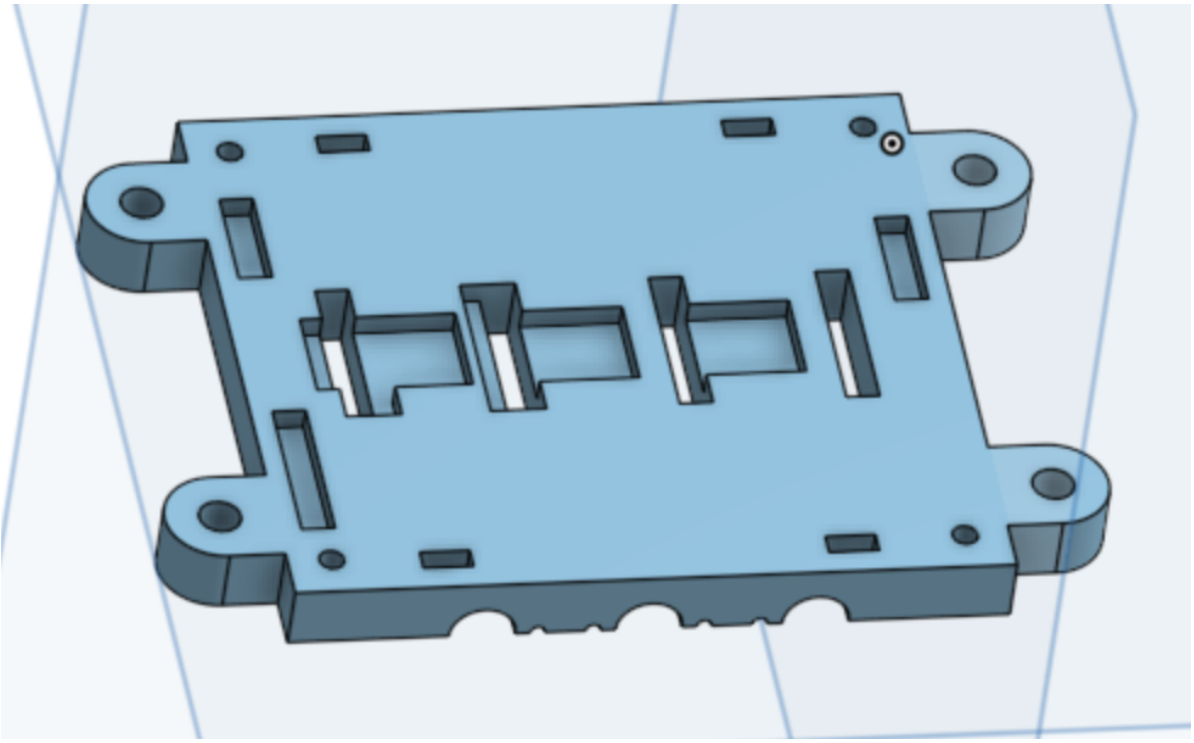

Figure 3-5: A CAD model of the top side of a yoke half. The yoke contained indents to fit the magnetic field sensors and terminal block pins so that the PCB board would lay flush on top of the yoke. The yoke also contained four slots for the vertical PCB boards to fit through.

digital readings were 5.11542 mV apart, which is slightly more than $\frac{20\ V}{2^{12}-1}$ since the true range of the device was slightly wider than +/- 10 volts. Likewise, the USB-231 was a 16-bit device and we confirmed all digital readings were 0.32213 mV apart, which is slightly larger than the theoretical value $\frac{20\ V}{2^{16}-1}$.

Due to the nature of an ADC, the digital readings were corrupted with a small amount of Gaussian white noise. To characterize this noise, we applied a constant voltage from a signal generator to the ADC, ensuring with a multimeter the voltage was constant to the fifth decimal place. We then plotted a histogram of the collected readings and fit a Gaussian curve to them, such as the one shown in Figure 3-12. We found the standard deviation of the noise in the USB-205 ADC to be 1.50 mV, and the standard deviation of the noise in the USB-231 ADC to be 0.29 mV.

Furthermore, we found the noise behavior was the same regardless of the magnitude of the applied voltage. Thus, we modeled the signal detected by the ADC as

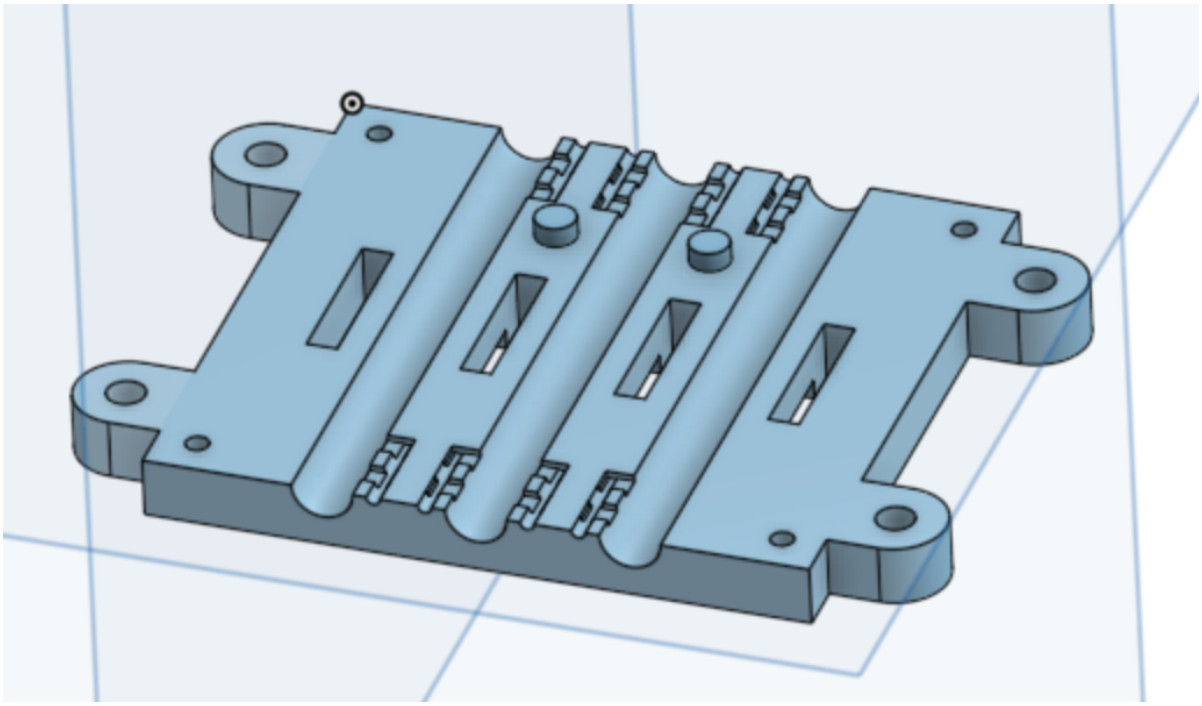

Figure 3-6: A CAD model of the bottom side of the yoke. The three channels that clipped around the cables are visible, as well as smaller channels to fit the coaxial cables that were soldered onto the voltage sensors.

$s[n] = v[n] + m[n]$, where $s[n]$ is the value recorded by the ADC, $v[n]$ is the value that would be recorded by an ideal ADC without noise, and $m[n]$ is a random number taken from a Gaussian distribution.

3.3 Anti-Aliasing Filter

To reduce detection of undesired high frequency signals such as WiFi and Bluetooth, we constructed an anti-aliasing hardware filter. The electrical schematic of this filter is found in Figure 3-13. This filter was implemented as a set of 8 low-pass Sallen-key filters using two 5.1 $K\Omega$ resistors, two 0.01 μF capacitors, and a ua741 op-amp. These values were selected to create a cutoff frequency close to 3000 Hz, since that is the highest frequency we were interested in analyzing in our experiments. A photograph of the filter is shown in Figure 3-14. Table 3.1 shows the gain and phase shift of a series of voltage signals that were applied to the hardware filter as measured by an oscilloscope.

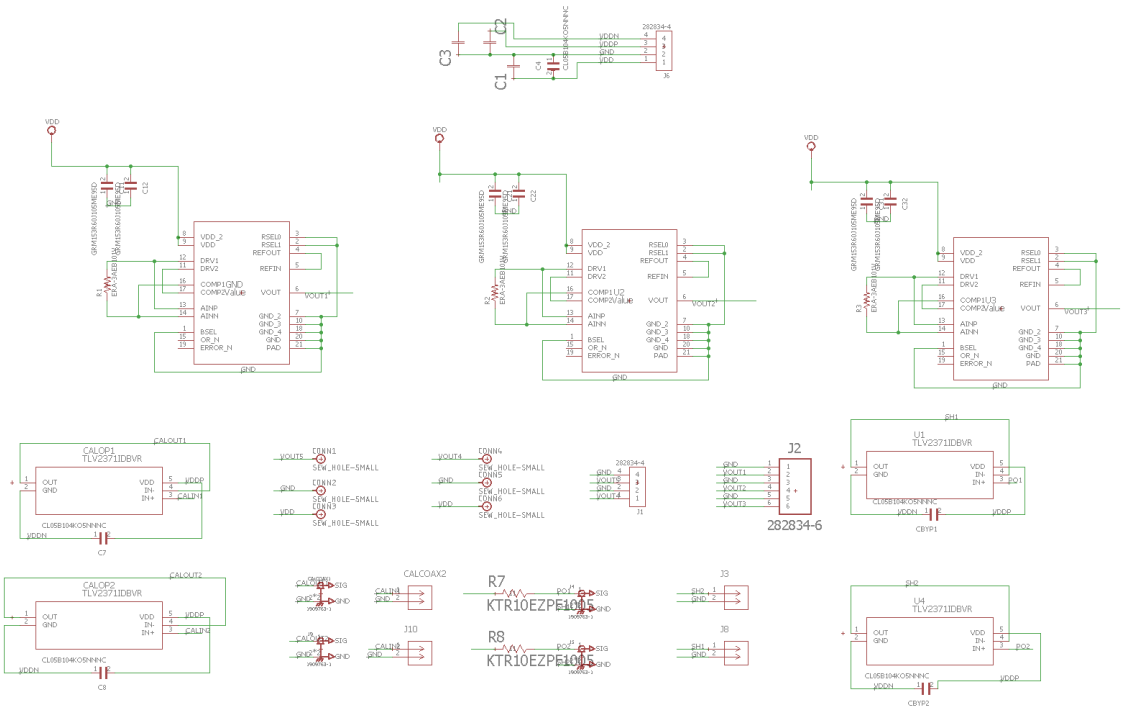

Figure 3-7: An electrical schematic of the main PCB boards. Two of these boards are used in the detector.

Table 3.1: Output of anti-aliasing hardware filter.

Frequency	**Gain**	**Phase Shift**
10 Hz	~1.0	~0°
100 Hz	~1.0	3.5°
1000 Hz	0.91	33°
2800 Hz	0.57	81°
3000 Hz	0.54	86°
3200 Hz	0.50	90°
5000 Hz	0.29	112°
10000 Hz	0.10	139°

To reduce the effect of the filter on frequencies below 3000 Hz, we developed a digital filter to process the readings output by the hardware filter. The theoretical transfer function of the Sallen-Key filter is

$$H(j\omega) = \frac{1}{1 + j\omega 2RC + (j\omega RC)^2} \tag{3.1}$$

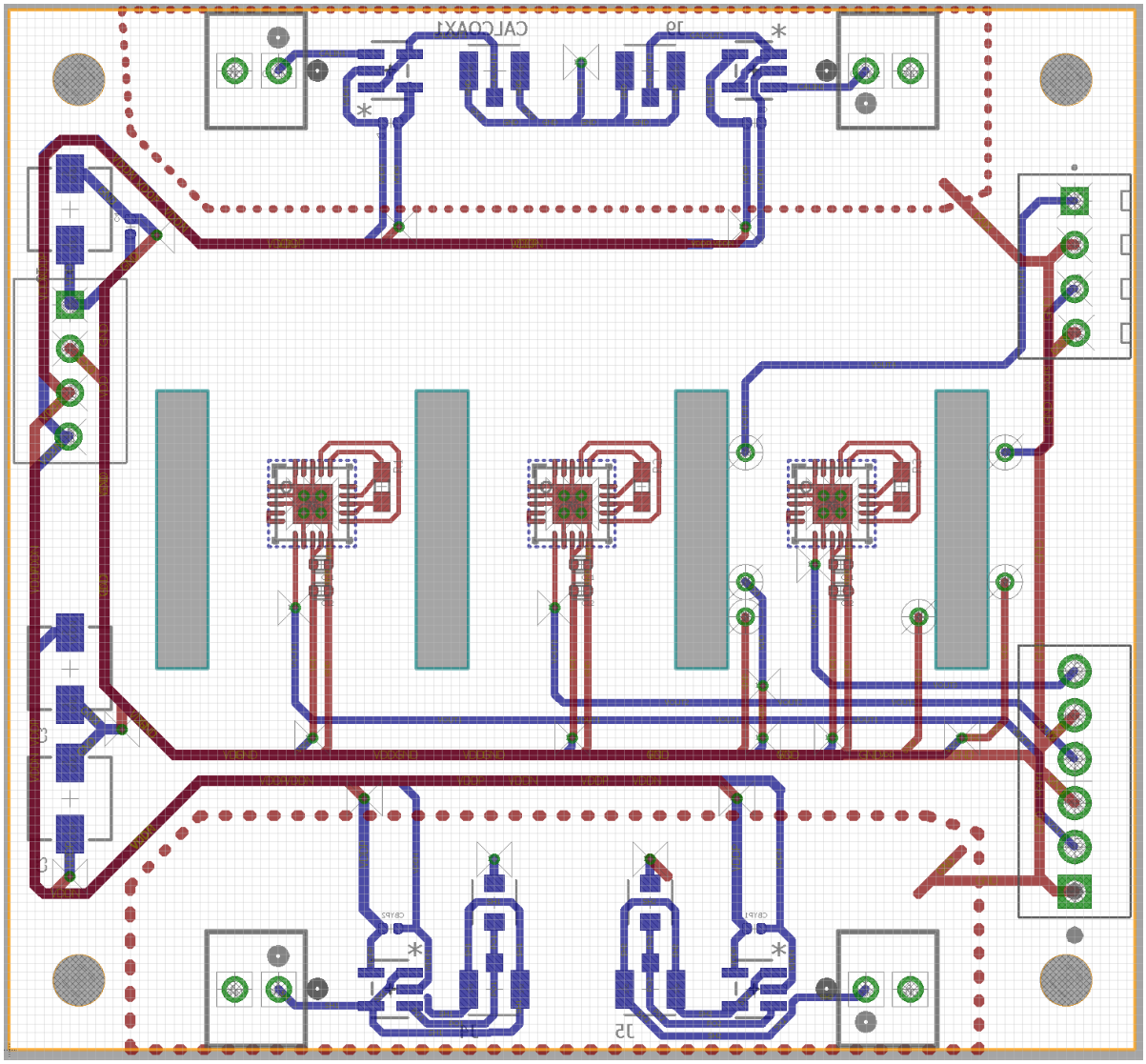

Figure 3-8: The board schematic of one of the main PCB boards. Two different PCB board layouts were required to keep each group of components on the same side when the two boards were attached to the yoke.

The digital filter we created digitally applies the inverse of the above signal by taking the Fourier transform of the input signal, multiplying by the inverse transfer function, and returning the inverse Fourier transform of the result. The filter was implemented in Python. The code of the implementation can be found as the antiantialiasingfilter() function in the file preprocessor.py found in Appendix A.

Table 3.2 shows the gain and phase shift of the signals after passing through the hardware filter and then being processed by the digital filter. The signals were

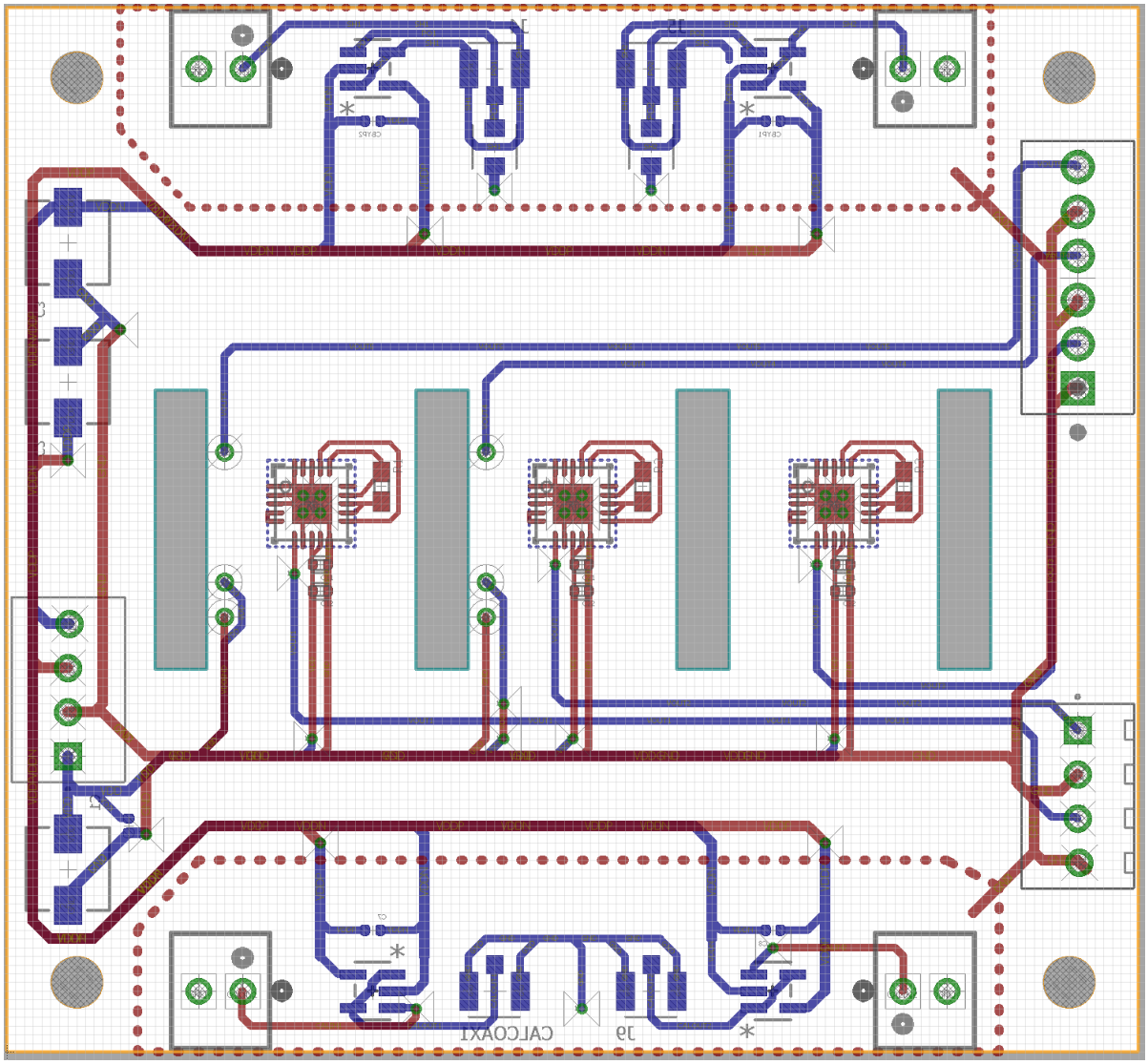

Figure 3-9: The board schematic of the other main PCB board., which complements the boards shown in Figure 3-8.

sampled by the ADC at 6250 Hz. As the results in the table show, the combination of the both filters was effective in correcting the gain and phase shift of signals under 3000 Hz while attenuating signals above this frequency.

3.4 Test Beds

To validate our detector hardware and algorithms, we built two test bed validation environments that would allow us to control and monitor the true current and voltage

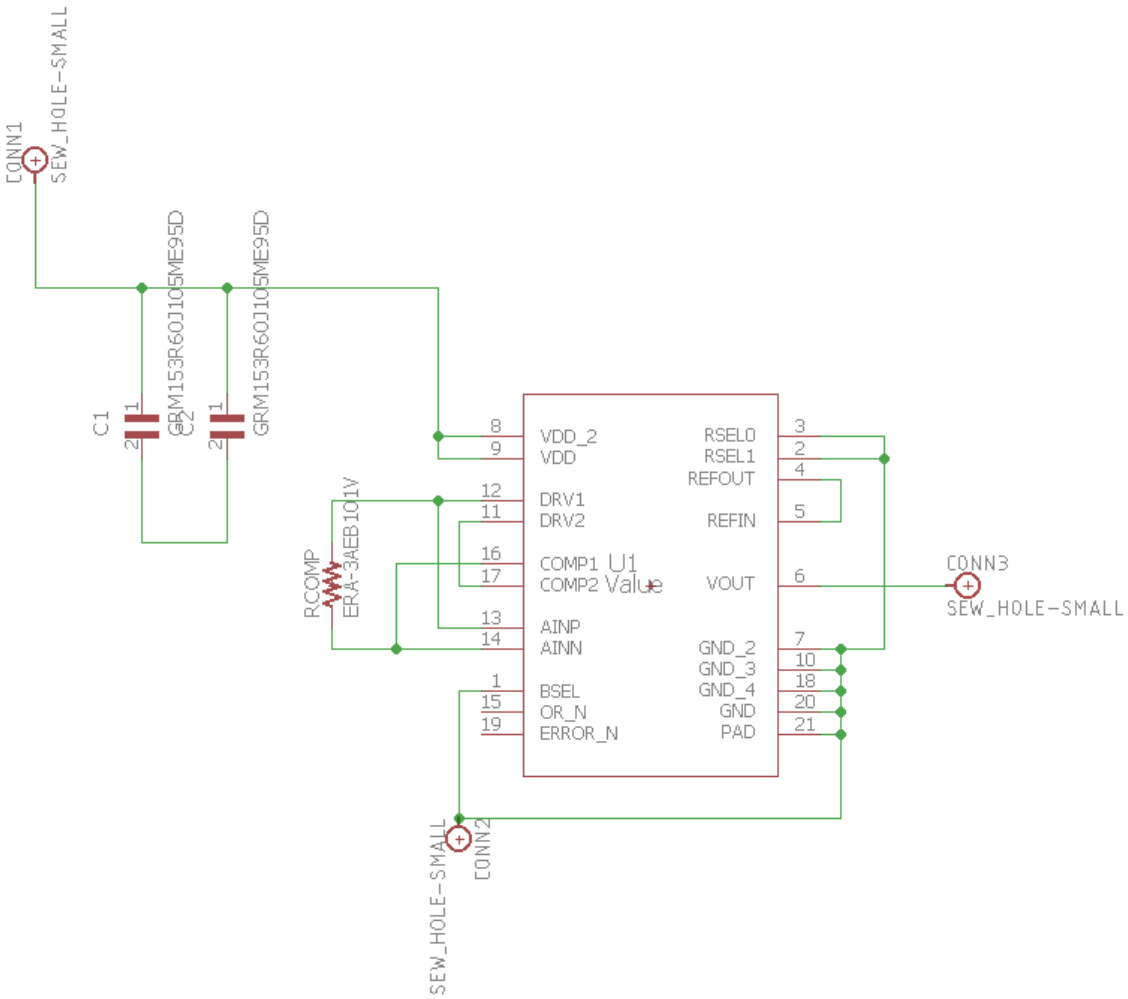

Figure 3-10: The electrical schematic of a vertical PCB board. Four such boards were used in the detector.

Table 3.2: Output of anti-aliasing hardware and software filter.

Frequency	**Gain**	**Phase Shift**
100 Hz	0.999	0.01°
1000 Hz	1.001	0.20°
2800 Hz	1.007	0.01°
3000 Hz	1.007	0.03°

values being applied to a set of cables.

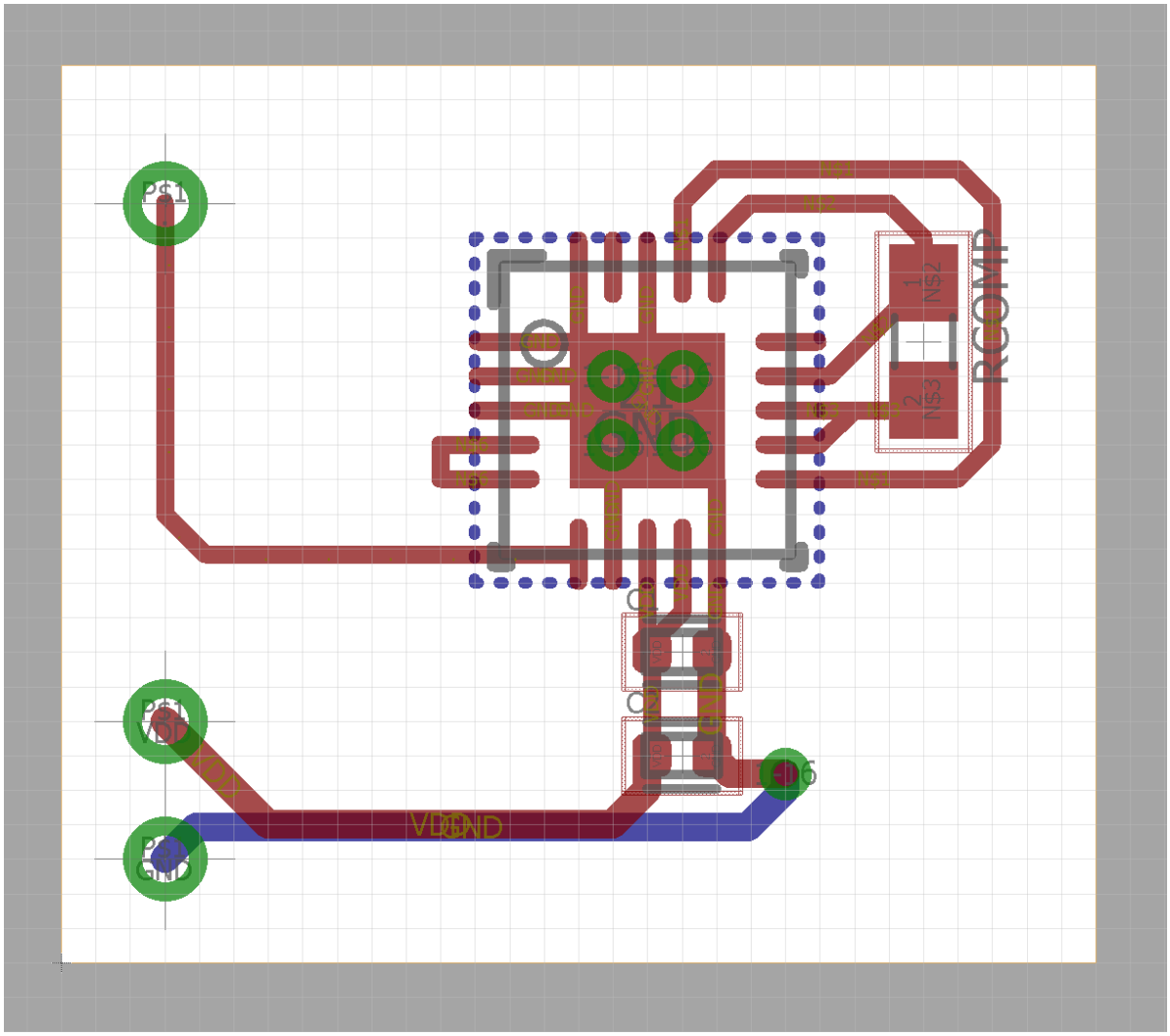

Figure 3-11: The board schematic of a vertical PCB board.

3.4.1 Parallel Cables Test Bed

The first test bed consisted of three parallel 8 AWG cables connected to two terminal blocks, as shown in Figure 3-15. The terminals in these blocks could be connected in different ways to create a variety of circuit configurations. To control the currents, we used a set of TI OPA549 op-amps that were capable of outputting voltages of +/- 30 V and currents of up to 8 A. They are shown in Figure 3-16. These op-amps allowed us to run low-noise currents at frequencies and amplitudes of our choosing. An electrical schematic of the board that housed each op-amp is shown in Figure 3-17.

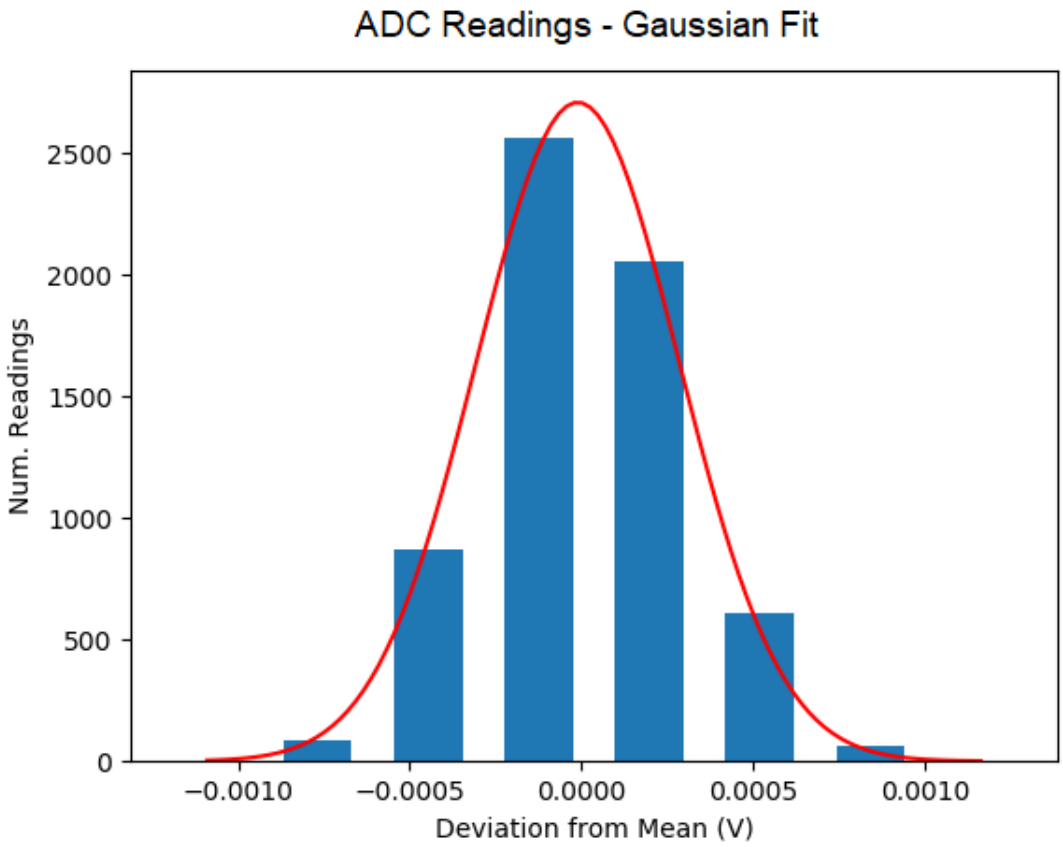

Figure 3-12: A histogram of USB-231 ADC readings collected while a constant voltage was applied, with a Gaussian distribution fit over the readings.

A common configuration we tested involved a set of balanced three phase currents. In this configuration, the cables are running three AC currents that are of equal amplitude and frequency but with 120 ° phase shifts relative to each other. The result is that at every moment in time, the three currents sum to zero. To create this configuration, we connected the terminal blocks to create the circuit shown in Figure 3-18, making the center cable current the sum of the first and third cable currents. We then passed the AC output of a signal generator through an all-pass filter that we tuned to create a second AC signals phase shifted by 120 °. These signals were applied to the two power op-amps, creating a set of balanced three phase currents. The oscilloscope shot shown in Figure 3-19 shows that the two output voltages of the power op amps were phase shifted by 120 °.

We also constructed several supports that allowed us to place external parallel

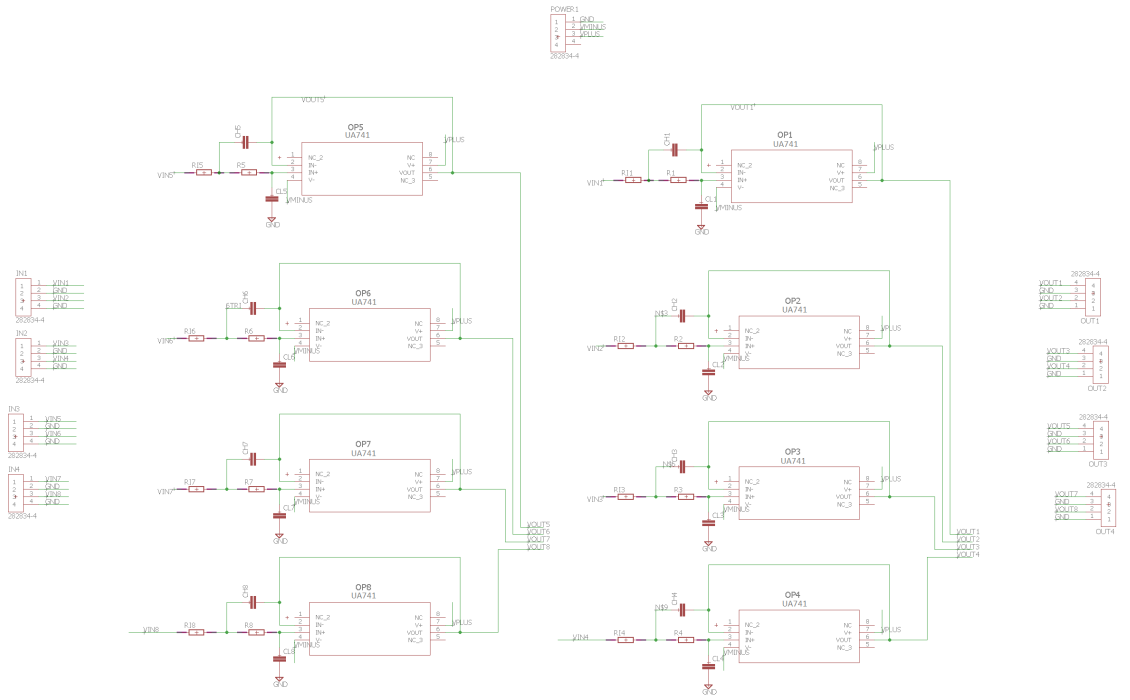

Figure 3-13: An electrical schematic of the anti-aliasing filter.

cables around the detector so we could introduce external interference at known locations and analyze the effect that this interference had on our estimates.

The parallel nature of the cables was useful not only for correctly calibrating the detector in an environment free of external magnetic fields, but also for replicating the conditions inside many industrial cable cabinets, where adjacent cables are often organized in long parallel runs.

3.4.2 Lightbulb Demo

To further validate the performance of our detector, we built a hardware demo that involved measuring cables that powered lightbulbs using 120 V RMS 60 Hz wall power. A photo of the demo is shown in Figure 3-20. The demo consisted of a box capable of holding two lightbulbs. By choosing which lightbulbs to connect to the box, we could control how much current each cable would draw. The yoke was placed around three cables protruding from the box, as shown in the photo. The cables were configured according to the schematic in Figure 3-21. The middle cable contained the return

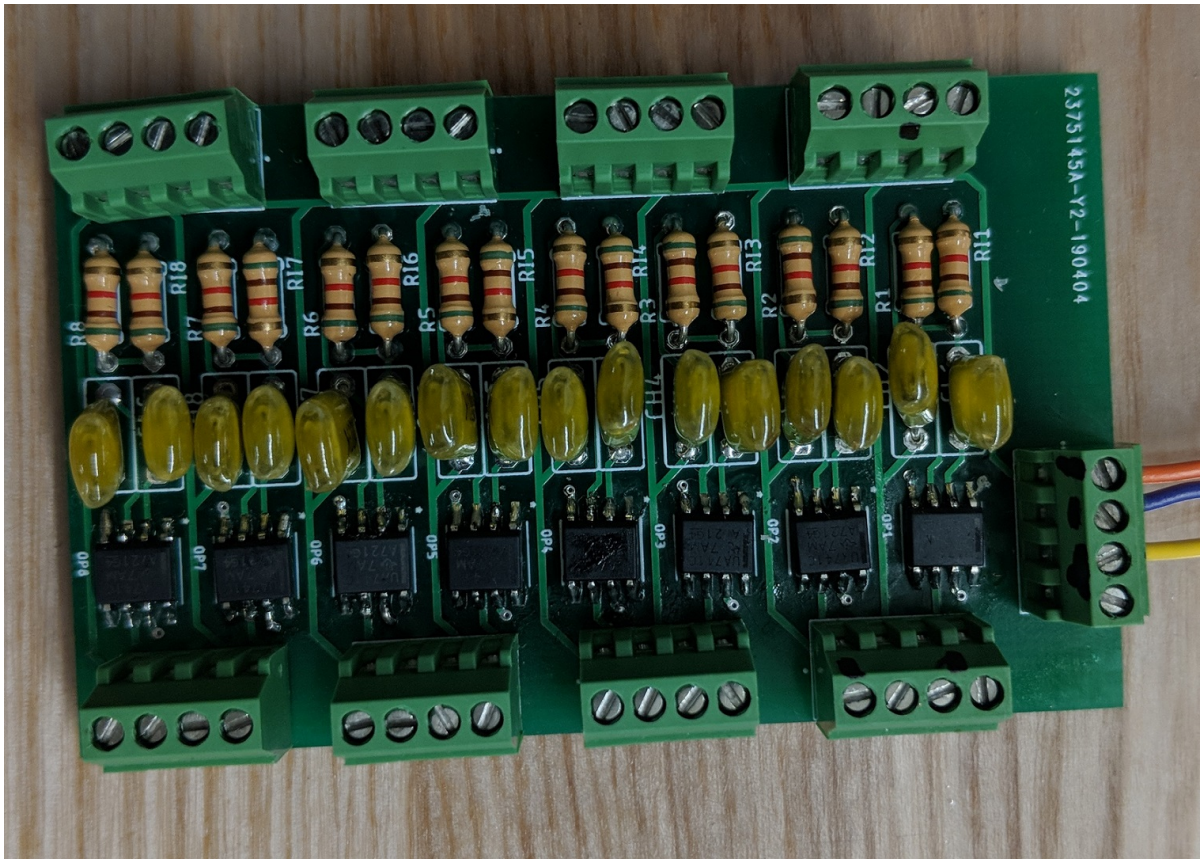

Figure 3-14: A photo of the anti-aliasing hardware filter.

current and thus the negative sum of the other two currents.

We obtained contact measurements of the voltage of the demo by building a voltage divider across the hot and neutral cables which output a voltage between +/- 5.06 V, making it safe for the ADC units to read. Since lightbulbs are non-ohmic, we obtained contact measurement of the currents by installing 1.08 Ω power resistors between ground and the neutral cable of each lightbulb and measuring the voltage drop across the resistors.

3.5 Software and Data Storage

All scripts and collected data were saved in a private GitHub repository. Relevant scripts are also included in Appendix A. Sensor readings processed by the computer were either saved to permanent storage or displayed in real-time using a graphical

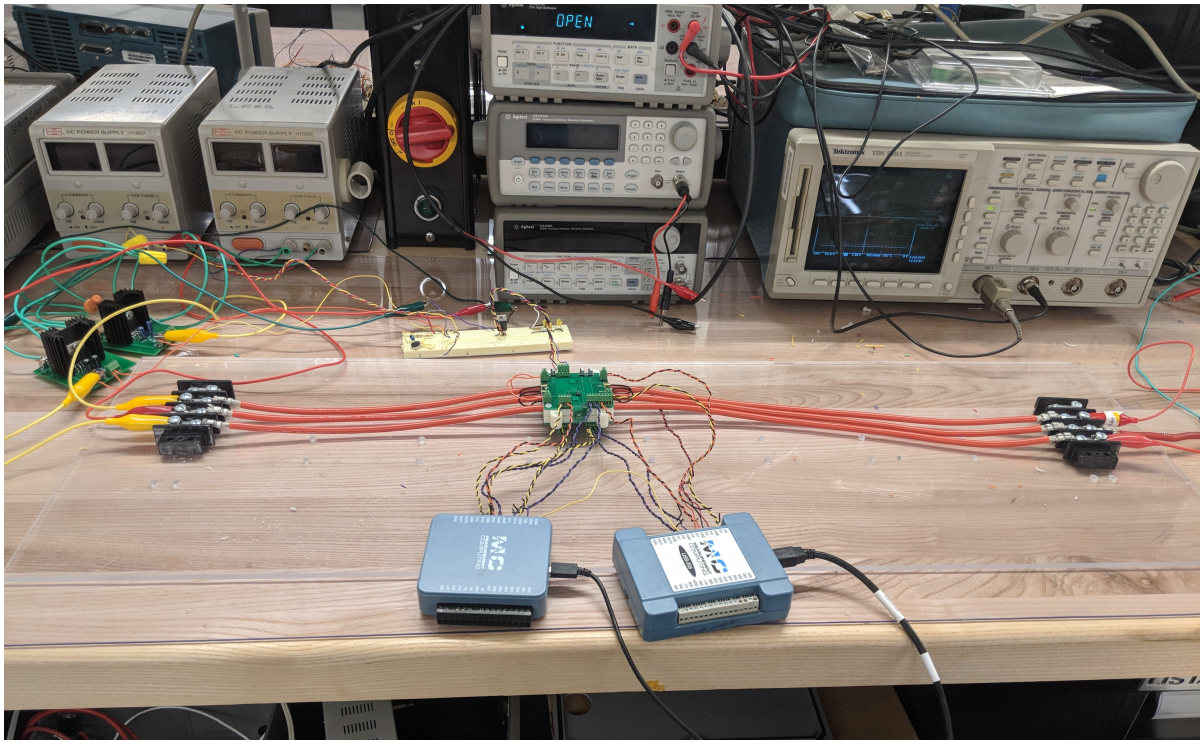

Figure 3-15: The parallel cables test bed.

user interface.

The saved data was stored in CSV files. For example, 2 seconds of readings from 10 magnetic field sensors collected at a sampling frequency of 6,000 Hz would be saved as a CSV file containing 12,000 rows (the number of time samples) of 10 columns each.

The graphical user interface allowed us to analyze collected data in real-time. It was capable of displaying sensor readings and current and voltage estimates as a real-time time display, a real-time frequency display, or a real-time text display where the frequency and amplitude of the frequency with greatest magnitude was shown. The interface is shown in Figure 3-22.

Figure 3-16: The OPA549 power op-amps used to power the parallel cables test bed.

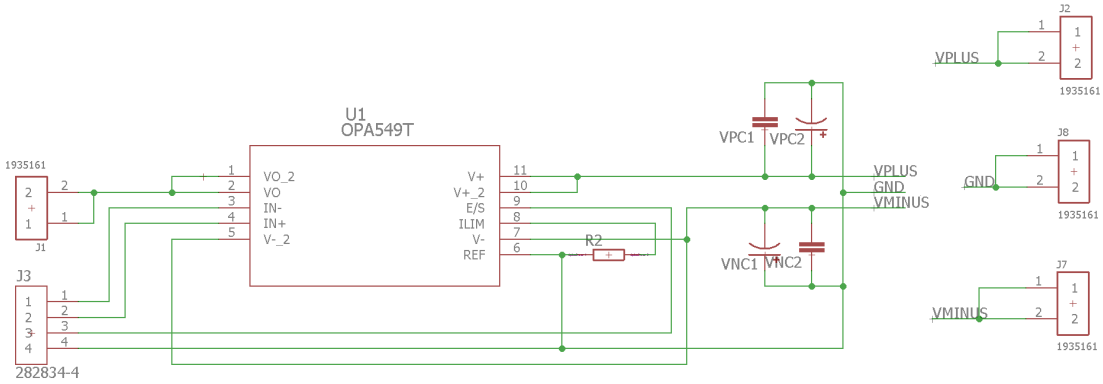

Figure 3-17: An electrical schematic of the board that housed the OPA549 op-amp.

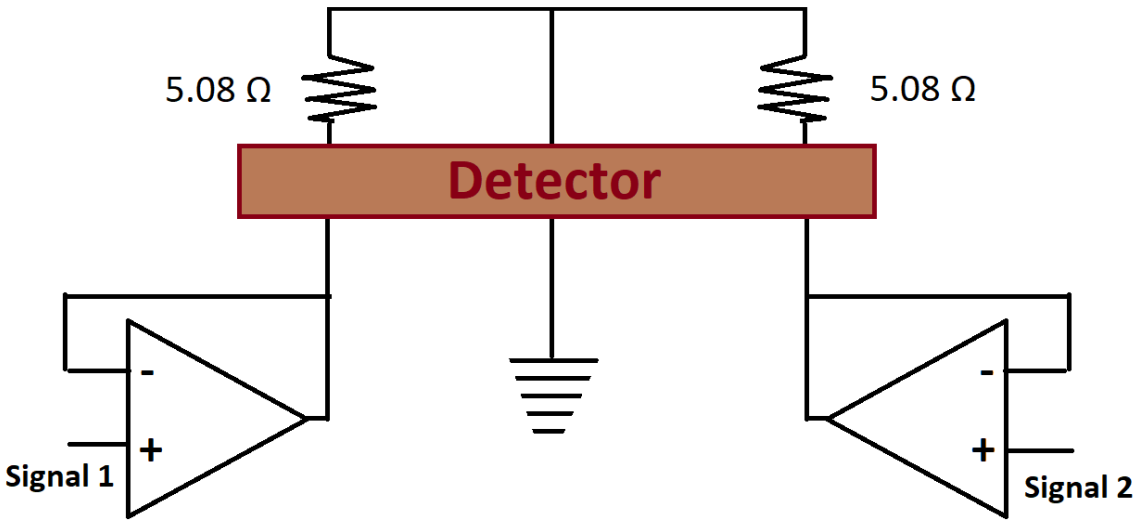

Figure 3-18: A schematic of the configuration used to create a balanced set of three phase currents in the parallel cables test bed.

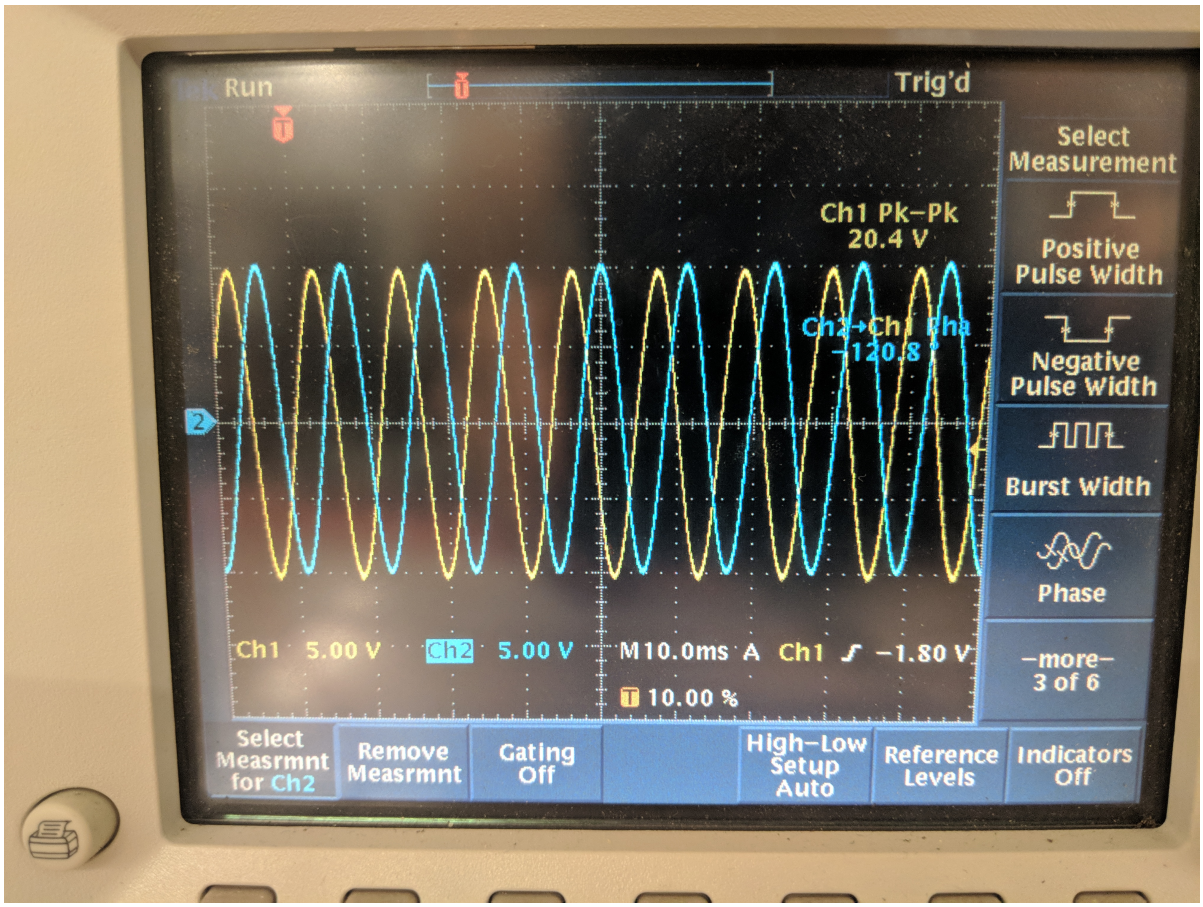

Figure 3-19: An oscilloscope reading of the voltage output of the two op-amps in the balanced three phase configuration, showing that they have been tuned to be 120 ° out of phase.

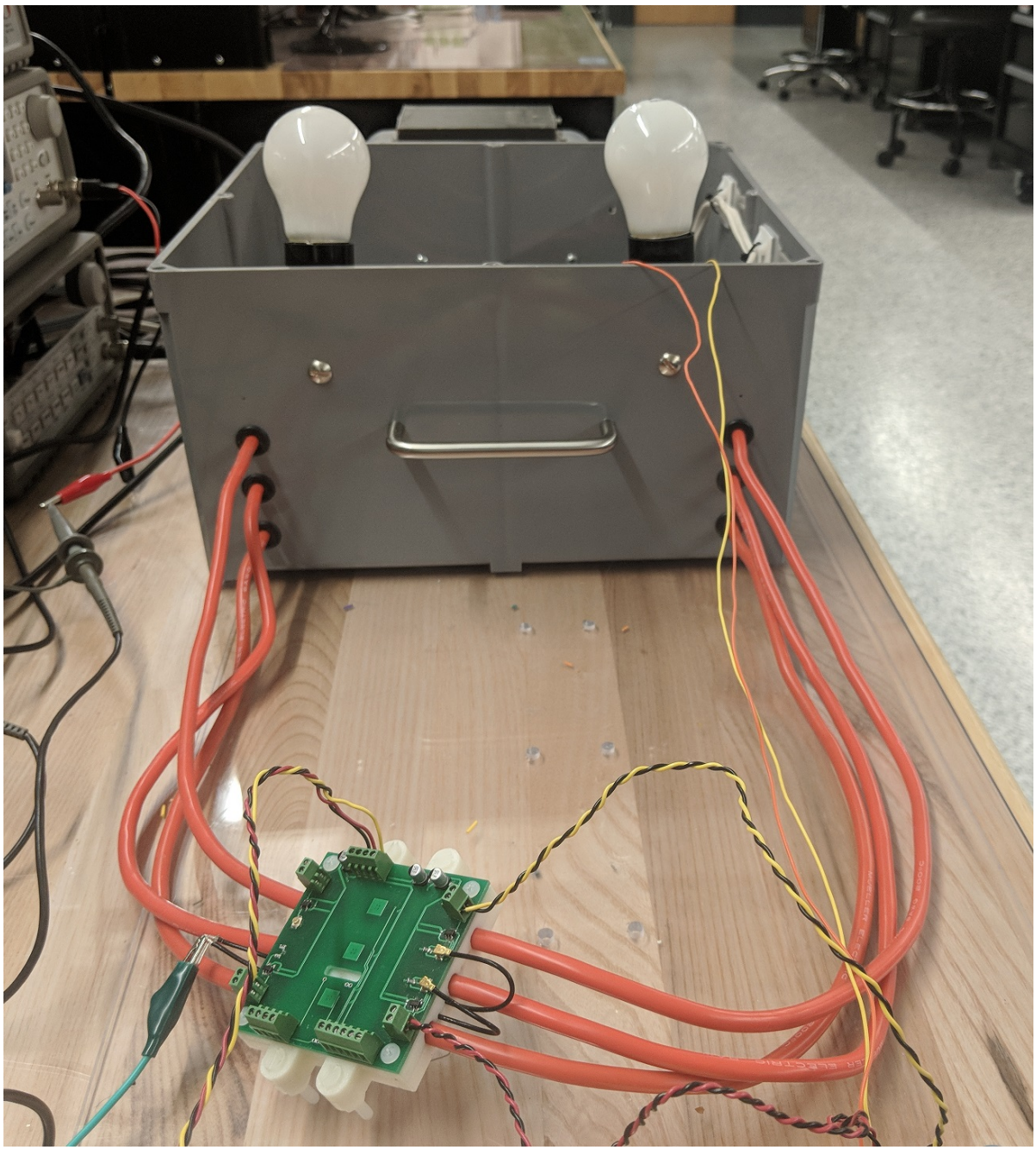

Figure 3-20: A photo of the detector attached to the cables of the lightbulb demo.

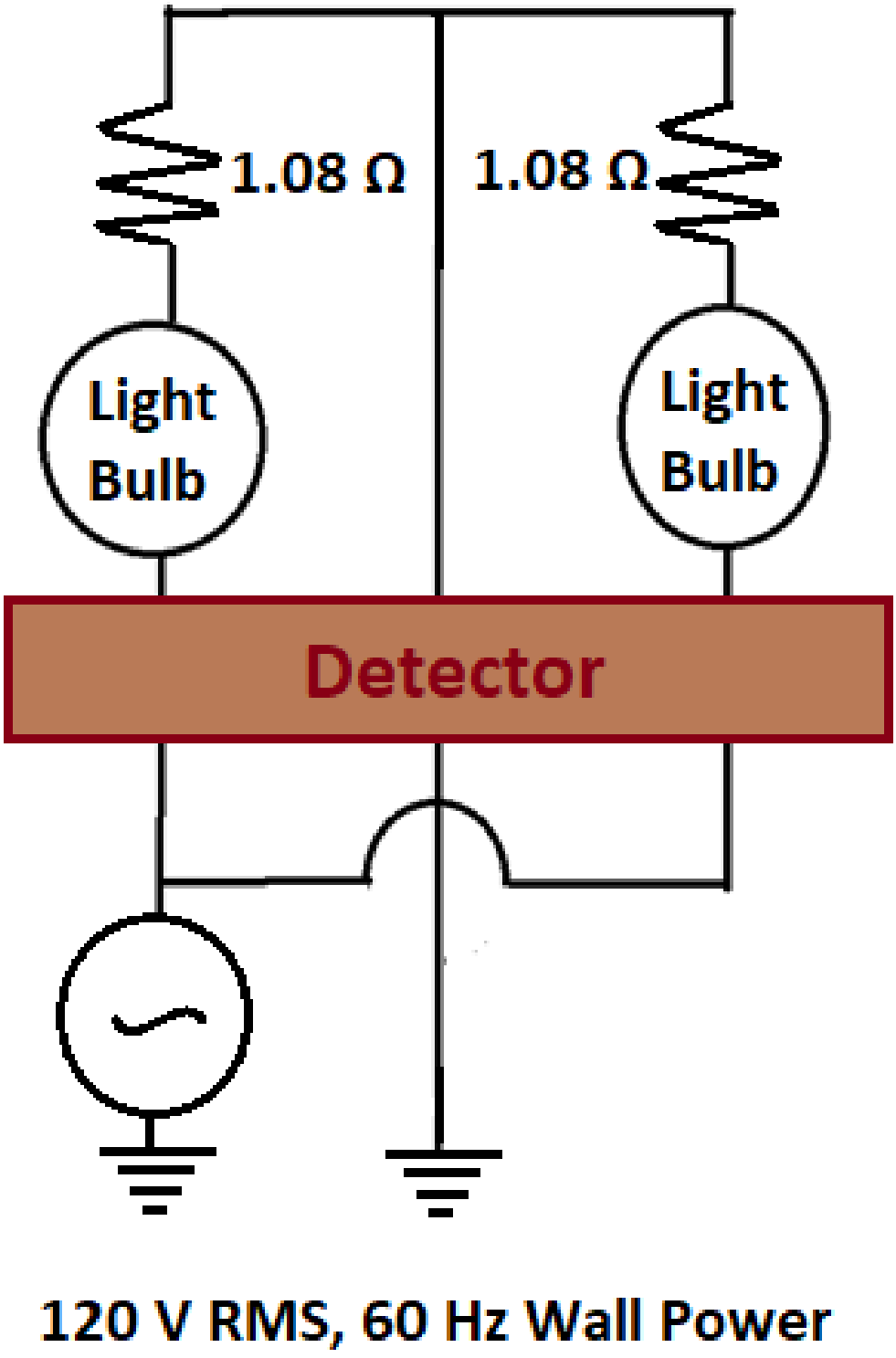

Figure 3-21: A schematic of the lightbulb demo.

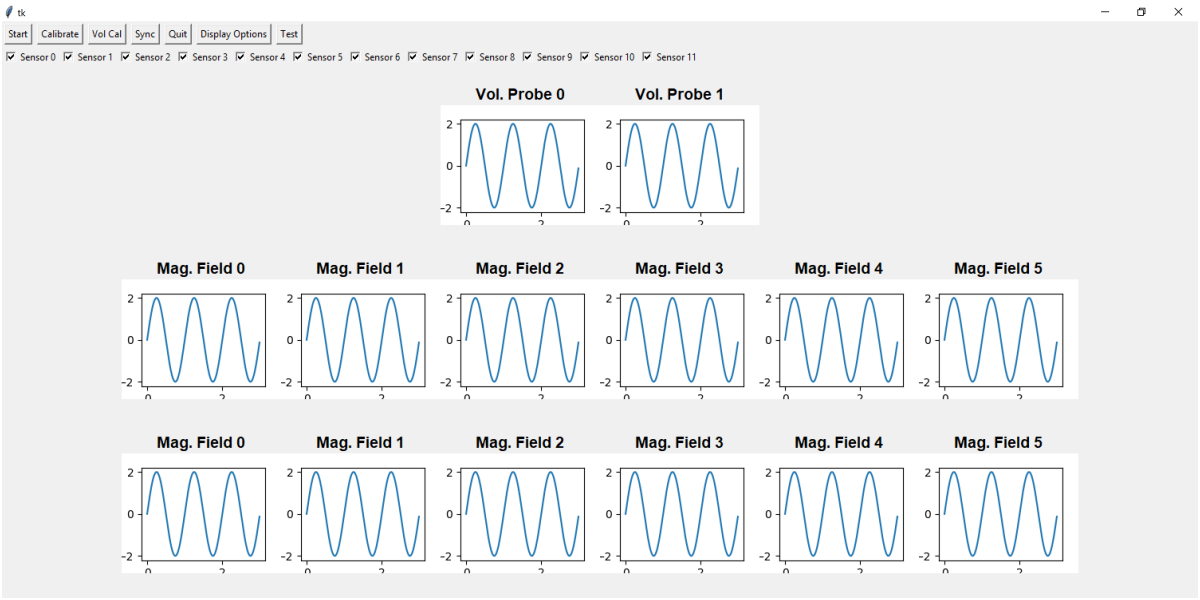

Figure 3-22: A screenshot of the live display.

Chapter 4

Current Estimation Methods

In this chapter we describe the magnetic field sensor hardware as well as the various current estimation algorithms we developed and tested.

4.1 Magnetic Field Sensors

To measure magnetic fields, we used the Texas Instruments DRV425. This chip uses a fluxgate sensor and outputs a voltage linearly proportional to the component of the magnetic field along the axis of sensitivity of the sensor. The output voltage V_{Out} is given by

$$V_{Out}(V) = 48.8\frac{mA}{mT} * R_{Shunt}(\Omega) * B(mT) + V_{Offset}(V) \tag{4.1}$$

where R_{Shunt} is the value of a shunt resistor whose value we chose to be 100 Ω and V_{Offset} is the voltage output by the sensor when no magnetic field is detected. Note that the sensor measures the component of the magnetic field along its axis of sensitivity, so if there exists an angle Θ between the magnetic field vector $\vec{B}$ and the sensor orientation, the sensor will detect the quantity $Bcos(\Theta)$. We operated the DRV425 at 5 V and selected hardware pins to set V_{Offset} to 2.5 V regardless of the power voltage level.

As seen in (4.1), the gain between voltage output and magnetic field can be controlled by the value of the shunt resistor. We selected a value of 100 Ω for our

in-lab experiments because we felt it would allow us to detect a reasonable amount of current. A 100 Ω shunt resistor would set the DRV425 output to saturate at 0.51 mT, which corresponds to a V_{Out} value of 5 V, the maximum it can output. Assuming the fluxgate sensor was located 0.7 cm from the center of the cable, this magnetic field would be created by a current of 17.5 A, which was much higher than the currents we experimented with in the laboratory.

We chose the DRV425 fluxgate sensor because of its high accuracy, low noise level, and its resistance to temperature change. According to the TI datasheet, the DRV425 has a gain of 48.76 ($\frac{mA}{mT}$) at 125 ° C, which is almost the same as its gain of 48.80 ($\frac{mA}{mT}$) at -40 ° C. Additionally, the datasheet claims the TI sensor has a noise floor of 2 nT per Hz squared, which is well below the noise of the Analog-to-Digital converter we used. Thus, in our experiments we considered the noise of the DRV425 to be negligible compared to the noise from other sources in our measurements.

4.2 Physics Simulator

We developed a physics simulator to simulate current detection experiments in software. While assembling new sensor arrays to test different configurations is a long process involving PCB board design, shipment, and manufacturing, the physics simulator allowed us to test different sensor layouts quickly and inexpensively.

The simulator was built using Python and Numpy. It consists of a set of classes that work together to simulate real life electromagnetic principles. The sources files are found in the simulator folder of the source code in Appendix A.

The simulator/sources.py file contains the definition of several classes of sources which contain a get_magnetic_field() method that returns a vector representing the x, y, and z magnetic field components. The most commonly used sources in our simulations were the UniformField class and the Wire class, which are both subclasses of the Source class. Each Source object can be assigned either a single magnetic field reading or a time series of magnetic field readings.

The simulator/sensors.py file defines the Sensor class, which requires a 3D lo-

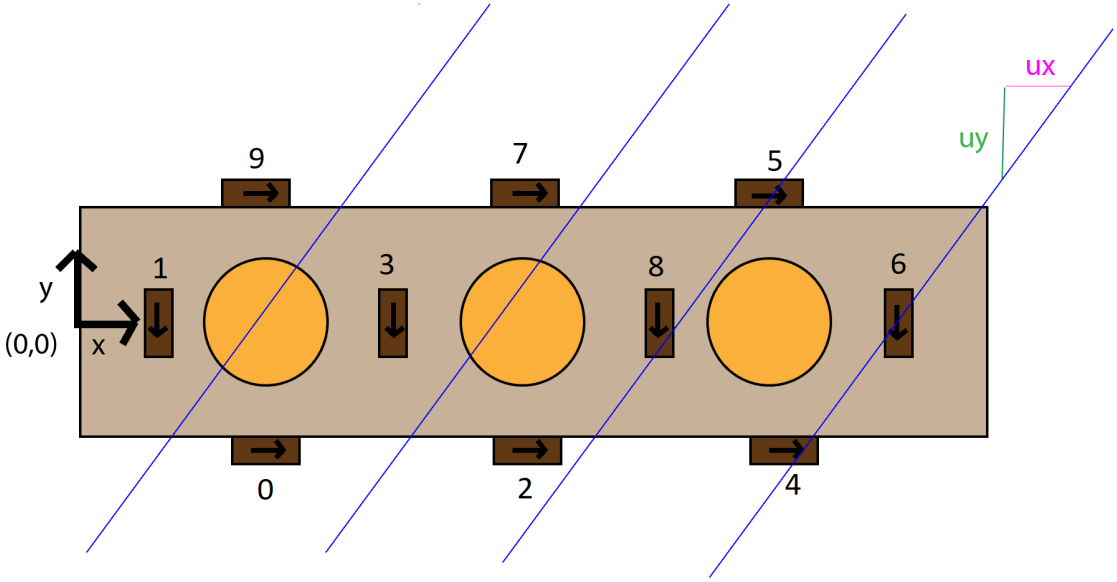

Figure 4-1: The final implementation of the magnetic field sensor array placed ten magnetic field sensors around and between the cables. The sensors are represented as dark rectangles in the figure above. The origin of the coordinate system for the elements in the array is on the left side of the yoke, aligned with the center of the cables. The yoke is 6 cm in width and 1.2 cm in height. Arrows in the magnetic field sensors show their axis of sensitivity. Also shown in this figure are uniform field lines representing one possible orientation of Earth's magnetic field. The vertical and horizontal decomposition of Earth's magnetic field, u_x and u_y, are also shown.

cation and 3D orientation to be instantiated. To detect magnetic fields, a sensor object is passed a list of Source objects into its detect() method, where it calls get_magnetic_field() on every Source in the array, calculates the dot product of the magnetic field vectors and its orientation vector, and return the sum of the results.

The simulator contains other files, such as simulator/sensor_placer.py, which contains a function that returns an array of a specified number of Sensor objects. There is also a test suite to ensure that the physics simulator is working correctly. This suite tests the results of the simulator against an answer derived by hand for a given set of test cases. These tests can be found in the file simulator/test.py.

4.3 Magnetic Field Readings and Interference

Although the locations of the magnetic field sensors varied in each of our implementations, our final implementation used 10 magnetic field sensors around a set of three cables as shown in Figure 4-1. In our simulations, we considered the origin of the x,y coordinate system to be on the left side of yoke and aligned with the center of the cables, as shown in the figure. The length of the yoke was 6 cm along the x axis and 1.2 cm along the y axis. The x,y locations of the three cables are (1.5,0.0), (3.0,0.0), and (4.5,0.0) in cm. The x,y locations of the top three sensors are (1.5,0.6), (3.0,0.6), and (4.5,0.6) in cm. The x,y locations of the bottom three sensors are (1.5,-0.6), (3.0,-0.6), and (4.5,-0.6) in cm. The x,y locations of the four vertical sensors are (0.7,0.0), (2.2,0.0), (3.7,0.0), and (5.2,0.0) in cm.

As described in Chapter 2, the magnetic field vector produced by a current will have the magnitude $|B| = \frac{u_0 I}{2\pi r}$ and will have a direction perpendicular to the vector from the center of the cable to the point at which the measurement is taken. Since the DRV425 only detects the component of a magnetic field along its axis of sensitivity, the magnetic field it will detect will be a scalar with the value $B = \frac{cos(\theta_{i,j}) u_0 I}{2\pi r_{i,j}}$, where $r_{i,j}$ is the distance between current i and sensor j, and $\theta_{i,j}$ is the angle between the axis of sensitivity of sensor j and a vector extending from cable i to sensor j.

The magnetic field $B_i(t)$ detected by each sensor can be represented as a function of the currents of the three cables as given by

$$B_i(t) = \frac{cos(\theta_{i,0}) u_0 I_0(t)}{2\pi r_{i,0}} + \frac{cos(\theta_{i,1}) u_0 I_1(t)}{2\pi r_{i,1}} + \frac{cos(\theta_{i,2}) u_0 I_2(t)}{2\pi r_{i,2}} \tag{4.2}$$

Our current estimation algorithms assume the cables and magnetic field sensors are stationary with respect to each other. Thus, the location terms $\theta_{i,j}$ and $r_{i,j}$, as well as all other constants, can be represented as a single constant for each current, and the expression simplifies to

$$B_i(t) = \alpha_{i,0} I_0(t) + \alpha_{i,1} I_1(t) + \alpha_{i,2} I_2(t) \tag{4.3}$$

where the constants are in the form $a_{i,j}$.

Since the sensors are stationary with respect to the currents, the equation describing their relationship becomes linear. Furthermore, with N sensors, we have a set of N equations that can be used to solve for the 3 unknown currents. These N equations can be expressed in the matrix form $AI = b$, where the matrix A is an Nx3 matrix. Since we have many more sensors than the number of currents being estimated, we were able to use the redundant measurements to reject noise and disturbances.

Our system does not include hardware shielding around the magnetic field sensor array, since it was our goal to perform all necessary filtering with software processing, effectively replacing the size and cost of shielding hardware with software. Thus, several types of external magnetic fields will also be detected by our sensor array and affect the accuracy of our current estimate. It was a major goal of this thesis to filter out these external sources and produce the most accurate current estimates possible.

There are several sources of external magnetic fields we considered. The most important and common are discussed below.

Earth's Magnetic Field We treated the magnetic field generated by the Earth as a spatially uniform and temporally constant field. A set of magnetic field sensors with the same axis of sensitivity would detect the same value from Earth's magnetic field, regardless of the field's orientation. For example, if the axis of sensitivity of the horizontal sensors in Figure 4-1 are pointing to the right, they will all detect the same value, u_x. A set of magnetic field sensors with the opposite axis of sensitivity would detect the value $-u_x$. A perpendicular set of sensors would detect a completely different value u_y, which cannot be determined from u_x without knowing Earth's magnetic field beforehand. We took this model into account as we developed our current estimators, as will be described in a later section.

Ambient Fields Our sensors will detect ambient magnetic fields originating from the building power infrastructure. In the US these fields exist at 60 Hz, while in other countries such as Japan they will exist at 50 Hz. Since the origin of these fields

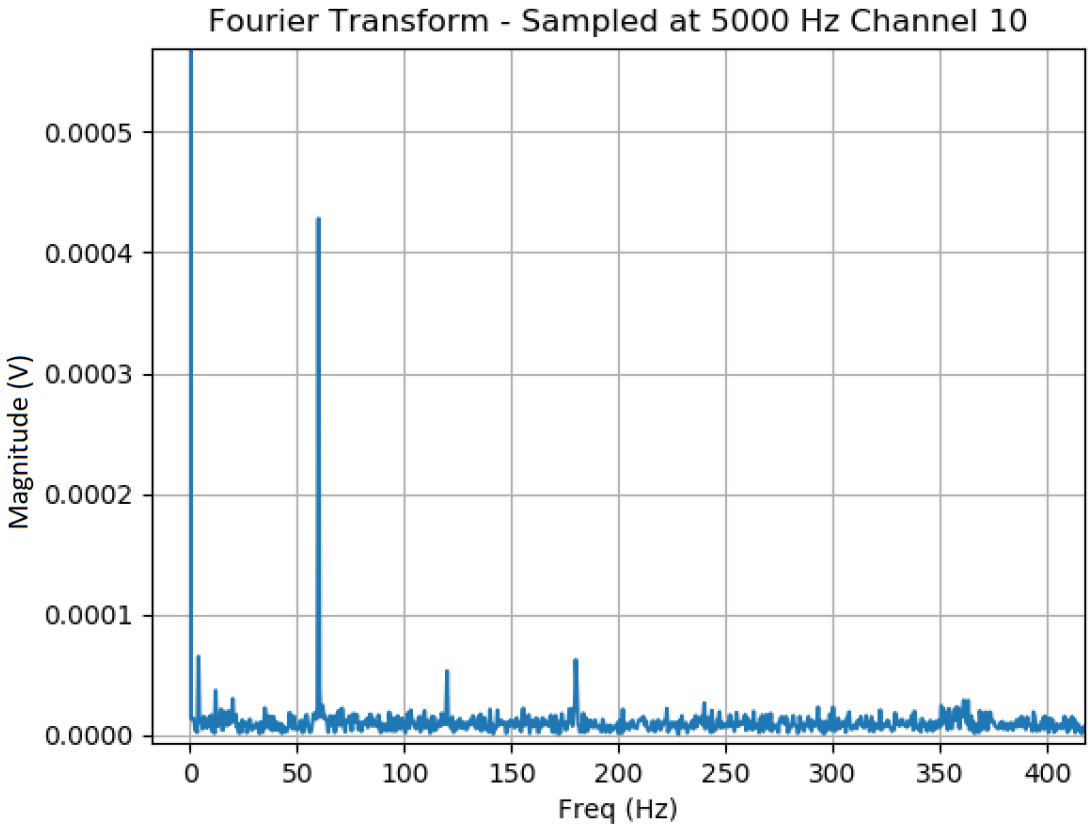

Figure 4-2: Fourier transform of a 2 second sample of magnetic field readings collected at 5000 Hz when no current was applied to the cables.

are very far away, they are effectively spatially uniform when detected by our sensor array. Thus, we modelled these fields as a time-varying but spatially uniform field.

Figure 4-2 shows the Fourier transform of magnetic field readings detected by a sensor in the array when there was no current running through the cables. As the figure shows, there is a large component at 60 Hz coming from far away cabling and building infrastructure.

Although we are aware of the ambient fields, we cannot use frequency filtering to separate these fields from the currents we are trying to detect because the currents themselves will typically also be running at the same frequency. Instead, our estimation algorithms will focus on the spatial properties of the internal and external currents and it is for this reason that we use a redundant sensor array with sensors

at different locations to collect readings and perform estimation.

We can now introduce the spatially uniform magnetic field terms into the equation modelling the fields detected by each sensor as a pair of perpendicular uniform terms, $u_x(t)$ and $u_y(t)$,

$$B_i(t) = \alpha_{i,0} I_0(t) + \alpha_{i,1} I_1(t) + \alpha_{i,2} I_2(t) + u_x(t) + u_y(t) \tag{4.4}$$

External Wires External wires running parallel to the internal wires are also detected by our sensors. However, we do not know the location of external wires beforehand and thus cannot reduce the location terms to a constant value. The magnetic field detected by an environment that includes P external wires is given by

$$B_i(t) = \sum_{j=0}^{3} \alpha_{i,j} I_j(t) + u_x(t) + u_y(t) + \sum_{k=0}^{P} \frac{cos(\theta_{i,k}) u_0 I_k(t)}{2\pi r_{i,k}} \tag{4.5}$$

External Plates A magnetic field will induce an eddy current in a conductor, which will in turn produce its own magnetic field. According to the method of images, if a current-carrying cable induces a current in a nearby thin perfectly conducting plate, the induced magnetic field will be equivalent to the magnetic field produced if instead there was a conductor on the opposite side of plate. If the plate is not perfectly conducting, the equivalent current will have a different phase and magnitude than the cable current. Thus, the magnetic field created by an eddy current in a metallic plate running parallel to the internal cables of our system can be modeled as a parallel wire and the model of (4.5) is still valid for this case.

These are the principal sources of external interference that we modeled and were concerned with. The reason we focused on parallel wires and plates is because when visiting industrial locations in the field, we observed that these are the most common forms of interference that the system we have designed will encounter. Other forms of interference, such as as magnets or spatially moving wires, are not usually present.

$$\begin{bmatrix} 0.16764 & 0.02541 & 0.00803 \\ 0.10976 & 0.04360 & 0.02613 \\ 0.02445 & 0.16388 & 0.02475 \\ -0.12797 & 0.11080 & 0.04306 \\ 0.00806 & 0.02455 & 0.16577 \\ -0.00524 & -0.02142 & -0.15877 \\ 0.02545 & 0.04530 & 0.13308 \\ -0.02222 & -0.15933 & -0.02404 \\ 0.04487 & 0.13839 & -0.10962 \\ -0.15754 & -0.02428 & -0.00603 \end{bmatrix}$$

(a) The empirical matrix

$$\begin{bmatrix} 0.16016 & 0.024495 & 0.00656 \\ 0.12093 & 0.04238 & 0.02567 \\ 0.02061 & 0.160164 & 0.02449 \\ -0.13925 & 0.121880 & 0.04242 \\ 0.00596 & 0.020615 & 0.16016 \\ -0.00596 & -0.02061 & -0.16016 \\ 0.02636 & 0.04431 & 0.13784 \\ -0.02061 & -0.16016 & -0.02449 \\ 0.04435 & 0.13925 & -0.12188 \\ -0.16016 & -0.02449 & -0.00656 \end{bmatrix}$$

(b) The theoretical matrix

Figure 4-3: Two different gain matrices for the 10 sensor current estimation system. The values are the gains between the cable currents and the output of the DRV425 sensors, and the units are in V/A. The empirical matrix was obtained by running known currents through one cable at a time using real hardware. The theoretical matrix was generated using the physics simulator.

4.4 The Gain Matrix

Our estimation algorithms depended on knowing the gains between each current and the detected magnetic field, expressed as $\alpha_{i,j}$ in (4.3). These gains are the terms of the matrix A when expressing the relationship between the current vector I and the magnetic field vector b as the matrix equation $AI = b$.

Figure 4-3b shows the matrix calculated by our physics simulator for the configuration of 10 sensors shown in Figure 4-1. Figure 4-3a shows the matrix calculated by the calibration procedure using hardware measurements. Observe that this matrix is similar, but not equal to, the theoretical matrix calculated by the physics simulator. To calculate the matrix using our hardware, we applied a range of DC currents through each cable, one cable at a time. Figure 4-4 shows the measured output of sensor 3 for three ranges of currents run in each cable. As the figure shows, the magnetic fields detected are a linear function of the current being applied. The slope of the linear function detected by sensor j when a range of currents was applied to cable i is the i, j-th term of the matrix A.

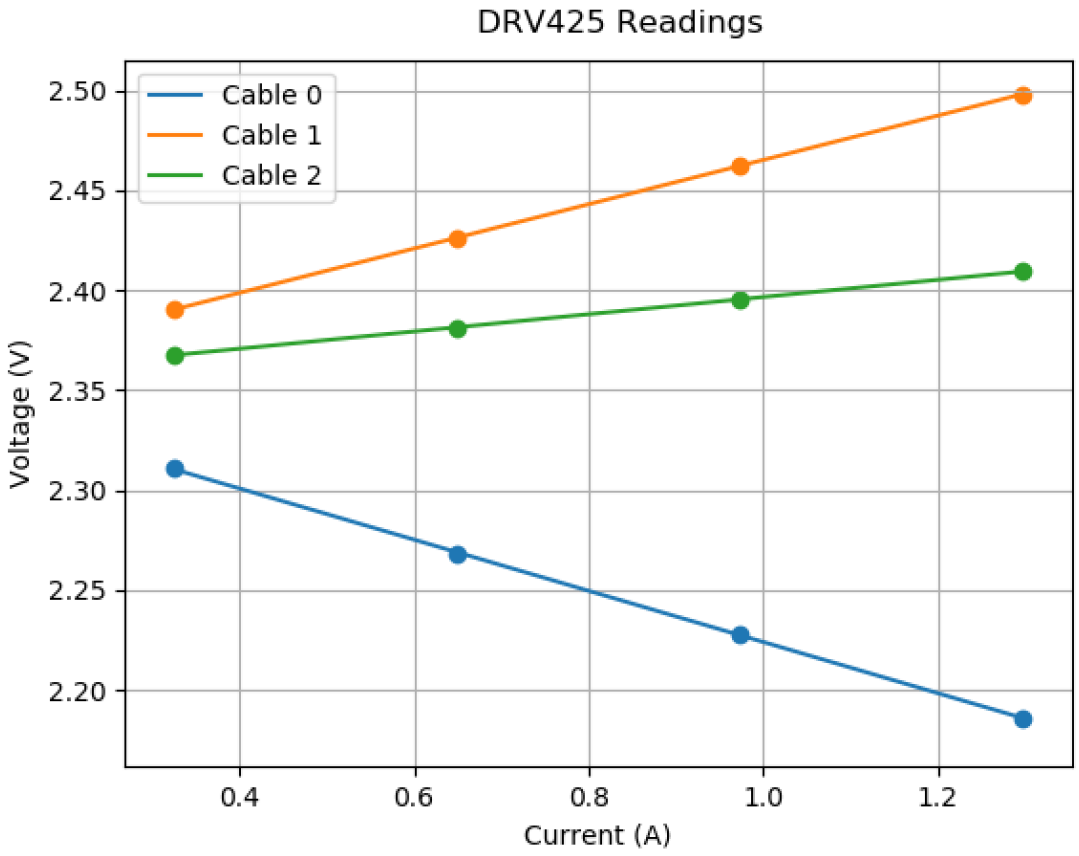

Figure 4-4: A graph showing the measured output values of a sensor in the sensor array when known currents were independently run through each cable.

The matrix we measured using hardware was repeatable and consistent. The difference between the theoretical and empirical matrices could be attributed to two factors. One factor is inaccurate placement of the magnetic field sensors. For example, screwing a nylon screw too loosely could cause the PCB board housing the sensors to be misaligned by a few microns relative to the yoke holding the cables. However, this cause alone would not create deviations large enough to explain the discrepancy between the theoretical and empirical matrices.

Instead, a more important factor was the effect of the magnetic fields generated by the currents powering the sensors. Despite our best efforts to run power and ground traces in the PCB board in parallel and to maintain them at an orientation that would minimize pickup by the magnetic field sensors, these power currents will still contribute a small amount of magnetic field. Since the DRV425 is a closed-loop

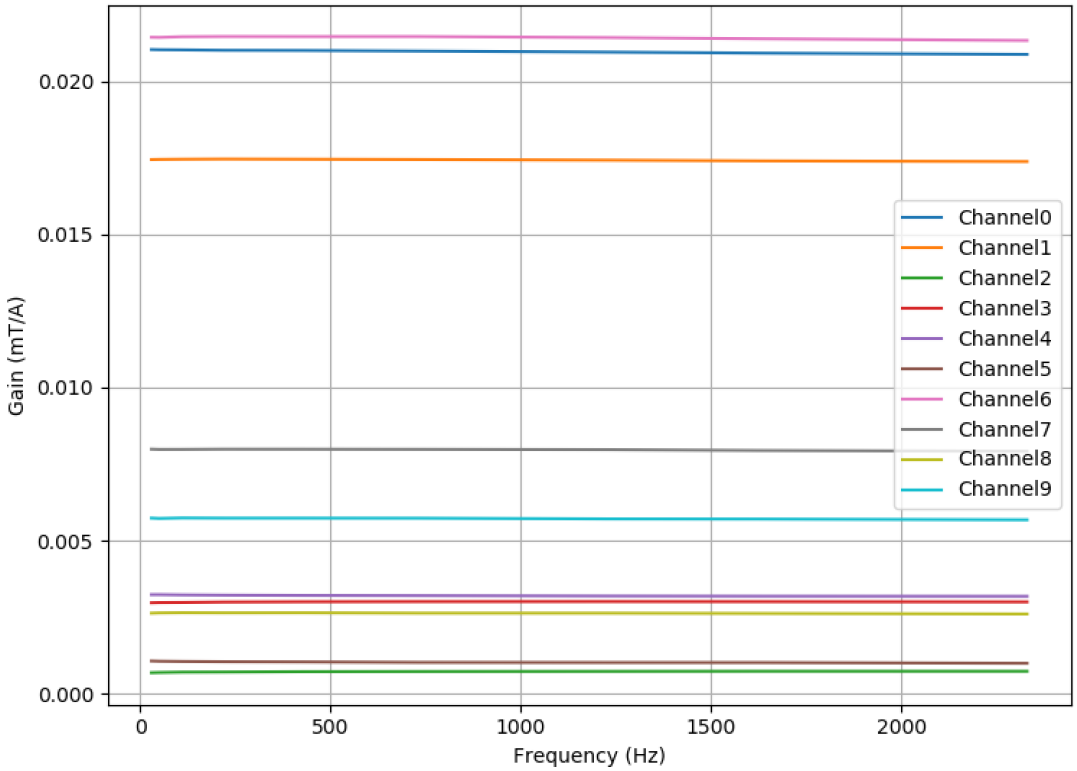

Figure 4-5: Sensor gains measured as a function of current frequency.

sensor, the greater the magnetic field it detects, the more current it will draw, thus contributing to the gain between cable current and detected magnetic field.

However, the gains between magnetic field sensors and cable currents are both linear and repeatable. Thus, once we obtain the empirical matrix, we can use it for all our estimation methods. We will discuss how we dealt with the issue of power trace interference as we describe the methods.

We also tested the effect that the frequency of the current had on the gains between cable and sensor. As Figure 4-5 shows, the gains did not change significantly up to the frequencies of interest.

4.5 Sensor Placement

We investigated the optimal placement of the magnetic field sensors to minimize current estimation error. We were free to place the sensors along the top and bottom planes formed by the PCB boards attached to the two halves of the yoke. In addition, we also considered placing sensors vertically, along the sides of the yoke as well as in slots between the cables.

Since the magnetic fields created by the internal and external cables is the superposition of the fields created by the cables individually, and since the majority of the current estimation methods we tested are linear, the estimation error can be expressed as the result of applying the magnetic fields from the external cables through the estimator. For example, the error between the true current I and the current estimate $\hat{I}$ formed by the Ordinary Least Squares Estimator is

$$I - \hat{I} = (A^T A)^{-1} A^T b_{internal} - (A^T A)^{-1} A^T b_{total} \tag{4.6}$$

where $b_{internal}$ is the vector of magnetic fields created by the internal currents and $b_{external}$ is the vector of magnetic fields created by external sources. The detected field b_{total} is the sum of the fields from the internal and external cables, so

$$\begin{aligned} I - \hat{I} &= (A^T A)^{-1} A^T b_{internal} - (A^T A)^{-1} A^T (b_{internal} + b_{external}) \\ &= (A^T A)^{-1} A^T b_{external} \end{aligned} \tag{4.7}$$

Thus, one approach to optimizing sensor placement is to find a placement that will minimize (4.7). While the location of external cables are not known beforehand, we aimed to find the position of the sensors with the smallest worst case error, or, in other words, a sensor placement configuration that performed the best under the most adversarial placement of a single external cable. This optimization problem can be represented as

$$min_{(x_0,y_0),\ldots,(x_N,y_N)} max_{(x_e,y_e)} sum(|(A^T A)^{-1} A^T b_{external}|_1) \tag{4.8}$$

where the location of the external wire is given as (x_e, y_e) and the locations of the sensors are given by $(x_0, y_0), ..., (x_N, y_N)$.

We sum the absolute value of $(A^T A)^{-1} A^T b_{external}$ because this expression is a vector of three currents. Note that the terms in the matrix A will involve the sensor location variables $(x_0, y_0), ..., (x_N, y_N)$ and the terms in the magnetic field vector will involve those variables as well as the external cable location variables (x_e, y_e). In this scenario, we will assume the external current is running 1 A.

To solve this problem and gain an intuitive understanding of the solution, we decided to create a graphical user interface in which we could experiment with different positions for each sensor. We created a program in which we could drag and drop magnetic field sensors with a computer mouse. Upon pressing a 'simulate' button, the program would generate a heatmap where each point around the yoke was colored a shade of red corresponding to the magnitude of the term $sum(|(A^T A)^{-1} A^T b_{external}|_1)$ due to a single 1 A external cable at the given point. The source code of the program can be found in the file display_heatmap.py in Appendix A. Heatmaps for four location configurations for a set of six sensors are shown in Figures 4-6, 4-7, 4-8, and 4-9.

What we found through the use of this program was that the optimal location of the sensors is when they are as close as possible to the internal cables and as far away from each other as possible. For six sensors, this condition is satisfied in Figure 4-6, in which the maximum error is 0.177 A. Moving two sensors closer together, but further from the cables, as shown in Figure 4-7, while still staying away from the outside area in which there can be external cables, still causes an increase in maximum error, bringing it up to 0.277 A. The worst of the four cases is shown in Figure 4-7, where the sensors are close to the area where an external cable is allowed. Since the sensors are so close to this area, they strongly detect external cables, and the worst case error is 1.304 A.

We considered these results when designing the previously mentioned sensor placement function in the physics simulator. The sensor placement function places the first 6 sensors directly above and below the three cables, as shown in Figure 4-10. When

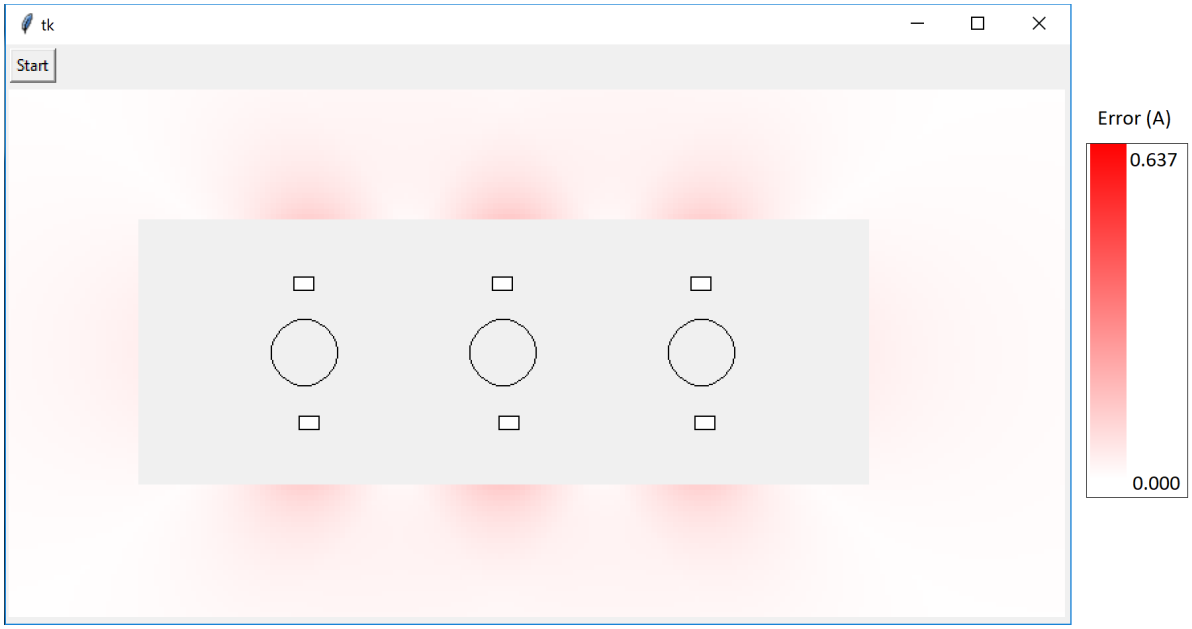

Figure 4-6: A heatmap showing estimation error when the sensors were placed directly over and under the sensors. The worst case error is 0.177 A.

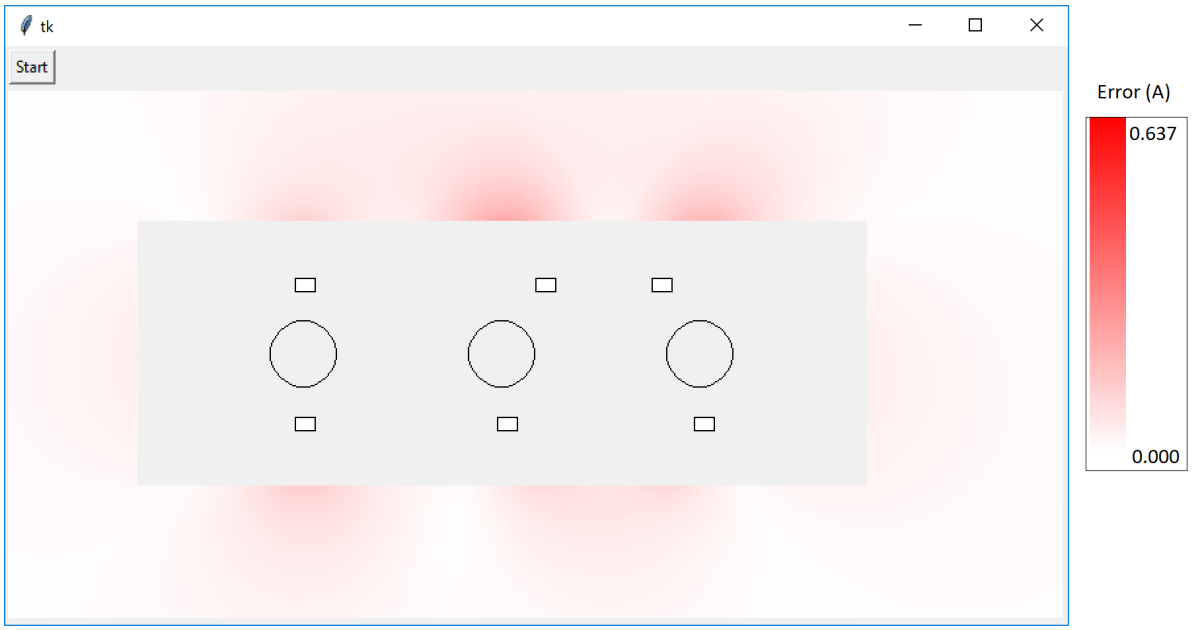

Figure 4-7: A heatmap showing estimation error when two of the sensors were placed closer to each other. The worst case error is 0.277 A.

more sensors are requested from the placement function, they are added vertically, until 10 sensors are placed as shown in Figure 4-11. Note that this configuration of 10 sensors is very similar to the sensor configuration that the final version of our system

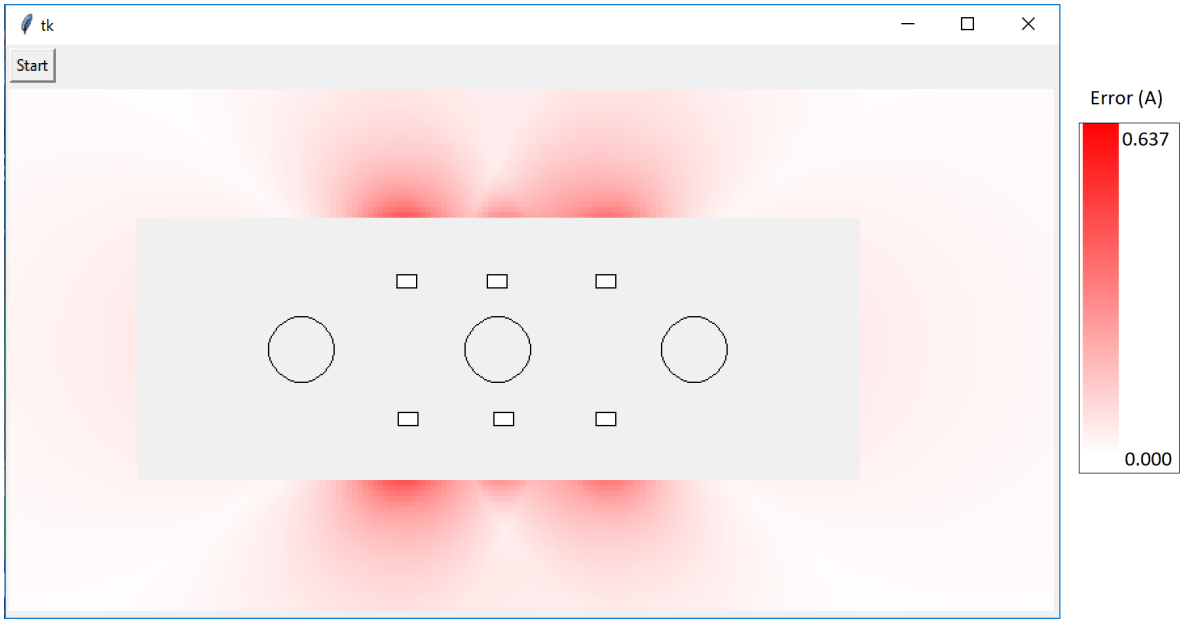

Figure 4-8: A heatmap showing estimation error when several sensors have been placed closer to each other. The worst case error is 0.572 A.

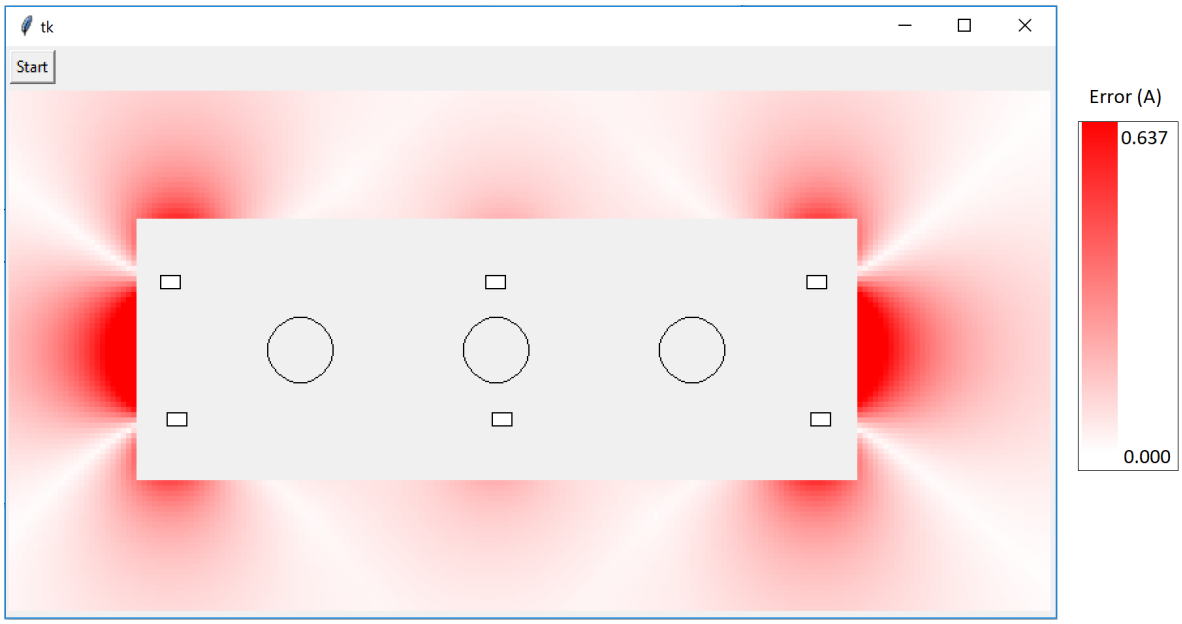

Figure 4-9: A heatmap showing estimation error when four of the sensors have been placed at the edge of the area in which no external sources of interference can exist. The worst case error is 1.304 A.

used.

As more sensors are placed by the sensor placement function, they are added in between existing sensors to fill in remaining gaps in a way that keeps sensors are far

Figure 4-10: A heatmap showing estimation error when six sensors are used. This heatmap is similar to the one shown in Figure 4-6, but the color scale has been changed to facilitate comparing this heatmaps with other heatmaps containing different numbers of sensors. The worst case error was 0.189 A.

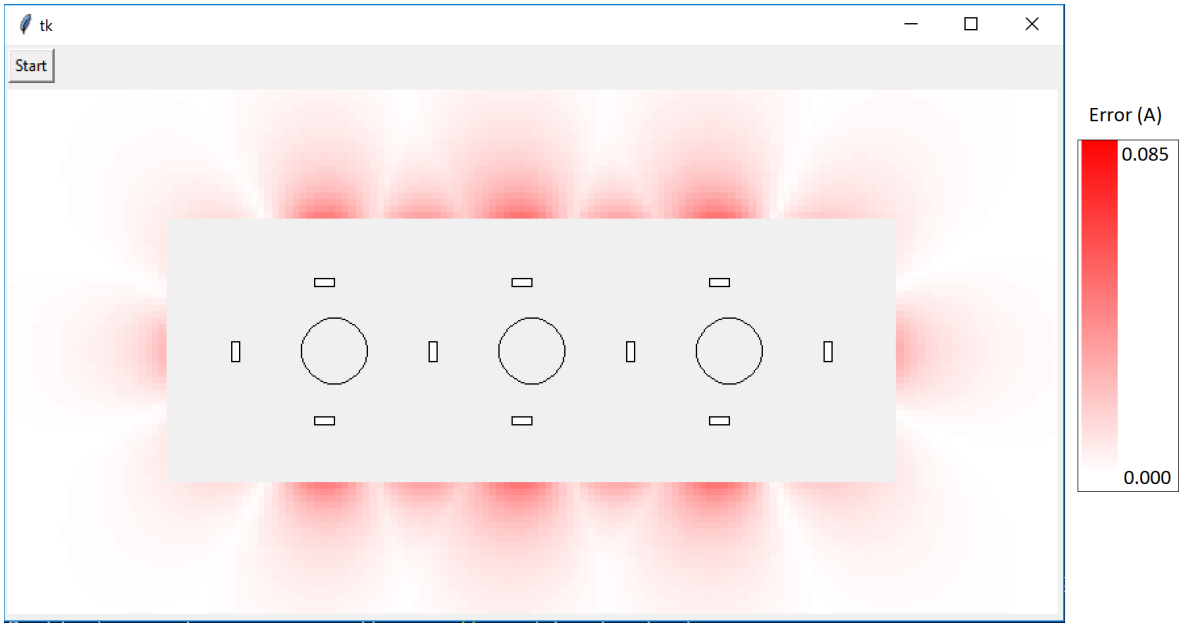

Figure 4-11: A heatmap showing estimation error when 10 sensors are used. The worst case error was 0.084 A.

apart from each other as possible. Figures 4-12 and 4-13 show the placements for 36 and 76 sensors, respectively.

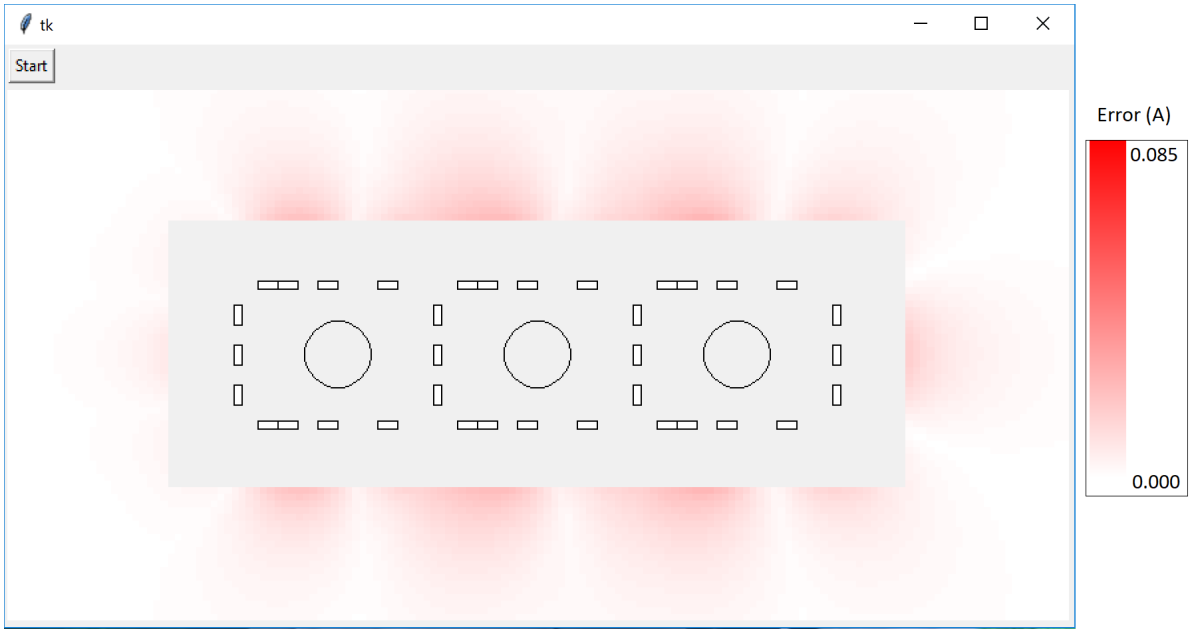

Figure 4-12: A heatmap showing estimation error when thirty-six sensors are used. The worst case error was 0.025 A.

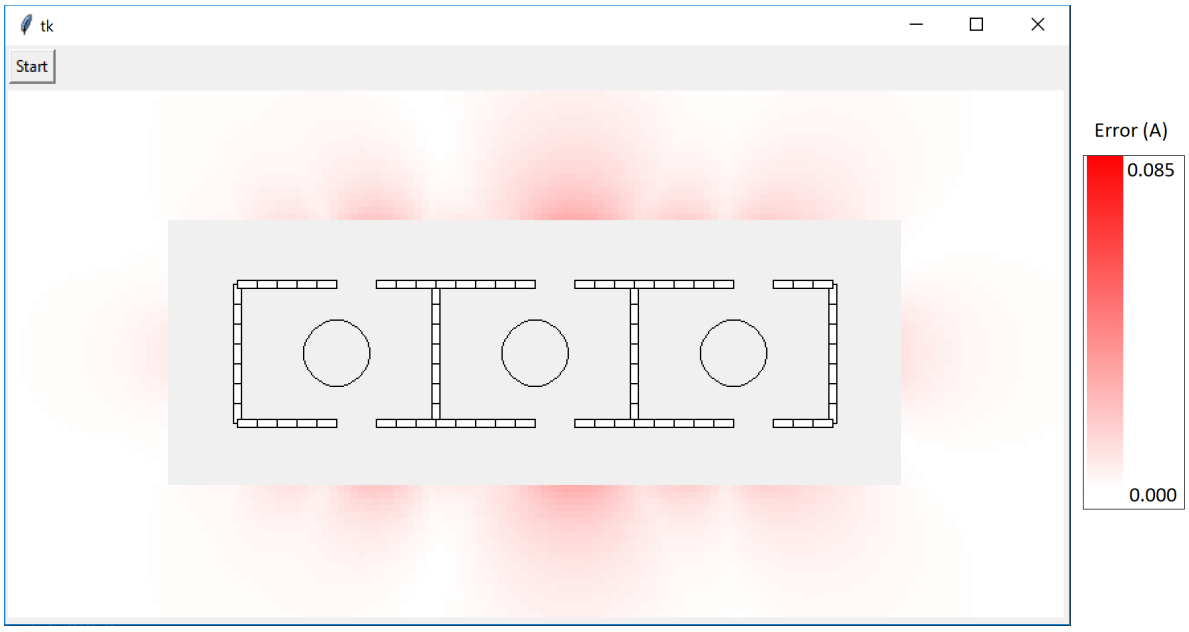

Figure 4-13: A heatmap showing estimation error when seventy-six sensors are used. The worst case error was 0.027 A.

Since the internal fluxgate sensor of the DRV425 effectively measures the magnetic field over a line integral of 1.5 mm, it is possible to approximately measure the field around a closed loop using enough DRV425 sensors. In the scheme we use in our

sensor placer, it takes 88 sensors to form a closed loop around each cable. Thus, the maximum amount of sensors that the sensor placement function can place is 88.

This sensor placement function was used to evaluate the performance of different current estimation methods. The results of different estimators as a function of the number of sensors used will be presented in later sections. The final sensors system has 10 sensors, and they are located in the positions stated at the beginning of Section 4.3.

4.6 Error Measurement

To evaluate different current estimation methods, it was important to have a common error measurement and test scenario with which to compare their performance.

The error measurement we chose is

$$\%Error = \sum_{n=0}^{2} \frac{|I_n - \hat{I}_n|_1}{|I_n|_1} * 100 \tag{4.9}$$

where I_n is the true current in cable n and $\hat{I}_n$ is the estimated current in cable n. We chose this error measurement because in many cases the currents we evaluate are balanced three phase currents, and in such a case the value $\sum_{n=0}^{2} |I_n|_1$ is never zero, allowing us to avoid division-by-zero problems.

Selecting a common test scenario was a more nuanced challenge. The test scenario we considered in the previous section, one where there is a single external cable with 1 A of current, is not the worst case scenario and may not be indicative of the performance of an estimator when many external cables are present. In addition, if there are multiple external cables present, the estimation error can in some cases go down, since it is possible for the magnetic field of another external cable to cancel the field from the first external cable, which is a common scenario in real industrial environments where external cables exist in triplets of balanced three phase currents.

We briefly considered using an integral of infinitely many external cables with 1 A of current located at all possible points outside the yoke area as a worst case scenario,

but this integral does not converge. For example, even a contiguous section of this integral, such as the area above the yoke bounded by the coordinates x=0.0 m to x= 0.060 m and y=0.01 m to y=infinity, produces an infinite magnetic field,

$$\int_{0.01}^{\infty}\int_{0.0}^{.060}\frac{u_0(x_i-x_e)I}{2\pi(y_i-y_e)^2+(x_i-x_e)^2}dxdy=\infty \tag{4.10}$$

In addition, (4.10) is not physically realistic and would never be found in an actual industrial environment.

Instead, we decided to create a set of four realistic test cases for each estimation method that represents situations typically found in industrial environments that our detector would have to deal with. These four cases are described below.

No Interference We tested the case when there is no external magnetic field interference. The three internal wires are running currents of -0.7 A, 1.0 A, and -0.3 A, respectively (a balanced configuration).

One External Cable We tested the case when there is a single parallel external wire running 1 A located at the point (0.03 m, 0.01 m), above the middle cable. The internal wires are running current as in the case with no interference.

Parallel Plate We tested the case when there is a perfectly conducting metallic plate located at y=0.01 m. The internal wires are running current as in the case with no interference.

Six External Cables We tested the case when there are six external wires. These wires are made up of two sets of three balanced currents. The first set is made up of wires located at (0.015 m, 0.015 m), (0.03 m, 0.015 m), and (0.045 m, 0.015 m) running -0.7 A,1.0 A,and -0.3 A, respectively. The second set is made up of cables located at (0.006 m,-0.015 m), (0.03 m,-0.015 m), and (0.04 m, -0.015 m) running currents of -0.7 A, 1.0 A, and -0.3 A, respectively. The internal wires are running current as in the case with no interference.

We felt the above cases represented situations that our system would typically encounter in industrial environments, and used these cases to compare the performance of our estimators.

4.7 Signal Pre-Processing

Before using the magnetic field readings to form a current estimate, we had to pre-process the readings for three reasons. First, we needed to remove noise in our readings introduced by the ADC. Second, we needed to correct a time shift that existed in the readings because the ADC sampled voltages consecutively, not simultaneously. Third, we needed to filter Fourier components in our readings that we knew did not originate from the currents.

4.7.1 De-Noising Filter

As explained in Chapter 3, the readings from the ADC are corrupted with a small amount of Gaussian sensor noise which introduced error in our current and voltage estimates. To graphically illustrate this problem, we performed an Ordinary Least Squares estimate on magnetic field readings collected using 10 sensors when 1.30 A of DC current was run through a single cable. Figure 4-14 shows the current estimate corrupted by noise superimposed on the true current measured using a contact measurement. Figure 4-15 shows the current estimate after the magnetic field readings were cleaned with our custom noise removal algorithm. We will now present the two digital noise removal algorithms that we experimented with: a custom filter we developed ourselves and a Wiener filter.

Custom Noise Removal Filter

The custom filter came from the observation that the magnitudes of the Fourier component of the signals we were estimating were significantly larger than the Fourier components of the sensor noise, which exhibited as white noise, as seen in Figure 4-2. Our custom filter involved removing all Fourier components with a pre-selected

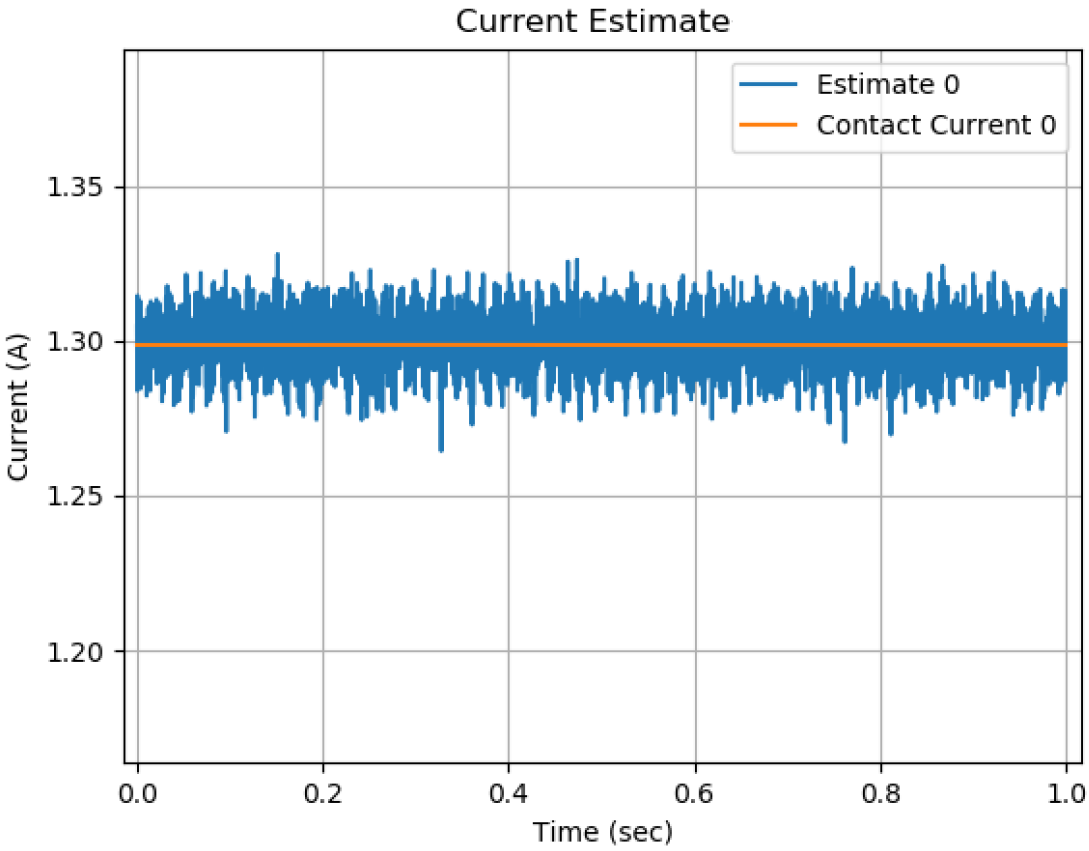

Figure 4-14: The true DC current applied to cable, calculated by measuring the voltage drop across a 5.08Ω resistor, superimposed over a current estimate for which noise has not been removed.

threshold value. The transfer function of our filter is

$$H(j\omega) = \begin{cases} 0 & \text{if } |S(j\omega)| < \alpha \\ 1 & \text{if } |S(j\omega)| >= \alpha \end{cases}$$

where $S(j\omega)$ is the Fourier transform of the signal and α is the threshold value.

This filter is not linear. To show this, let us consider the sum of two signals, $s_1(t)$ and $s_2(t)$, where each signal has a Fourier component at some frequency f of magnitude 0.75α. In this case, the Fourier component of $S_1(j\omega)H(j\omega)+S_2(j\omega)H(j\omega)$ at frequency f will be 0. However, if the filter were linear, the Fourier component at frequency f of $(S_1(j\omega) + S_2(j\omega))H(j\omega)$ would be also be 0, but instead it is 1.5α, so

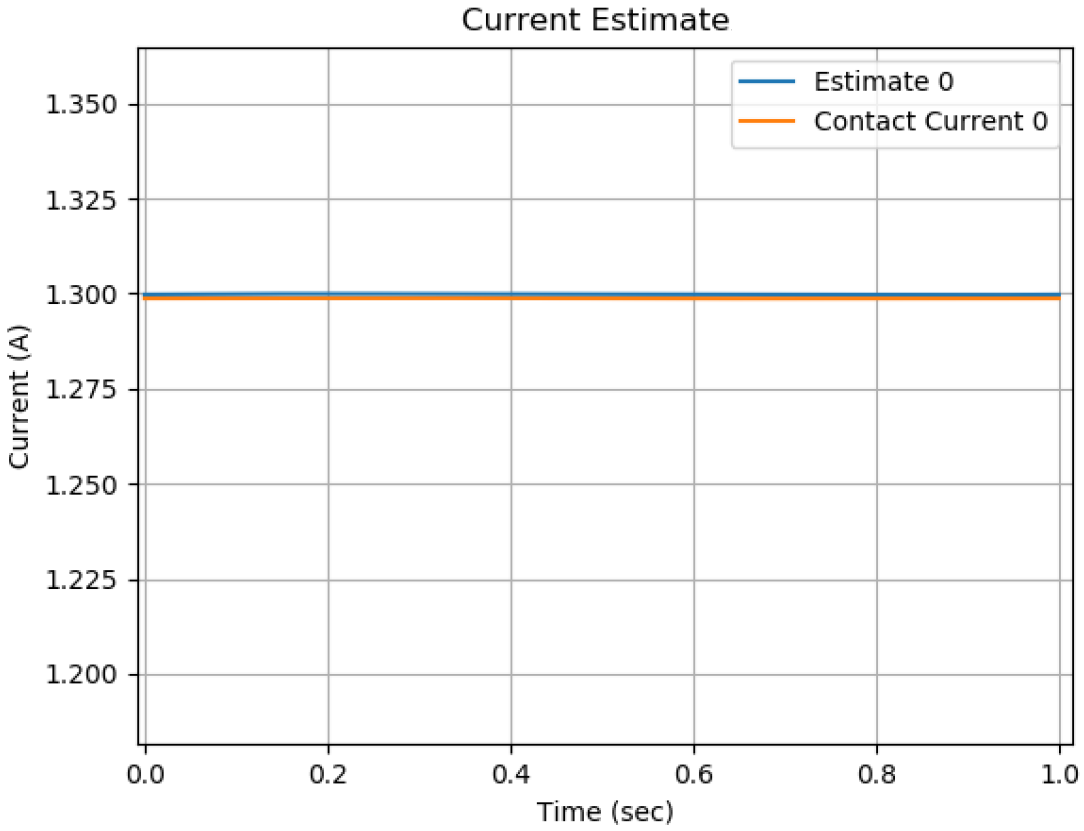

Figure 4-15: The true DC current applied to a cable, superimposed over an estimate in which noise has been removed with the Custom Noise Removal filter.

the filter is not linear.

Nevertheless, we believed the non-linear properties of the filter would have a minimal impact on the current and voltages estimation methods we later experimented with. An area of future research can be to investigate how the non-linear properties of this noise removal filter affects the performance of the linear estimation methods that are used after it.

A requirement of our filter was choosing an appropriate threshold α. A threshold that is too high can eliminate signals useful to our estimate while a threshold that is too low can leave sensor noise in our readings. We thus chose the threshold appropriately by observing a Fourier transform of a sample reading. A typical value of α for the output of a DRV425 sensor was 0.0005 V.

This filter was very effective in removing noise and improving the accuracy of

our estimates. For example, in an experiment in which we estimated 1.67 A 90 Hz currents in a balanced three phase configuration using the Parallel Cables test bed, using the noise filter reduced current estimation error from 0.93% to 0.77%.

Wiener Filter

We also tested the noise removal performance of a Wiener filter. To do this, we used the wiener() function from the scipy library, located in `scipy.signal.wiener`.

Although this filter improved the current estimate, reducing error from 0.93% to 0.88% in the balanced three-phase current experiment, it did not perform as well as the custom filter we developed. Thus, we continued using the custom filter for all our experiments. It is worth noting, however, that the generic SciPy implementation of the Wiener filter can potentially be customized for our signals, and attempting to design a Wiener filter specifically for the signals we process could be an area for further research.

4.7.2 Time-Shifting Filter

The readings collected by each ADC were consecutive, not simultaneous. This means that for a given time sample, the voltages collected by each of the eight channels were not collected at the same instant in time t_0. Rather, at a given sampling frequency f_s, each channel i sampled its voltage at the time $t_0 + \frac{1}{f_s}\frac{i}{8}$. Since our current estimation algorithms assume all magnetic field readings are collected at the same moment in time, this time delay between ADC channels introduces error into our estimates.

To correct this, we created a digital time shifting filter. The filter takes the Fourier transform of a signal, multiplies it by a unity gain phase shift filter, and returns the inverse Fourier transform of the resulting signal. The transfer function of this time shift filter is below.

$$H(j\omega) = e^{j\omega t_0} \tag{4.11}$$

Figure 4-16 shows a plot of readings collected by the ADC when a 2 VPP 130 Hz

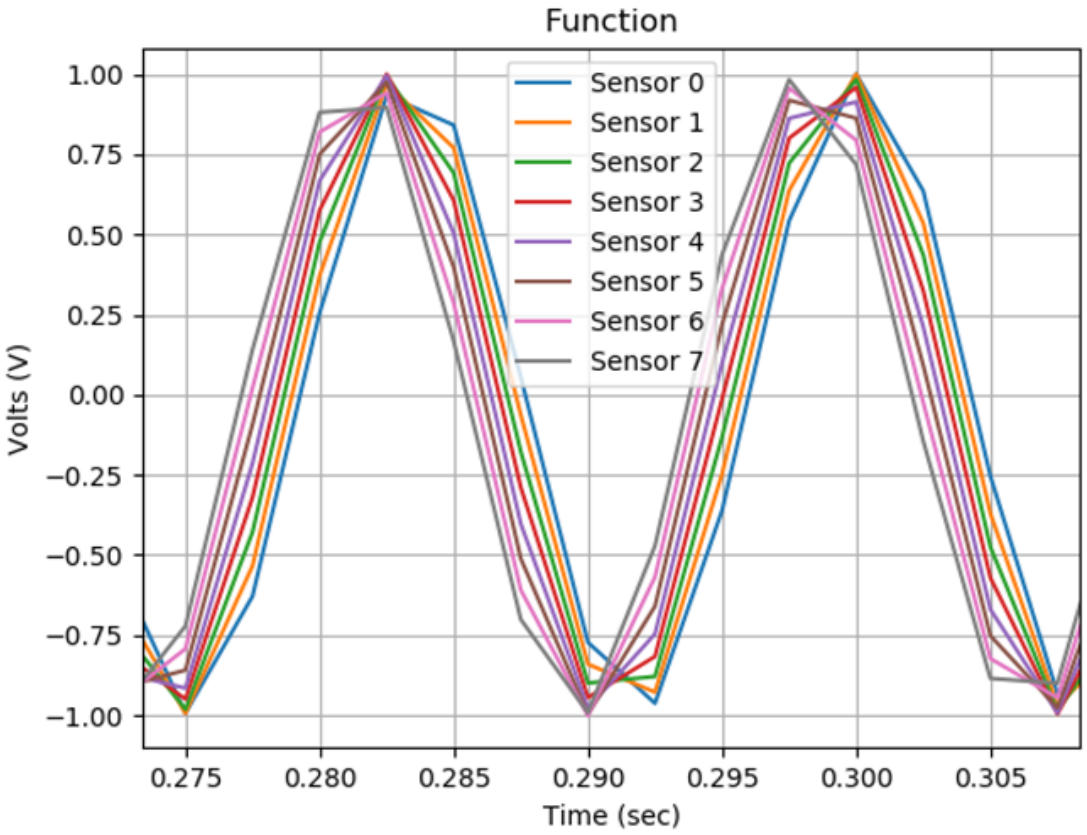

Figure 4-16: A plot showing readings from the ADC when 2 VPP 130 Hz voltage was simultaneously applied to all 8 channels. Since the ADC collects readings consecutively, the signals appear out of phase.

signal was simultaneously applied to all 8 channels of the USB-205. The plot assumes all channels were sampled at the same time. Since they were not, the signals appear shifted. Figure 4-17 shows a plot of the same readings after our time shift filter was applied. The eight signals are now lined up.

One problem with our time shift filter is that the inverse Fourier transform of the shifted signals will contain an imaginary term for even-length input signals. The reason for this is that the Fourier transform of real signals must possess the conjugate symmetry property, which states that $S(j\omega) = S(\bar{-}j\omega)$. However, the DFT of an even-length signal will contain only a single entry for the Nyquist frequency, without an entry for the negative Nyquist frequency. Thus, if the Nyquist frequency

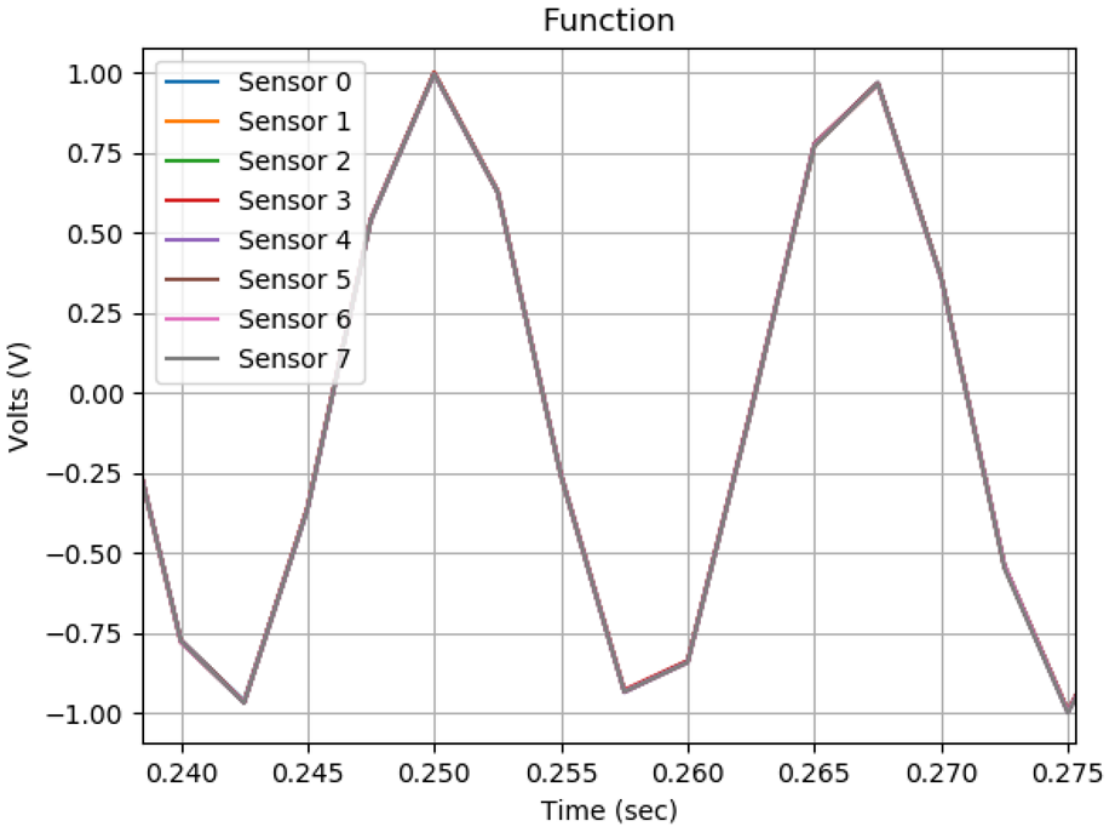

Figure 4-17: A plot of the the readings from the ADC when 2 VPP 130 Hz voltage was simultaneously applied to all 8 channels. The signals have been time shifted, and now correctly appear in phase.

component is not 0, it is not possible to maintain the conjugate symmetry property for the Nyquist frequency, and application of our filter will result in a small complex component in the output.

One possible solution is to always use odd-length signals, perhaps by dropping a time sample to turn an even-length signal into an odd-length signal. Another solution is to simply drop the imaginary term, as the Nyquist frequency component is typically small in magnitude as long as the sampling frequency has been appropriately selected. We chose the latter solution, since the value of the imaginary term was small, typically in the 10^{-9} V range compared to the values of the real term which were typically in the 10^{-3} V range.

We saw significant improvement in our current estimation when applying the time-

shift filter. In the experiment running 1.67 A 90 Hz balanced three phase currents, the estimation error without the time-shifting filter was 2.04%, but with the filter it was 0.77%.

An additional use of the time shift filter was to synchronize readings from multiple ADC units. Since our system used a total of 13 sensors, 10 magnetic field sensors and 3 voltage sensors, and each ADC only offered 8 analog channels, we used two ADC units to collect readings. However, the two USB-205 and USB-231 ADC units we used did not have the ability to operate synchronously in a slave-master configuration. Instead, we connected channel 0 from the first ADC, which was also collecting magnetic field readings, to channel 0 from the second ADC. We then detected the frequency with the largest magnitude (without considering the 0th frequency) and compared the phase shift ϕ of this frequency between the two channels. Using the formula $t_{lag} = \frac{\phi}{2\pi f}$, we identified the time lag between the two ADC units. We then used the time shift filter to shift all samples from the second unit by t_{lag}.

This synchronization scheme was effective. In the balanced three-phase current experiment, applying this synchronization algorithm reduced the error 60.4% to 0.77%. Thus, we conclude that the time shift filter is correctly synchronizing the inputs of the two ADC units.

4.7.3 Rogue Frequency Filter

Before using the anti-aliasing filter, another source of noise in our measurements manifested in the form of frequencies that were not common to all magnetic field sensors. Due to the proximity of the sensors to the currents, we expected all magnetic field sensors to detect signals at the same frequencies, albeit at different magnitudes for each sensor. However, as Figure 4-18 shows, signals from different sensors exhibited a significant amount of noise at different frequencies, a problem we termed 'rogue frequencies'.

To fix this issue, we applied a filter that would eliminate a frequency if its magnitude was not above the noise threshold value α for all sensor channels, similar to our custom noise removal algorithm. This was very effective in eliminating rogue

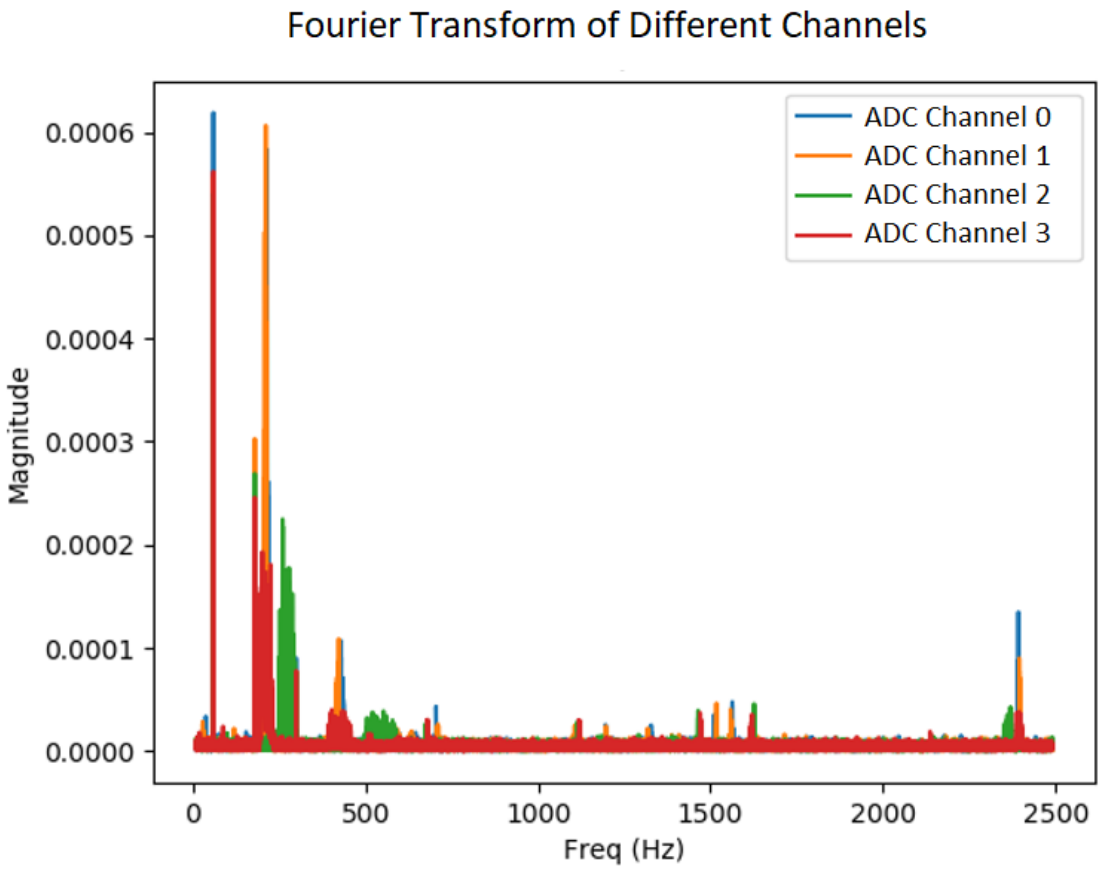

Figure 4-18: Fourier Spectrum showing that different sensors exhibited unique noisy Fourier components.

frequencies. For the balanced three-phase experiment, applying this filter reduced estimation error from 0.90% to 0.77%. However, when we started using the anti-aliasing filters, we did not see the rogue frequencies in our readings, indicating they were high-frequency signals. It is possible these were WiFi or Bluetooth signals being picked up by the wiring connecting the PCB board to the ADC units.

4.8 Current Estimation Methods

We will now present the different current estimation methods we developed, adapted, and evaluated. These methods include physics-based methods, in which we modeled not only the internal currents but also sources of interference, as well as machine-

learning based methods, in which we fit a training set of magnetic field readings to their corresponding true current values. As previously mentioned, to determine which method performed best, we used the physics simulator and sensor placement function to simulate each method when using a different number of sensors for each of the four test cases previously introduced.

4.8.1 Ordinary Least Squares Estimator

Ordinary Least Squares (OLS) is a common technique used to solve a system of equations when the number of equations is greater than the number of unknowns. In our setup, the number of sensors N is greater than the number of currents we are measuring, and we can use OLS to form a current estimate. Since the relationship between the vector of currents and the vector of magnetic field readings is given by $AI = b$, the Ordinary Least Squares Estimate is

$$\hat{I} = (A^T A)^{-1} A^T b \tag{4.12}$$

where $\hat{I}$ is the estimate, a vector of three currents.

The residuals of a least squares estimate indicate how well the magnetic field readings fit the matrix A. The residual vector can be defined as the difference between the magnetic field readings and the result of multiplying the matrix A by the estimated current, which is effectively the estimated magnetic field reading vector implied by the estimated current. This is written as

$$r = b - \hat{b} = b - A\hat{I} \tag{4.13}$$

where r is the residual vector.

Theoretically, if the matrix A was correctly calibrated, the current-carrying cables were ideal and infinite, and there was no sensor noise or external magnetic field interference, the detected magnetic field readings b would be a perfect fit using Ordinary Least Squares and the residual vector magnitude would be 0. In practice, sensor noise

and magnetic field interference create some residual, and the sign of a good fit is when the residual is very small.

We can enhance the matrix A by solving for another magnetic field source: the spatially uniform magnetic fields. In a set of 10 sensors where six sensors align with the horizontal plane of the PCB board and four sensors are perpendicular to this plane, two different components of a spatially uniform field will be detected by each set of sensors, which we can call u_x and u_y. This uniform field can be modeled as a column containing 1, -1, or 0, depending on the orientation of the sensor relative to the uniform field component.

To illustrate this point, for the configuration found in Figure 4-1 with 10 sensors, the matrix augmentation is

$$\begin{bmatrix} \alpha_{0,0} & \alpha_{0,1} & \alpha_{0,2} & 1 & 0 \\ \alpha_{1,0} & \alpha_{1,1} & \alpha_{1,2} & 0 & -1 \\ \alpha_{2,0} & \alpha_{2,1} & \alpha_{2,2} & 1 & 0 \\ \alpha_{3,0} & \alpha_{3,1} & \alpha_{3,2} & 0 & -1 \\ \alpha_{4,0} & \alpha_{4,1} & \alpha_{4,2} & 1 & 0 \\ \alpha_{5,0} & \alpha_{5,1} & \alpha_{5,2} & 1 & 0 \\ \alpha_{6,0} & \alpha_{6,1} & \alpha_{6,2} & 0 & -1 \\ \alpha_{7,0} & \alpha_{7,1} & \alpha_{7,2} & 1 & 0 \\ \alpha_{8,0} & \alpha_{8,1} & \alpha_{8,2} & 0 & -1 \\ \alpha_{9,0} & \alpha_{9,1} & \alpha_{9,2} & 1 & 0 \end{bmatrix} \begin{bmatrix} I_0 \\ I_1 \\ I_2 \\ u_x \\ u_y \end{bmatrix} = \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \\ b_8 \\ b_9 \end{bmatrix} \tag{4.14}$$

This augmentation to the matrix model is effective not only in separating the Earth's uniform field from current-created magnetic fields, but also in separating ambient 60 Hz magnetic fields, since those fields can be considered spatially uniform.

Figure 4-19 shows the Fourier transform of an OLS current estimate formed without augmenting the gain matrix with the uniform field columns. The estimated was performed on a balanced set of 1.67 A 90 Hz currents. There is a noticeable 60 Hz component. Figure 4-20 shows an OLS estimate using the same magnetic field readings but using a gain matrix augmented with the uniform field columns. In

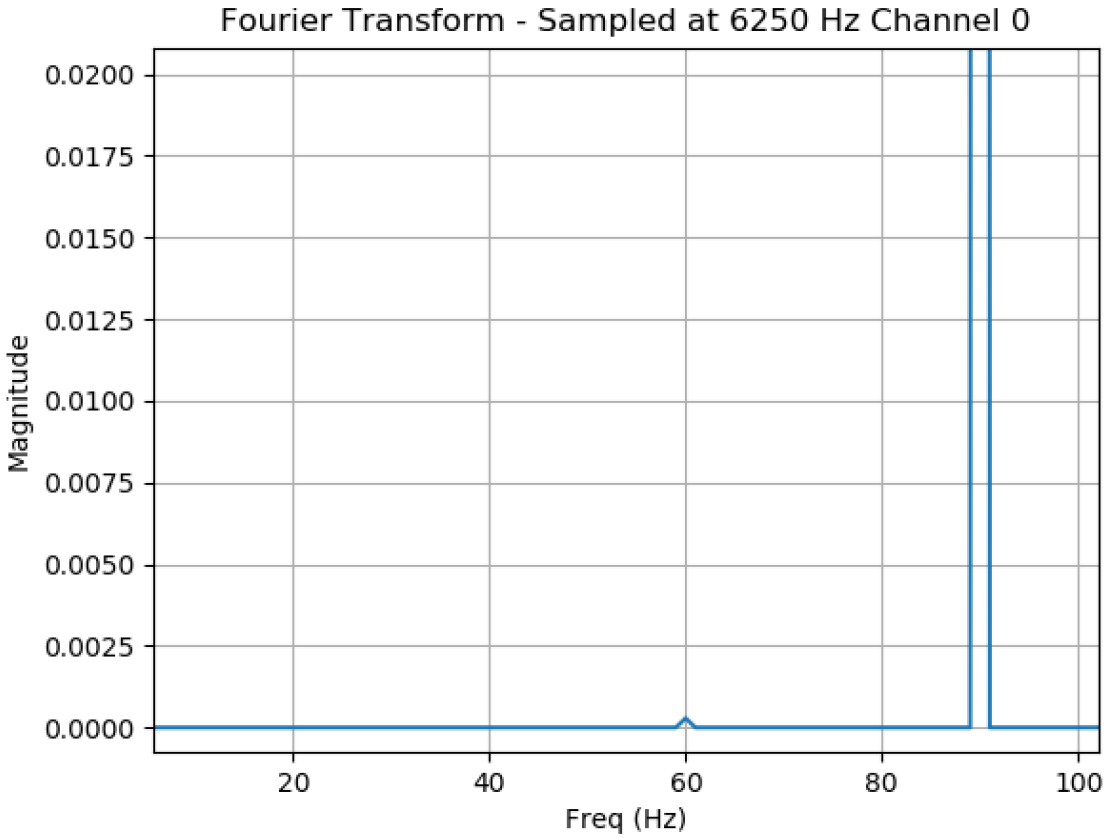

Figure 4-19: A plot showing the Fourier transform of an OLS estimate of 90 Hz current. In this estimate, the gain matrix A was not augmented with uniform field columns, and there is a 60 Hz component visible in the estimate caused by ambient 60 Hz magnetic fields. The readings shown are the voltage output of the DRV425 and thus the magnitude is in volts.

this second estimate, the 60 Hz component has been largely eliminated, making the estimate more accurate.

Hypothetical Two-Sensor Scenario

To better understand how the OLS estimator works and the error that external cables introduce, let us consider for a moment a simplified version of the problem we are solving. We will consider a situation in which we wish to estimate the current in one cable using two magnetic field sensors. Since we will have two equations to solve for one unknown, we can use the OLS estimator. In this simple experiment, we will not

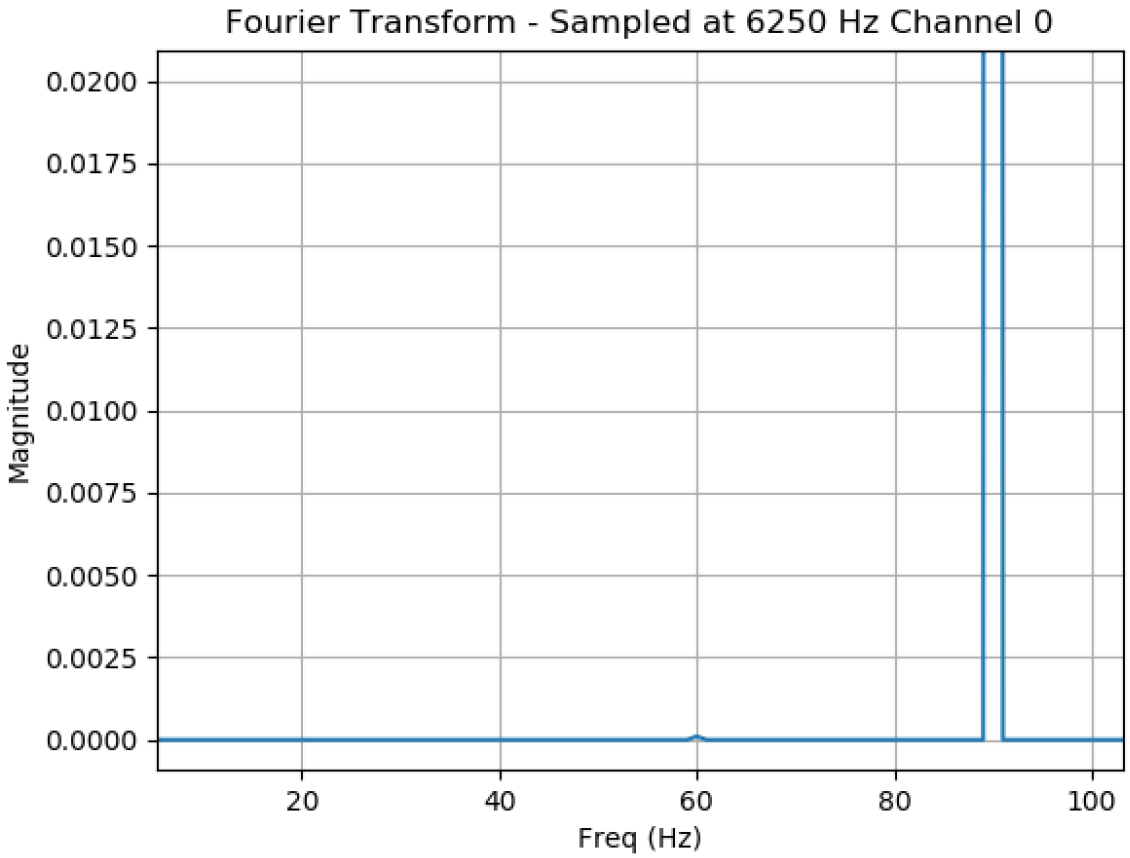

Figure 4-20: A plot showing the Fourier transform of an OLS estimate of 90 Hz current formed using a gain matrix A augmented with uniform field columns. The 60 Hz component is largely eliminated from the estimate. The readings shown are the voltage output of the DRV425 and thus the magnitude is in volts.

include the uniform field components in the OLS estimator.

Figure 4-21 illustrates this scenario. Each sensor will be located at a distance of d_1 above and below the internal cable, and will be oriented to align with the direction of the magnetic field from the internal cable. Furthermore, we will include an external cable at a distance of d_2 from one of the sensors to introduce an error into our estimate. The two detected magnetic fields will be b_1 and b_2. The matrix relationship $AI = b$ in this hypothetical scenario is given by

$$AI_1 = b \Rightarrow \begin{bmatrix} \frac{\alpha}{d_1} \\ \frac{\alpha}{d_1} \end{bmatrix} I_1 = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} \tag{4.15}$$

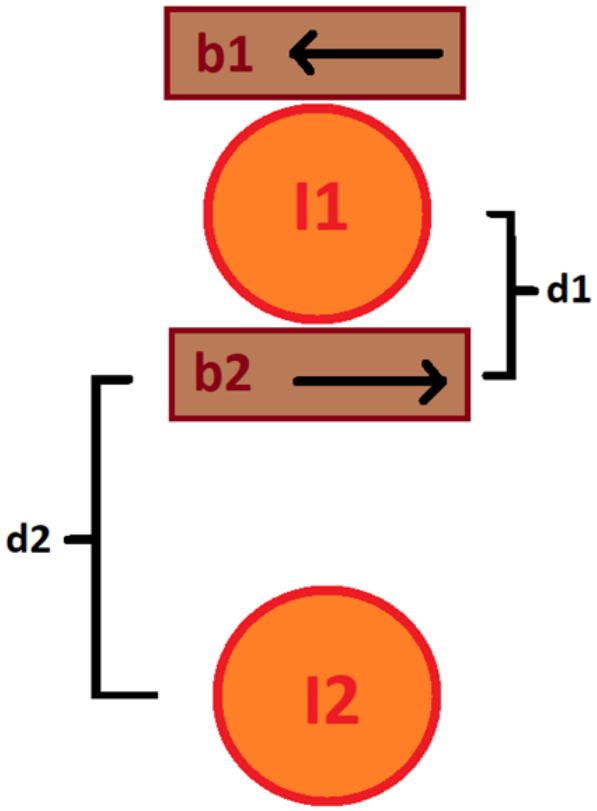

Figure 4-21: A diagram of the hypothetical scenario.

where $\alpha = \frac{\mu_0}{2\pi}$.

Note that both entries in the matrix A are the same. This illustrates the function of the matrix A: it describes the spatial distribution of the magnetic fields produced by the internal cables. In this case, the equal matrix entries indicate that both sensors are equally distant from the current and are both oriented along the direction of the current-produced magnetic fields. The OLS estimate is then given by

$$\hat{I}_1 = (A^T A)^{-1} A^T b = \frac{d_1(b_1 + b_2)}{2\alpha} \tag{4.16}$$

The fractional error of the estimate is

$$|\frac{\hat{I}_1 - I_1}{I_1}| = \frac{1}{2(2\frac{d_2}{d_1} + (\frac{d_2}{d_1})^2)} \frac{I_2}{I_1} \tag{4.17}$$

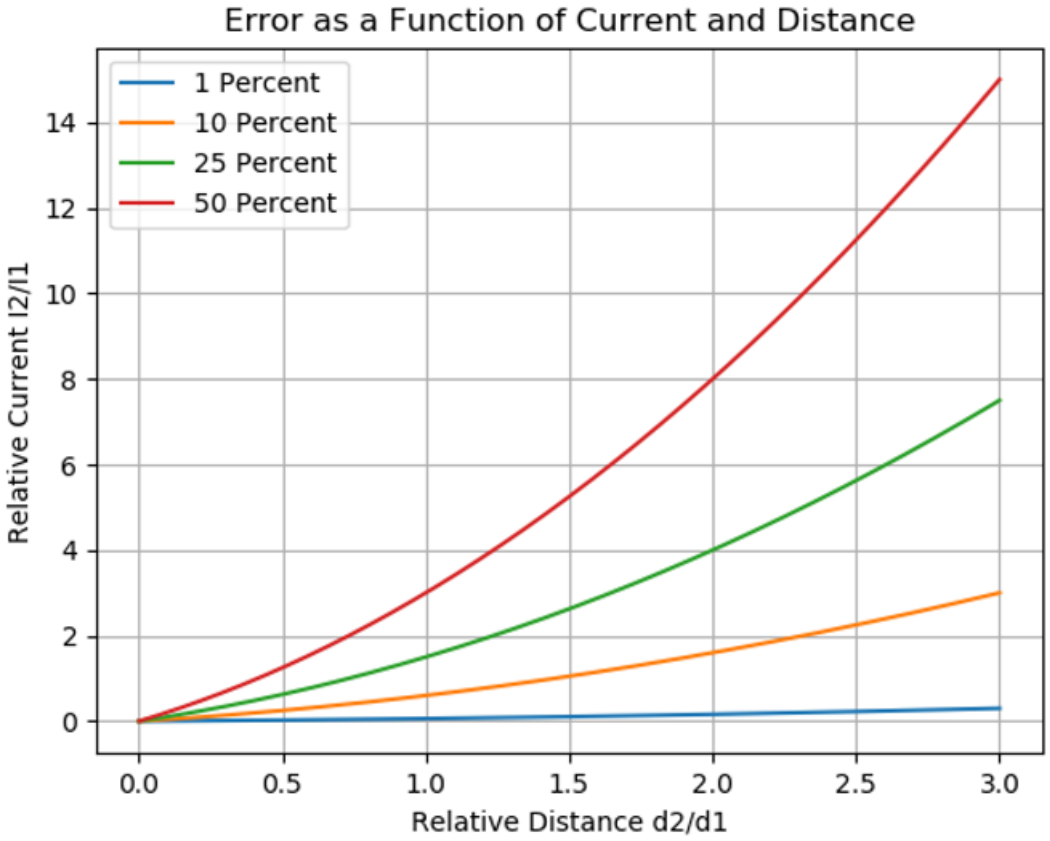

Figure 4-22: Error contours as a function of distance and current of the external cable.

Observing (4.17), we see that the fractional error will decrease as the wire causing external interference is moved further away, since the error will tend to 0 as d_2 increases to infinity. The error will also tend to 0 as the current I_2 tends to 0.

Figure 4-22 shows contours of 1%, 5%, 10%, and 25% error as a function of the ratio between internal and external current $\frac{I_2}{I_1}$ and the ratio of the distances $\frac{d_2}{d_1}$.

Figure 4-23 shows a plot of percent error when the external current has the same magnitude as the internal current and d_1 is set to 5 mm, a value similar to our setup. Under these conditions, the external cable would have to be at a distance of 30.7 mm, more than 6 times the distance between the internal cable and the sensors, for the current estimate to be at less than 1%. This suggested to us that the OLS estimator alone would not be able to achieve the thesis goals, and we decided to consider other

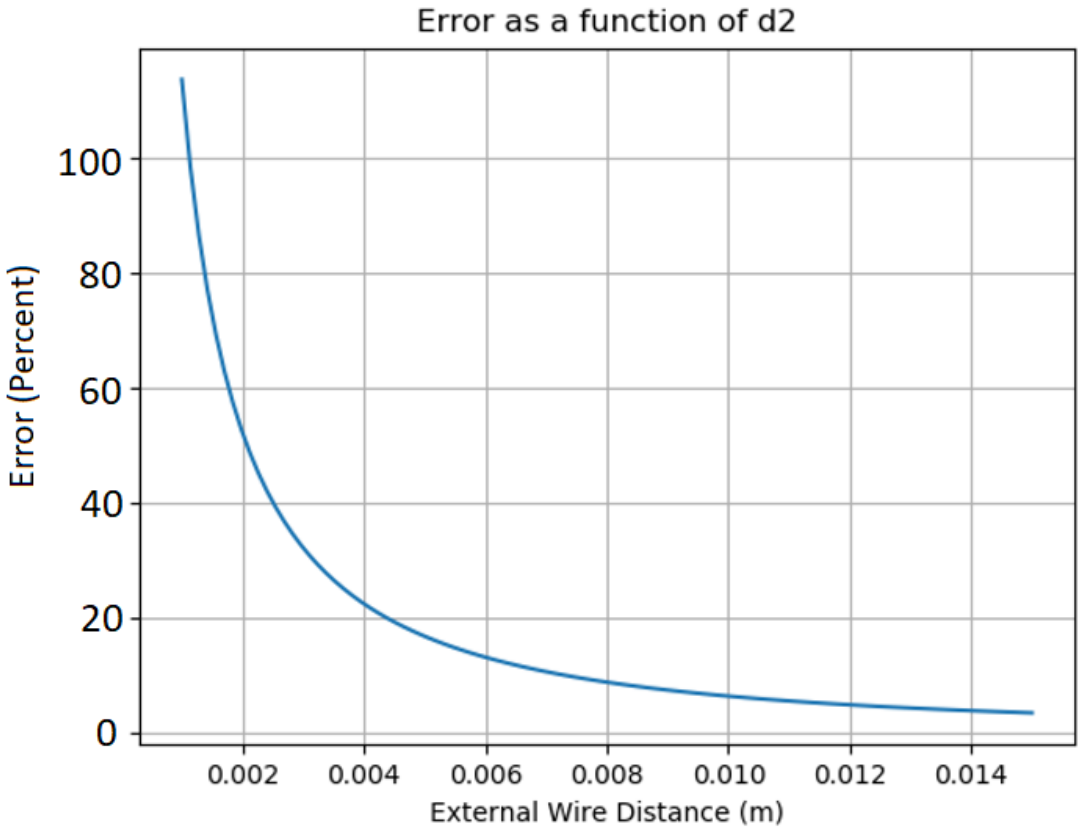

Figure 4-23: Estimation error of the simple experiment as a function of the distance of the external cable.

current estimation techniques, which will be discussed in later sections.

Although an analysis similar to the one above can be performed for the case of 3 cables and 10 sensors, it will be much more complex to analyze. Instead, we now turn to our physics simulator to gain insight into the performance of the OLS estimator when estimating three separate currents.

Multiple Sensor Performance

We now present the performance of the OLS estimator in simulated experiments using the physics simulator to estimate currents in the four test cases presented previously. Since the problems of sensor noise, time shift, and spatially uniform field interference were solved by using our pre-processing algorithms and augmenting the matrix A,

Table 4.1: OLS Estimation error in the four test cases.

Case	**With Six Sensors**	**With Ten Sensors**
No Interference	0%	0%
External Wire	6.5%	2.3%
Plate	4.73%	1.46%
Six Wires	16.8%	5.71%

the experiments do not simulate these forms of interference. Rather, the simulations only involve interference originating from external cables and plates and the OLS estimator for these simulations is used without including the uniform field columns.

Table 4.1 shows the percent error of the OLS estimator in the four test cases when the simulated system used six sensors and when it used ten sensors. In the case of six sensors, the locations correspond to the six horizontal sensors of Figure 4-1. In the case of ten sensors, the locations are the same as all sensors in Figure 4-1. Figures 4-24 and 4-25 show the percent error of the four test cases as a function of the number of sensors in the array. The sensors are placed according to the pattern described in Section 4.5.

We can make several observations about these simulations. First, the estimation error when there is no external interference is 0% regardless of the number of sensors. This is a desirable result and is a property of the OLS estimator.

Secondly, the estimation error is not 0% for all cases when there are 88 sensors, enough to form a closed loop around all three cables. The OLS model attempts to fit the data to the model of internal cables and does not completely reject external fields even when there is a complete sensor loop.

Third, the estimation error is not a monotonically decreasing function of the number of sensors. In some cases, adding sensors worsened the error. This happened because in these cases, the sensors that were added were close to the locations of the external cables in our tests. In general, we do not know the location of external cables beforehand, so we cannot know whether adding a sensor in a given location will increase or decrease estimation error.

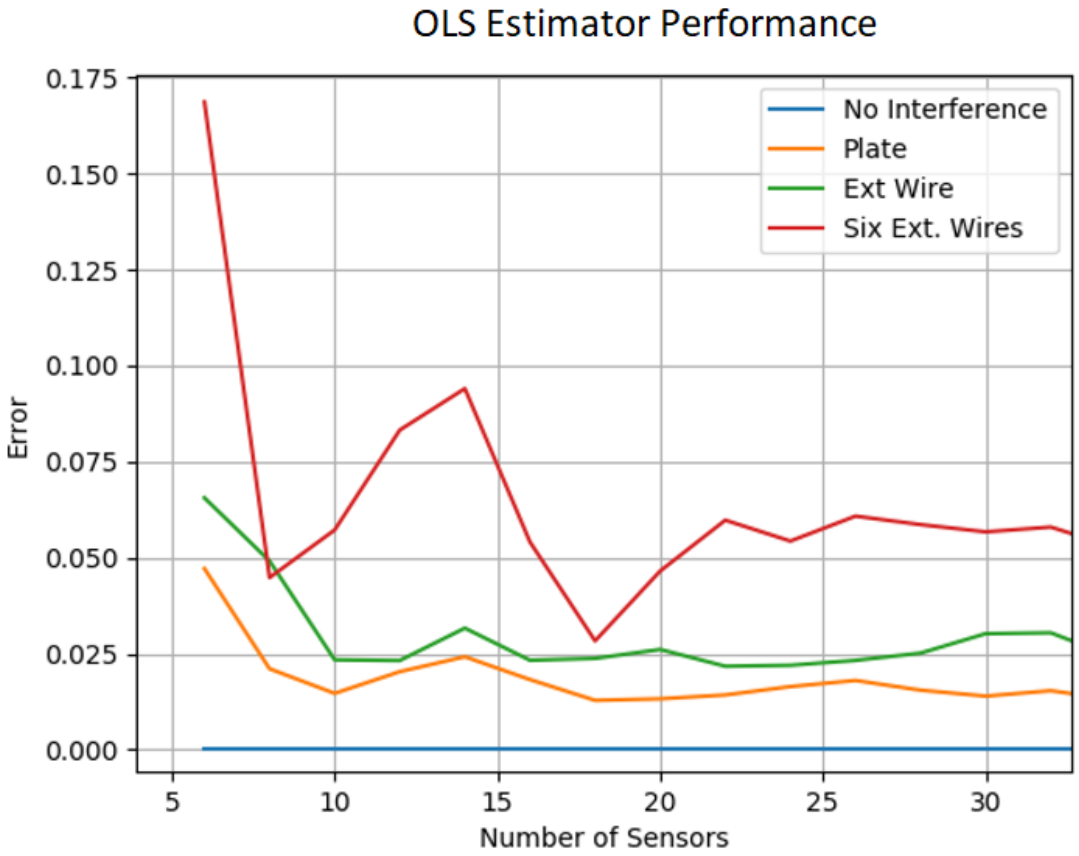

Figure 4-24: Estimation error using the OLS Estimator as a function of up to 25 sensors.

4.8.2 Ampere's Law Estimator

We also developed an estimator that approximates the measurement of closed loop integrals of magnetic fields around each cable. The motivation behind this estimator was Ampere's Law, which states

$$\mu_0 I = \oint \vec{B} \cdot dl \tag{4.18}$$

Ampere's Law implies that if we were to measure the magnetic field along a closed-loop around one cable, the magnetic fields generated by sources outside of the loop would sum to zero, and the magnetic fields generated by the cable and measured along the closed loop would be proportional to the cable current. Thus, measuring

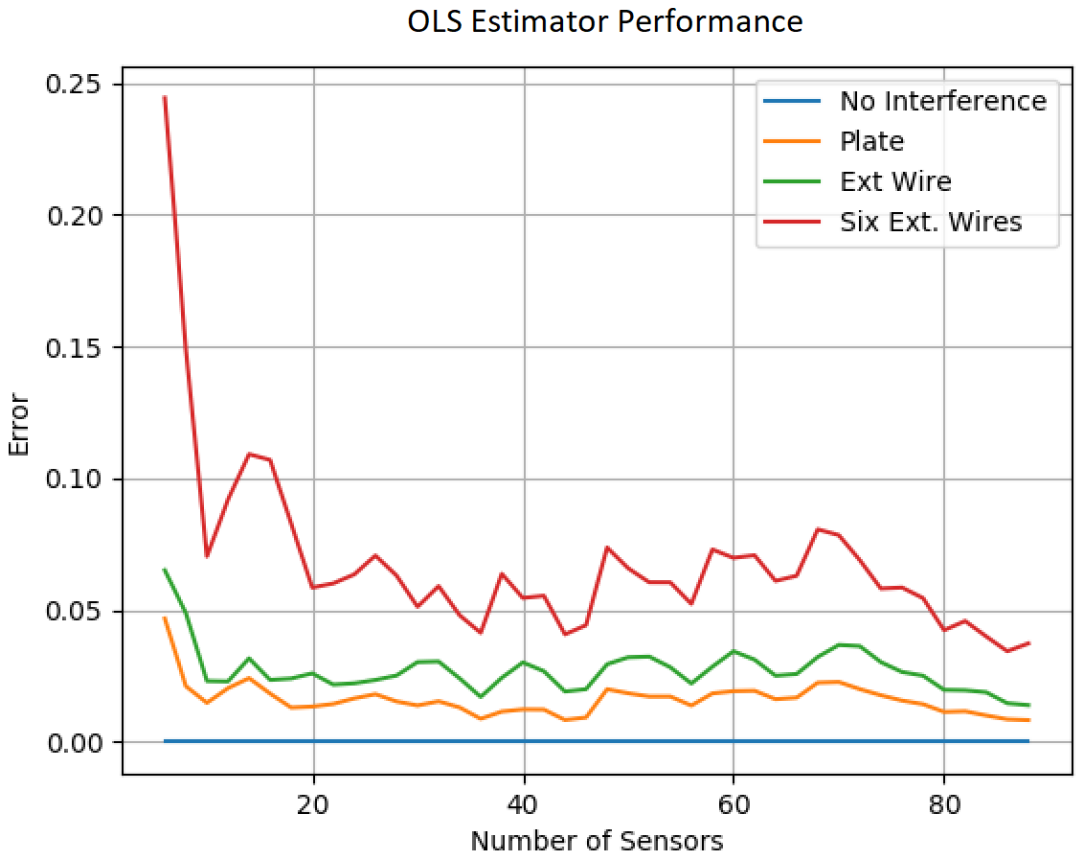

Figure 4-25: Estimation error using the OLS Estimator as a function of up to 88 sensors.

this integral would facilitate a current estimate free from external magnetic field interference. However, as previously mentioned, 88 sensors are required to measure a closed loop path around the three cables in our setup, which are too many to realistically include in a viable detector. Thus, we decided to explore the performance of approximating Ampere's Law with a limited number of sensors.

The estimator we tested was based on the idea that the sum of the external magnetic fields detected by an array of sensors around a single cable should be zero. Say there are M sensors around a cable being used to form the Ampere's Law estimate. Each sensor m will detect a total magnetic field $b_{total,m}$ that is the sum of two magnetic fields: the field produced by the internal cable $b_{internal,m}$ and the field produced by external sources $b_{external,m}$. In a set of sensors that form a closed loop around a cable, $\mu_0 I = \sum_{m=1}^{M} b_{internal,m}$ and $0 = \sum_{m=1}^{M} b_{external,m}$. Furthermore, we consider the

magnetic field created by the current estimate $\hat{I}$ to be $a_{i,j}\hat{I}$, where $a_{i,j}$ is the gain between cable i and sensor j from the gain matrix discussed in the previous section.

We can express the magnetic fields coming from external sources as

$$b_{total,m} = b_{internal,m} + b_{external,m} \tag{4.19}$$

$$b_{external,m} = b_{total,m} - b_{internal,m} \tag{4.20}$$

We can then impose the condition that the external fields detected by all sensors sum to zero,

$$\sum_{m=1}^{M}(b_{external,m}) = 0 \tag{4.21}$$

$$\sum_{m=1}^{M}(b_{total,m} - b_{internal,m}) = 0 \tag{4.22}$$

$$\sum_{m=1}^{M}(b_m - a_j\hat{I}_0) = 0 \tag{4.23}$$

$$\sum_{m=1}^{M} b_m - \sum_{m=1}^{M} a_m\hat{I}_0 = 0 \tag{4.24}$$

where b_m is a more concise way of representing the total magnetic field detected by each sensor. We can rearrange the last expression to produce an estimate for the current $\hat{I}_0$,

$$\hat{I}_0 = \frac{\sum_{m=1}^{M} b_j}{\sum_{m=1}^{M} a_j} \tag{4.25}$$

Note that this estimator does not use all magnetic field sensors in the array to estimate each current. Rather, it uses a subset of the sensors that form a closed loop around each current. Thus, each current estimate I_0, I_1, and I_2 will be formed using three different subsets of sensors, although some sensors will appear in more than one subset. The estimation results for the four standard test cases when using six and ten sensors are found in Table 4.2. Figure 4-26 shows the estimation error as a function of the number of sensors.

A major flaw of this estimator is the non zero estimate error when there is no

Table 4.2: Ampere's Law Estimation error in the four special cases.

Case	With Six Sensors	With Ten Sensors
No Interference	19.3%	12.8%
External Wire	23.6%	11.8%
Plate	15.8%	13.3%
Six Wires	38.5%	10.5%

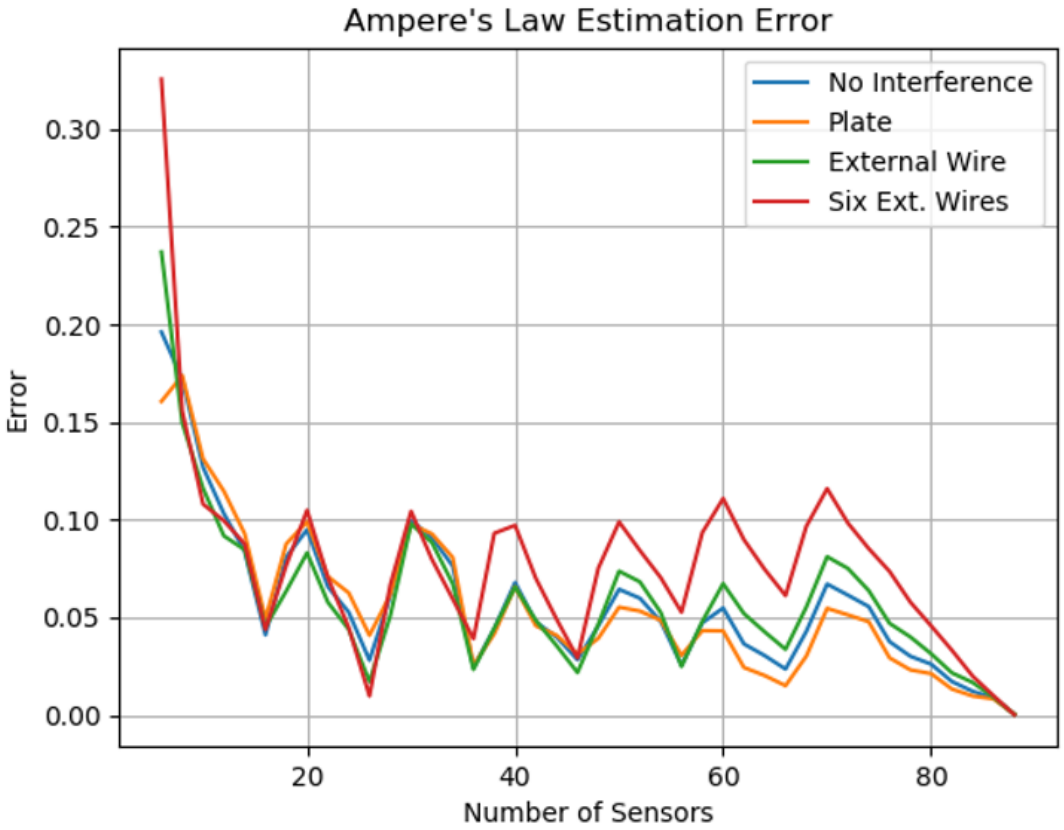

Figure 4-26: Estimation error using the Ampere's Law Estimator.

external interference. This error occurs because the estimator attempts to estimate each current individually, without considering the other two cable currents within the yoke, which are thus treated as external currents.

As the graph shows, the error estimate tends to 0% for all cases as the number of sensors forms a complete loop around all three cables. However, the error does not begin to monotonically decrease below 4% error until there are more than 76

sensors. A system with this many sensors would be too expensive and would not be preferable to other existing sensors such as Hall Effect sensors. Thus, the Ampere's Law Estimator did not prove satisfactory to achieve thesis goals.

4.8.3 Non-linear Model Estimator

Another estimator we developed involved modeling external sources of magnetic fields, and using these models to determine what components of the detected magnetic fields originated from external sources and what components came from the internal cables. As previously stated, the external sources of interference we are most concerned with are parallel external cables and parallel external plates, whose effects can also be modeled as external cables. The magnetic field detected by a sensor i as a result of the internal cables, an external uniform field, and P external cables is given by

$$b_i(t) = \sum_{j=1}^{3} \alpha_{i,j} I_j(t) + u(t) + \sum_{k=1}^{P} \frac{u_0}{2\pi} \frac{(x_i - x_k) I_k(t)}{(x_i - x_j)^2 + (y_i - y_k)^2} \tag{4.26}$$

where x_k and y_k are the coordinates of each external cable, and $I_k(t)$ is the current of each external cable.

In a system using N sensors, we will have a system of N equations of the above form. The parameters $\alpha_{i,j}$,x_i, and y_i are known. The unknown variables are the three $I_j(t)$ terms, the uniform field $u(t)$, the locations of each external cable x_k and y_k, and each external current $I_k(t)$. Thus, in a setup with P cables, there will be 4+3P unknown variables. This suggests a disadvantage of this estimator: we are limited in how many external cables we can model by the number of sensors in our system. A system with 10 sensors can model no more than 2 external cables, and a system with 25 sensors can model no more than 7 external cables. In our visits to industrial areas we found that an electrical closet can easily contain bundles of 6 or more cables running together, and therefore this estimator can easily encounter a situation in which there are more external cables than it can model.

Another problem with the Non-Linear Model Estimator is that it is non-linear, as (4.26) shows, and we cannot solve it using linear least squares estimator. Despite

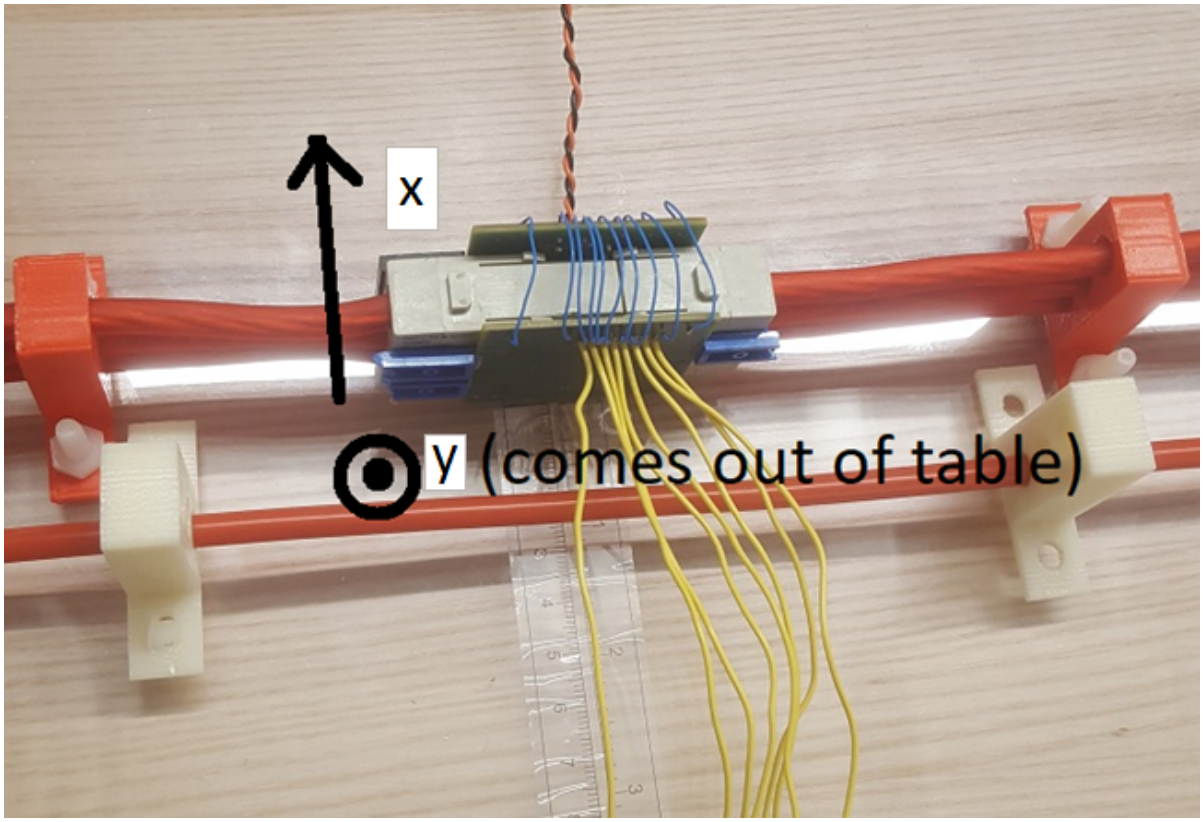

Figure 4-27: An external cable experiment using 3D printed supports.

these challenges, we decided to explore the effectiveness of using this model to estimate currents.

We began by considering the case in which there was a single external parallel cable. As shown in Figure 4-27, we used a 3D printed external support to hold the external cable while knowing its x and y location. Since there were more sensors than parameters, we solved the system of equations by minimizing the squared error between the readings and estimated readings. That is to say, we were finding parameters that minimized the expression

$$min_{(I_0,...,I_N,x_0,...,I_P)} \sum_{i=0}^{N} (b(t) - \hat{b_i}(t))^2 \tag{4.27}$$

where $\hat{b_i}(t)$ is the magnetic field vector implied by a set of variable estimates $(\hat{I_0}, ..., \hat{I_N}, \hat{x_0}, ..., \hat{I_P})$. To perform this optimization we used a minimization function in the SciPy library, scipy.optimize.minimize. The code we used, including the function

fun() where we defined the squared error, is found in the file minimize_fun.py in Appendix A.

After experimenting with non-linear optimization, we found that although it was possible to generate accurate estimates, the results were very sensitive to the initial values we entered into the minimize() function. This was especially a problem when choosing what initial values to select for x_k and y_k, since there were several regions around the yoke where an external cable could be located, and initializing these values to 0 returned poor results.

For example, the Table 4.3 shows the true values and estimated values of one particular experiment in which we ran 0.735 A of current through one cable, no current through the other two cables, and -0.735 A through an external cable. The estimate error is calculated using only the estimates of the three internal cables. Our first estimate had an 8.8% error. However, after adjusting the initial values of the external cable to be closer to the true values, we produced a second, more accurate estimate.

Table 4.4 shows the results of a similar experiment in which the external cable was placed in a different location. As in the previous case, the first experiment return a large 14.4% error, but after adjusting the initial location parameters to values we knew were closer to the true values, the estimate greatly improved.

Unfortunately, we cannot know beforehand where the external cable will be located, and cannot know at runtime how to set the initial values for the location of the external cable. Experiments in which we tried running the optimizer with different initial values were too computationally expensive.

Because of the dependence of the estimator on initial values and the computa-

Table 4.3: Estimation results for an experiment in which two different sets of initial values were used with the Non-Linear Model Estimator.

Type	I_0 (A)	I_1 (A)	I_2 (A)	x (m)	y (m)	$I_{external}$ (A)	Percent Error
True Value	0.735	0	0	0.04	0.03	-0.735	N/A
First Estimate	0.748	-0.045	-0.139	0.006	0.02	0.919	8.8%
Second Estimate	0.725	-.004	0.001	3.0	4.0	-.736	0.6%

Table 4.4: Estimation results for a second experiment in which two different sets of initial values were used with the Non-Linear Model Estimator.

Type	I_0 (A)	I_1 (A)	I_2 (A)	x (m)	y (m)	$I_{external}$ (A)	Percent Error
True Value	0.926	0	0	0.02	-.0025	-0.926	N/A
First Estimate	0.912	0.196	0.182	0.9	0.0	0.029	14.1%
Second Estimate	0.936	-0.023	0.007	0.005	-.022	-.752	1.4%

tional expense of running non-linear optimization, we decided to try other estimation methods. Although it would be possible to develop a closed-form solution to the least squares minimization of a system of equations of the form of (4.26), we decided to explore linear estimation methods instead due to their faster computation speed and the previously mentioned issues with this estimator.

4.8.4 Linear Model Estimator

Because of the problems we encountered when trying to minimize a set of non-linear equations, we decided instead to create linear models of external interference to use for current estimation. For example, the magnetic field created by one external cable would be approximated as

$$b = a_1 x_e + a_2 y_e + a_3 I_e \tag{4.28}$$

where x_e and y_e are the coordinates of the external cable and I_e is the current. We also explored creating polynomial models of external interference. In a second order polynomial model of a single external cable, the magnetic field would not only be modeled as a linear function of the location and current variables, but also the pairwise products of these variables. This model would take the form

$$b = a_1 x_e + a_2 y_e + a_3 I_e + a_4 x_e^2 + a_5 y_e^2 + a_6 I_e^2 + a_7 x_e y_e + a_8 x_e I_e + a_9 y_e I_e \tag{4.29}$$

In a third order polynomial model, the three-way products would be used, and so forth.

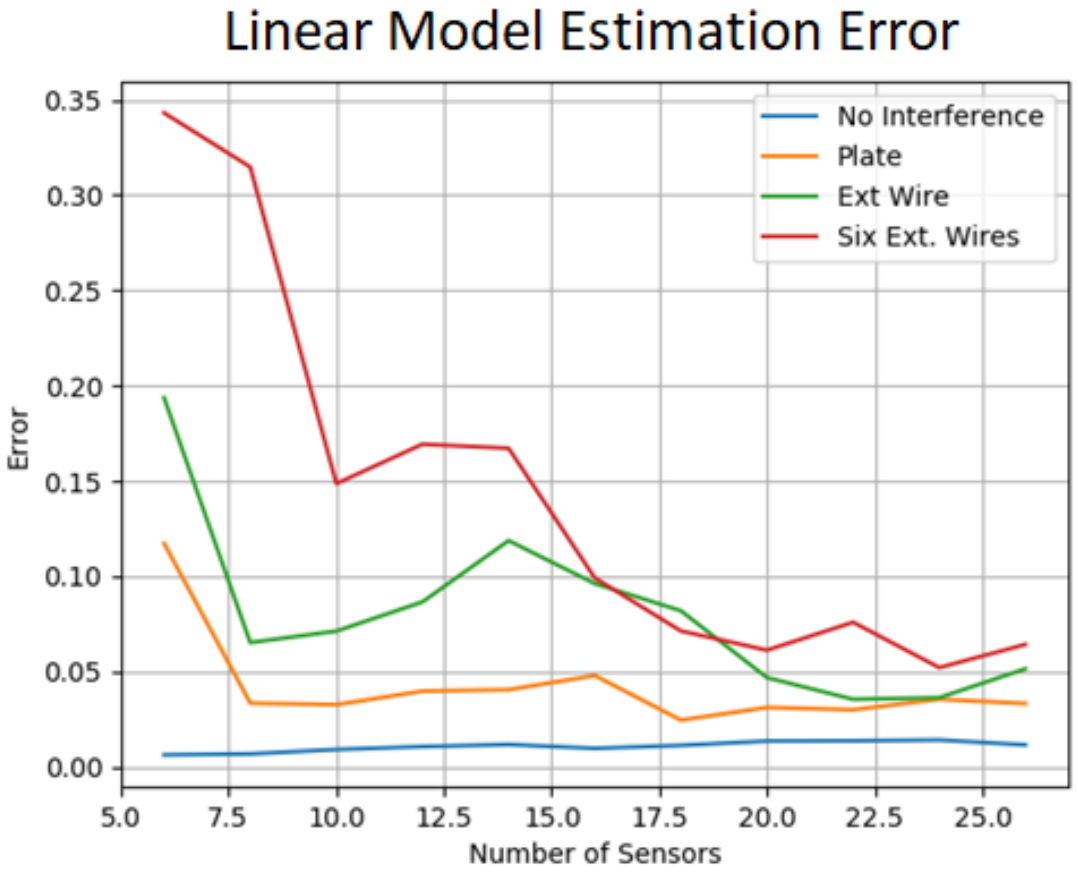

Figure 4-28: Estimation error of the Linear Model Estimator as a function of the number of sensors used.

To generate a linear model, our goal was to calculate coefficients such as a_0,a_1, and a_2 in (4.31), so that the resulting sum b is as close to the magnetic field predicted by the non-linear model as possible. To calculate these coefficients, we generated a training set of 100,000 samples. Each sample contained randomly chosen values for x_e, y_e and I_e, with one of the constraints being that the location of the cable had to be outside the yoke. Each sample also contained the magnetic field b predicted by the non-linear model of an external cable, produced by the physics simulator. We then used the LinearRegression.fit() function of scikit-learn to create a linear fit. Effectively, we were generating a matrix D to fit $Dx = b$ for the training samples, where x is a vector of cable parameters and b is a vector of magnetic fields. The value being minimized when using the LinearRegression function is the RMSE, so

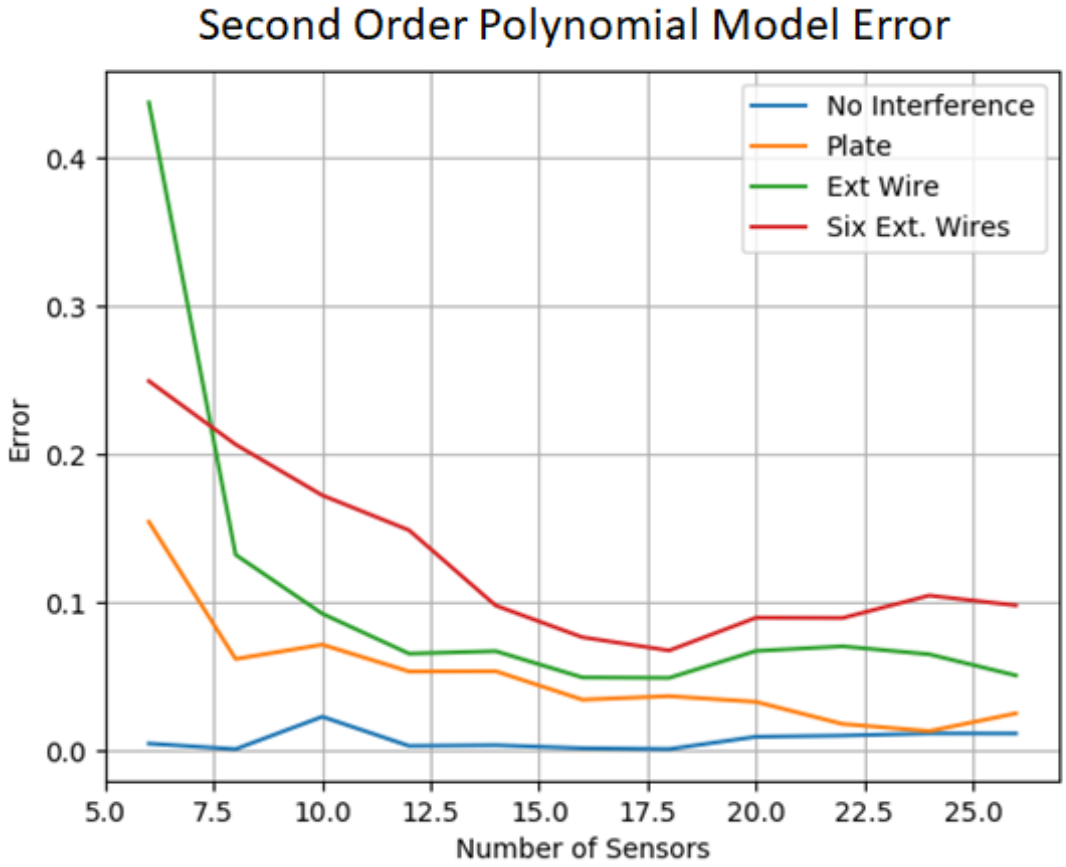

Figure 4-29: Estimation errors as a function of the number of sensors used for the Second Order Polynomial Estimator.

essentially we were minimizing the function

$$(a_1 x_e + a_2 y_e + a_3 I_e - \frac{\mu_0 (y_e - y_s) I_e}{(x_e - x_s)^2 + (y_e - y_s)^2})^2 \tag{4.30}$$

Once the linear model consisting of the matrix D was found, the full model for magnetic field readings could be formed by concatenating the gain matrix A that characterized the internal currents I with the matrix D, in the form

$$\begin{bmatrix} A & D \end{bmatrix} \begin{bmatrix} I \\ x \end{bmatrix} = b \tag{4.31}$$

Although this matrix formulation can be augmented with columns to model external

Table 4.5: Linear Model Estimator error in the four special cases.

Case	**With Six Sensors**	**With Ten Sensors**
No Interference	0.6%	0.9%
External Wire	19.3%	7.1%
Plate	11.7%	3.2%
Six Wires	34.4%	14.9%

uniform fields, we did not include those columns since the simulations we ran did not include uniform fields. To form the current estimate, we used the magnetic field vector b to solve for the parameters in I and x using the Ordinary Least Squares fit corresponding to the matrix relationship in (4.31).

However, the model of a single external cable did not yield good results. A graph of the estimate error for the four test cases as a function of the number of sensors used in shown in Figure 4-28 and the estimation errors for 6 and 10 sensors is shown in Table 4.5. We then analyzed the estimation error for a second order polynomial model of an external cable. This model also performed poorly. The graph of the estimate error is shown in Figure 4-29 and the Table 4.6 shows estimation errors for the four test cases.

We believe the reason these models performed so poorly is because it was linearized around the values $x_e = 3.0$ cm, $y_e = 0.0$ cm, and $I_e = 0.0$ A. However, since the cables are location outside the yoke, they will often be located far away from this point in space, which increases the linearization error. To overcome this, we could include linearization of cables in different regions around the yoke. However, each additional cable introduced into the linear model adds three new variables, and even more in a polynomial model. Due to the limited number of variables we could solve for using

Table 4.6: Second Order Polynomial Estimator error in the four special cases.

Case	**With Six Sensors**	**With Ten Sensors**
No Interference	0.4%	2.2%
External Wire	43.7%	9.2%
Plate	15.4%	7.1%
Six Wires	24.9%	17.2%

a sensor array with 10 sensors, we concluded the Linear Model Estimator would not achieve our thesis goals, and instead focused our attention on other current estimators.

4.8.5 Spatial Harmonics Estimator

We implemented the estimator derived by the research team at Politecnico di Milano described in Chapter 2. [24] This estimator uses the harmonic expansion of the solution to Laplace's Equation to create a general linear representation of external sources of magnetic fields. These linear components appear in pairs and will be referred to as $a_m(t)$ and $b_m(t)$. The linear representation of the magnetic field vector using the first M harmonics measured by a particular sensor i is repeated for convenience,

$$\begin{aligned} H_i(r,\phi,t) = &-\sum_{m=1}^{M} m r_i^{m-1}(a_m(t)cos(m\phi_i) + b_m(t)sin(m\phi_i))\hat{r} \\ &+\sum_{m=1}^{M} m r_i^{m-1}(b_m(t)cos(m\phi_i) - a_m(t)sin(m\phi_i))\hat{\theta} \end{aligned} \tag{4.32}$$

where r_i is the distance of the sensor from the origin of the coordinate system, θ_i is the angle of the sensor location with respect to the coordinate system, and ϕ_i is the orientation of the axis of sensitivity of the sensor.

In a system with N sensors, there will be N equations of the form given in (4.32). The number of unknowns will depend on the number of components of the linear representation that we include in the estimator. If we refer to the vector of $a_m(t)$ and $b_m(t)$ components as C, and the coefficients of these components as detected by the sensors as the matrix Q, then the full representation of the problem we are solving, including both internal currents and external components, is given by

$$\begin{bmatrix} A & Q \end{bmatrix} \begin{bmatrix} I \\ C \end{bmatrix} = b \tag{4.33}$$

In the case of this estimator, augmenting the gain matrix A with columns to represent uniform magnetic fields is unnecessary, because the first harmonic of the linear representation of external magnetic fields corresponds to the uniform field columns.

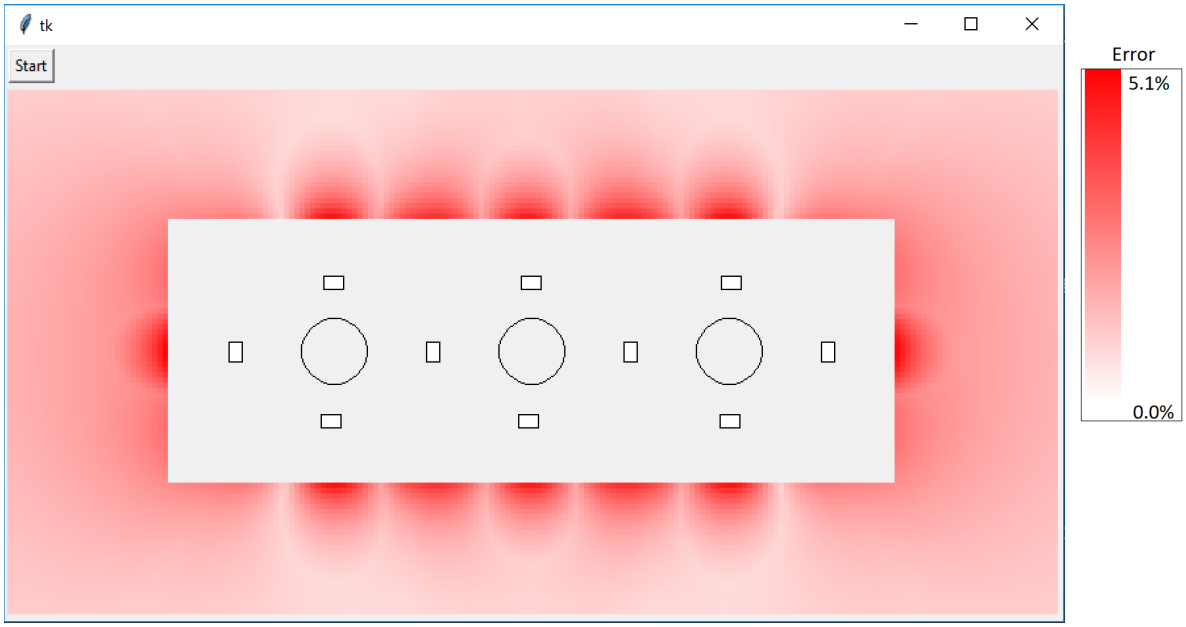

Figure 4-30: A heat map showing the percent error introduced by an external cable when the Spatial Harmonics Estimator is used with M=0.

The code we used to implement this estimator can be found in the code file current_estimator.py in Appendix A. To validate that our implementation of the estimator was correct, we conducted an experiment similar to the one described in the paper that introduced the estimator. In that paper, the research team used a simulator to test the percent estimation error that an external cable running 1 A created in a system that was estimating currents in three bus bar current conductors running 1 A. What they found was that the percent error decreased as the number of harmonic components that the estimator included increased.

In our experiment, we simulated three circular conductors running 1 A and generated heat maps in which darker red corresponded to a greater amount of estimation error induced by a single external cable running 1 A when placed at the corresponding location in the heat map. We show the results when M=0 in Figure 4-30, when M=1 in Figure 4-31, when M=2 in Figure 4-32, and when M=3 in Figure 4-33. As the heat maps show, as the number of harmonic components included in the estimator are increased, the percent error of the estimator decreases. With 10 sensors, at most 3 harmonic components can be included, since this introduces 6 unknowns that must be

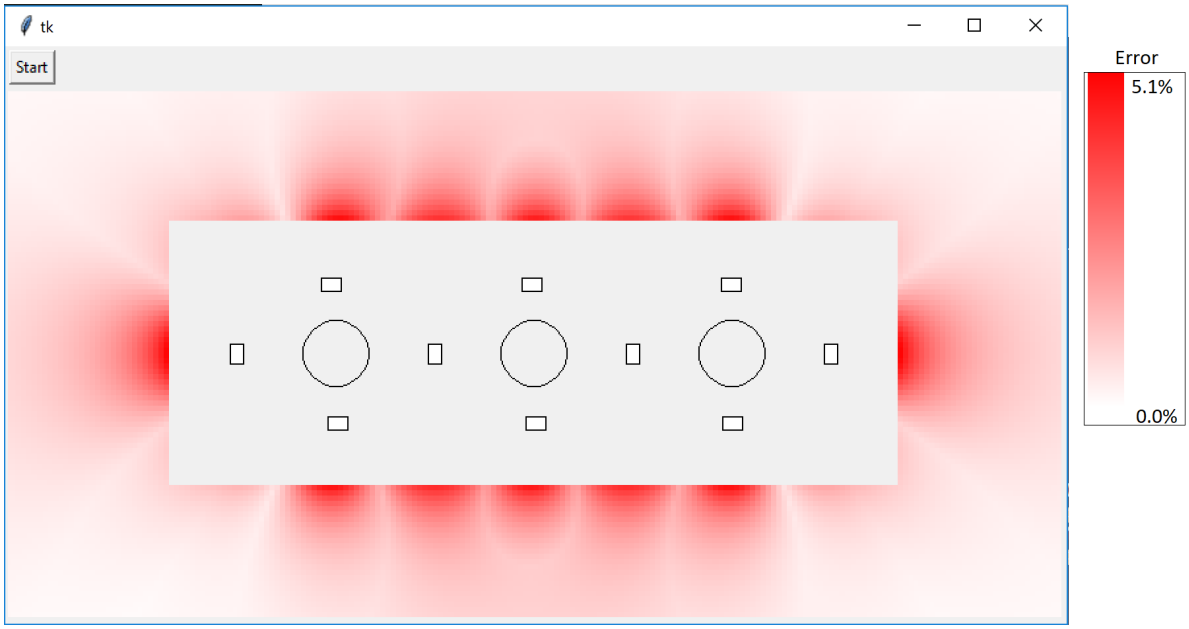

Figure 4-31: A heat map showing the percent error introduced by an external cable when the Spatial Harmonics Estimator is used with M=1.

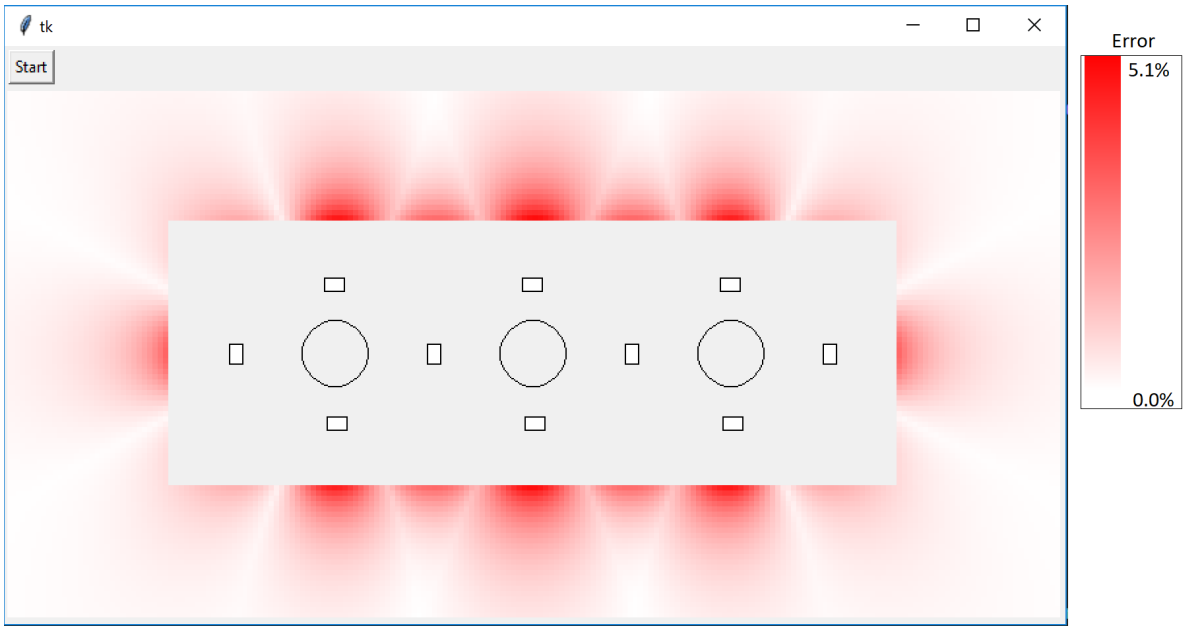

Figure 4-32: A heat map showing the percent error introduced by an external cable when the Spatial Harmonics Estimator is used with M=2.

solved in addition to the 3 unknown internal currents. The results of this experiment gave us confidence that our code correctly implements the estimator described in the paper.

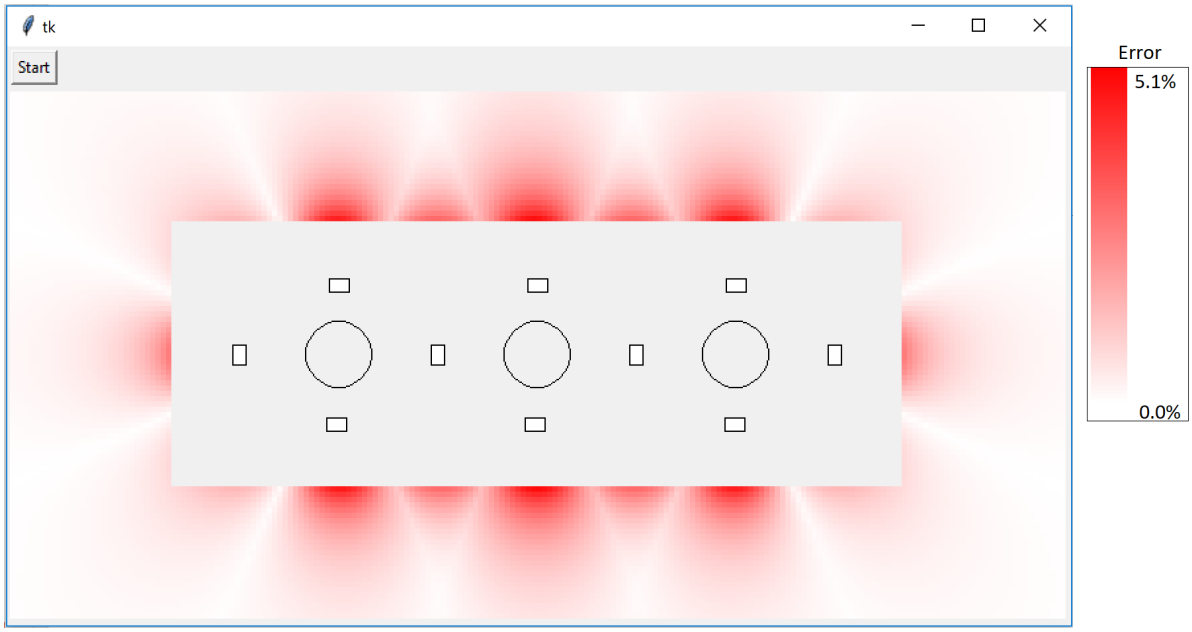

Figure 4-33: A heat map showing the percent error introduced by an external cable when the Spatial Harmonics Estimator is used with M=3.

Table 4.7: Spatial Harmonics Estimator error in the four special cases.

Case	**With Six Sensors**	**With Ten Sensors**
No Interference	0%	0%
External Wire	6.5%	1.7%
Plate	4.7%	1.0%
Six Wires	16.8%	5.4%

Note that the case M=0 is equivalent to the OLS estimator without augmenting the matrix A with uniform field columns. In fact, the first harmonic component of the Spatial Harmonics Estimator corresponds to the columns representing uniform fields, so the case in which M=1 is equivalent to the OLS estimator in which the matrix A has been augmented with uniform field columns.

We tested the estimator on the four standard test cases. The estimation error when using 6 sensors and when using 10 sensors is shown in Table 4.7. The performance of the Harmonics Estimator does offer an improvement over the OLS estimator when using 10 sensors, although it struggles in the test case of six cables. Figure 4-34 shows the estimation error for the four test cases as a function of the number of sensors used.

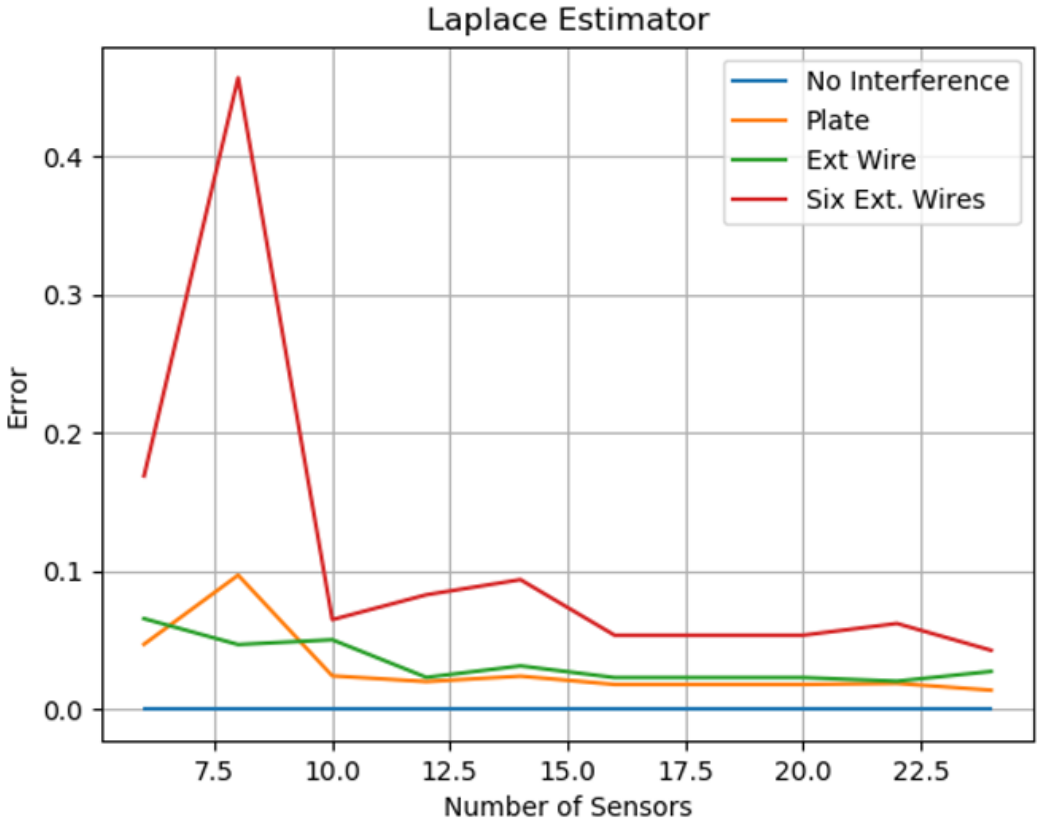

Figure 4-34: Estimation error using the Spatial Harmonics Estimator as a function of the number of sensors.

4.8.6 BLU Estimator

A shortcoming of the OLS estimator is that it is optimized for the case in which external interference is uncorrelated across all sensors, which is not the case for external sources of magnetic fields. As Figure 4-35 suggests, magnetic fields from an external cable will be detected by each sensor according to a particular pattern. Thus, it is reasonable to believe that the external magnetic fields detected by the sensors will exhibit some correlation.

More precisely, if we define a probabilistic distribution representing the possible locations and current values of external cables, then the probabilistic magnetic fields detected by sensor i can be represented as a random variable B_i. This random variable is a scalar, the flux density that will be measured by each sensor. In the

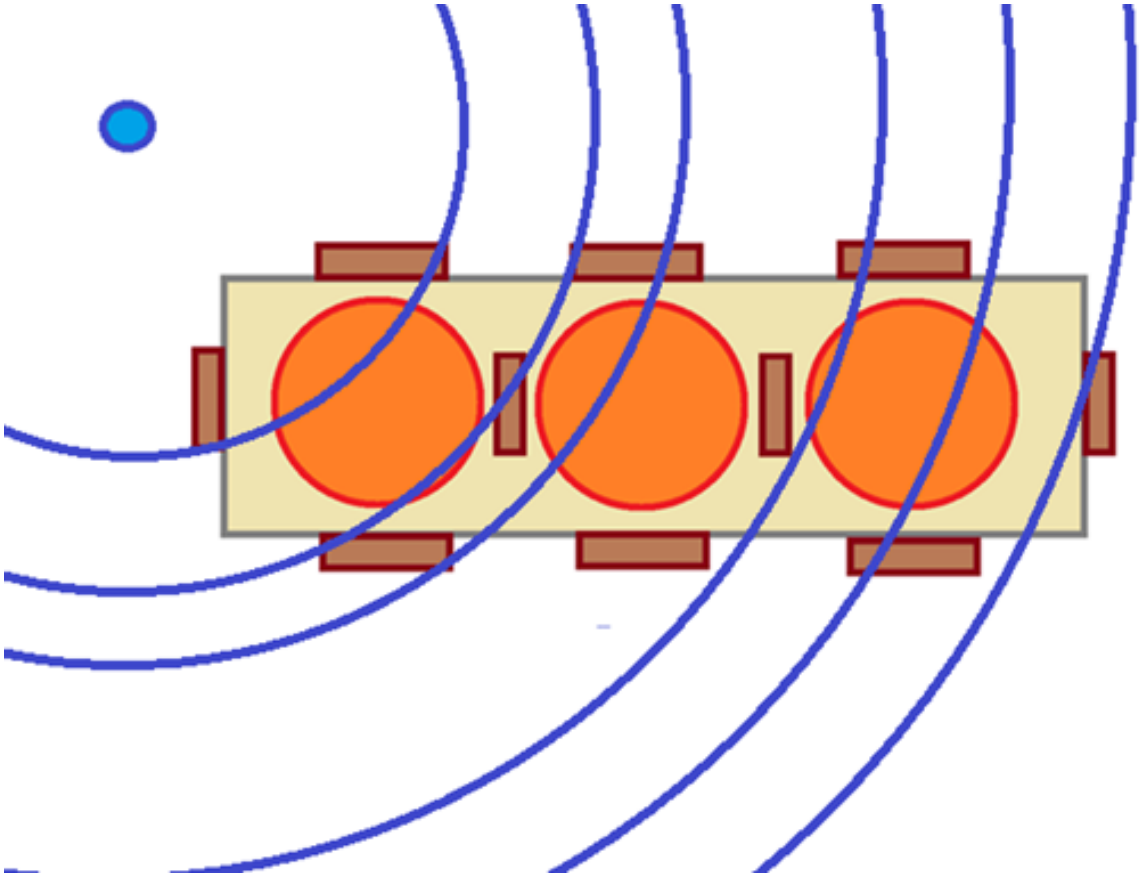

Figure 4-35: Magnetic fields from an external cable will be detected by the sensor array in a particular pattern described by the laws of electromagnetism.

probabilistic model we will describe, the current value of an external cable will be characterized by a probabilistic distribution whose mean is 0, which means each random variable will have a mean of zero, or $E[B_i] = 0$. This means that each pair B_i and B_j will have a correlation $E[B_i B_j]$ and covariance $cov(B_i, B_j)$ that are equal. Furthermore, the correlations between every pair of sensors can be concisely represented by a covariance matrix, a matrix where the i, j-th term corresponds to the correlation $E[B_i B_j]$. The value of the correlations will depend on the probabilistic model of external disturbances we are using.

This covariance matrix can be used to form a current estimate using the Best Linear Unbiased (BLU) Estimator. Let us assume our system contains N sensors. In our problem for which we have characterized the relationship between current and

detected magnetic fields as $AI = b$, the BLU Estimate is given by

$$\hat{I} = (A^T \Sigma^{-1} A)^{-1} A^T \Sigma^{-1} b \tag{4.34}$$

where A is the gain matrix and Σ is an N by N matrix where the i, j term is the correlation $E[B_i B_j]$. If the sensor readings of the external fields were uncorrelated, the matrix Σ would be a diagonal matrix and the above expression would simplify to the form of the OLS estimator.

The performance of our estimator depends on the covariance matrix and thus the probabilistic model we use to derive the matrix. The first probabilistic model we created involved three random variables: X_e, Y_e, the locations of an external cable, and I_e, the external current. X_e and Y_e were taken from a uniform distribution of locations around the yoke, excluding the yoke area. In other words they were taken from the interval $\{-.01\ m < X_e < 0.055\ m\}, \{-0.03\ m < Y_e < 0.03\ m\}$ while excluding the center yoke area $\{0.0\ m < X_e < 0.045\ m\}, \{-0.012\ m < Y_e < 0.012\ m\}$. The current I_e was taken from a uniform distribution $\{-10.0\ A < I_e < 10\ A\}$.

Each random variable B_i is a function of the location and current random variables according to the same physical law that governs the detected magnetic field. The random variable B_i for a sensor whose axis of sensitivity lies in the direction of the x axis can be expressed as

$$B_i = \frac{u_0}{2\pi} \frac{(X_e - x_s) I_e}{(X_e - x_s)^2 + (Y_e - y_s)^2} \tag{4.35}$$

where x_s and y_s are the coordinates of the sensor. The random variable B_i for a sensor whose axis of sensitivity lies in the direction of the y axis can be expressed as

$$B_i = \frac{u_0}{2\pi} \frac{(Y_e - y_s) I_e}{(X_e - x_s)^2 + (Y_e - y_s)^2} \tag{4.36}$$

The random variable I_e is independent of the random variables X_e and Y_e. Furthermore, the expectation $E[I_e]$ is 0, since it is a distribution symmetrically centered

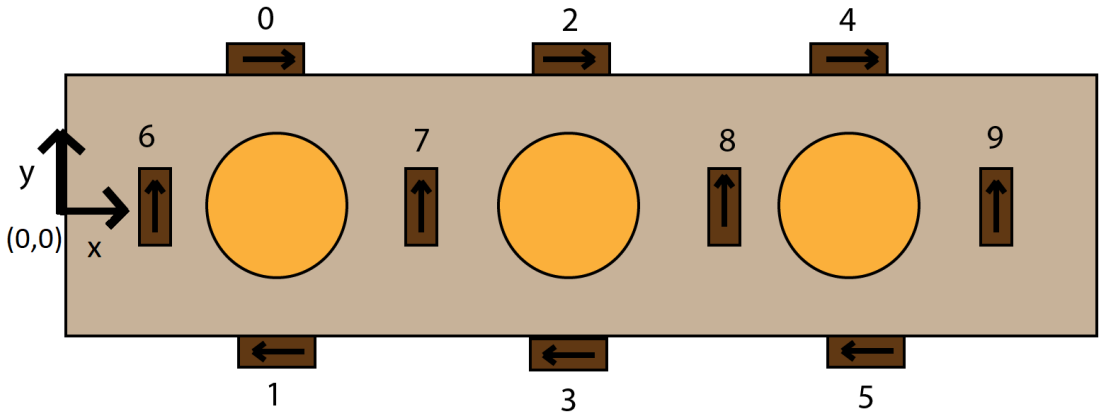

Figure 4-36: The numbering of the first 10 sensors used in the physics simulations involving the BLU estimator. Note the numbering of these sensors is not the same as the numbering of the sensors of the hardware prototype given in Figure 4-1.

around 0 A. Therefore, the expectation of B_i is

$$\begin{aligned} E[B_i] &= \int_{x_0}^{x_f} \int_{y_0}^{y_f} \int_{-10}^{10} \frac{u_0}{2\pi} \frac{(X_e - x_s)I_e}{(X_e - x_s)^2 + (Y_e - y_s)^2} p(X_e, Y_e) p(I_E) dI_e dX_e dY_e \\ &= \int_{x_0}^{x_f} \int_{y_0}^{y_f} 0 \cdot p(X_e, Y_e) dX_e dY_e \\ &= 0 \end{aligned} \tag{4.37}$$

We approximated the covariance matrix using the physics simulator. We ran the simulator over a mesh of the previously stated ranges for the three random variables. During each run of the simulator, the program computed the detected magnetic field of each sensor as well as the pairwise products of the detected magnetic fields to compute both the expectation $E[B_i]$ and the correlation $E[B_i B_j]$. The numbering of the sensors used in these simulations is shown in Figure 4-36. Note that the numbering presented here is not necessarily the same as the numbering of the sensors used in the hardware prototype, which is given in Figure 4-1.

To check that our program was working correctly, we inspected the calculated expectations. The values for the first four sensors were $-1.17 * 10^{-22}$ T^2, $5.90 * 10^{-24}$ T^2, $-9.97 * 10^{-23}$ T^2 and $3.39 * 10^{-23}$ T^2. These values are close to 0, and are only non-zero because of limitations of computer hardware in representing decimal

numbers.

The first four rows and columns of the covariance matrix our program calculated are

$$\Sigma(0:3,0:3) = \begin{bmatrix} 2.845 & -1.758 & 1.192 & -1.261 \\ -1.758 & 2.845 & -1.261 & 1.192 \\ 1.192 & -1.261 & 2.739 & -1.801 \\ -1.261 & 1.192 & -1.801 & 2.739 \end{bmatrix} * 10^{-9}\ T^2 \tag{4.38}$$

The complete covariance matrix was calculated for 25 sensors, since up to 25 sensors were used in the physics simulations of the BLU estimator presented later in this section.

To appreciate the covariance matrix, we should consider the locations of the sensors in Figure 4-36. Sensor 0 is located on the opposite side of sensor 1, while sensor 2 is located next to sensor 0, and sensors on opposite sides are oriented in opposite directions. Thus, an external cable over the yoke will cause sensor 0 and sensor 2 to have readings with the same sign, while sensor 1 and 3 will have the opposite sign. The covariance matrix is in agreement with these observations. The $(0,1)$ term is negative, indicating that sensor 0 and sensor 1 will tend to have opposite readings due to external magnetic fields. The $(0,2)$ term is positive, indicating that sensor 0 and sensor 2 will have similar readings. Furthermore, all the diagonal terms are large and positive, because each sensor will have the most similar reading with itself.

We then used this covariance matrix to estimate simulated currents. In these simulations, the gain matrix A did not contain columns to model the uniform field. Table 4.8 shows the estimation error for the four test cases when using 6 and 10 sensors. When using 10 sensors, which include horizontal sensors, the performance of this estimator indicates that it is superior to any of the estimators we have previously considered.

The performance of the estimator as a function of the number of sensors used is shown in Figure 4-37. Note that unlike the OLS estimator, there are no cases when the error dramatically increases when adding a new sensor. This is because the

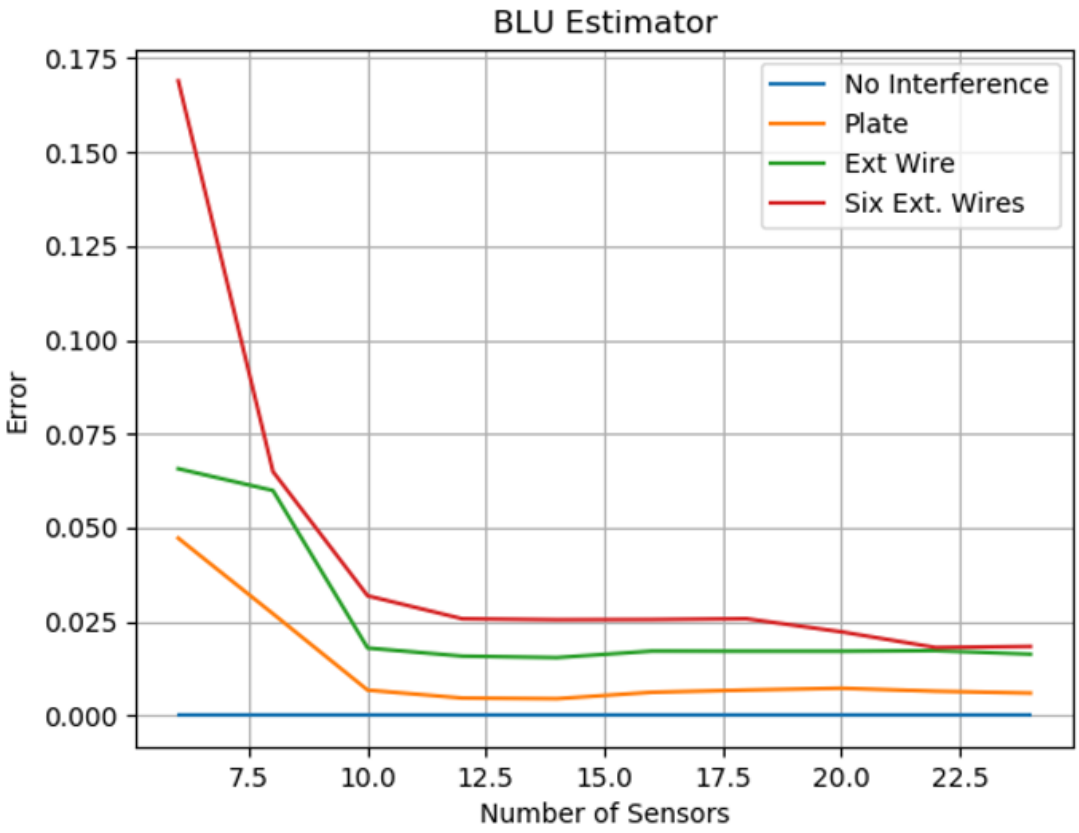

Figure 4-37: Estimation error using the BLU Estimator with the first probabilistic model.

BLU estimator takes into account the covariance between sensors when forming an estimate.

The probabilistic model we previously presented only models one external cable for each realization of the random variables. We thus created a new probabilistic model that would involve multiple external cables. Of course, it is unknown beforehand

Table 4.8: BLU Estimator error in the four special cases using the first probabilistic model.

Case	**With Six Sensors**	**With Ten Sensors**
No Interference	0%	0%
External Wire	6.5%	1.8%
Plate	4.7%	0.7%
Six Wires	16.9%	3.1%

how many external cables the system will encounter. Thus, our next model involved 22 evenly spaced external cables located around the yoke. We were attempting to approximate a model with an infinite number of external cables holding random currents, but had to work with the limitations of the computer hardware we used to run the physics simulator. The logic used to generate the 22 locations can be found in the file multicorrelation.py, found in Appendix A.

In the model of 22 evenly spaced external cables, each external current I_k is random, so for each realization of an experiment there are 22 random variables. That is to say, the random variable B_i is a sum of functions of 22 random variables and the locations x_k and y_k are constants. In this model, for a sensor whose axis of sensitivity is along the x-axis, B_i is given by

$$B_i = \sum_{k=0}^{22} \frac{u_0}{2\pi} \frac{(x_i - x_k)I_k}{(y_i - y_k)^2 + (x_i - x_k)^2} \tag{4.39}$$

The generated covariance matrix is

$$\Sigma(0:3, 0:3) = \begin{bmatrix} 5.172, -4.733, 4.118, -4.160 \\ -4.733, 5.184, -4.139, 4.148 \\ 4.118, -4.139, 6.627, -5.605 \\ -4.160, 4.148, -5.60, 6.628 \end{bmatrix} * 10^{-9}\ T^2 \tag{4.40}$$

Interestingly, it appears the magnitude of the values in this covariance matrix are greater than those found in the previous matrix. The performance of the BLU estimator improved when using this covariance matrix. Table 4.9 shows the error in the four test cases when using 6 and 10 sensors.

The performance of the estimator as a function of the number of sensors is shown in Figure 4-38. It appears that the performance of the estimator with this matrix degrades when 20 or more sensors are used. The reasons for why this occurred can be an area for future research. However, since our hardware prototype contains 10 sensors, and the BLU estimator produces the most accurate estimates of any estimator previously discussed, we considered the BLU estimator to be very promising current

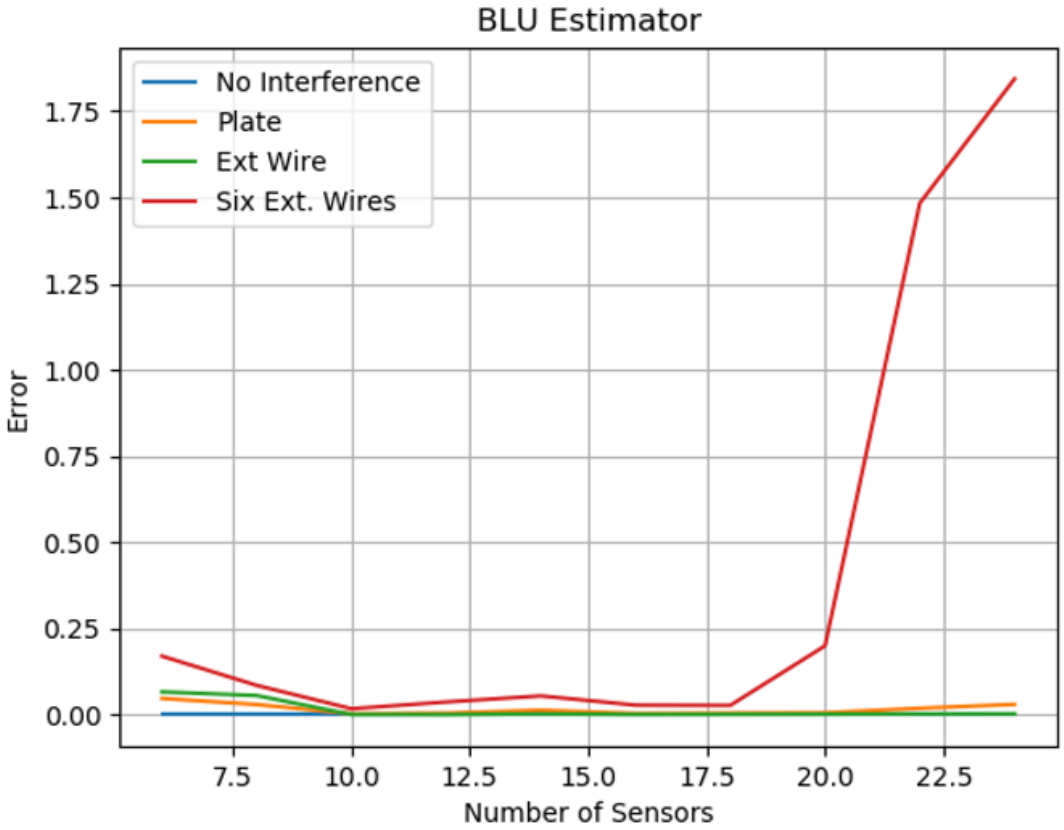

Figure 4-38: Estimation Error using the BLU Estimator with the second probabilistic model.

estimator to use in the detector. When used with the hardware prototype, the gain matrix A of the BLU Estimator was augmented to contain columns modeling uniform magnetic fields.

As previously mentioned, there is a discrepancy in the gains between the cable currents and sensor outputs calculated in the simulation and measured empirically.

Table 4.9: BLU Estimator error in the four special cases using the second probabilistic model.

Case	**With Six Sensors**	**With Ten Sensors**
No Interference	0%	0%
External Wire	6.6%	0.1%
Plate	4.7%	0.3%
Six Wires	17.0%	1.8%

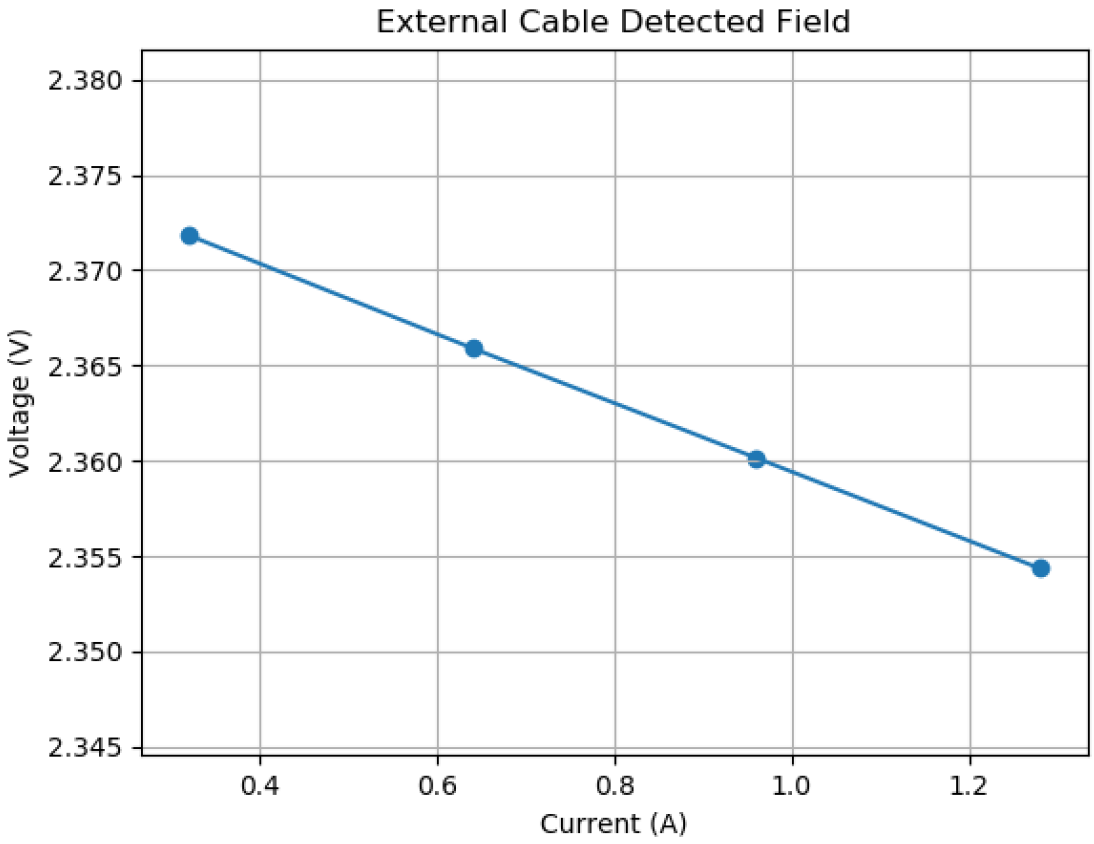

Figure 4-39: A plot of the voltage output of a particular DRV425 sensor when DC currents of different magnitudes were run through an external cable.

This is due to the interference from PCB traces powering the sensors. The measured gain matrix A produced OLS estimates containing less error than the estimates formed using the simulated matrix. Similarly, to use the BLU estimator with real hardware, it is best to generate the covariance matrix using values measured with hardware.

It is possible to generate the covariance matrix using measurements because the magnetic field is detected by a sensor from an external cable is a linear function of the external cable current. We confirmed this was the case by measuring the magnetic fields generated by an external current. The results are shown for a particular sensor in Figure 4-39.

To obtain the sensor gains used to generate the covariance matrix we placed an external cable around a detector with no internal cables and measured the magnetic

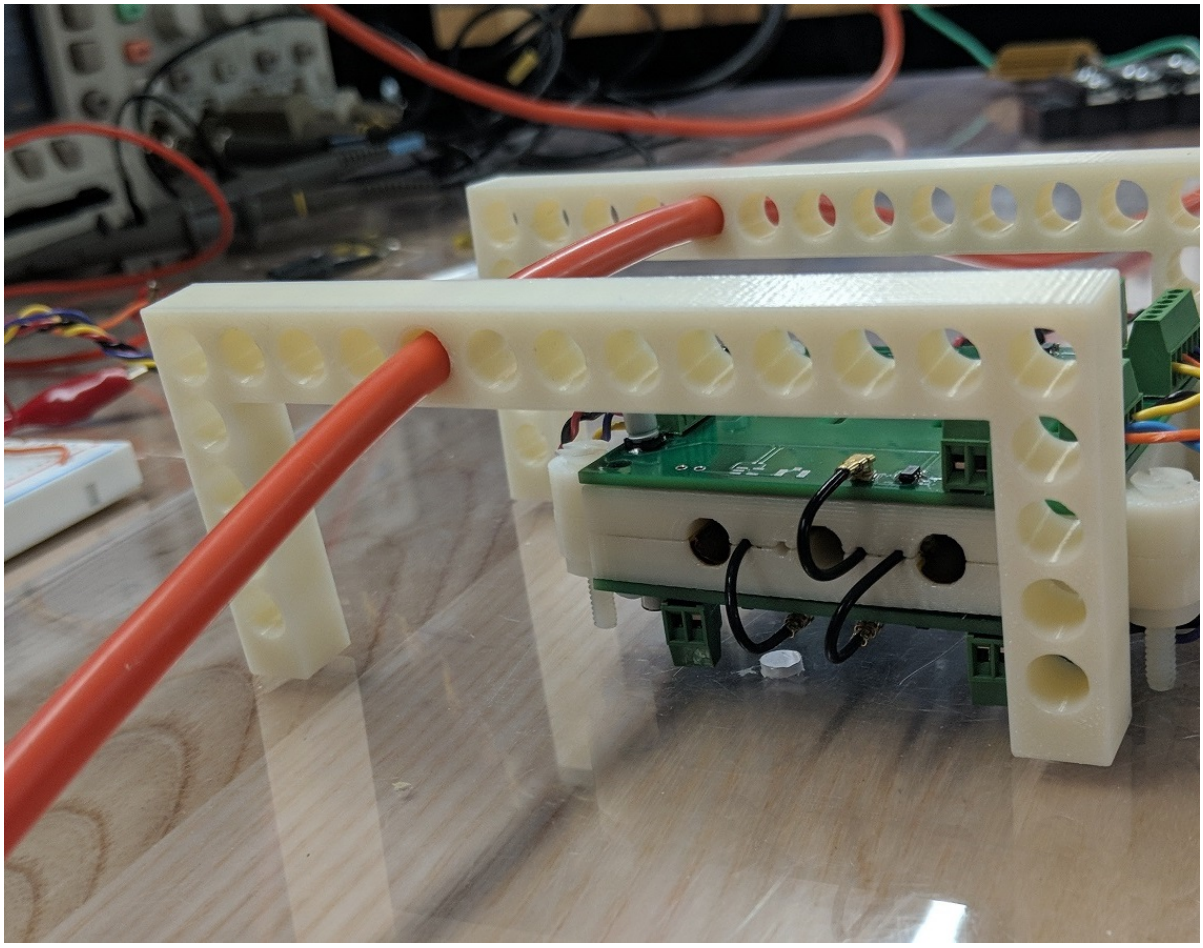

Figure 4-40: A photo of an external cable placed outside the yoke with 3D printed supports.

fields with two different DC currents. We collected readings for 14 locations of an external cable. Two of these locations are shown in Figures 4-40 and Figure 4-41. Although we only measured the sensor gains for one external cable at a time, in the presence of multiple external cables, the sensor will detect the sum of the fields generated by these external cables. Because of this, using the gains we obtained from two measurements of DC magnetic fields for each location of an external current, we were able to generate a covariance matrix similar to the second probabilistic model previously mentioned, but this time with 14 cables located around the yoke simultaneously. The script that generated the covariance matrix from these gains is named multi_covariance_hardware.py and is found in Appendix A. The measured covariance matrix generated using this procedure was used to form the BLU estimates in

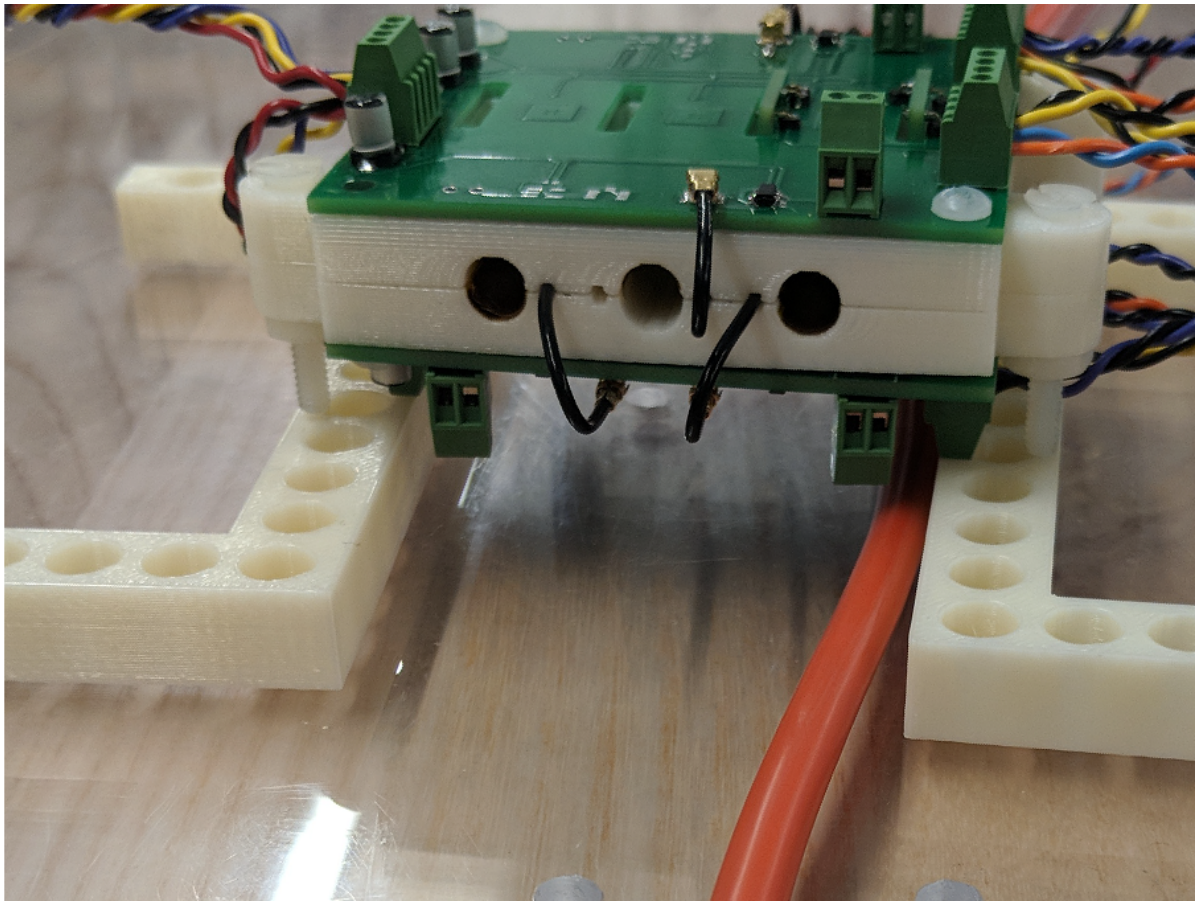

Figure 4-41: A photo of an external cable placed underneath the detector.

the test bed validation procedures presented in Section 4.10.

The performance of the BLU Estimator with the second probabilistic model was superior to the performance of the other estimators we tested. The estimation errors in the four test cases were consistently lower than the estimation errors of other estimators, and the BLU Estimator can theoretically produce estimates with no error in the absence of sensor noise and external interference, which is not true of all estimators. Therefore, we use the BLU Estimator in test bed validation experiments and it is the estimator we have chosen to use in our final current detector system.

4.9 Machine Learning Methods

We also tested supervised machine learning methods to explore whether these methods could produce an estimator superior to any of the ones described in the previous section. The two principal machine learning techniques we experimented with were regression and neural net training.

4.9.1 Regression Estimator

The regression estimator involved fitting a training set of magnetic field readings, both with and without external field interference, to the correct current readings for each training sample. The learning algorithm was effectively finding a matrix D and bias term f that would best fit the relationship $Db + f = I$ for all training samples. It did this by minimizing the total squared error between the true and estimated current. In other words, it minimized the expression given by

$$min \frac{1}{N} \sum_{n=0}^{N} (\hat{I_n} - (Db_n + f))^2 \tag{4.41}$$

We used the physics simulator to generate a training set made of 1,000,000 samples. In each sample, a randomly chosen number of external cables was placed in the simulation. With 50% probability, this number was 0, so half of the training samples did not include external interference. In the other cases, 1-7 external cables were placed randomly around the yoke among the same range of locations used in the BLU Estimator simulations, presented in Section 4.8.6. Each sample thus consisted of the magnetic field readings detected by the sensor and the three true current values.

We used linear regression to fit the magnetic field readings to the current values. To do this, we used the sklearn.linear_model.LinearRegression class from the sklearn toolkit. Given the low number of parameters in this problem, we used the closed form linear regression procedure rather than an iterative approach such as Gradient Descent. The code used for this procedure can be found in the sim_estimator_per_sensors function of the run_simulation.py file, found in Appendix A.

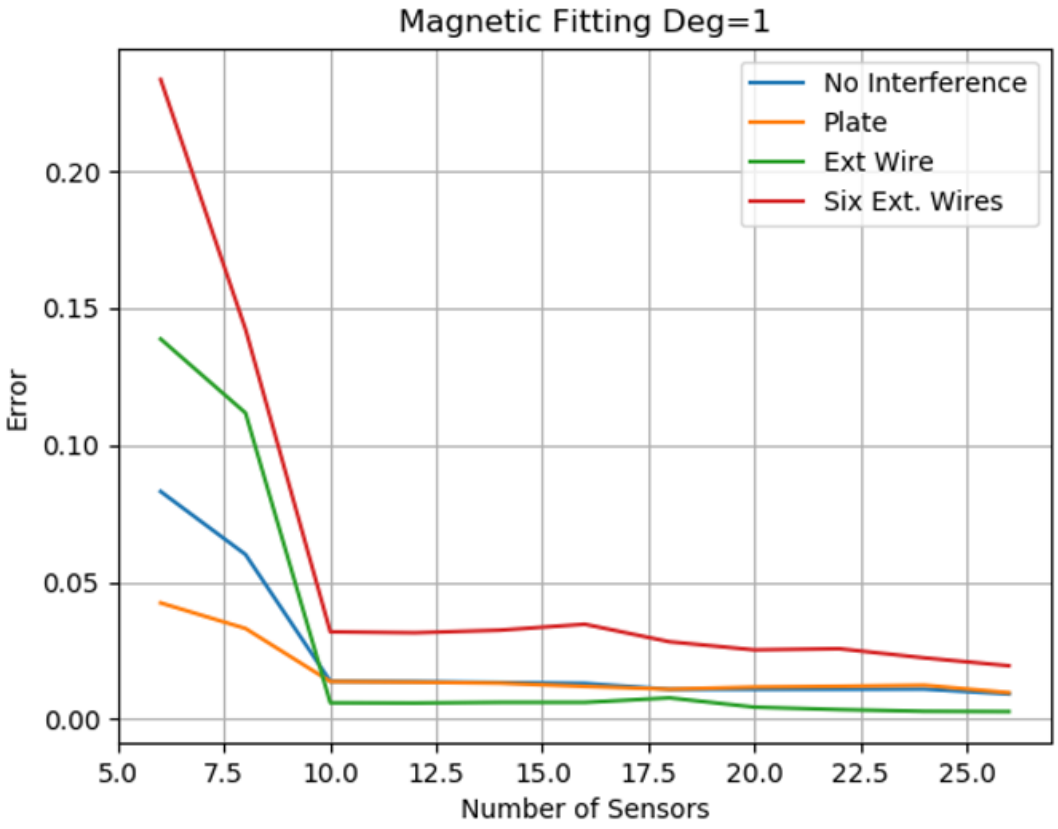

Figure 4-42: Estimation error using the Regression Estimator as a function of the number of sensors used.

Table 4.10 shows the estimation errors achieved for the four test cases when using 6 and 10 sensors. Figure 4-42 shows the percent errors as a function of the number of sensors used. Although the estimator exhibits good performance when 10 sensors are used, a flaw of the estimator is that it suffers non-zero estimation error when there are no sources of magnetic field interference.

Table 4.10: First order Regression Estimator error in the four special cases.

Case	**With Six Sensors**	**With Ten Sensors**
No Interference	8.3%	1.3%
External Wire	13.8%	0.5%
Plate	4.2%	1.3%
Six Wires	23.3%	3.2%

We then experimented with higher-level polynomial regression. This involved transforming the input vector of magnetic fields into a vector containing higher level products of the vector elements. In a 2nd degree transformation, this would mean transforming the input vector $b_0, b_1, ..., b_N$ into a vector that also includes all possible pairwise products, $b_0^2, b_0b_1, b_0b_2, ..., b_N^2$. In a third degree transformation, the vector is expanded to include all possible three-term product combinations, and so on.

Once the vector b is transformed into a higher-level polynomial version of itself, we proceed with regression using the same library function as before. This method allows us to fit a function based not only on magnetic field readings, but also on products of those readings, thus creating a function that better fits the relationships between readings.

The results for 3rd degree regression for the four test cases are shown in Table 4.11 in the case of 6 and 10 sensors, and Figure 4-43 shows the results as a function of the number of sensors used. This estimator also exhibited non-zero error in the case of no external interference.

The results of the two estimators discussed in this section were mixed. Although the first order Regression Estimator outperformed the OLS estimator when there was external interference, it does not achieve 0% error when there is no interference. The third order Regression Estimator formed very good estimates in the case when there was a single external wire or plate, but performed poorly in the case of six wires. Since the BLU estimator is more consistent in its ability to produce low error estimates, we decided to focus on it instead of the Regression estimators when performing experiments with hardware.

Table 4.11: Third order Regression Estimator error in the four special cases.

Case	**With Six Sensors**	**With Ten Sensors**
No Interference	7.9%	1.8%
External Wire	7.4%	3.2%
Plate	13.1%	2.0%
Six Wires	19.2%	1.7%

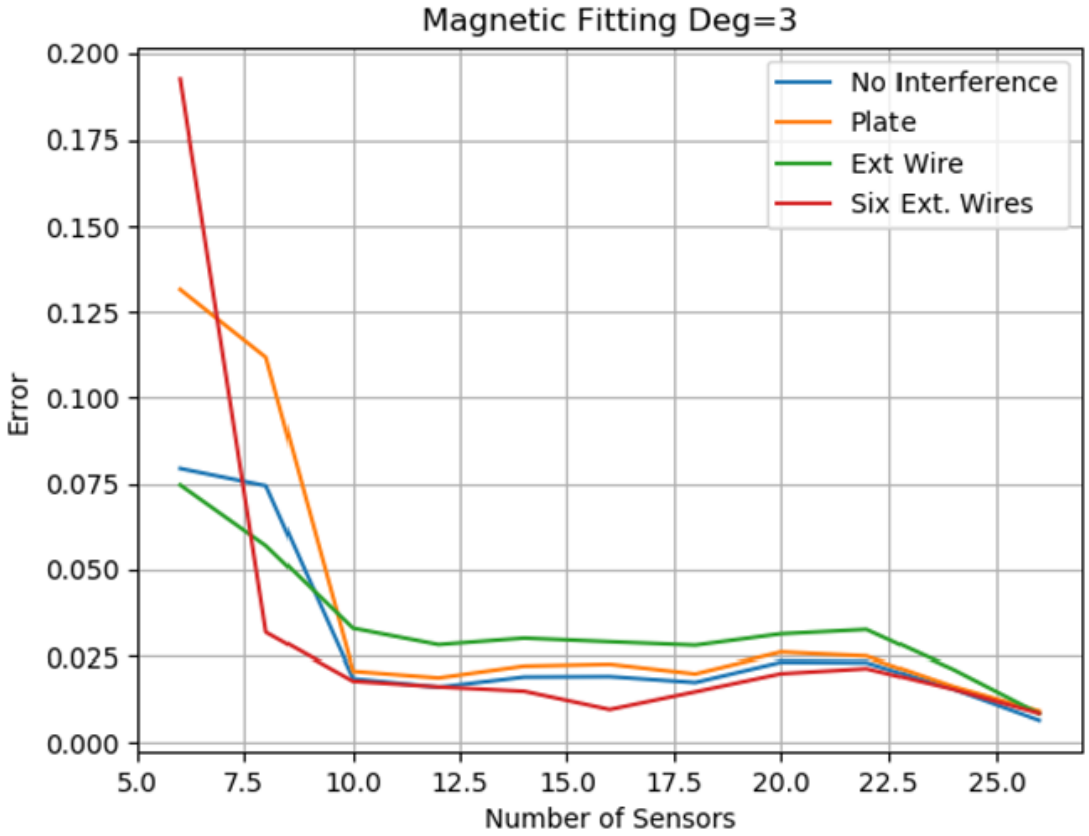

Figure 4-43: Estimation error using the Regression Estimator with a third degree polynomial transformation as a function of the nubmer of sensors used.

4.9.2 Neural Net Training

We trained neural nets to explore how well these could estimate currents in the presence of interference. To train the nets, we used the same training set that we used for the Regression Estimator discussed in the previous section, which simulated magnetic field readings collected from a set of currents in the presence of 0-7 external cables located at random locations around the yoke.

We trained the neural nets using the tensorflow framework. [1] The optimizer was the Adam Optimizer [8] and the learning rate was set to 0.1. The function being minimized was the RMSE, $\sum_{i=0}^{2}(y_i - \hat{y}_i)^2$, where y is the vector of true current values in a training sample and $\hat{y}$ is the current vector estimate produced by the neural net. The input to the neural net was a 10x100 tensor, representing 10 magnetic field

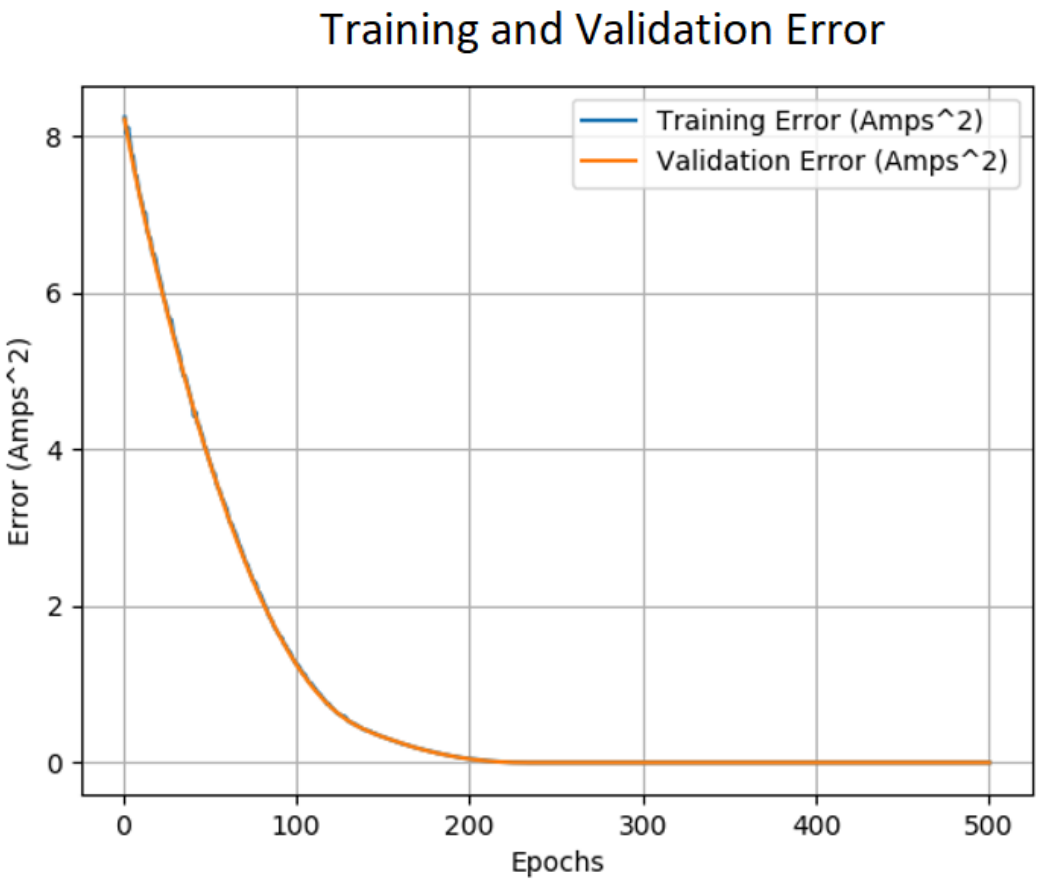

Figure 4-44: The training and validation errors of a neural network as a function of the number of epochs trained. The training set used did not include magnetic field interference.

readings in units of mT collected over 100 time samples. The training sample output was a 3x100 tensor, representing the 3 correct current values over 100 time samples.

To confirm our code was working correctly, we first trained neural nets in the case of no external interference. The resulting neural nets were able to produce extremely accurate results. After 450 epochs of training, the RMSE training and validation error were under 0.00005 A^2. The average current estimation error 0.25%. The training and validation errors as a function of the number of epochs trained are shown at different scales in Figures 4-44 and 4-45. The neural nets were able to achieve performance comparable to the OLS and BLU estimator in the case when there was no external interference, validating that the training data and the neural net itself were correctly

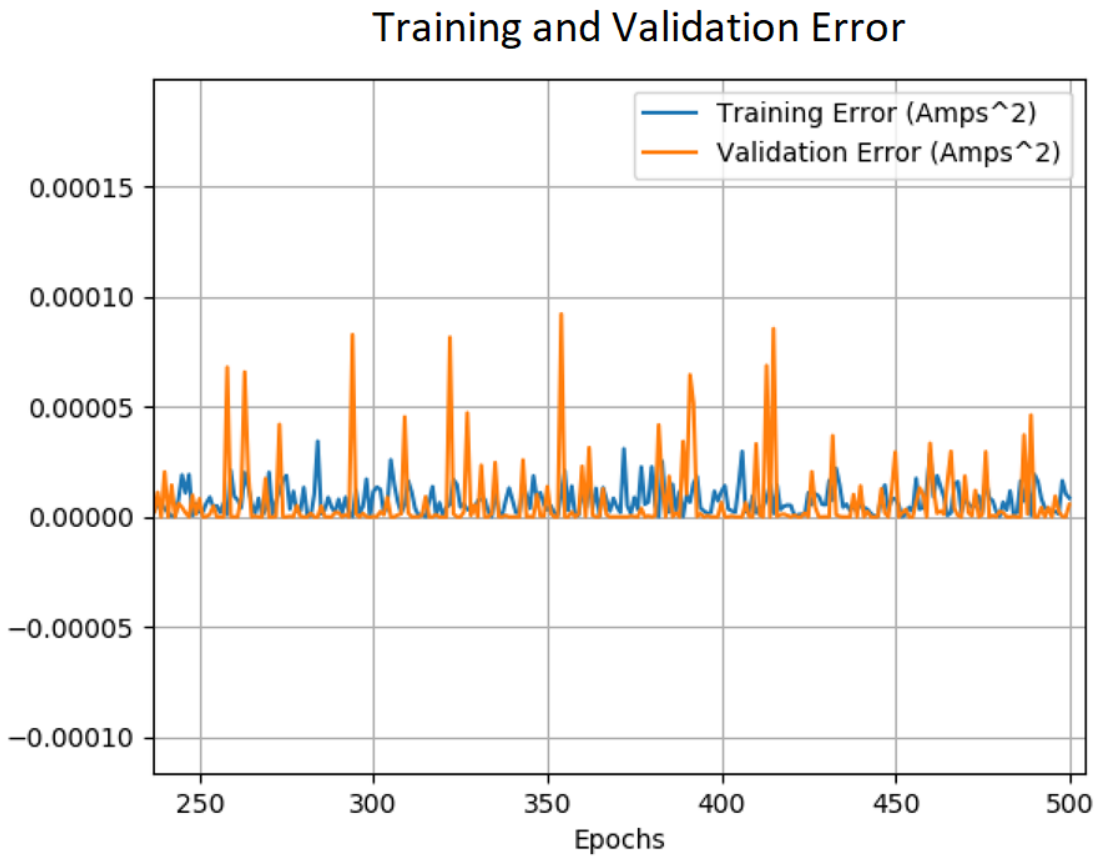

Figure 4-45: A closer view of the training and validation errors after training for more than 250 epochs for a training set that did not include external field interference.

coded.

We then trained the neural nets on the full training set, which included magnetic field readings in cases with external interference. As previously mentioned, half the training samples were free from interference and half the training samples had 1-7 external cables as interference. However, the neural nets were not able to achieve a low error estimate. After 2000 epochs, the training error was still at 0.05 A^2. This corresponded to an average current estimation error of around 5%. The error as a function of training is shown at different scales in Figures 4-46 and 4-47. Since the Neural Network Estimator did not produce estimates with less error than the BLU Estimator, we decided to focus on the BLU Estimator when performing test bed validation experiments.

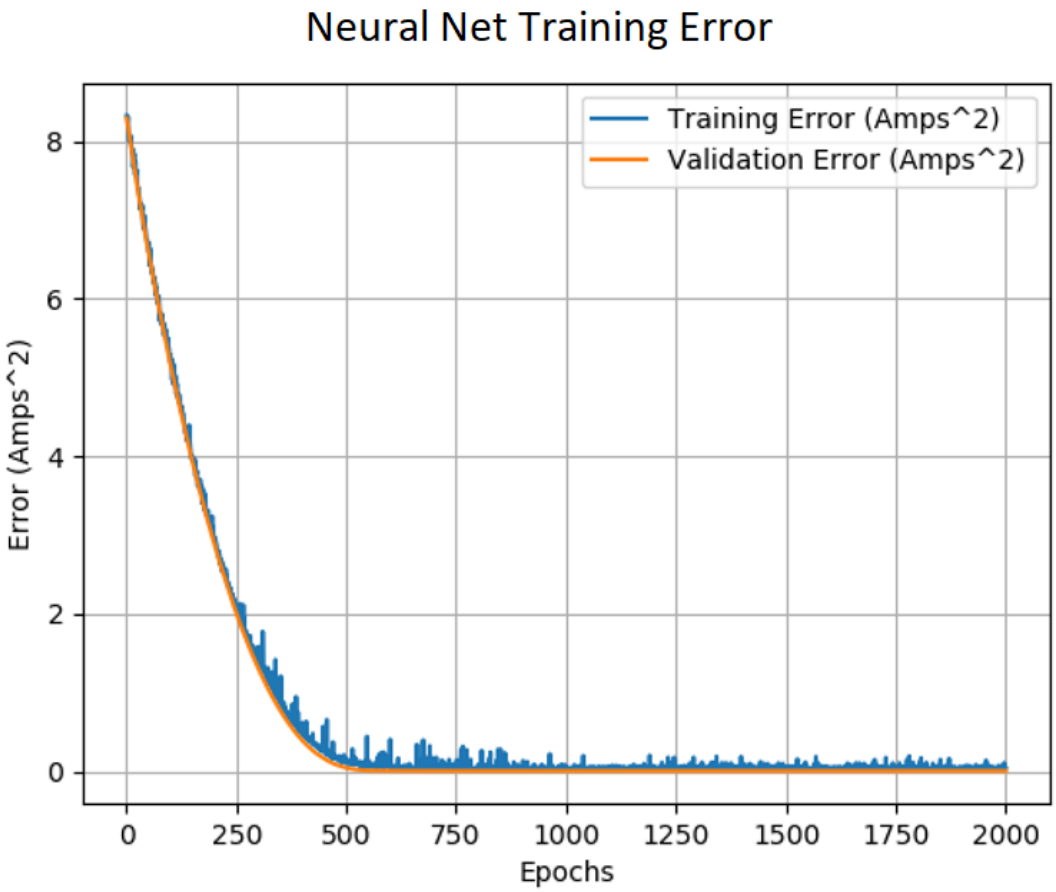

Figure 4-46: The training and validation errors of a neural net as a function of the number of epochs trained. The training set used included readings with external magnetic field interference.

4.9.3 Summary of Current Estimation Methods

We decided to use the BLU Estimator to estimate currents in the presence of challenging interference. Although the Regression estimator also produced estimates with very little error, we chose the BLU estimator because of its better performance and because its error is zero when there is no interference. Although other estimators we considered, such as the Non-Linear Estimator, attempt to more accurately model external interference, a fundamental weakness of these estimators is that the number of parameters they can model is limited by the number of sensors in the system. This is a weakness the BLU estimator does not have to overcome.

The neural networks we trained did not perform as well as other methods. We

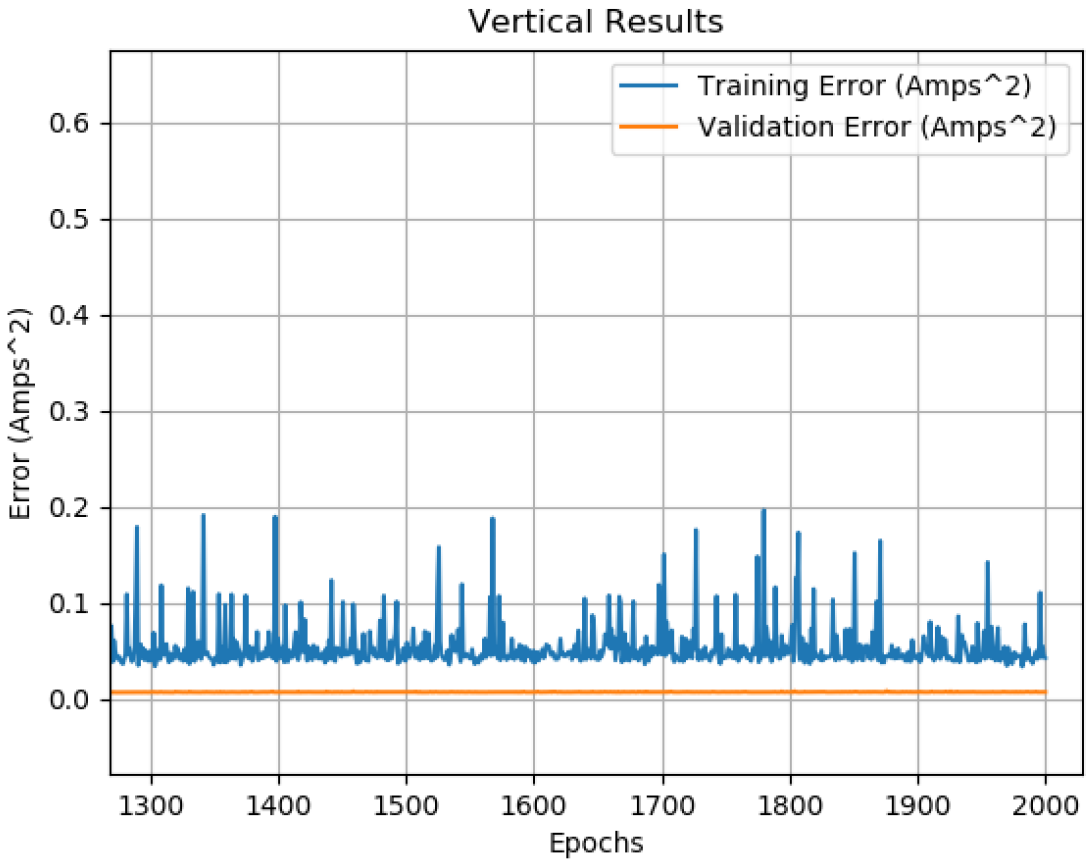

Figure 4-47: A closer view of the training and validation errors after 1400 epochs of training on a training set that included external magnetic field interference..

believe this was because the training set contained both samples with and without interference. The neural net tries to fit a function around all these samples, without any knowledge of which samples are free of interference. An area for future research may be the use of neural networks especially designed to distinguish between correct and incorrect training samples, such as Siamese networks.

In the test bed validation experiments that follow, either the OLS Estimator or the BLU Estimator using the second probabilistic model are used. Each experiment will mention which estimator was used. The estimators are used with gain and covariance matrices that were calculated using real hardware, not simulated values. Lastly, although the OLS and BLU Estimators did not include the uniform field columns in the simulated experiments that plotted estimate error as a function of the number of

sensors, they do use gain matrices augmented with the uniform field columns in the hardware experiments presented in the next section.

4.10 Test Bed Validation

4.10.1 Parallel Cable Test Bed

We used the parallel cable test bed to run 90 Hz current in a balanced three phase configuration while placing different forms of external interference around the detector. We ran these experiments at 90 Hz to be able to distinguish between the interference created by the sources we introduced and ambient 60 Hz magnetic fields.

No external interference

In the first validation experiment, we applied 8.3 V 90 Hz voltages to the parallel cable setup, producing a set of 1.67 A 90 Hz currents. We confirmed that the contact voltage and current measurements we collected were correct by also measuring the voltages with an oscilloscope. Figure 4-48 shows the readings output by the oscilloscope, which were saved into a floppy disk as a CSV file.

Figure 4-49 shows the three OLS current estimates superimposed over the three cable currents measured using contact measurements in the case when there is no nearby interference. The contact measurements were performed by measuring the voltage drops across the two resistors in the parallel cables test bed, shown in Figure 3-18, and assuming the return current was the sum of the first and third currents. The constraint that the second current is the sum of the first and third currents is unknown to the OLS estimator. The percent error between the estimate and the measured current is 0.43%.

To confirm the gains of the estimator were scaling correctly, we applied 10 V 90 Hz voltages to the parallel cables setup, which produced a set of 2 A 90 Hz currents. The oscilloscope readings of these voltages are shown in Figure 4-50.

The OLS estimate of these currents is shown in Figure 4-51. The error is 0.46%.

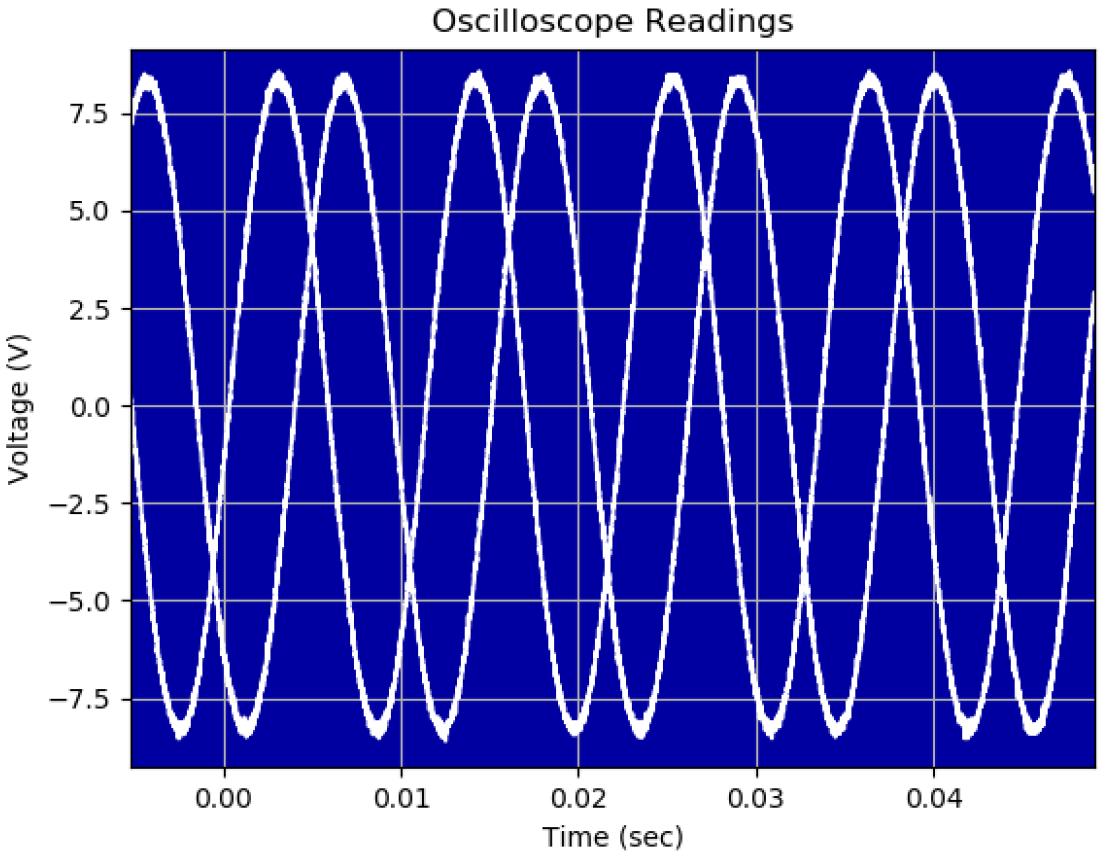

Figure 4-48: A plot of the voltage data exported by the oscilloscope when 8.3 V 90 Hz voltage was output by the op-amps.

As the Figure shows, the detector is able to accurately estimate currents of different magnitudes.

External Cables Interference

We placed a pair of external cables 1 cm above the detector as shown in Figure 4-52. The cables were running currents of the same magnitude and frequency as the internal cables and had opposing signs with respect with each other. The OLS estimate is shown in Figure 4-53. The error was 0.52%. As this experiment shows, the OLS estimator was able to produce an accurate estimate even in the presence of interference. This was due to the placement of the sensors, especially the vertical sensors, since experiments with earlier versions of the detector that did not contain

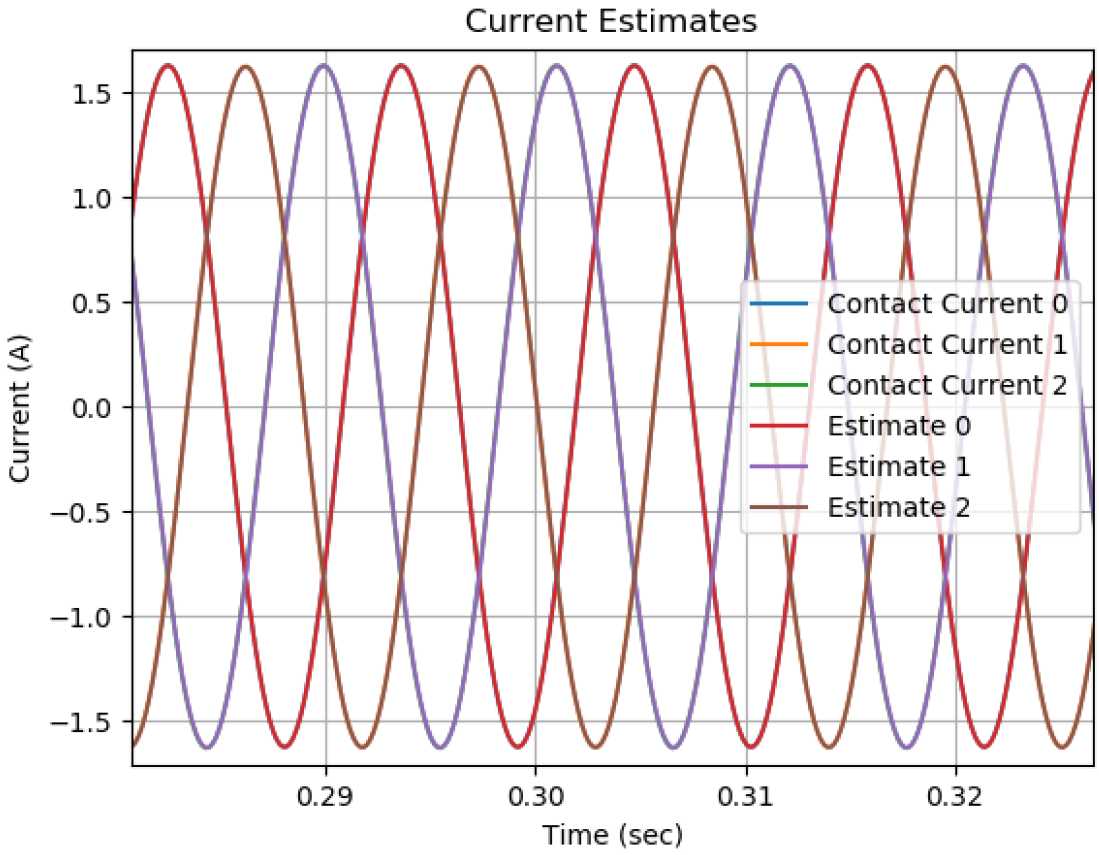

Figure 4-49: The OLS estimate of a set of 1.67 A 90 Hz currents.

vertical sensors performed much worse in similar experiments.

External Plate Interference

We placed an external plate 1 cm above the detector as shown in Figure 4-54. The estimate is shown in Figure 4-55. The error was 0.54%.

Cable Bundle Interference

We placed a bundle of six external cables 1 cm above the detector as shown in Figure 4-56. The OLS estimate is shown in Figure 4-57. The error was 0.59%.

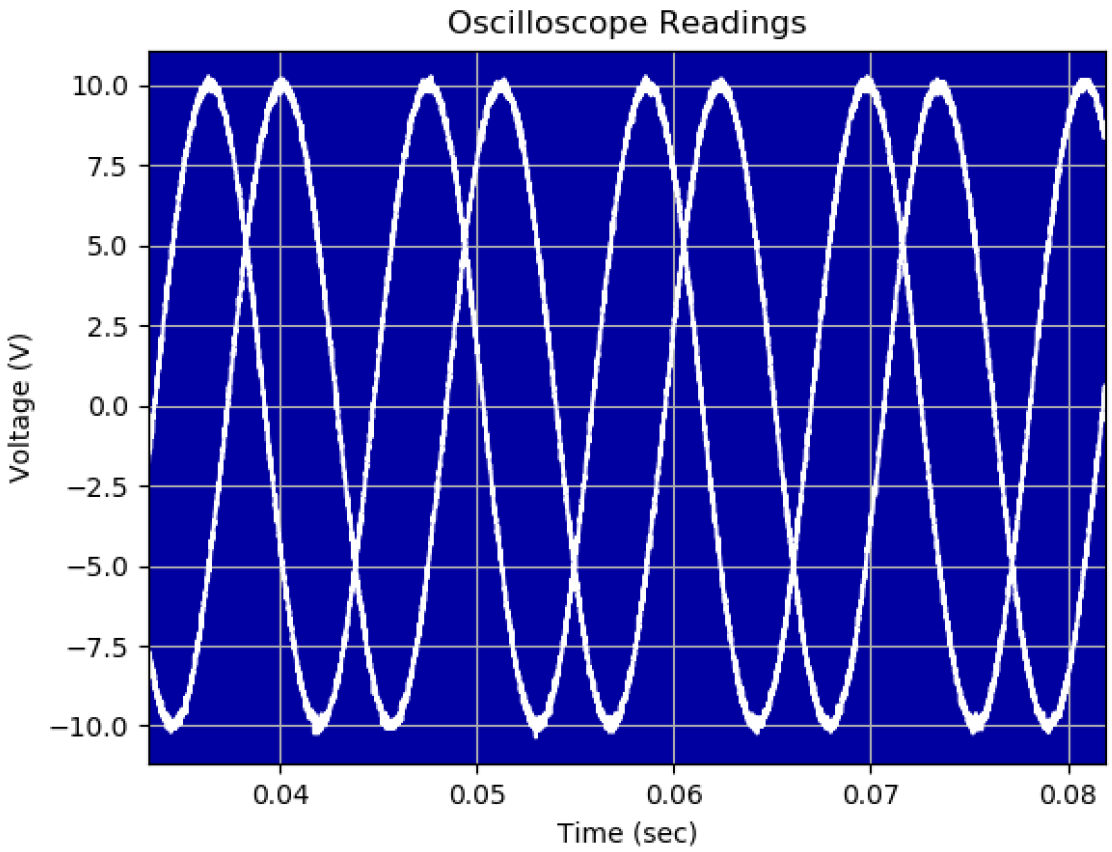

Figure 4-50: A plot of the voltage data exported by the oscilloscope when 10 V 90 Hz voltage was output by the oscilloscopes.

Iron Core Interference

In the previous experiments, the OLS estimator was sufficient to estimate the currents to less than 1% accuracy. However, we wanted to test a case in which a significant estimation error was created by interference, to test whether the BLU estimator could overcome this interference.

We placed an iron core 1.5 cm above the detector as shown in Figure 4-58. The OLS estimate is shown in Figure 4-59 and had an error of 2.79%. We then applied the BLU estimator to these readings and achieved the estimate shown in Figure 4-60, which had an error of 0.90%. This demonstrated the covariance matrix generated using hardware readings was effective in rejecting external interference.

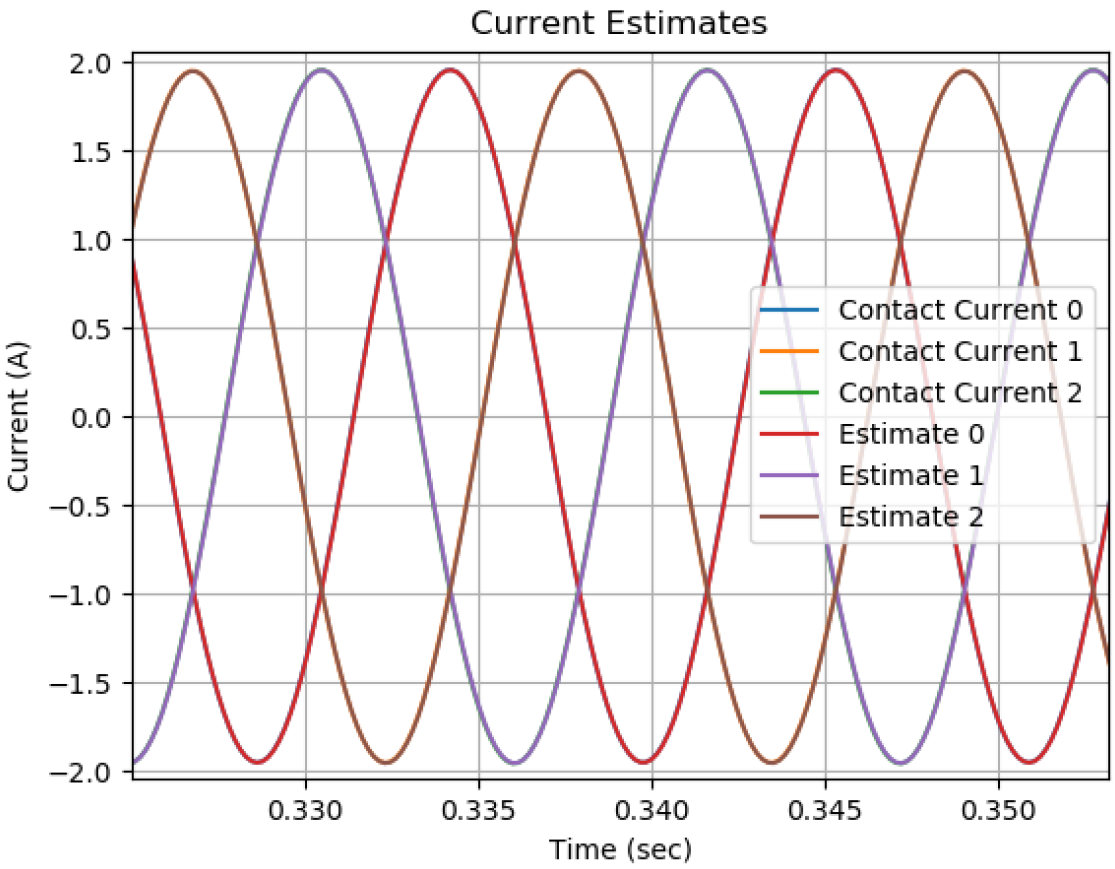

Figure 4-51: The OLS estimate of a set of 2 A 90 Hz currents.

No Current

We removed the cables from the detector to analyze the output of the estimator when there were no internal currents. Figure 4-61 shows the estimator output. The current estimates have a very low frequency and an amplitude in the range of 1 mA.

4.10.2 Lightbulb Demo

We connected the detector to the lightbulb demo. We placed two 15 W lightbulbs on the demo. As previously mentioned, we used 1.08 Ω resistors to measure the current in the demo box by measuring the voltage drop across the resistors. Figures 4-62 and 4-63 shows the currents across both resistors, as calculated by measuring the voltage drop across both resistors and dividing the voltage by 1.08 Ω.

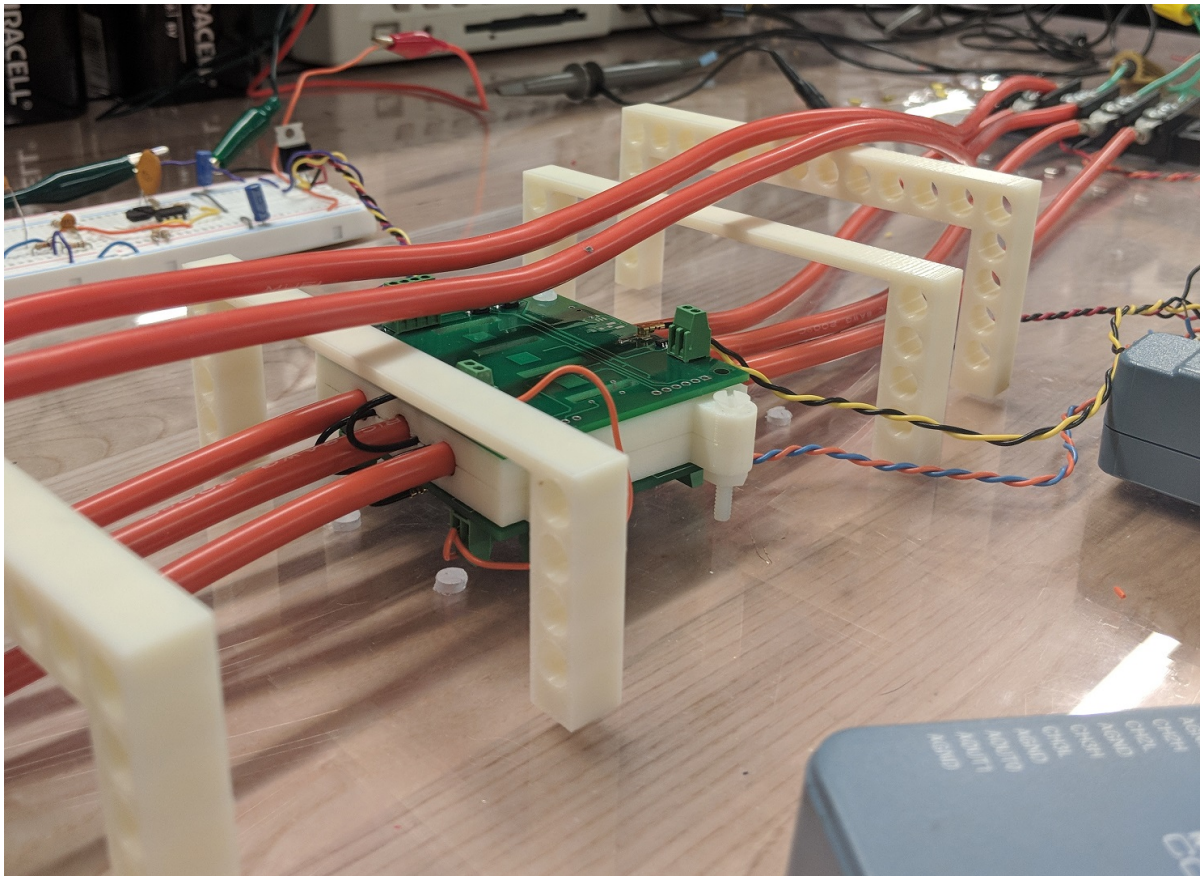

Figure 4-52: A pair of external cables were placed 1 cm above the detector.

No Interference

Figure 4-64 shows the estimates produced by the detector superimposed over the currents as measured with contact measurements. The two light bulb currents are equal and in phase since the hot ends of the bulbs are both connected to 120 V RMS power. The third current, however, is the sum of the first two and is 180 ° out of phase. The estimate error was 0.77%.

External Cables Interference

A pair of external cable were placed 1.5 cm above the detector. The cables ran 0.2 A 90 Hz current. The estimate is shown in Figure 4-65. The error was 0.80%.

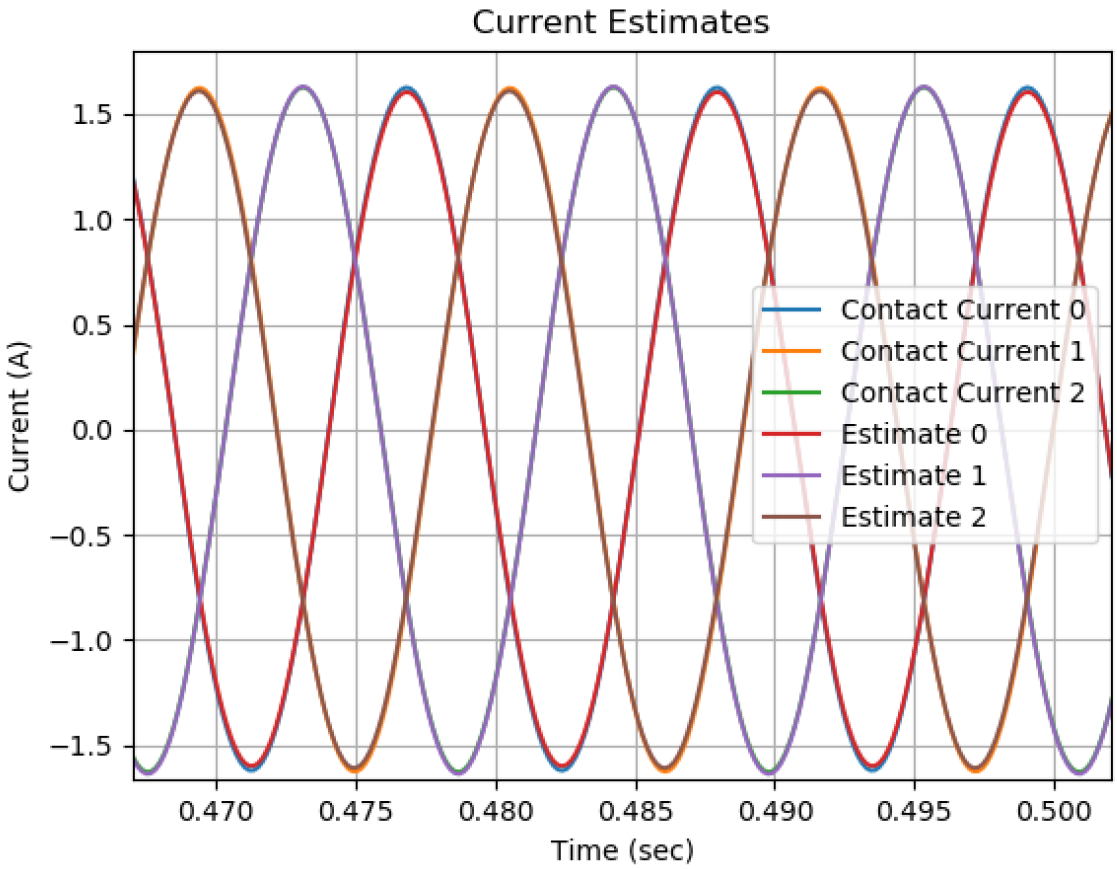

Figure 4-53: The OLS current estimates.

External Plate Interference

An aluminum plate was placed 1.5 cm above the detector. The estimate is shown in Figure 4-66. The error was 0.83%.

Cable Bundle Interference

A bundle of six cables was placed 1.5 cm above the detector. The cables ran 0.2 A 90 Hz current. The estimate is shown in Figure 4-67. The error was 0.88%.

4.11 Summary

The results presented in this chapter demonstrate that the current detector is able to achieve current estimates with less than 1% error in the presence of challenging

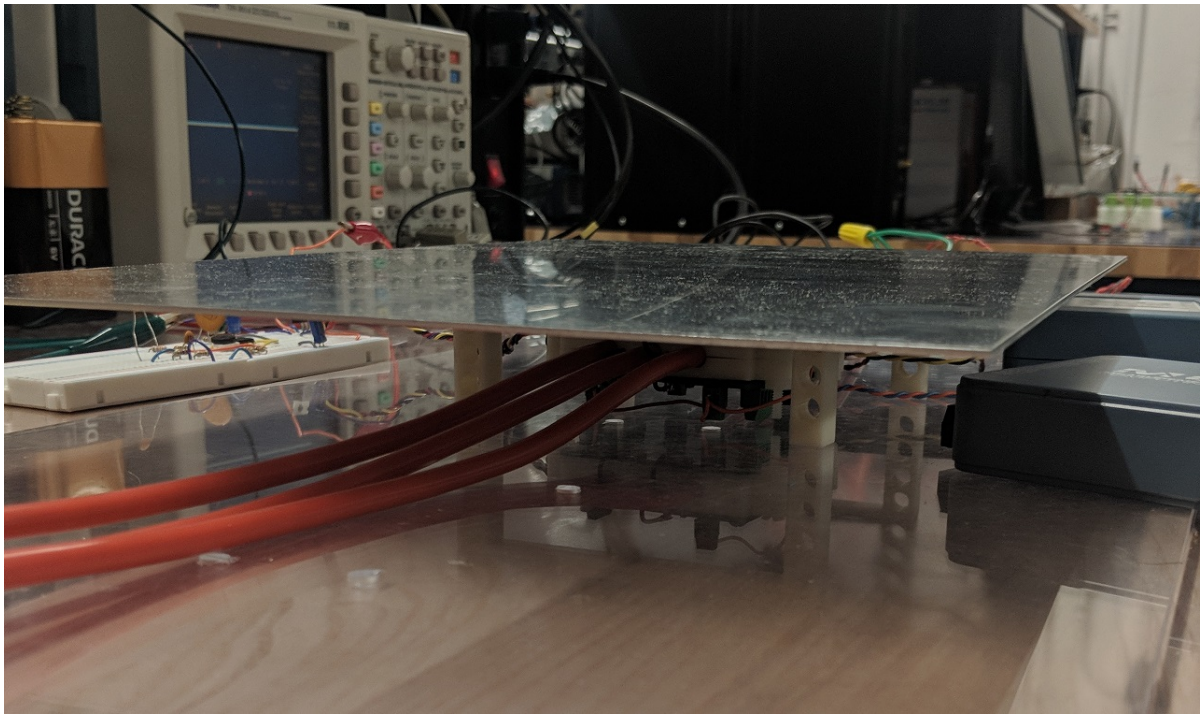

Figure 4-54: A plate was placed 1 cm above the detector.

interference. It is able to produce such accurate estimates due to both the placement of the sensors and the algorithm used to form the estimate. We presented several different current algorithms and tested them with the same set of computer simulations. The simulations showed that the BLU estimator was the best algorithm due to its high accuracy and its ability to produce zero error estimates in simulated cases where there is no sensor noise or interference. Furthermore, we showed that the BLU estimator was able to improve on an estimate that had been corrupted by a nearby large iron core. Using the DRV425 sensor, the maximum current frequency that can be detected is 47 kHz. Furthermore, the sensor placement and algorithms presented in this chapter are valid for a sensor array using any magnetic field sensor that takes point measurements. An area for further research is the use of different magnetic field sensors that can detect a larger range or wider bandwidth of magnetic fields.

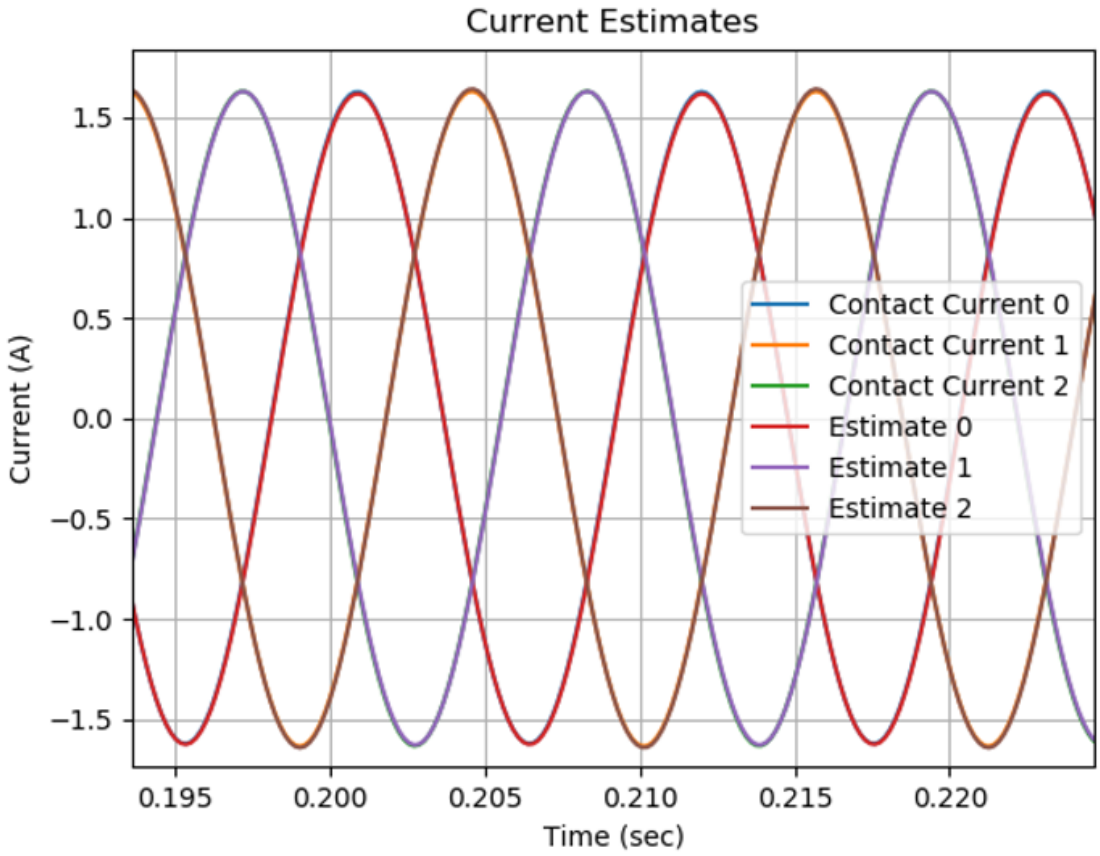

Figure 4-55: An experiment in which currents were estimated in the presence of an external plate.

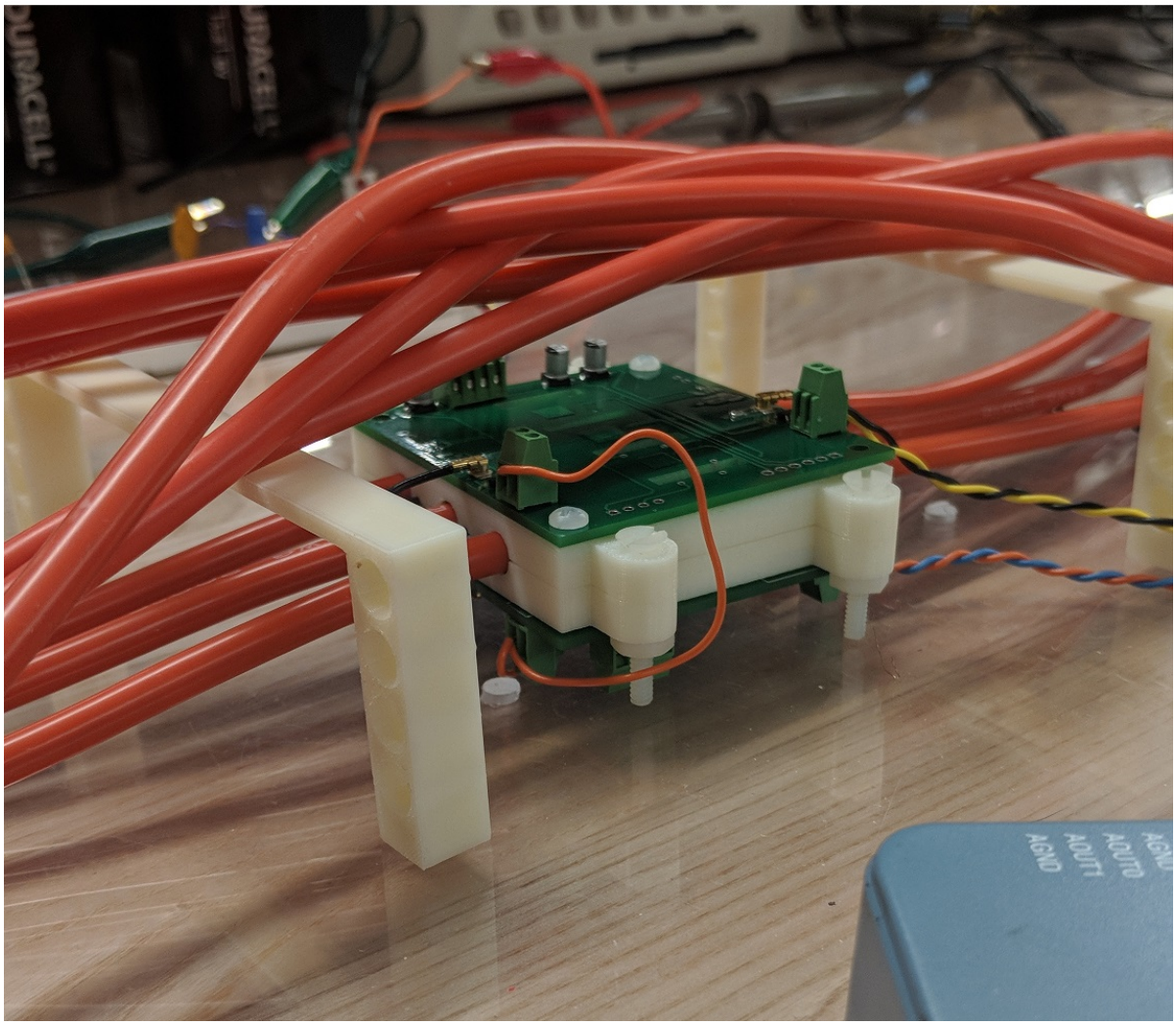

Figure 4-56: The bundle of six cables.

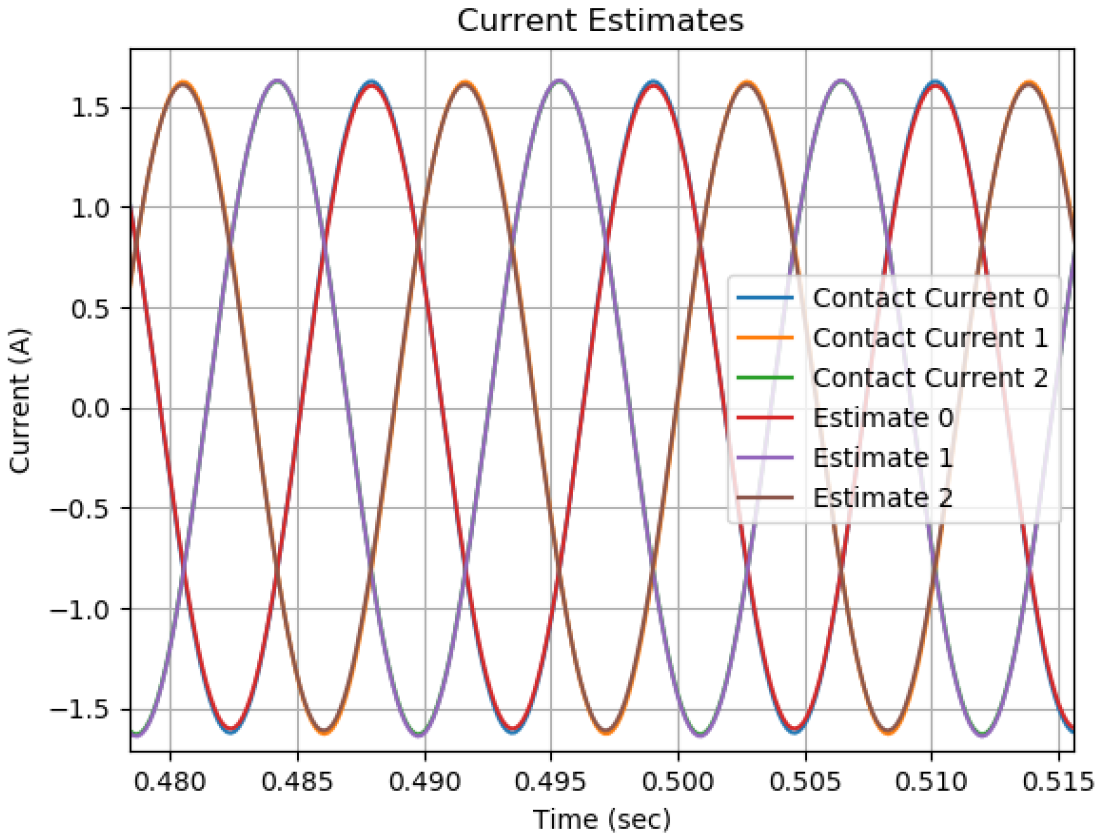

Figure 4-57: An experiment in which current was estimated in the presence of six external cables.

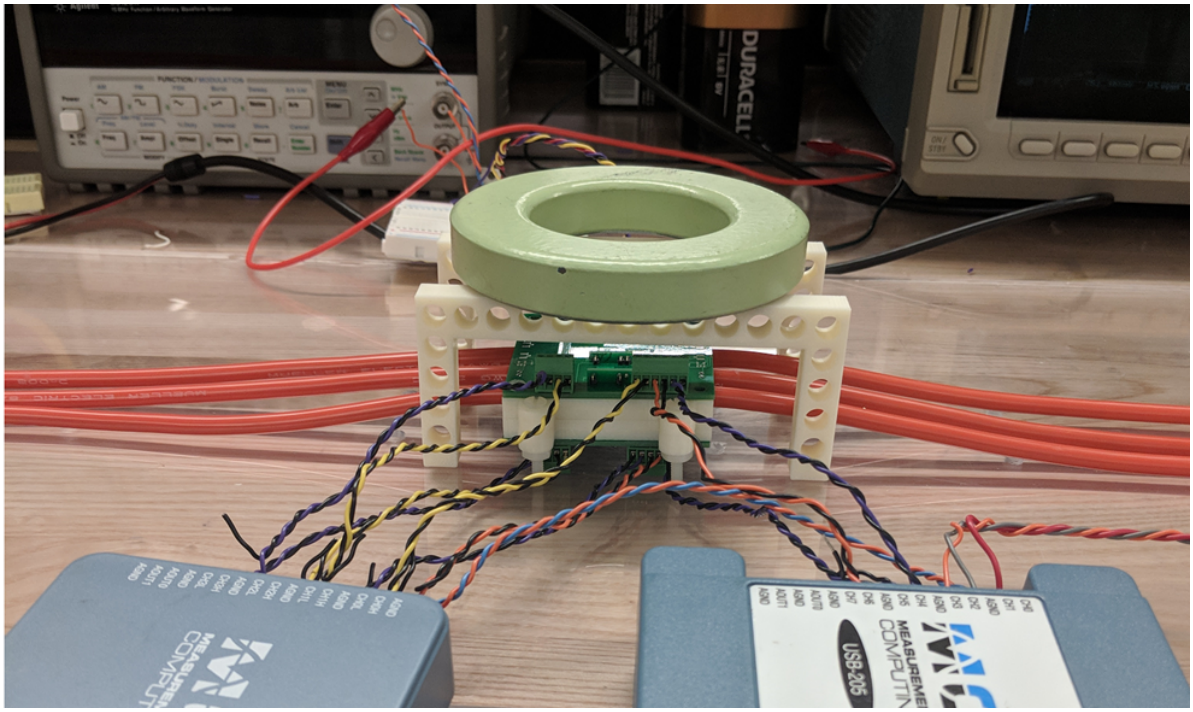

Figure 4-58: An iron core was placed 1.5 cm above the detector.

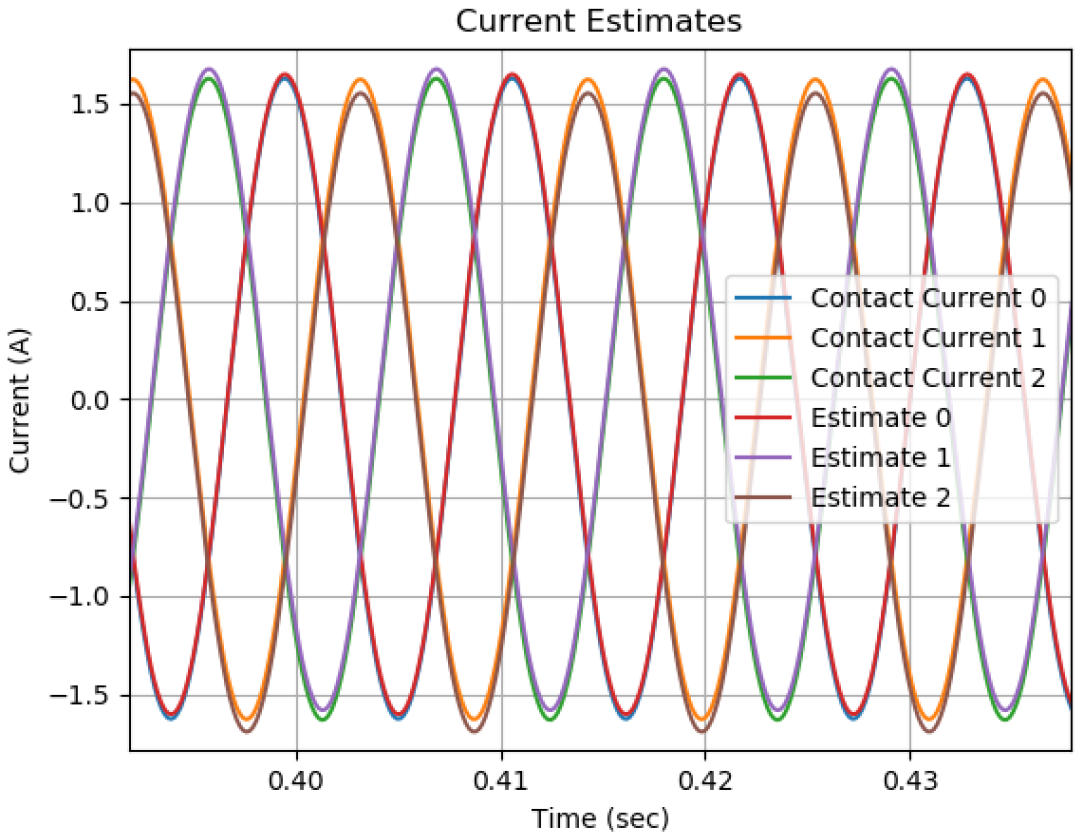

Figure 4-59: The OLS estimate with 2.79% error.

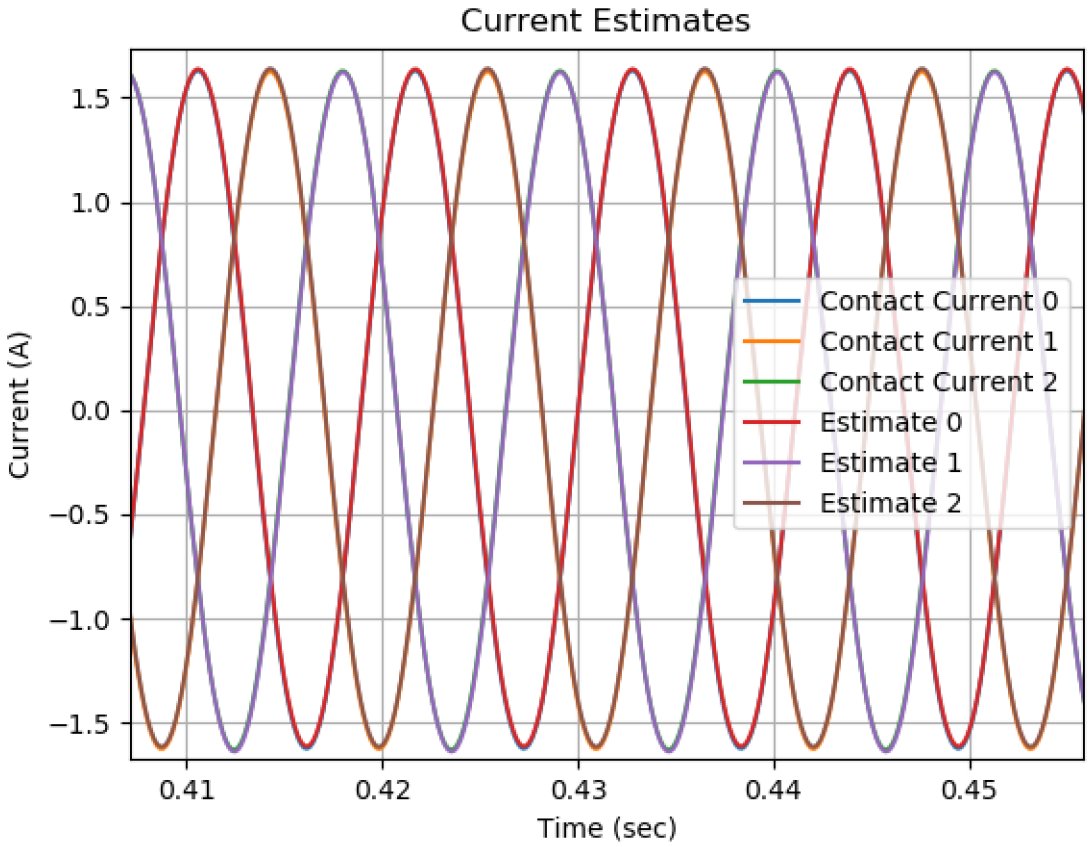

Figure 4-60: The OLS and the BLU estimators are used to estimate current in the presence of a large iron core.

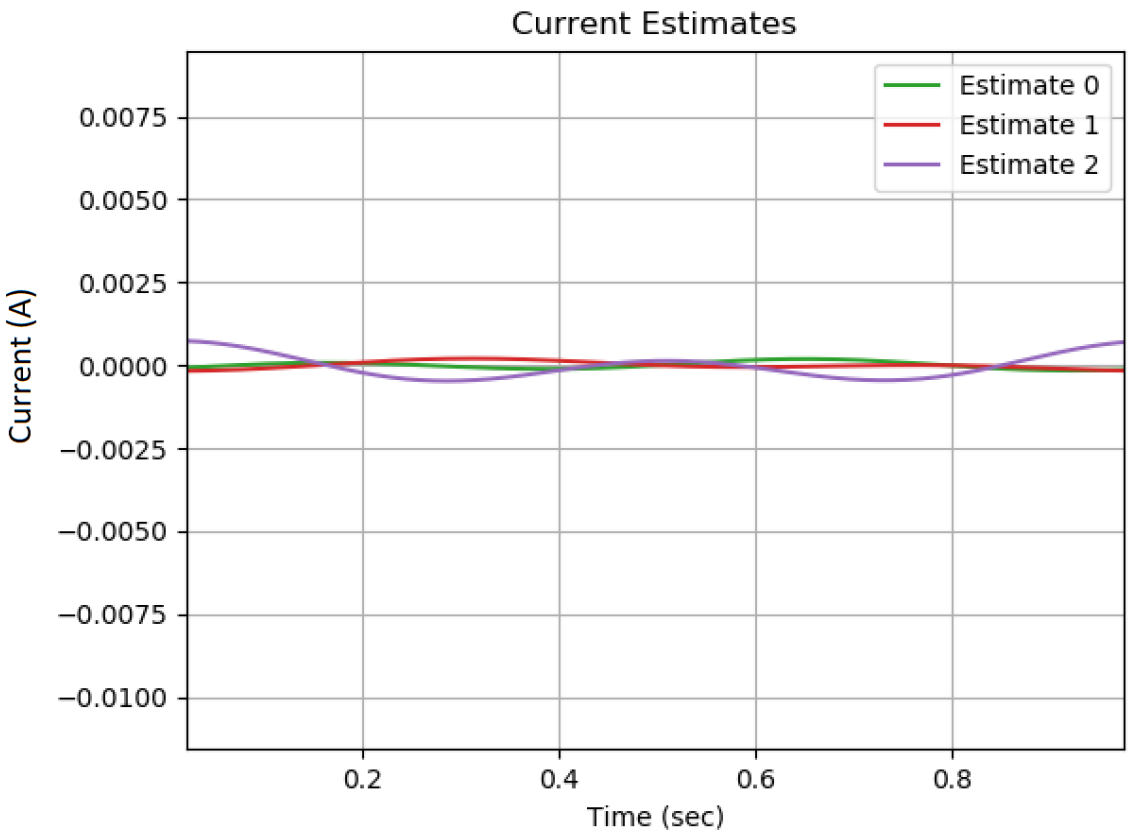

Figure 4-61: Current estimates when there are no internal currents.

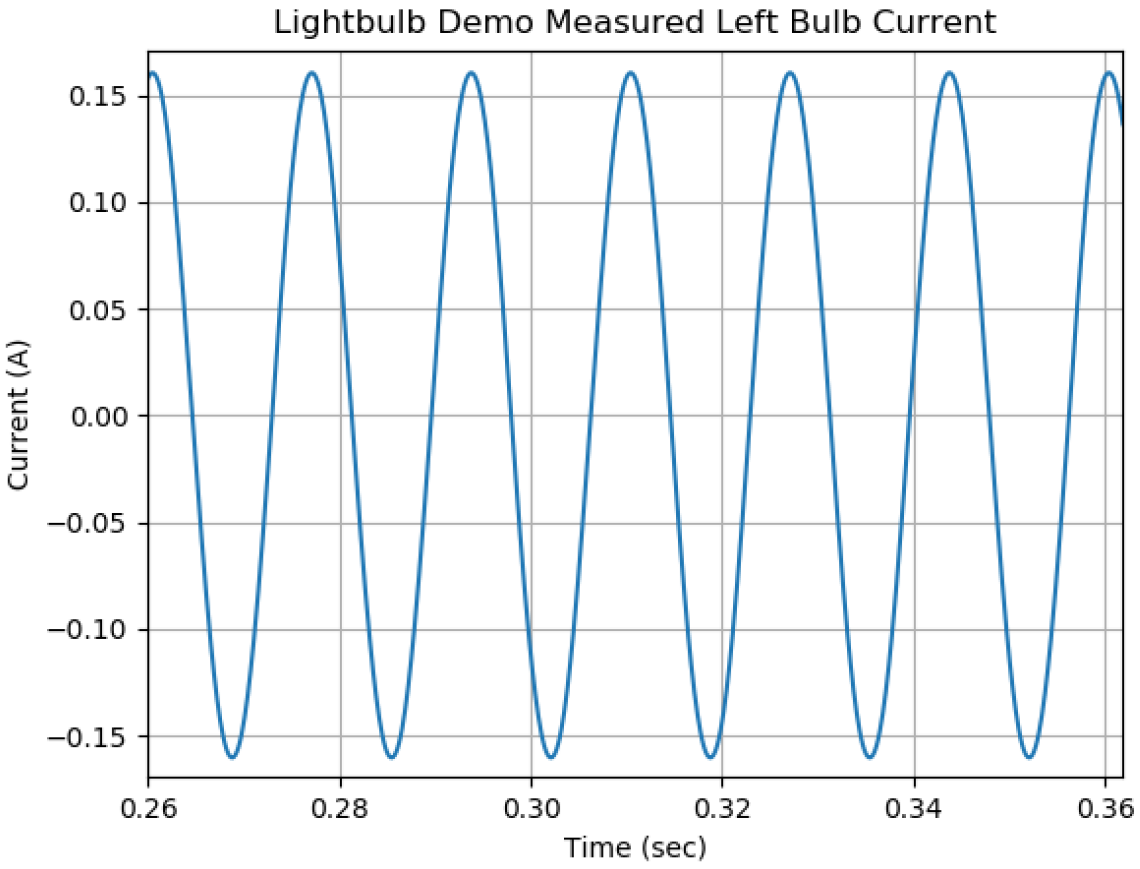

Figure 4-62: The current running through the left light bulb.

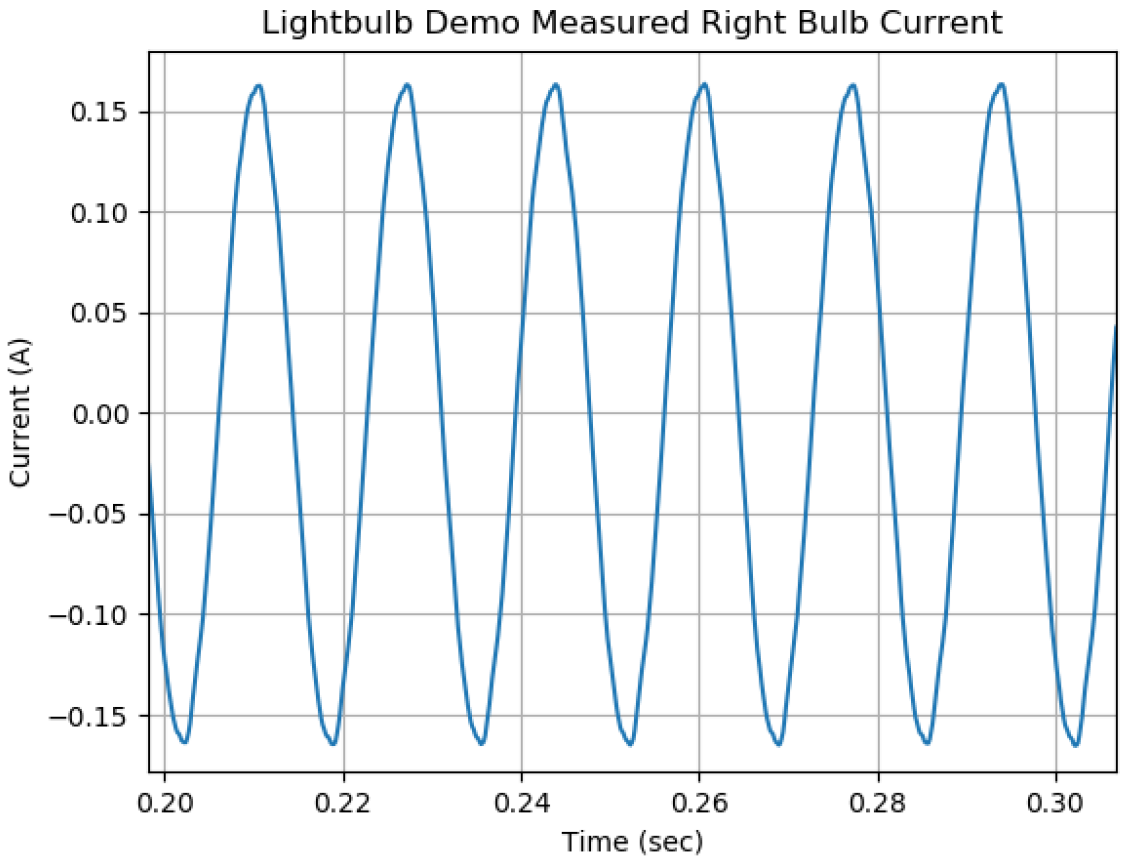

Figure 4-63: The current running through the right light bulb.

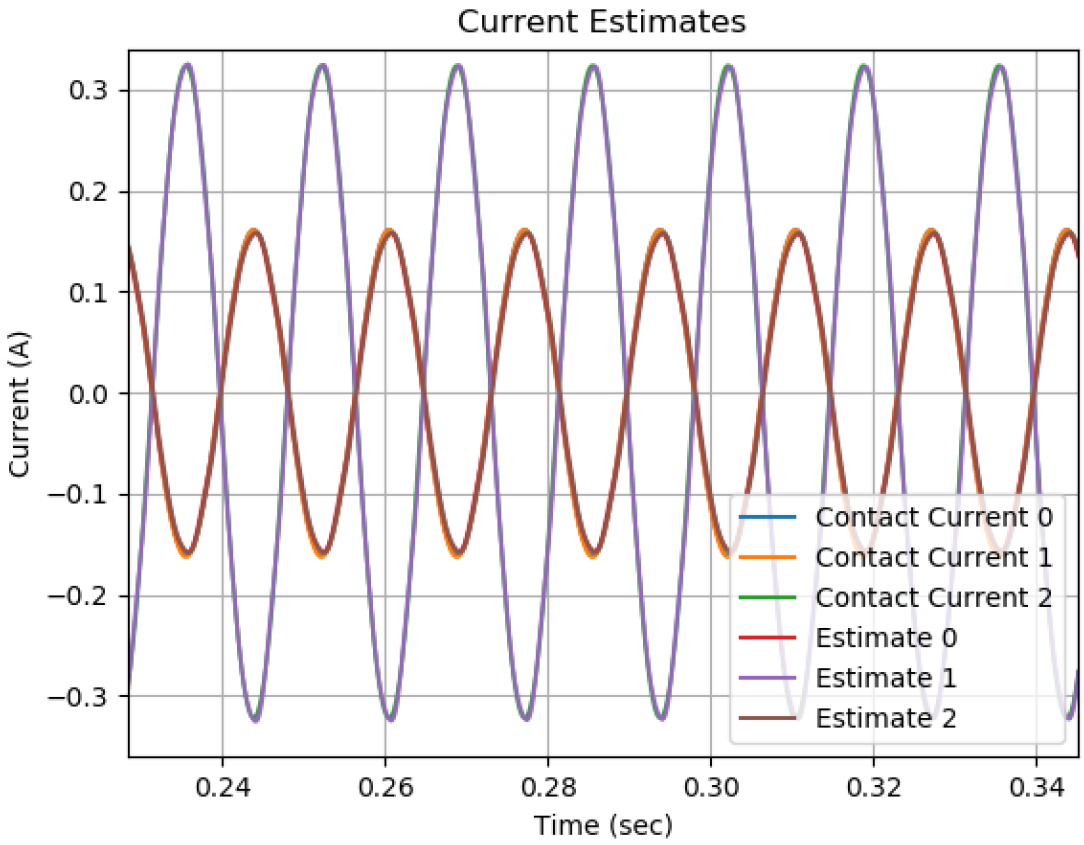

Figure 4-64: The current estimate of currents in the light bulb box.

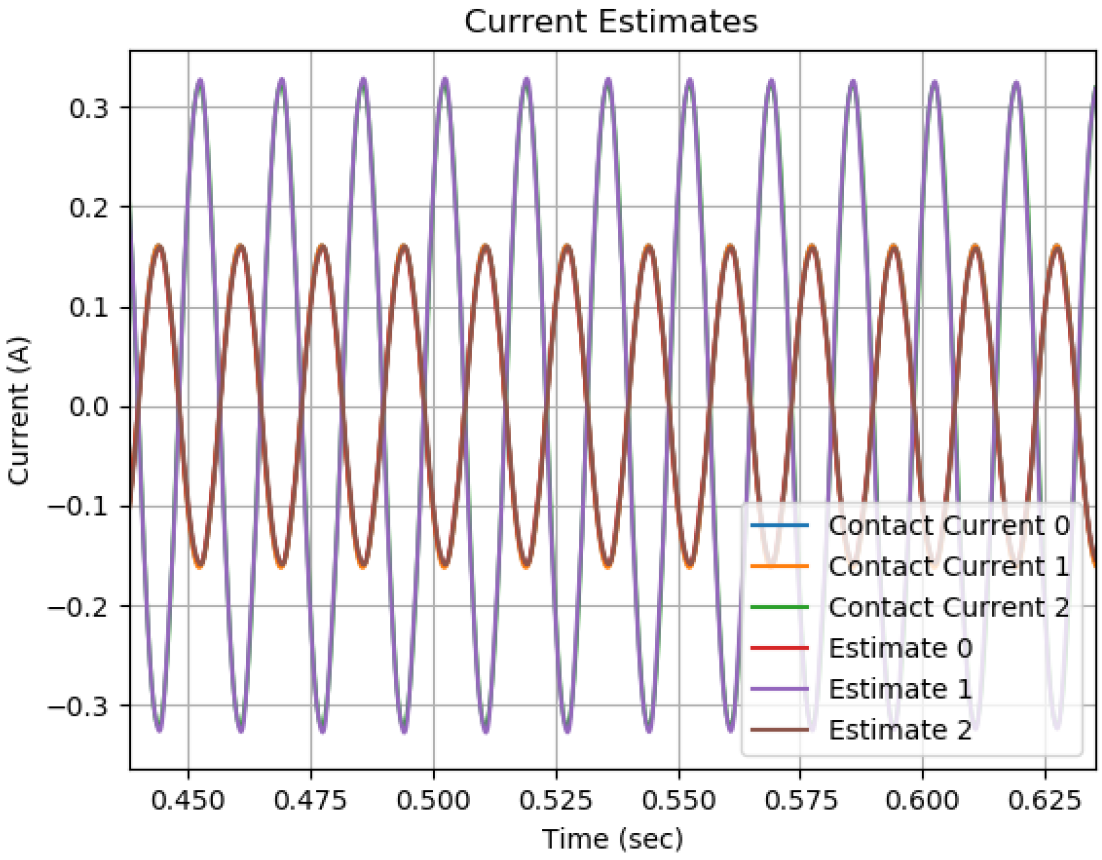

Figure 4-65: Current estimates of the light bulb box in the presence of a pair of external cables.

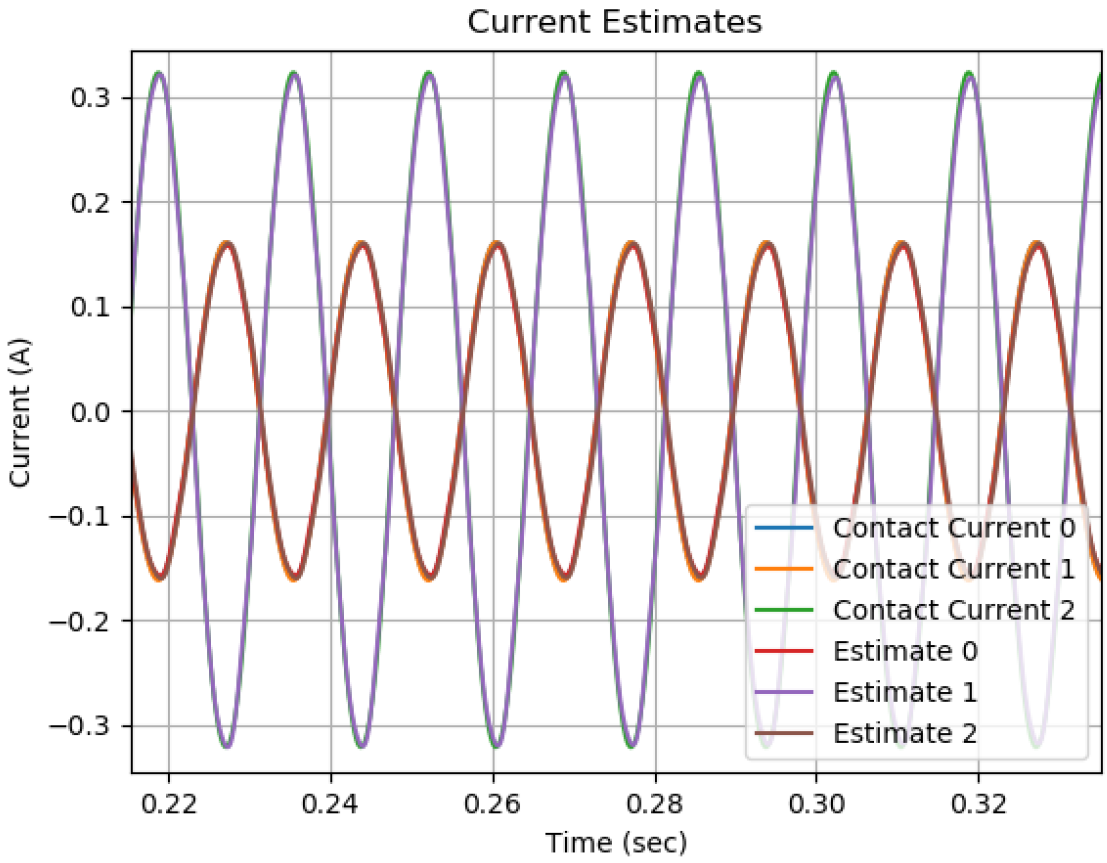

Figure 4-66: Current estimates of the light bulb box in the presence of a plate.

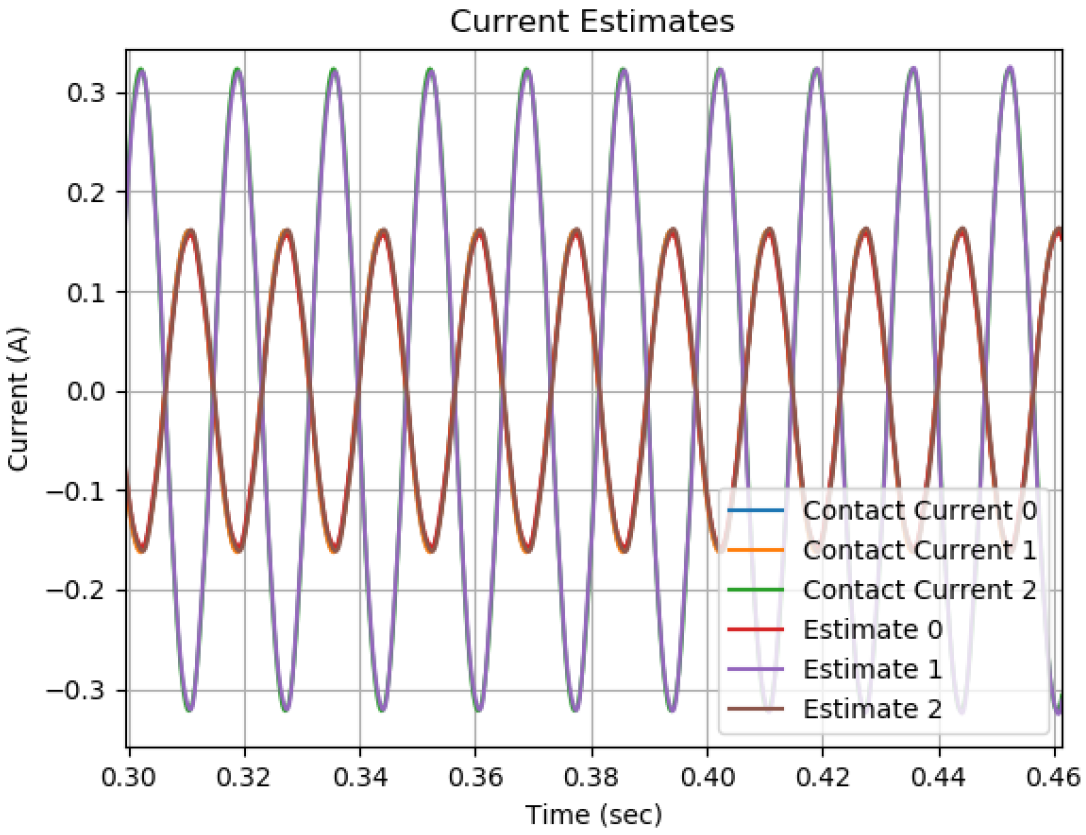

Figure 4-67: Current estimates of the light bulb box in the presence of six external cables.

Chapter 5

Voltage Estimation Methods

To achieve our goal of making contactless estimates of AC cable voltages we built electrodes that capacitively coupled with the cable conductor and digitally processed the electrode voltage. Our goal was to measure line-to-line voltage between adjacent cables, rather than the voltage between each cable and some ground. We had to keep two issues in mind while designing the voltage sensors: 1) ensuring the voltage picked up by the electrodes was large enough to detect and 2) rejecting electrical interference from nearby cables and machines. To achieve the former goal, we had to ensure the effective capacitance between our electrode and the cable was as large as possible. To achieve the latter goal, we used active shields around our sensing electrodes driven by a buffering op-amp.

We experimented with two different implementations of the voltage sensors. Both designs involved the use of electrodes to detect the electric fields between two adjacent cables. The first sensor design used a coaxial cable conductor as the sensing electrode. The second sensor design used a copper tape as the sensing electrode. We will now discuss the first sensor design, followed by a description of the reasons we pursued a different sensor design.

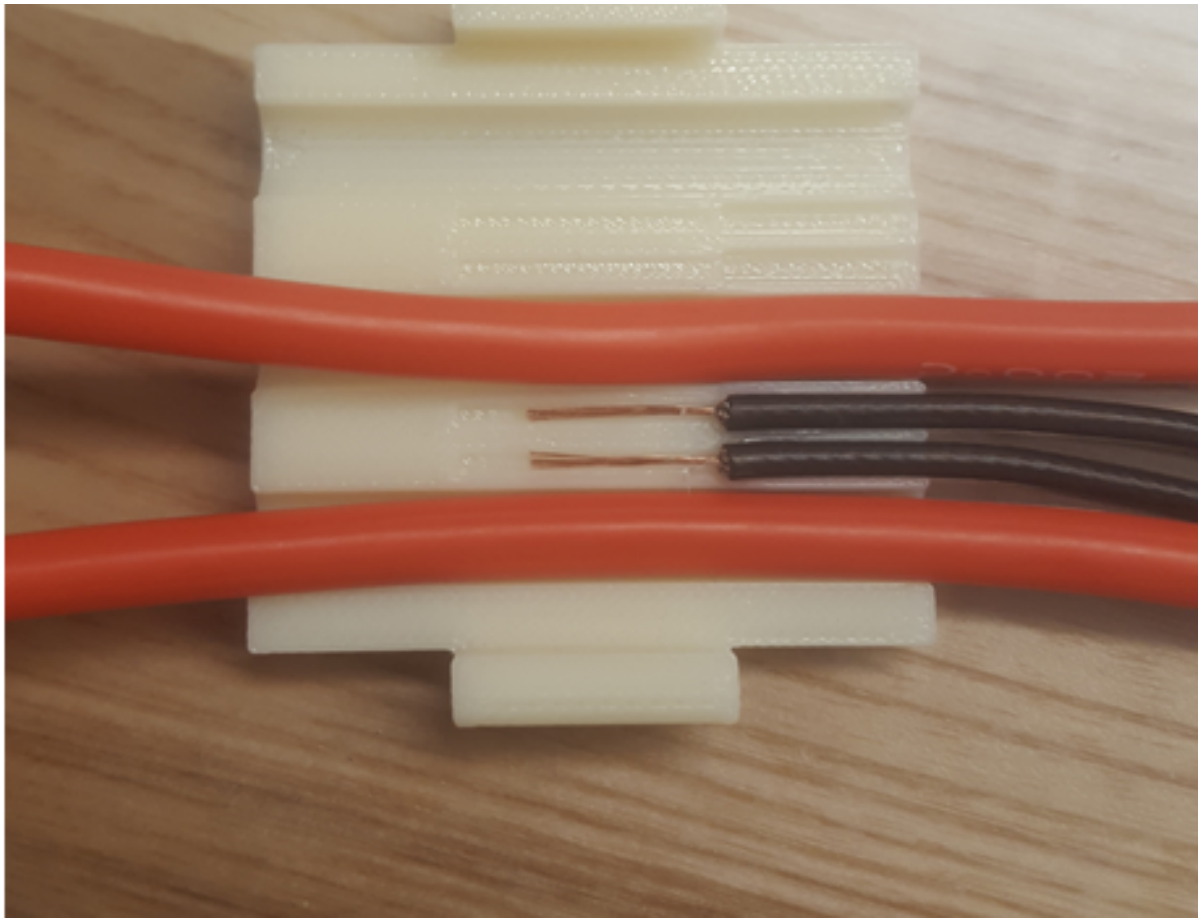

Figure 5-1: A photograph of the first implementation of our voltage detector. The exposed inner conductor of the coaxial cables can be seen. The red cables are the power cables that carried the voltage being estimated.

5.1 Coaxial Cable Electrode Sensor

In the first implementation of the voltage sensor, two coaxial cables were placed in between each pair of power cables in the yoke. The inner conductor of the coaxial cable was exposed for a length of 1 cm. This exposed conductor was meant to serve as the electrode that would pick up the electric field lines produced between the power cables, thereby capacitively coupling with the AC voltage in the cables. The dielectric between the cable conductor and the electrode consisted of both the power cable insulation and the ABS material of the yoke. Figure 5-1 shows a photograph of this implementation.

To protect the electrode voltage from external electric field interference, we used

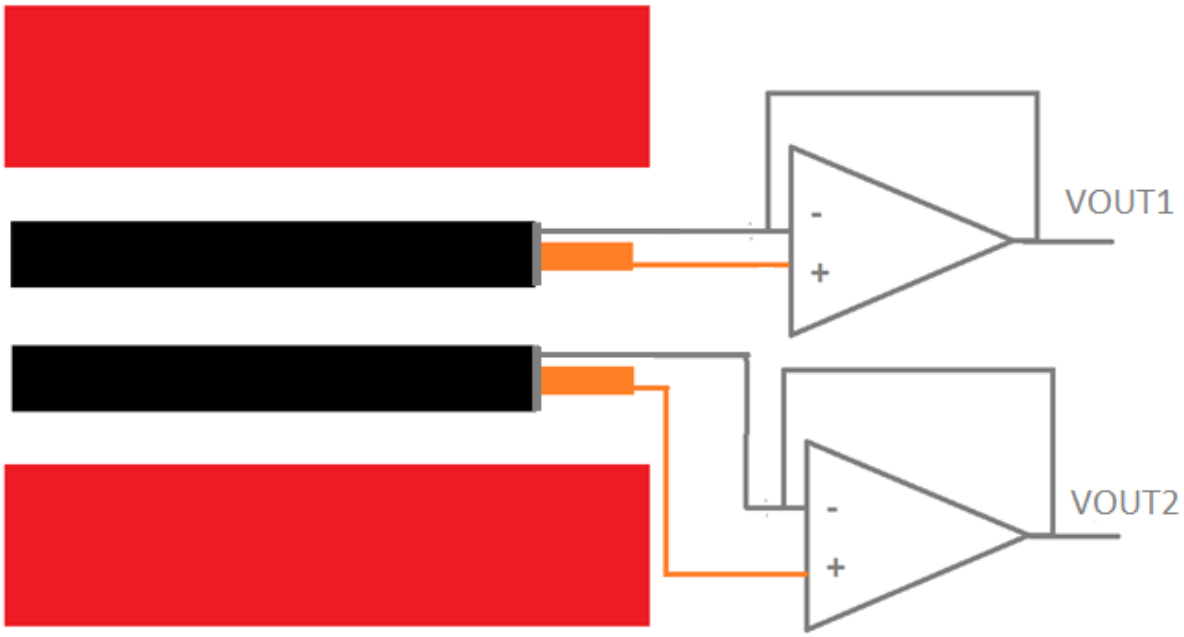

Figure 5-2: A photograph of the first implementation of our voltage detector. The exposed inner conductor of the coaxial cables can be seen. The red cables are the power cables that carried the voltage being estimated.

an op amp to drive the coaxial shield to the same voltage as the electrode. A schematic of this design in shown in Figure 5-2. Thus, with the exception of the exposed 1 cm segment, the shield formed an active guard around the inner conductor protecting it from external interference.

We tested the voltage sensors on the light bulb box described in Chapter 3. The box connected to a wall outlet, providing an opportunity to estimate 120 V RMS 60 Hz voltage and to compare it to the true voltage measured using the resistor divider.

We modelled each electrode and cable as two nodes connected by a capacitor. This meant that the electrode voltage would be the derivative of the cable voltage. Thus, to form our estimate, we formed a running sum of the electrode voltage signal.

The voltage estimate V_{est} was formed by

$$V_{sum}[i] = \sum_{n=0}^{i}(V_{electrode}[n] - avg(V_{electrode})) \tag{5.1}$$

$$V_{est}[i] = \alpha(V_{sum}[i] - avg(V_{sum})) \tag{5.2}$$

The value α was determined by the dielectric properties of the ABS material between the electrodes and cable. Since we had not yet created a calibration mechanism to determine this capacitance, we manually selected α to make the amplitude of the true voltage match the amplitude of the estimated voltage.

Initially, this procedure generated an estimate with low-frequency components that did not exist in the true voltage waveform. The estimate exhibited an enveloping effect when plotted as a function of time. However, after filtering components in the estimate below 30 Hz, the estimate resembled the true voltage reading. Figure 5-3 shows the estimated waveform superimposed over the true voltage.

Nevertheless, the voltage sensor had a major problem: the amplitude of the electrode voltage decreased with time, even though the amplitude of the true voltage was constant. Figure 5-4 shows a plot of the amplitude of the electrode voltage as a function of time. We ensured the cables fit snugly inside the yoke to rule out mechanical drift as a potential cause. We believe this problem occurred because the humidity and temperature of the ABS yoke material changed with time, causing a change in the properties of the dielectric between the electrode and cable conductor, and thus changing the capacitance of the sensor. Since the issue was due to the design of our electric field sensor, we decided to create a different sensing method.

5.2 Copper Tape Electrode Sensor

The second voltage sensor we designed used a rectangular piece of copper tape as the sensing electrode. This copper tape was taped onto the cable channels in the yoke so that when the yoke was clipped around a set of cables, each copper tape would be in direct contact with the cable insulation. An illustration is shown in Figure 5-5.

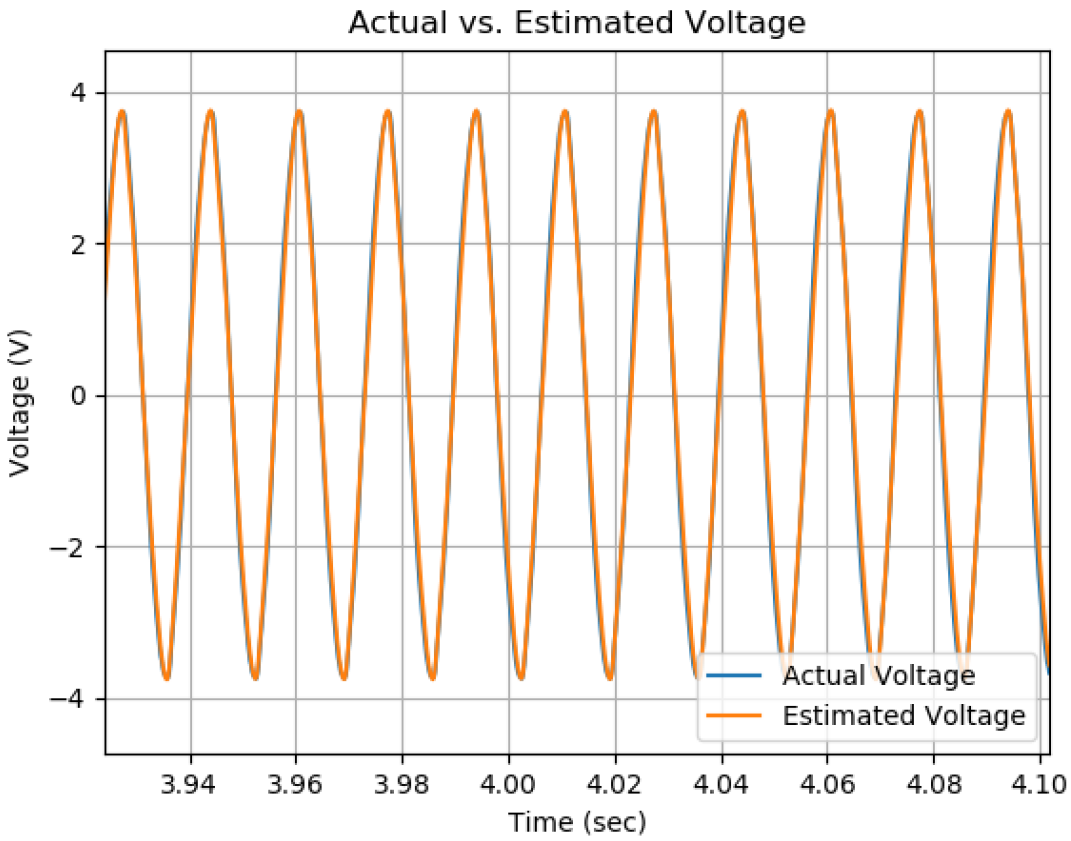

Figure 5-3: A plot of estimated and actual voltage in the light bulb box as detected by the coaxial cable electrodes. The estimated waveform shape lines up well with the actual waveform shape.

Thus, we eliminated the use of ABS plastic as a dielectric material between the cable and electrode in order to overcome the temperature and humidity issue that affected our previous design.

To shield the electrode from external electric fields, we also included a shielding piece of copper tape that is driven to the same potential as the sensing electrode by an op-amp. A piece of kapton tape separates the sensing electrode and the shielding electrode. We used a TLV2371 op-amp because of its high input impedance, which was desirable since the impedance of the capacitance between the cable and electrode was also very high. To provide a reference to the ground of our sensing circuit, we included a bypass resistor between the sensing electrode and ground. We experimented with different resistor values, including 470 $K\Omega$, 1 $M\Omega$, and 10 $M\Omega$. This setup is

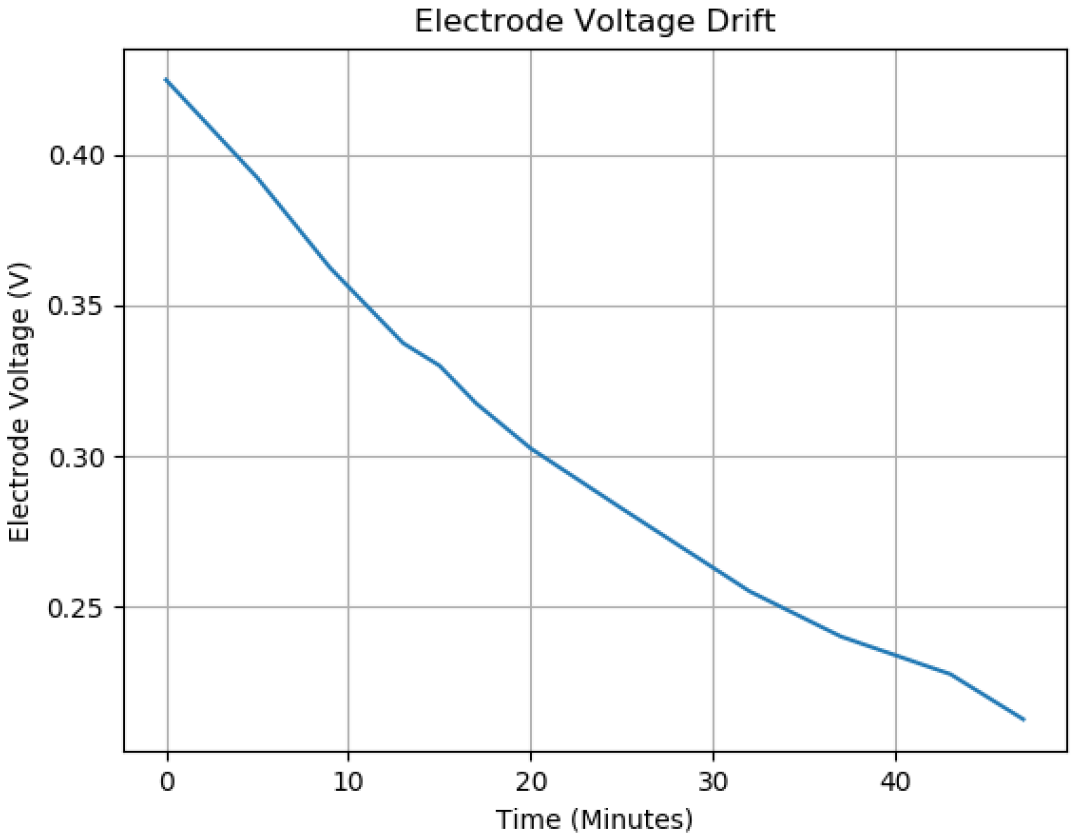

Figure 5-4: A graph showing the change in the line-to-line differential electrode voltage magnitude as a function of time.

illustrated in Figure 5-6.

Since the ground of our sensing system is not necessarily the same as the ground of the three phase system being measured, we did not estimate absolute voltage values for each cable. Rather, we estimated the voltage differences between adjacent pairs of cables. If we refer to the voltages in the three cables as $V_{cable,0}$, $V_{cable1,}$ and $V_{cable,2}$, then the two voltages we are estimating are $V_{cable,0} - V_{cable,1}$ and $V_{cable,1} - V_{cable,2}$.

Lumped Parameter Model

To estimate cable voltage it is necessary to understand the physical relationship between the cable and electrode. For this purpose, we developed a lumped parameter model from which we could derive a transfer function to form the voltage difference estimate.

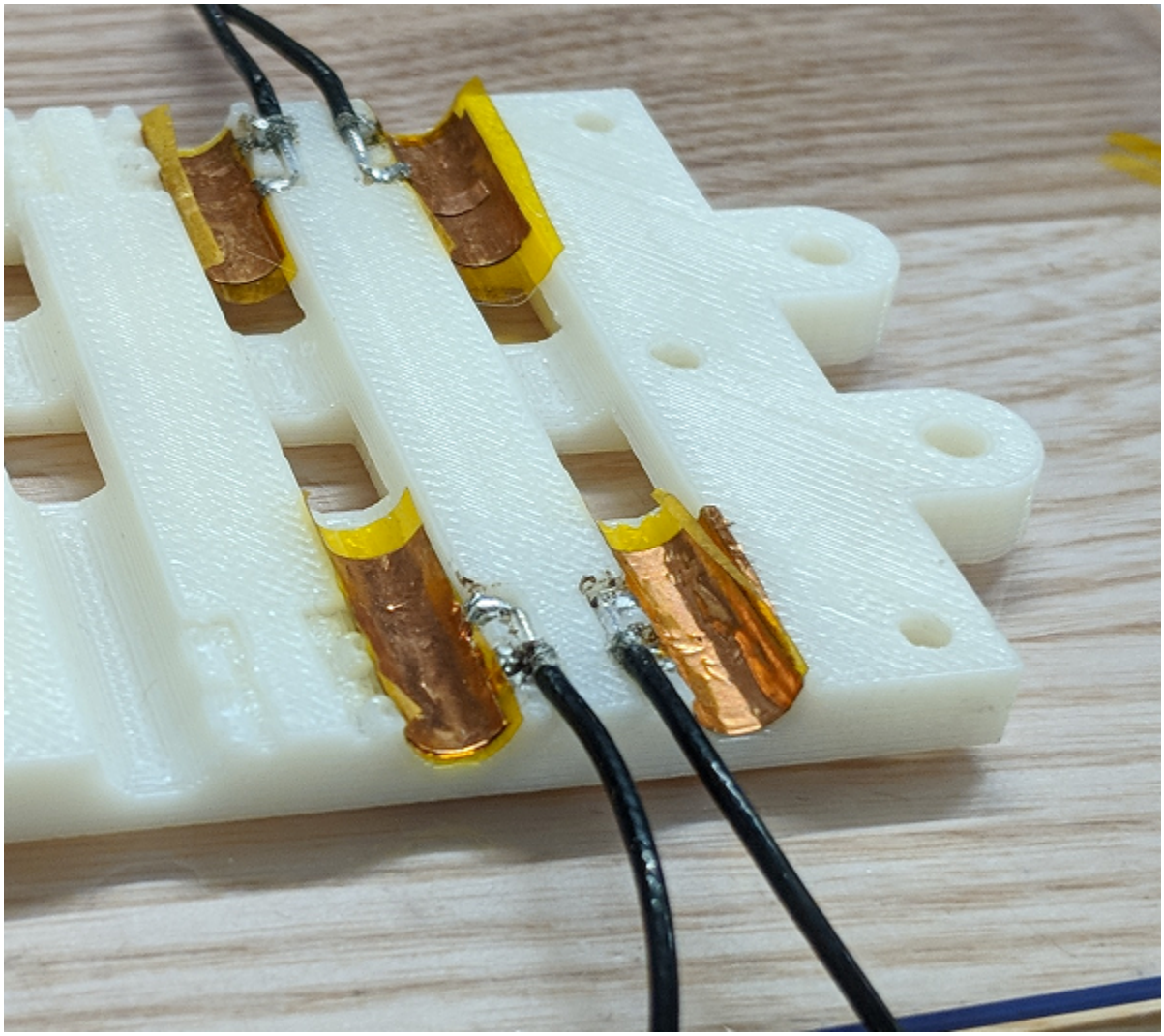

Figure 5-5: The electrode consisted of copper tapes taped along each cable channel of the yoke. Beneath each copper tape is a piece of kapton tape, followed by a second copper tape that served as the active shield.

Since the cable insulation serves as a dielectric separating the cable conductor and the copper tape electrode, it can be modeled as a capacitor. The capacitor value depends on the dielectric properties of the cable being used by the operator, which we do not know beforehand. Therefore, it is necessary for our system to be calibrated. This can be done by the operator by manually entering the correct voltage value for a certain point in time. The capacitance can be solved using this value and the system will then be able to estimate any voltage difference carried by the cable. However, we also developed a method of automatically determining the capacitance by contactless

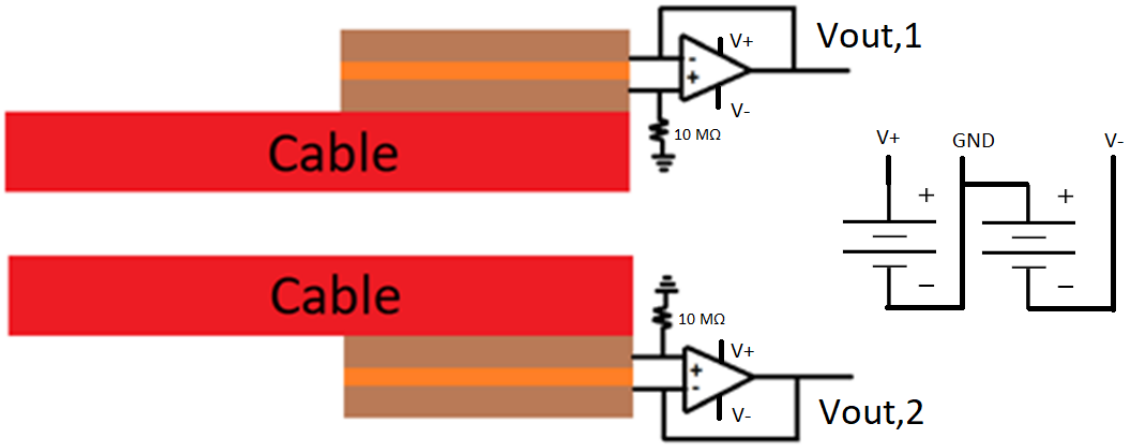

Figure 5-6: A schematic showing how an op-amp was used to drive the shield to the same potential as the sensing electrode. Signals $V_{Out,1}$ and $V_{Out,2}$ are read by the ADC. The op-amps are powered by a battery to keep the detector ground separate from the ground of the system being measured. This prevented the calibration signal, discussed in the section on Automatic Calibration, from being shunted. The detector, battery, and ADC all had the same ground.

injection of our own signal into the power cable. This method will be described in Section 5.3.

The capacitance between the cable conductor and electrode also depends on the dimension of the electrode. The larger the electrode is, the greater the capacitance and the larger the signal we will receive from the cable. The width of the electrode is limited by the cable diameter and the length of the electrode is limited by the size of our detector. Although the detector is 6 cm in length, we also included an electrode on the opposite end of the yoke meant for calibration, as we will describe in the Section 5.3. The detecting and calibrating electrodes had to be kept as far apart from each other as possible to avoid cross-capacitance. Thus, the electrodes in our prototype were 1.5 cm long, and the calibrating and detecting electrodes were at a distance of 3.0 cm from each other.

If the electric field lines between the cable conductor and electrode were straight, we could calculate the electrode capacitance using the capacitance formula. For example, with the 8 AWG cable used in our experiments, we can approximate the cable diameter to be 6 mm, the insulation thickness to be 1 mm, and the insulation

Table 5.1: Capacitance between different components of the voltage detector as measured by an impedance analyzer at 1000 Hz.

Components	**Capacitance**
Cable to Electrode	5.4 pF
Electrode to Electrode	0.4 pF
Cable to Cable	0.2 pF

dielectric constant to be 3.2 (since it is made of silicone). The theoretical capacitance is then

$$C = \frac{\epsilon_r \epsilon_0 A}{d} = \frac{(3.2)(8.854\ \frac{pF}{m})(3\ cm)\pi(1.5\ cm)}{1\ mm} \approx 4.0\ pF \tag{5.3}$$

However, even if we were to measure the dimensions of the copper tape electrode, the true capacitance would differ slightly from the value given by (5.3), not only because the dimensions are just estimates, but also because of fringing electric field lines at the edge of the electrode and curved field lines between the electrode and the semicircular area of the cable not enclosed by the electrode.

To further understand the capacitance values between the components of our voltage estimation system, we measured the capacitance between the electrode and cable, between adjacent electrodes, and between adjacent cables using an impedance analyzer. Ideally, we wanted the capacitance between the cable and electrode to be much larger than the other capacitances, to obtain a meaningful signal from our system. The capacitances are shown in the Table 5.1. The capacitance between the electrode and cable was 13.5 larger than the capacitance between neighboring electrodes and about 25 times larger than the capacitance between adjacent cables.

The true capacitance between the cable and electrode when the system is running is different than the one measured by the impedance analyzer, since we did not have the op-amps powered during the above analysis, and the introduction of live voltages into the PCB board would somewhat change the configuration of the electric field lines.

We now turned to the task of modelling the voltage detection system consisting

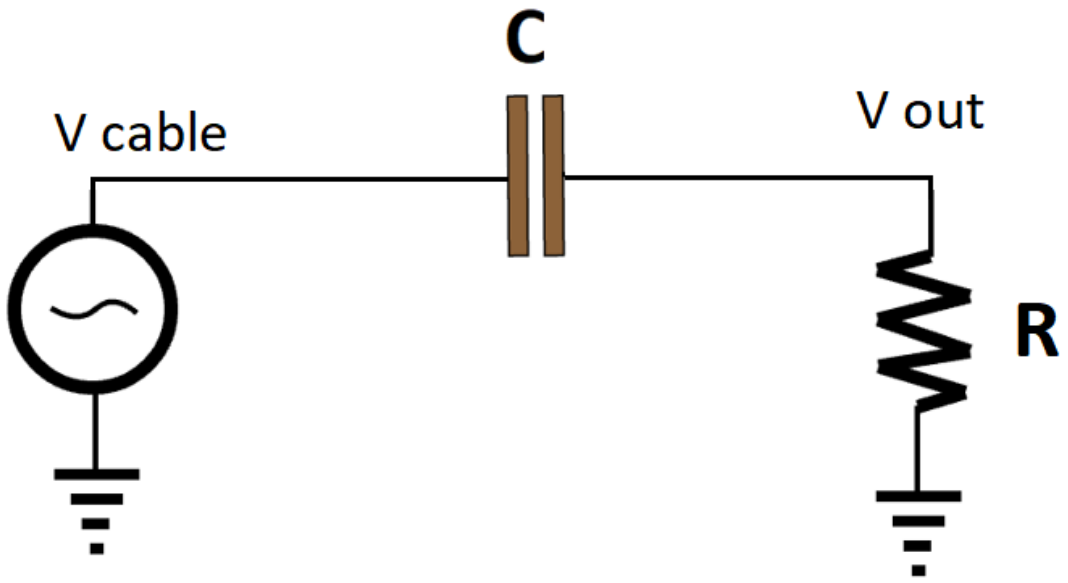

Figure 5-7: A model of the voltage detection system if the op-amp were ideal. The resistor represents the bypass resistor whose value we chose, and the capacitor models the dielectric effect between the cable voltage and the electrode voltage.

of the electrode, the bypass capacitor, and the op-amp driving the active shield. Assuming the op-amp is ideal, the system could be modeled as a high-pass RC filter in which there is a capacitor between the cable voltage and electrode voltage and a resistor between electrode voltage and ground, illustrated in Figure 5-7. The transfer function of this model is

$$\frac{V_{out}}{V_{cable}} = H_e(j\omega) = \frac{j\omega RC}{1 + j\omega RC} \tag{5.4}$$

where V_{cable} is the cable voltage, V_{out} is the electrode voltage that is read by the ADC, and H_e is the ratio between these two values.

(5.4) indicates that at low frequencies, the electrode voltage increases linearly with frequency with a slope of RC, while at high frequencies, the electrode voltage would be equal to the cable voltage. However, we found that the latter behavior did not match the results of our experiments. When we applied high-frequency voltages to the power cables, the electrode voltage did not match the amplitude of the cable voltage, but instead settled to an amplitude slightly over half of the cable voltage.

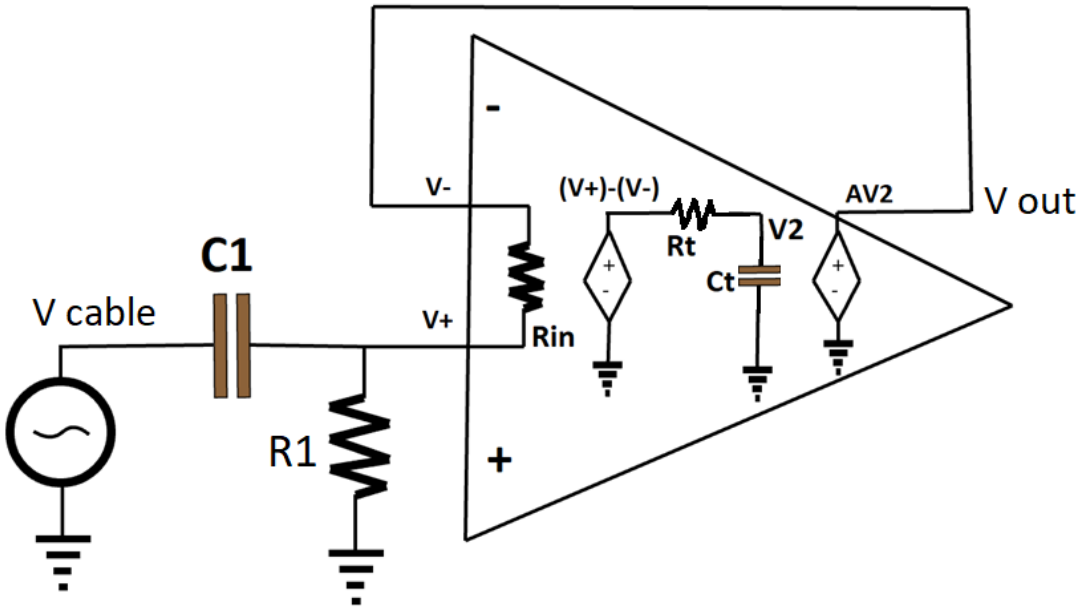

Figure 5-8: A lumped parameter model including the properties of the op amp. Specifically, the input resistance, gain, and cutoff frequency of the op-amp were modelled.

The measurements we collected are graphed in Figure 5-9.

Thus, we decided the op-amp was not behaving ideally. This was a reasonable assumption, since the impedance of the electrode, in the range of 4 pF, and the impedance of the bypass resistor, ranging from 470 $K\Omega$ to 1 $M\Omega$ for certain experiments, were very high. These large impedances meant that the input impedance, gain, and cutoff frequency of the op-amp could no longer be neglected, but rather had to be included in our model.

We created a lumped parameter model for our voltage detection system using the basic op-amp model, illustrated in Figure 5-8. This model involves an input resistance R_{IN}, a low-pass filter modeled by a resistor R_τ and capacitor C_τ, and a gain A. Solving for V_{Out} as a function of V_{Cable}, we derived the transfer function given

by

$$H_e(j\omega) = \frac{j\omega A R_1 R_{IN} C_1}{R_{IN} + R_1 + R_{IN} A\tau + j\omega(\tau(R_1 + R_{IN}) + R_1 R_{IN} C_1(A+1)) + (j\omega)^2 R_1 R_{IN} C_1} \tag{5.5}$$

where τ is equal to $C_\tau R_\tau$. Note that in an ideal op-amp, $A = \infty$, $R_{IN} = \infty$, and $\tau = 0$. Under these conditions, the transfer function above simplifies to the transfer function found in (5.4).

To prepare this function for curve fitting experiments, we isolated independent variables by dividing the denominator by the constants in the numerator, and making the approximation $(A+1) \approx A$, since A is typically around 100,000. The transfer function is given by

$$H_e(j\omega) = \frac{j\omega}{\frac{1}{R_1 C_1} + j\omega(\frac{(R_{IN}+R_1)\tau}{A R_1 R_{IN} C_1} + 1) + (j\omega)^2 \frac{\tau}{A}} \tag{5.6}$$

$$= \frac{j\omega}{a + j\omega b + (j\omega)^2 c} \tag{5.7}$$

The transfer function in (5.4) and (5.5) is in a form that can be curve-fit to solve for three parameters. In practice, however, we could not reliably excite the $(j\omega)^2$ term. Although applying voltages beyond 80,000 Hz caused the output voltage amplitudes to begin decreasing, the output voltage became triangular at these frequencies, indicating we were exceeding the slew rate of the op-amp. Thus, we did not consider the amplitude of these readings worthy of using for curve-fitting purposes. However, since the value of the resistor was known, for the purpose of solving the capacitance of the electrode C_1 it was sufficient to identify the parameter a. Solving the parameter b provided insight into the value of $\frac{(R_{IN}+R_1)\tau}{A R_1 R_{IN} C_1}$, which would become more useful for the calibration procedure described in the next section.

To curve-fit the parameters of the transfer function, we applied a range of voltage frequencies to the power cables with voltage detector attached. The voltages ranged from 150 Hz to 40,000 Hz. We then curve-fit the magnitude of the gain between electrode and cable voltages $\frac{|V_{out}|}{|V_{cable}|}$ to the magnitude of the transfer function using

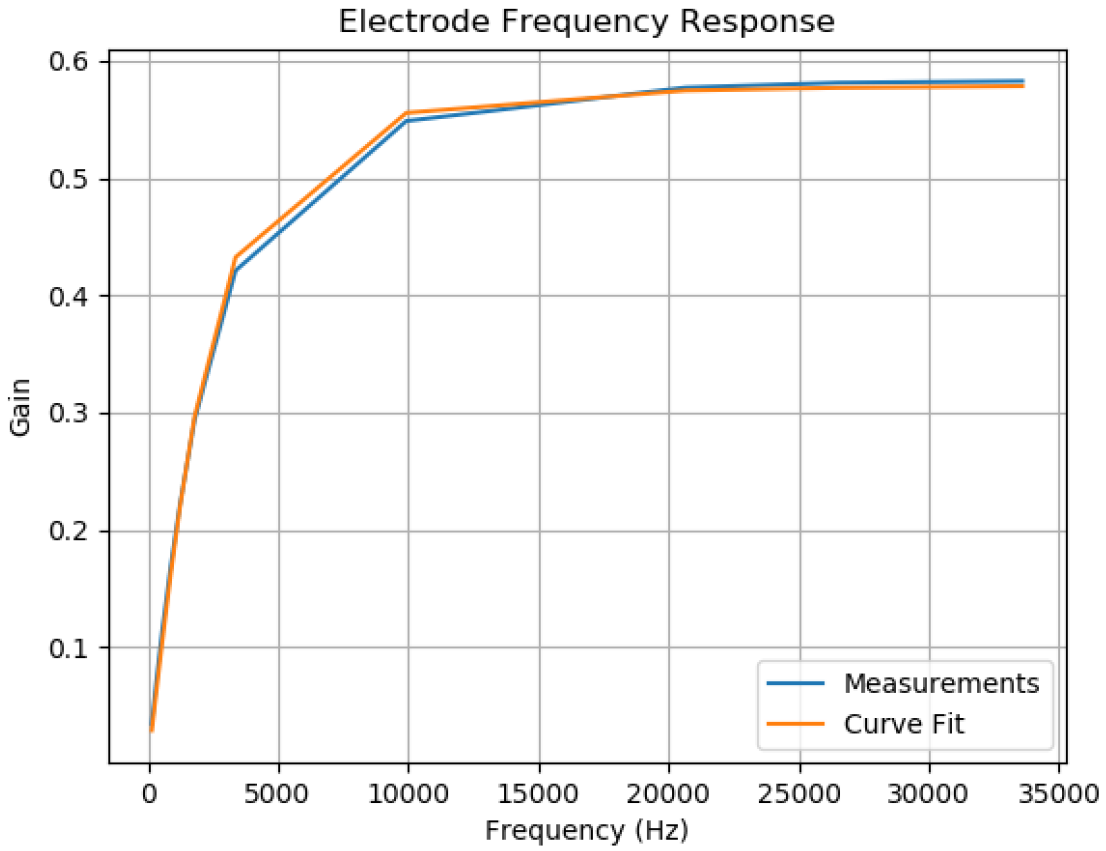

Figure 5-9: The results of curve fitting the readings to a model that includes op-amp properties. The measurements were collected using the USB-231 ADC.

the scipy.optimize.curve_fit function. The full script that performs this optimization is named voltage_freq_response_fit.py and is found in Appendix A.

This transfer function was a good fit for the data we collected. The estimated function is plotted against the data points in Figure 5-9. Parameter a was found to be 38101 s^{-1}, which implied a C_1 value of 2.6 pF, which is a reasonable value for our system setup. Parameter b was found to be 1.72.

The shape of the frequency curve can be adjusted by choosing the value of the bypass resistor. A larger bypass resistor will lead to larger signals at low frequencies, since the slope ωRC will be larger, but will also cause the corner frequency of the curve to move to a lower frequency. This will make the low-frequency portion of the curve less linear.

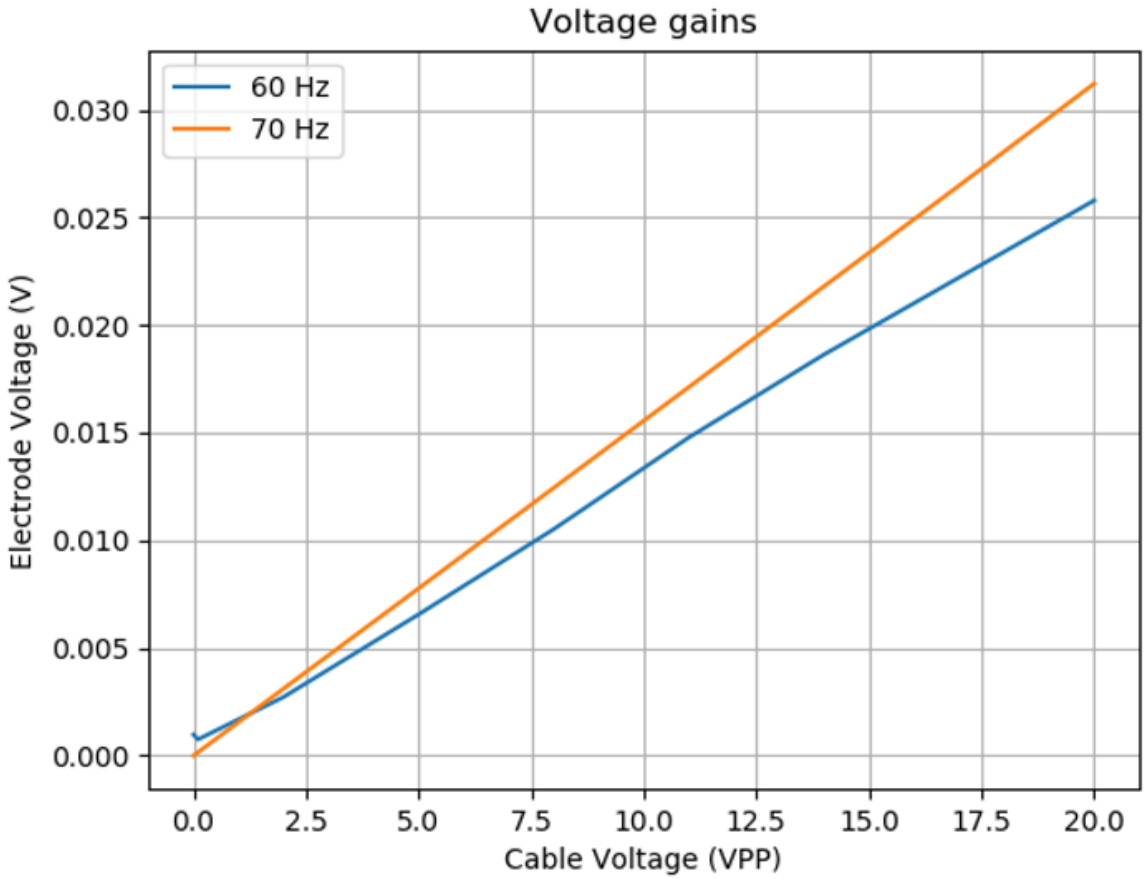

Figure 5-10: A graph showing the sensing electrode amplitude as a function of the cable voltage amplitude.

After experimenting with different resistor values, we selected a resistor of 10 $M\Omega$ for our system. One reason for this choice was to create a large electrode signal at 60 Hz. For example, a 400 V cable signal at 60 Hz would create an electrode signal of $(60\text{ Hz})(2\pi)(10^7\ \Omega)(3\text{ pF}) = 4.52$ V, comfortably under the 6 V limit of the buffering op-amps. In addition, moving the corner frequency of the frequency response curve to a frequency that could be more easily measured by our hardware was important for our automatic calibration mechanism, which will be discussed in Section 5.3.

The electrode voltage magnitude was a linear function of the cable voltage magnitude for both configurations of the voltage detector. Figure 5-10 shows a plot of this relationship for a range of voltage magnitudes applied at two different frequencies.

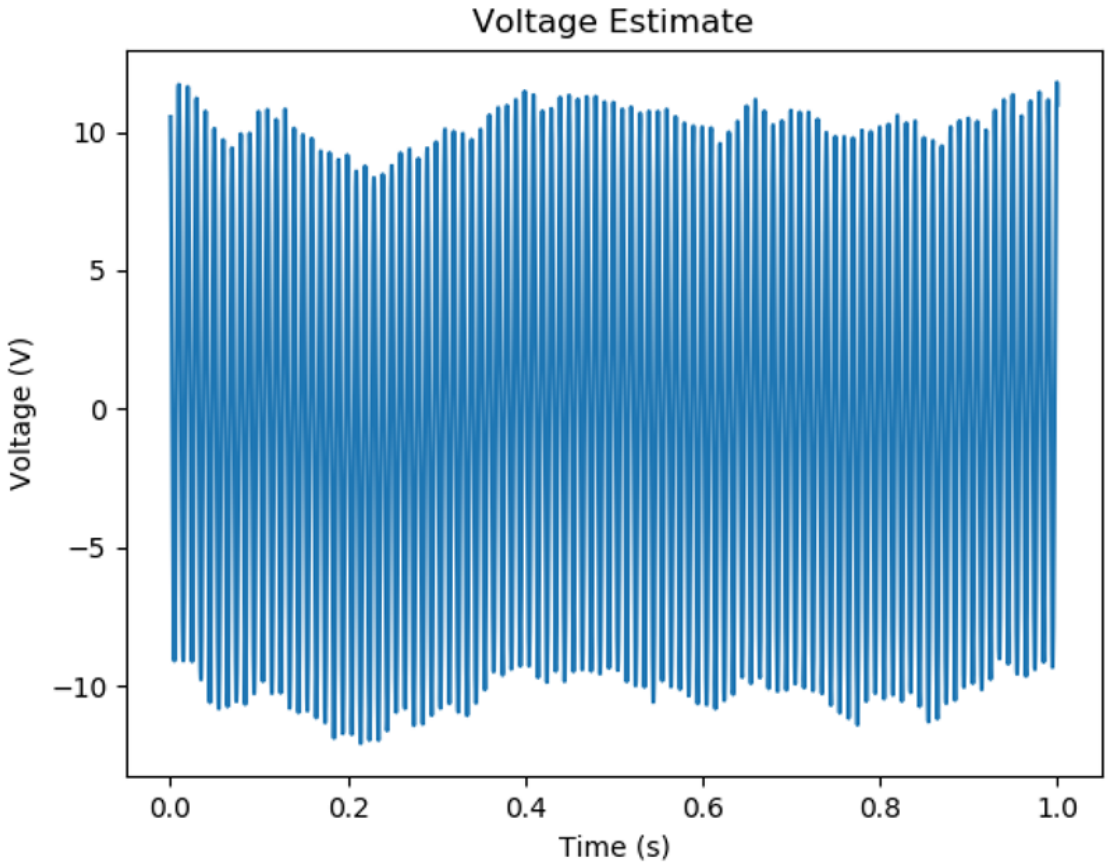

Figure 5-11: The noisy voltage estimate.

The data was collected using the detector with a 1 $M\Omega$ bypass resistor.

Voltage Estimation and Noise Explosion

To estimate cable voltage from electrode voltage, we apply the inverse transfer function to the electrode voltage, given by

$$V_{cable} = V_{out} \frac{\frac{1}{R_1 C_1} + j\omega(\frac{(R_{IN}+R_1)\tau}{AR_1 R_{IN} C_1} + 1)}{j\omega} \tag{5.8}$$

When estimating cable voltage, it was important to first filter sensor noise, since the inverse transfer function above can magnify low-frequency noise. For example, Figure 5-11 shows a voltage estimate in an experiment where a clean 20 VPP 70 Hz was applied to a cable. Despite the true signal being clean, the estimate is wavy.

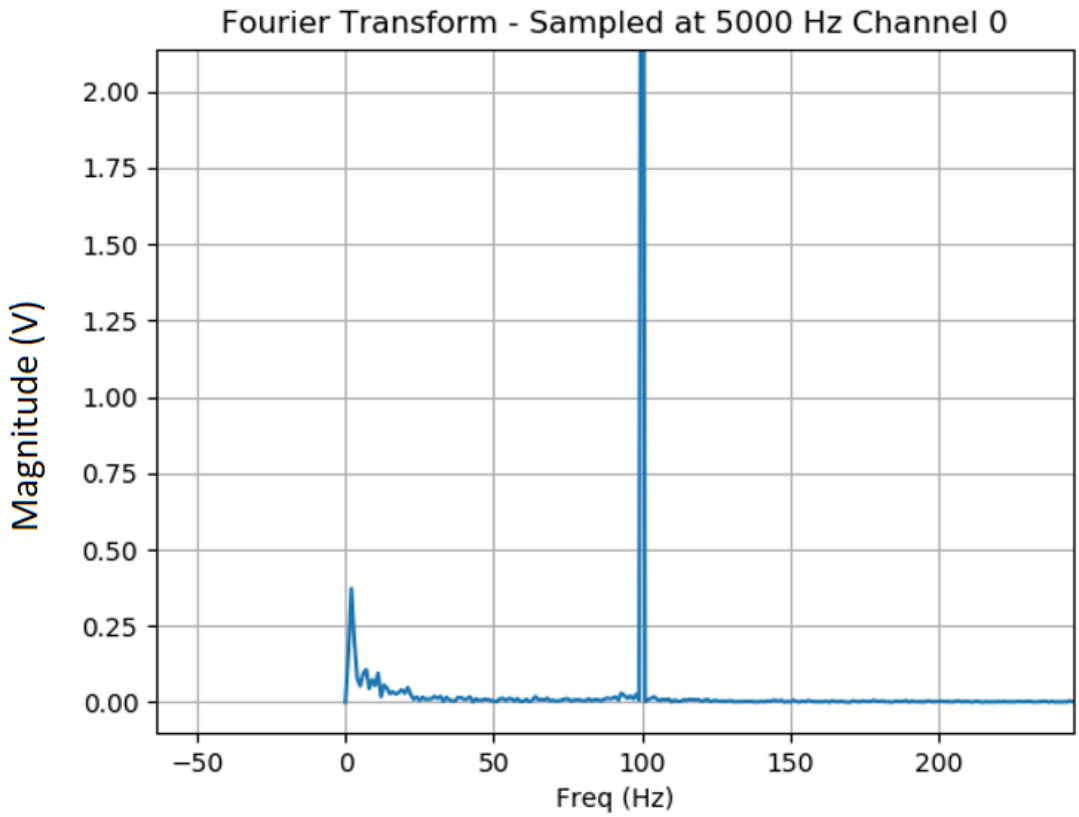

Figure 5-12: Fourier Transform showing large low-frequency voltage estimate components.

Figure 5-12 shows the Fourier Transform of the estimate, which reveals the problem: the inverse transfer function caused the low-frequency components of the white noise to take on large values in the estimate. Applying the custom noise filter described in the Section 4.7.1, however, eliminates this problem and produced an estimate that looked like a clean sinusoidal signal. A graph of two such signals can be seen in Figure 5-17.

Active Shielding

To test the effectiveness of the active shielding, we applied a 20 VPP 70 Hz signal to the cables while introducing different forms of interference. The forms of interference we experimented with were an aluminum plate 1 cm above the sensor, a human hand

Table 5.2: Comparison of electrode voltage change in the presence of interference with and without active shielding.

Case	No Shielding (% Error)	With Shielding (% Error)
No Interference	0.1087 V	0.0463 V
Aluminum Plate	0.1113 V (2.4%)	0.0461 V (0.4%)
Iron Plate	0.1111 V (2.2%)	0.0461 V (0.4%)
Hand	0.1121 V (3.1%)	0.0462 V (0.2%)

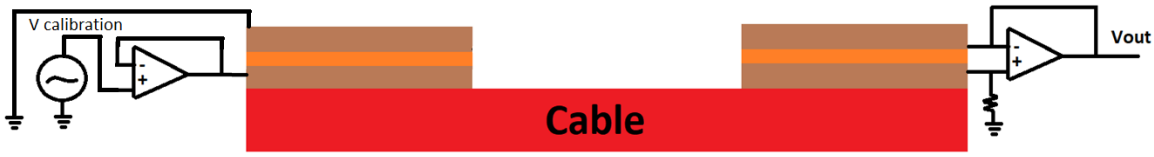

Figure 5-13: To calibrate the system, we apply a known signal to the calibrating electrode, represented on the left side of the figure, which capacitively injects a voltage into the cable that then creates a voltage in the sensing electrode.

1 cm above the sensor, and a round iron plate 1 cm above the sensor. Table 5.2 shows the values of the change in electrode voltage magnitude when sources of interference were introduced when the active shield was driven by the op-amp and in the case when it was not. The results show that the active shielding reduces the effect of external interference by more than a factor of 5.

5.3 Automatic Calibration

As previously mentioned, the capacitance between the sensing electrode and the cable is unknown beforehand because it depends on the dielectric properties and the exact dimensions of the cable insulation. We have designed a method to determine the electrode capacitance automatically, without requiring the operator to input the correct voltage value at any point in time.

To do this, we include a second electrode, which we will call the calibration electrode, in each yoke cable channel on the opposite end of the sensing electrode. The calibration electrode and the sensing electrode are equal in width and length, but are 3 cm apart from each other, and thus have minimal cross-capacitance. The calibra-

tion electrode is protected by a shielding electrode driven to ground. The calibration electrodes can be seen on a yoke in Figure 5-5.

A known voltage is applied to the calibration electrode through a buffering op-amp. The calibration voltage induces a voltage in the power cable through the calibrating electrode capacitance. The voltage in the power cable then induces a voltage in the sensing electrode through the sensing electrode capacitance. If these two capacitance values are equal, the system can be modeled as a circuit in which there are two capacitors in series between the calibration voltage and the sensing electrode voltage with the same capacitance C_1. Once the transfer function between the calibration voltage and the electrode voltage is known, we can solve for the electrode capacitance. Figure 5-13 shows a schematic of the automatic calibration system.

We invented this method in May 2018. We have recently found a method very similar to this described in [15]. However, the paper simply mentions that this is a potential technique they would like to develop, and no results have been presented as to its development. We believe we are the first team to present results with this method.

The illustration in Figure 5-13 is oversimplified to only show one power cable. In actuality, there are three power cables connected through unknown impedances with possible capacitances between them. Figure 5-14 shows a more complete model with the unknown impedances between two power cables. To avoid having to solve for these impedances during calibration, we apply the same calibration voltage to all three power cables. Since the cables are at the same potential at the calibration frequency, the impedances between them will not factor into the transfer function between the calibration voltage and the sensing electrode voltage.

To perform calibration, we used a pair of 6 V batteries to supply power to the detector system, rather than using a power supply connected to the wall. This is because when the detector is plugged into wall power, the unknown impedance between the cable and the ground of the detector is too low and provides a shunting path for the calibration voltage that will redirect the calibration signal away from the sensing electrode. This shunting path is shown in orange dashed lines in Figure 5-14.

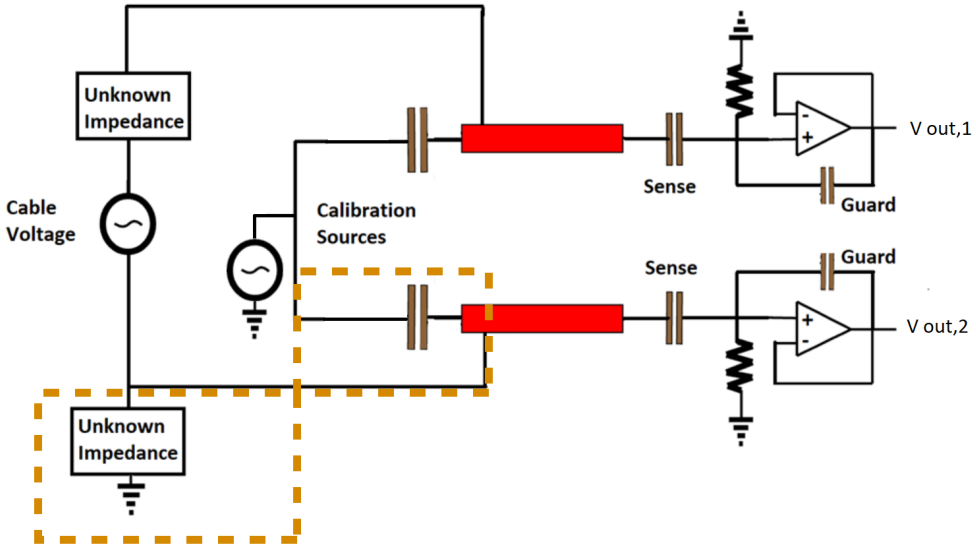

Figure 5-14: A model showing the full system model of a pair of power cables, the sensing electrodes, the op-amps driving the active shield, and the calibrating electrodes. If the impedance between the ground of the power cables and the ground of the sensing system is low enough, a shunting path will exist, shown in orange dashed lines, that will greatly reduce the output voltage created by the calibration signal.

Using a pair of batteries to power the detector system, and therefore keeping the detector system ground and the power cable grounds electrically isolated, prevents this shunting path from existing.

However, even when using batteries to power the detector system, we observed that the calibration at the sensing electrode was different depending on whether the power cables were physically disconnected or connected to the power supply. Additionally, we noticed that the calibration signal detected by the sensing electrodes changed depending on the position of the cables and the physical size of the load that was attached to the cables. This led us to conclude that there was another significant component to the model of the calibration system: a parasitic capacitance between the power cables and the ground of the detector system. This capacitance exists due to the electric field lines emanating through the air and connecting the power system and the detector system. Although this capacitance would normally be negligible in

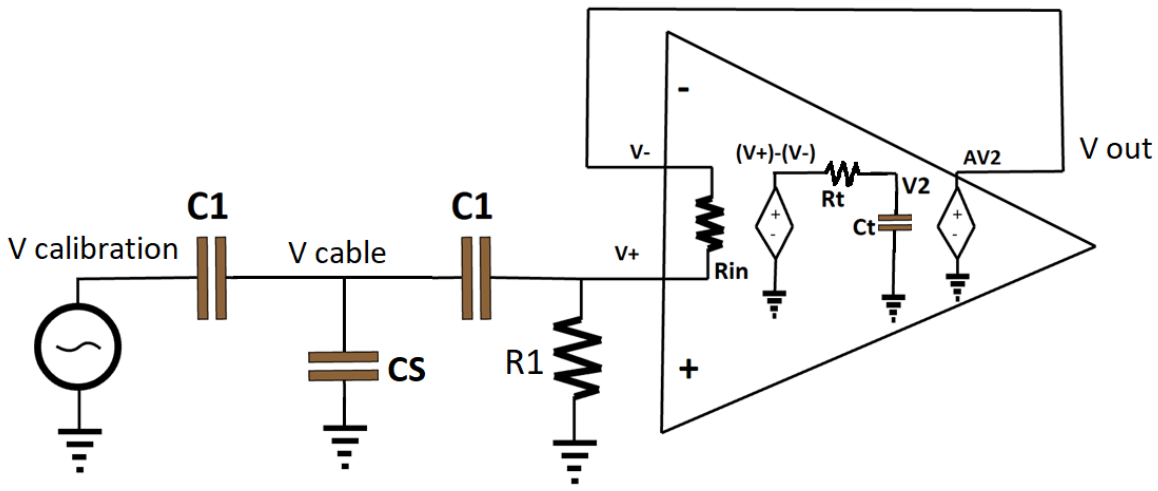

Figure 5-15: The lumped parameter model involving the shunt capacitance as well as the op-amp characteristics.

most electrical projects, the high impedances involved in the calibration system cause this parasitic capacitance to affect the outcome of our results.

Due to the parasitic capacitance, a more correct model of the calibration circuit for a single power cable involves a capacitance between the power cable and ground, as shown in Figure 5-15. We refer to this as the shunt capacitance. The transfer function of this lumped parameter model is

$$\frac{V_{out}}{V_{calibration}} = H_c(j\omega) = \frac{j\omega A R_1 R_{IN} C_1^2}{d(j\omega)} \tag{5.9}$$

$$\begin{aligned} d(j\omega) =& (2C_1 + C_S)(R_{IN} + R_1 + R_{IN}A) \\ &+ j\omega((2C_1 + C_S)\tau(R_1 + R_{IN}) + (C_1^2 + C_S C_1)R_1 R_{IN} C_1 (A+1)) \\ &+ (j\omega)^2 \tau R_1 R_{IN}(C_1^2 + C_S C_1) \end{aligned} \tag{5.10}$$

To prepare this function for curve fitting, we divided the numerator term into the denominator, and made the approximation that $A + 1 \approx A$, since A is typically

around 100,000. The transfer function then simplifies to

$$\frac{V_{out}}{V_{calibration}} = H_c(j\omega) = \frac{j\omega}{(\frac{2C_1+C_S}{C_1})\frac{1}{R_1C_1} + j\omega((\frac{2C_1+C_S}{C_1})\frac{(R_{IN}+R_1)\tau}{AR_1R_{IN}C_1} + \frac{C_1+C_S}{C_1}) + (j\omega)^2\frac{\tau}{A}(\frac{C_1+C_S}{C_1})} \tag{5.11}$$

$$= \frac{j\omega}{a_c + j\omega b_c + (j\omega)^2 c_c} \tag{5.12}$$

When the shunt capacitance C_s is set to 0, (5.11) simplifies to the function in (5.6) with the electrode capacitance set to $\frac{C_1}{2}$, since the input signal must now travel through two capacitors of equal capacitance in series.

5.3.1 Capacitance Estimation Procedure

We estimate the capacitance C_1 by taking voltage measurements to solve for the parameters of $H_e(j\omega)$, the transfer function between the cable and the output voltage, and $H_c(j\omega)$, the transfer function between the calibration voltage and the output voltage. Since our hardware did not allow us to produce frequencies high enough to observe the $(j\omega)^2$ terms, we could only solve for two parameters in either transfer function. However, the term $\frac{(R_{IN}+R_1)\tau}{AR_1R_{IN}}$, which we can refer to as the hardware term h, appears in both transfer functions. It was thus possible to use the four transfer function parameters to solve for the three unknown values, C_1, C_s, and h.

To estimate the capacitance, we first apply voltages directly to the power cable to determine the parameters a and b of transfer function $H_e(j\omega)$. The parameter b allows us to solve for the hardware term h, since $b = 1 + \frac{h}{C_1}$. This hardware term should stay the same when the detector is removed from one cable and attached to another, since it does not involve the value of the electrode capacitance. Therefore, this term can be determined when the detector is manufactured.

We can then estimate the capacitance by applying voltages to the calibration electrode to determine the values of the parameters a_c and b_c of transfer function

Table 5.3: Electrode voltage when different voltage frequencies were applied directly to the cable.

Frequency (Hz)	V_{cable} (V)	V_{out} (V)
70	3.8453	0.0217
140	0.0403	3.6017
51,179 Hz	0.7035	0.4246

Table 5.4: Electrode voltage when different voltage frequencies were applied to the calibrating electrode while the power cables were disconnected from any load.

Frequency (Hz)	V_{cable} (V)	V_{out} (V)
70	0.0043	3.8474
140	0.0080	3.6047
51,103	1.009	0.1149

$H_c(j\omega)$. Then, we can estimate C_1 using the formula

$$\hat{C_1} = \frac{1 + b_c - R_1 a_c h}{R_1 a_c} \tag{5.13}$$

where $\hat{C_1}$ is the estimate of the capacitance. This formula can be derived from (5.11).

However, our experiments did not yield accurate capacitance estimates, which we believe was due to limitations of the hardware we used. Below we will present the results we obtained, followed by an explanation of the next steps necessary to achieve better results.

To determine the hardware term h and the true capacitance C_1, we first applied voltages at low and high frequencies directly to the cable and measured the output at the sensing electrode. The values for low and high frequency terms are found in Table 5.3. These measurements yielded values of $a = 77980\ s^{-1}$ and $b = 0.6035$. This implied a capacitance of $C_1 = 1.26$ pF and a hardware term of $h = 0.803$ pF.

We then applied voltages at low and high frequencies to the calibration electrode while the power cables were disconnected from any loads. Relevant values are shown in Table 5.4. For these readings, $a_c = 395367\ s^{-1}$ and $b_c = 0.1138$. These values, together with the hardware term h, yielded a capacitance estimate of $\hat{C_1} = 1.63$ pF. This is an estimation error of 29%.

We also tested other cases, such as the case when the cables are connected to the power op-amps in the parallel cable test bed. We applied a range of voltage frequencies to the calibration electrode. Table 5.5 shows relevant voltage values. These measurements yielded values of $a_c = 648079\ s^{-1}$ and $b_c = 20.03$. The capacitance estimate is 2.39 pF, an 89% error.

These results indicate that the hardware we used was not a good fit for the model we developed. The most likely reason for this problem is that the capacitance between the calibrating electrode and the power cable is not equal to the capacitance between the power cable and the sensing electrode. This is likely due to the fact that we were using wires to provide a common mode calibration signal from circuits outside the PCB board. Although we pushed the calibration wires as far away from the cables as possible, as shown in Figure 5-16, they may still have created parasitic coupling with the power cables. In addition, our low-frequency voltage source and the high frequency voltage source were on opposite ends of the detector, and we had to move a wire to switch between these two sources, which could have affected the readings we received. A better approach would be to generate the signal from within the PCB board and use PCB board traces to supply the calibration signal, to ensure the calibration electrode capacitance is as close as possible to the sensing electrode capacitance. Additionally, it would be helpful to use an op-amp with a slew rate fast enough to process signals beyond 80,000 Hz. Producing such high frequencies will allow us to excite the ω^2 term of the transfer functions and even allow for estimating C_1 without first having to determine the hardware term with a separate frequency sweep.

Table 5.5: Comparison of electrode voltage change in the presence of interference with and without active shielding.

Frequency (Hz)	V_{cable} (V)	V_{out} (V)
70	3.8311	0.0026
140	3.5968	0.0052
54732	0.6031	0.0301

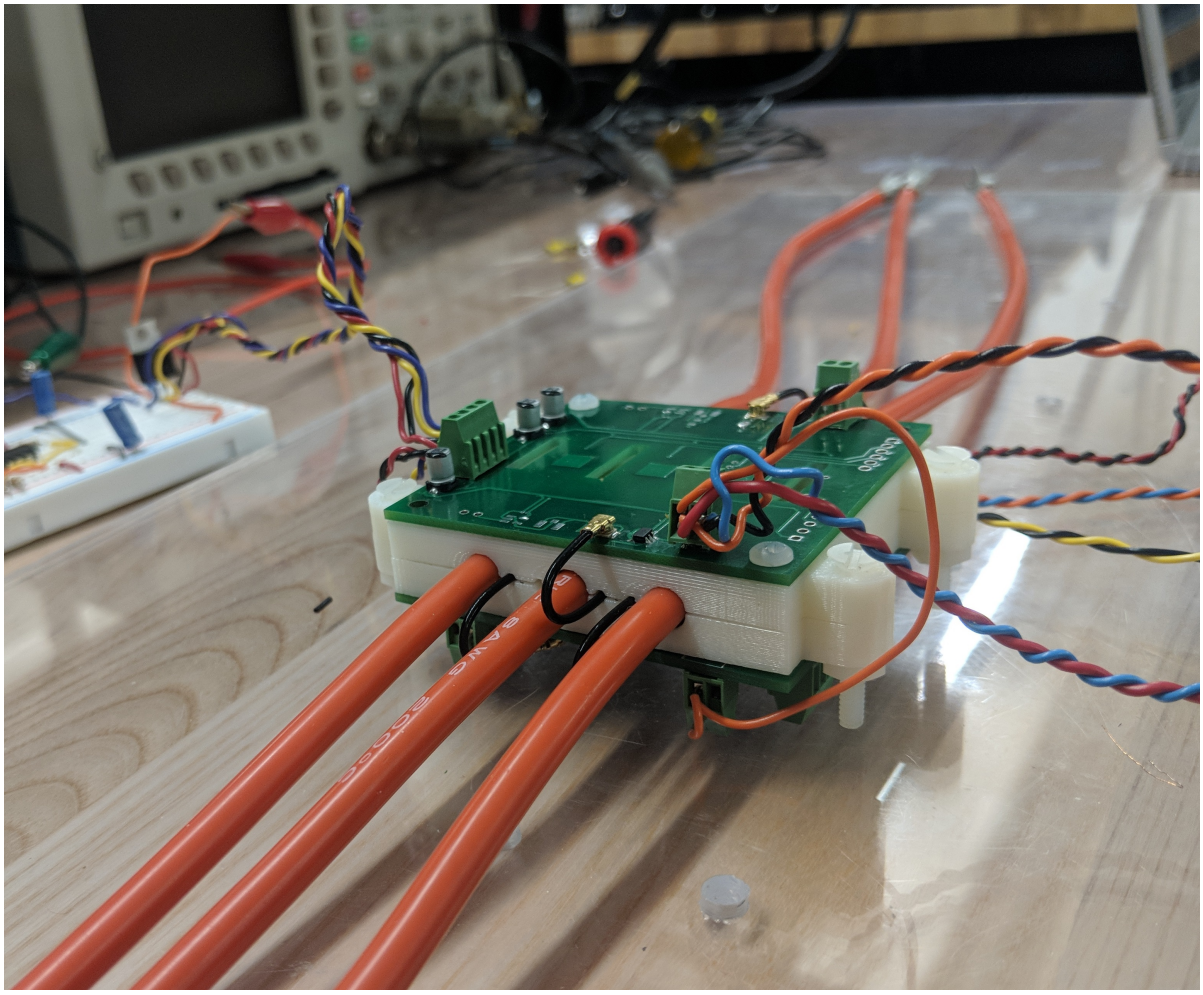

Figure 5-16: Although the wires used to supply calibration signal were pushed as far away from the red power cables as possible, they may still have coupled with the cables.

5.4 Test Bed Validation

5.4.1 Parallel Cable Test Bed

Just as we did for current estimation, we used the parallel cable test bed to validate our voltage estimation techniques. Since the calibration system is still under development, the electrode capacitance was calibrated manually by selecting the correct capacitance value that would make the peaks of the waveform match.

No external interference

We applied a pair of 8.3 V 90 Hz voltages phase shifted by 120 ° across the parallel cable test bed. This was done to be able to compare the amplitude of the voltage estimate to the amplitude of the 60 Hz pickup from ambient electric fields. Figure 5-17 shows the two voltage estimates superimposed over the contact measurements of the voltages. Since the voltage detection system must remain isolated from the system it is measuring, these waveforms were not obtained simultaneously. However, once superimposed, the estimate fits over the contact measurement very well. The capacitance between the cable and electrodes was found to be 3.03 pF in this experiment. The estimate error for this experiment was 0.43%. Since we manually adjusted the capacitance value to make the amplitudes of the estimates and measurements match, the source of this error was noise in the readings of the ADC.

To confirm the estimates scaled correctly, we applied a pair of 10.0 V 90 Hz voltages and used the same electrode capacitance value. The estimate is shown in Figure 5-18. The error was 0.62%.

External Cables Interference

We then placed a pair of external cables 1.5 cm above the detector as shown in Figure 5-19. The cables contained voltages at the same frequency and magnitude as the internal cables, 8.3 V and 90 Hz. The estimates and contact measurements are shown in Figure 5-20. The error was 0.67%.

External Plate Interference

We placed an aluminum plate 1.5 cm above the detector as shown in Figure 5-21. The estimate is shown in Figure 5-22. The estimation error was 0.68%.

Cable Bundle Interference

We placed a bundle of six cables 1.5 cm above the detector as shown in Figure 5-23. The cables contained the same magnitude and frequency voltage as the internal

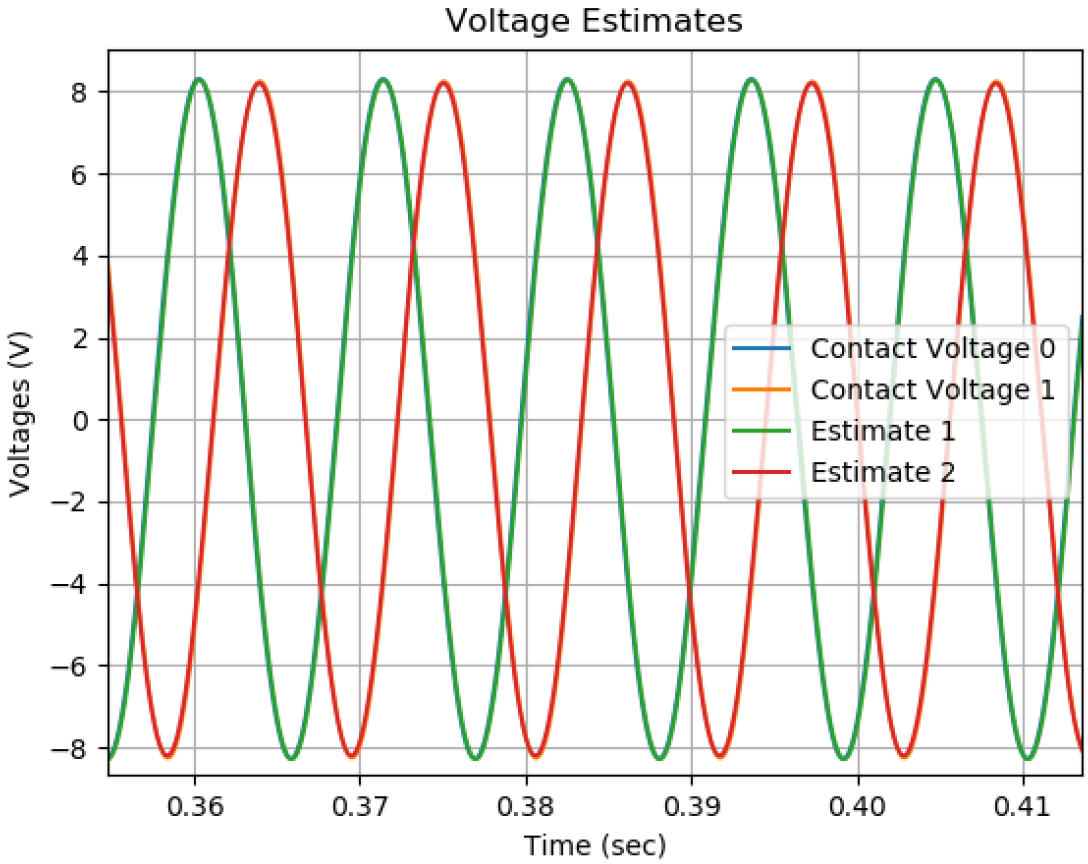

Figure 5-17: Voltage estimates superimposed over a contact measurements. There was no external interference. The error was 0.43%

cables, 8.3 V and 90 Hz. The estimates are shown in Figure 5-24. The error was 0.62%.

Empty Detector

We removed the cables from the detectors and ran a voltage estimate. The estimate is shown in Figure 5-25. Note that the y axis is scaled by 10^{-15}. This figure shows that in the absence of cable voltage differences the estimator output is practically zero.

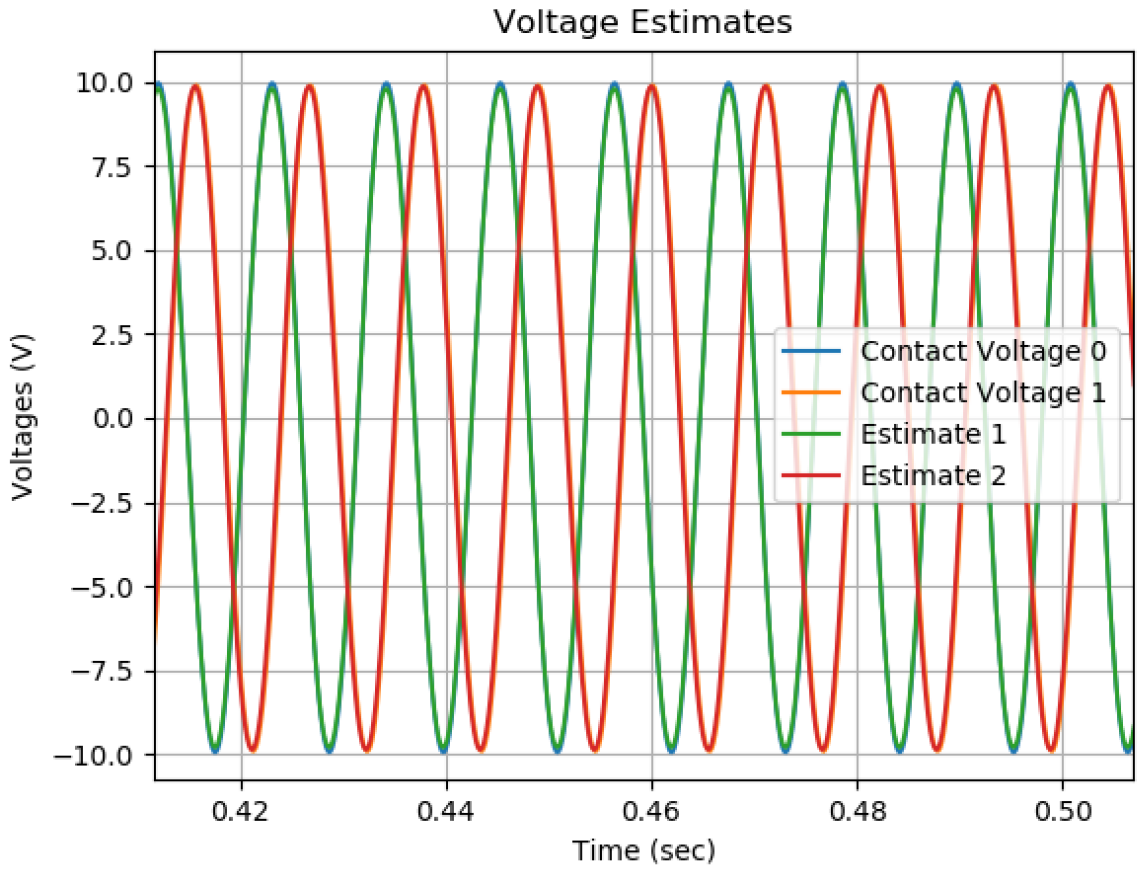

Figure 5-18: Voltage estimates of higher magnitude voltage. There was no external interference. The error was 0.62%.

5.4.2 Lightbulb Demo

We estimated voltages using the lightbulb demo to examine wall power and the harmonics within contained in it. As previously mentioned, the lightbulb demo contained a voltage divider that we used to safely perform contact measurements of voltage. The measured voltage waveform is shown in Figure 5-26. The Fourier transform of this measurement is shown in Figure 5-27. As the Fourier transform shows, the signal is not a clean sinusoidal signal, but rather contains many harmonics that our estimator must correctly match.

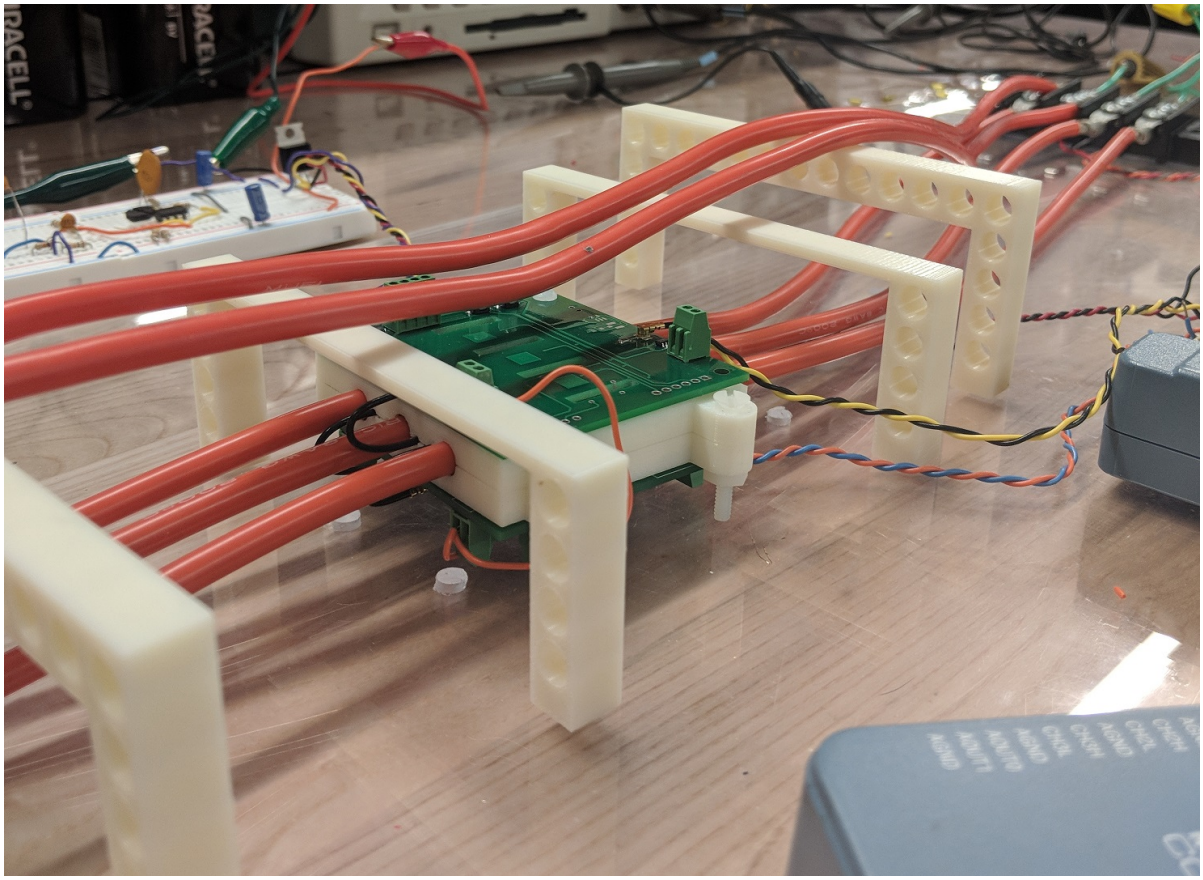

Figure 5-19: A pair of cables were placed 1.5 cm above the detector.

No Interference

Figure 5-28 shows the voltage estimate superimposed over the contact measurement. The estimate lined up very well with the measured voltage, recreating the characteristic flat slope after each peak. The error was 0.82%.

External Cables Interference

We placed a pair of cables 1.5 cm above the detector. The cables contained 8 V 90 Hz voltage. As Figure 5-30, the change in the output voltage was minimal. The error was 0.83%. Figure 5-31 shows a Fourier Transform of the voltage estimate, where the 90 Hz interference created by the external cables is visible, but two order of magnitudes lower than the 60 Hz signal. Note that the vertical axis of the graph in this Figure is logarithmic.

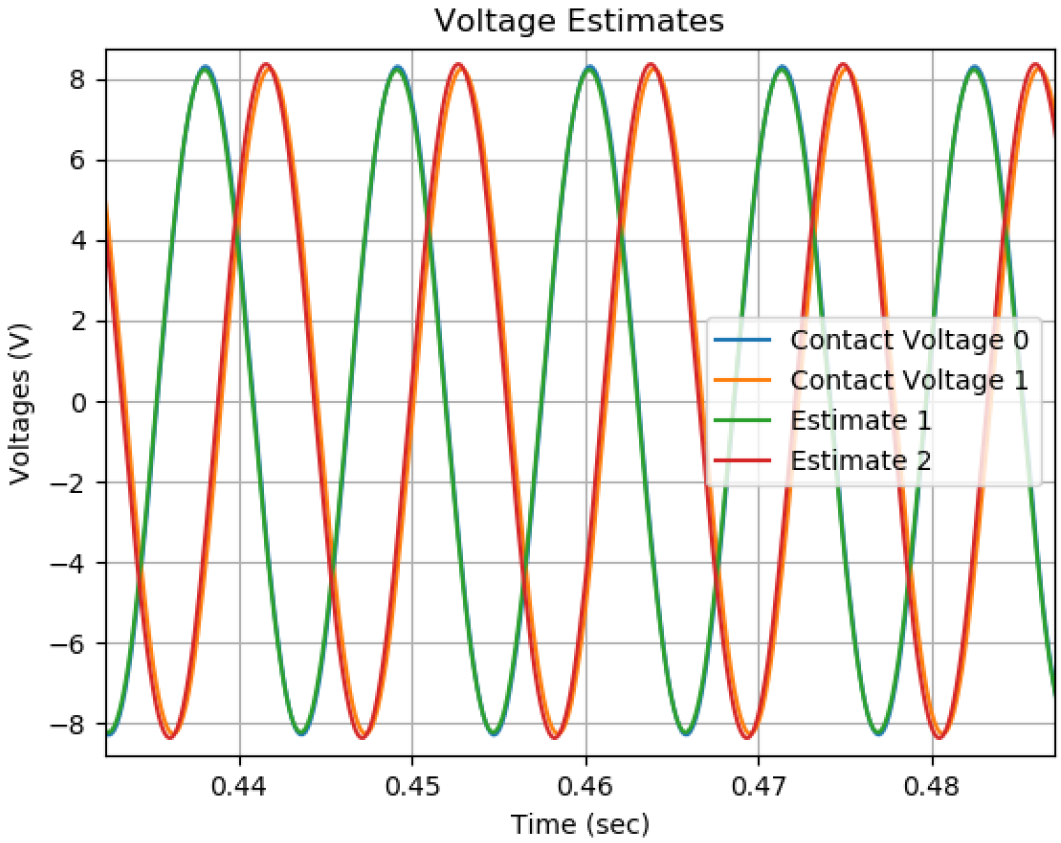

Figure 5-20: The estimated voltages superimposed over the measured voltages when two cables were placed above the detector. The voltage estimation error was 0.67%.

External Plate Interference

We placed a plate 1.5 cm above the detector. The voltage estimate is shown in Figure 5-34. The error was 0.83%.

Six Cables Interference

We placed six cables 1.5 cm above the detector. The cables contained 8 V 90 Hz voltage. The voltage estimate is shown in Figure 5-35. The error was 0.95%. The Fourier transform of this estimate is shown in Figure 5-36. The 90 Hz interference caused the electrodes to pick up a small, signal but it is two orders of magnitude less than the 60 Hz signal.

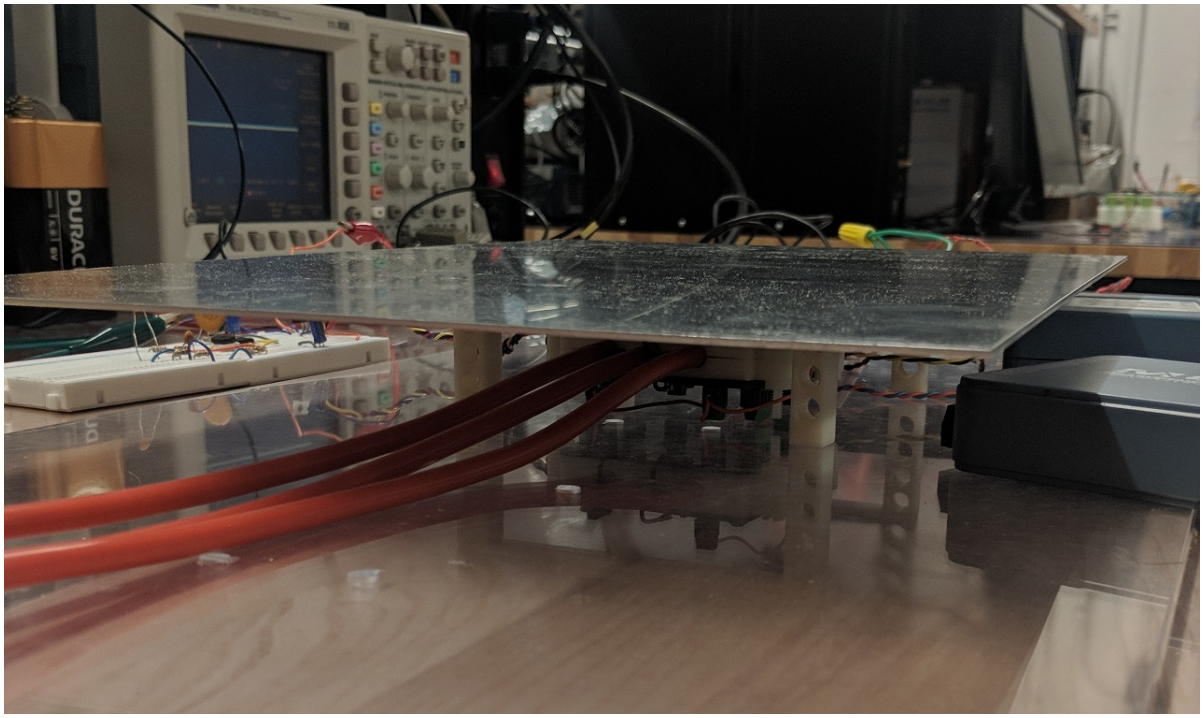

Figure 5-21: A plate was placed 1.5 cm above the detector.

5.5 Summary

The results presented in this chapter demonstrate that the voltage detector we have built produces accurate estimates and is very immune to noise. The test bed validation experiments showed that when external interference is introduced, the active shield of the voltage detector prevents the estimation error from exceeding 1%. The voltage detector is capable of detecting signals as high as 80 kHz, since that is close to the highest frequencies the TLV2371 op-amp used to drive the active shield can handle. We developed a lumped parameter model of the physical relationship between the cable voltage and electrode voltage and verified the model with experimental results. Although we also developed a model for the physical relationship between the calibration electrode and the cable, the experimental results did not fit into the model. However, we believe this occurred due to limitations with the hardware we used and that a different prototype can yield results that will fit the model. Specifically, a single signal generator source capable of producing both low-frequency and high-frequency outputs should be used. Since we used two separate sources, we had to disconnect,

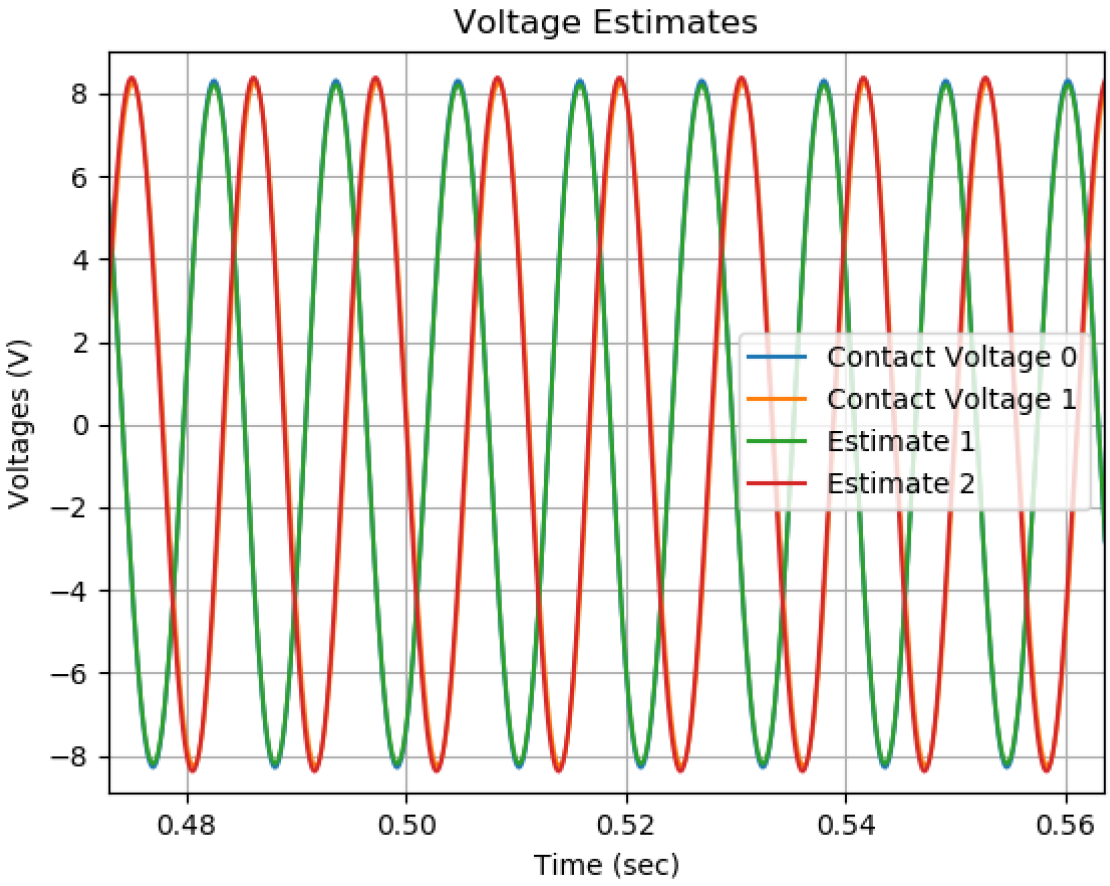

Figure 5-22: The estimated voltages superimposed over the measured voltages when a plate was placed above the detector. The voltage estimation error was 0.68%.

move, and reconnect the wires carrying the calibration signal during our frequency sweep, which could have affected our results. It would also help to use a signal generator mounted on the PCB to minimize stray capacitance between detector wires and power cables. Additionally, using an op-amp capable of outputting sinusoidal signals greater than 80,000 Hz would be useful to observe the ω^2 components of the voltage detection transfer functions.

One weakness of the voltage detector is that, although the capacitance value was consistent and stable for a single sensing electrode, the capacitance values varied greatly among different electrodes. For example, one electrode may have typically had a capacitance around 3 pF when placed around a cable, while another electrode could have a capacitance of 1.2 pF when placed around the same cable. In one

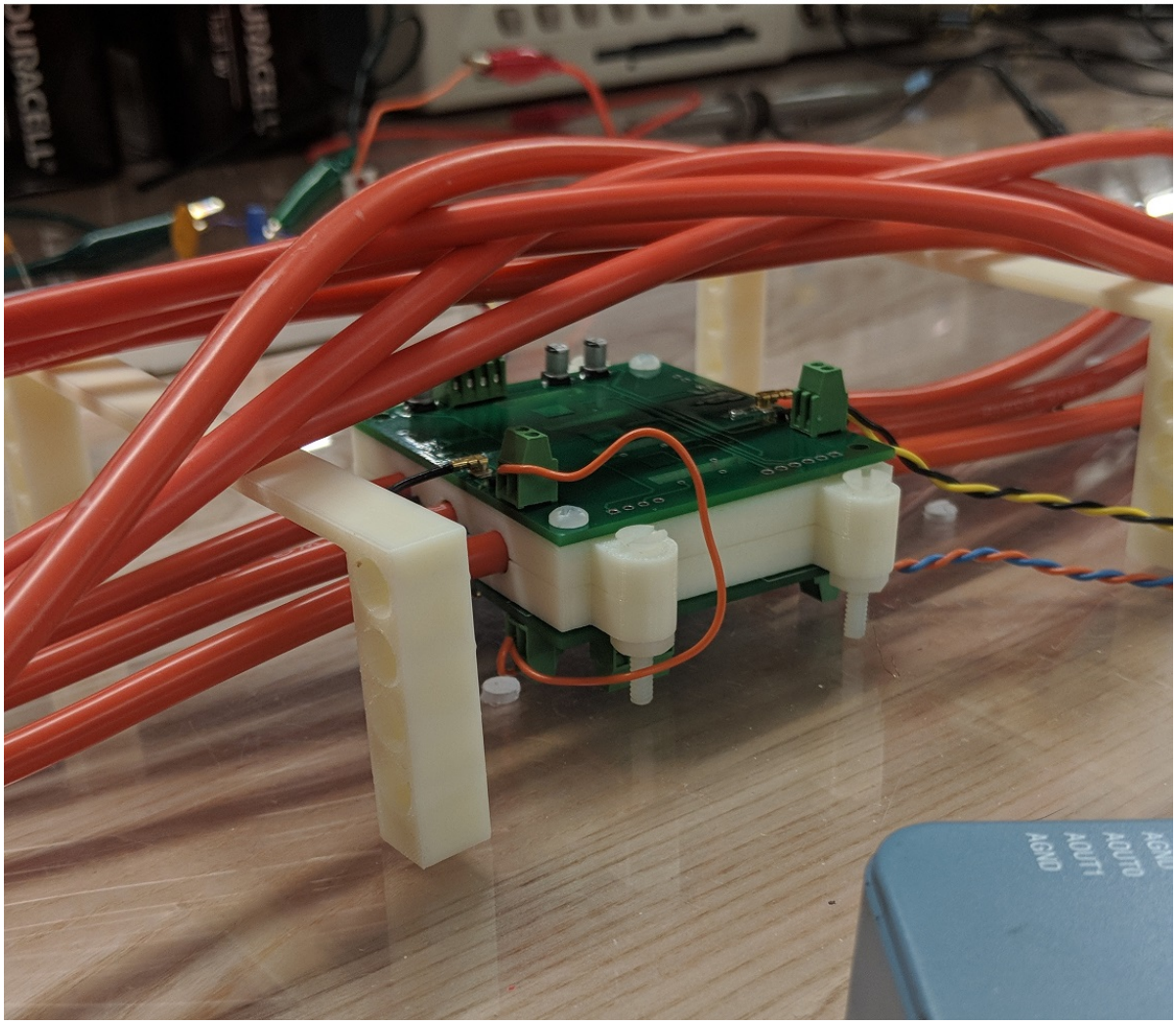

Figure 5-23: A bundle of six cables was placed 1.5 cm above the detector.

instance, reinforcing the soldering on one of the op-amps changed the capacitance of the electrode. An area of future research can be to investigate why this happens and what design decisions are necessary to manufacture electrodes that will have consistent capacitances.

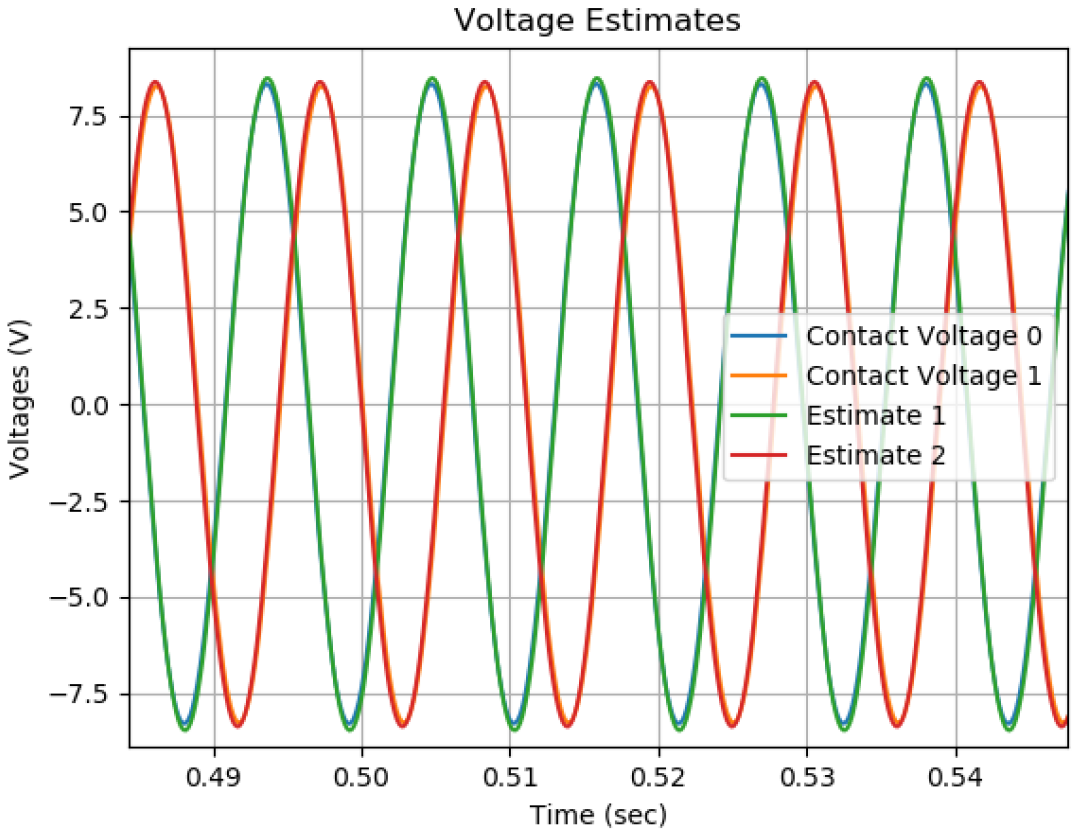

Figure 5-24: The estimated voltages superimposed over the measured voltages when a a bundle of six cables was placed above the detector. The estimation error was 0.62%.

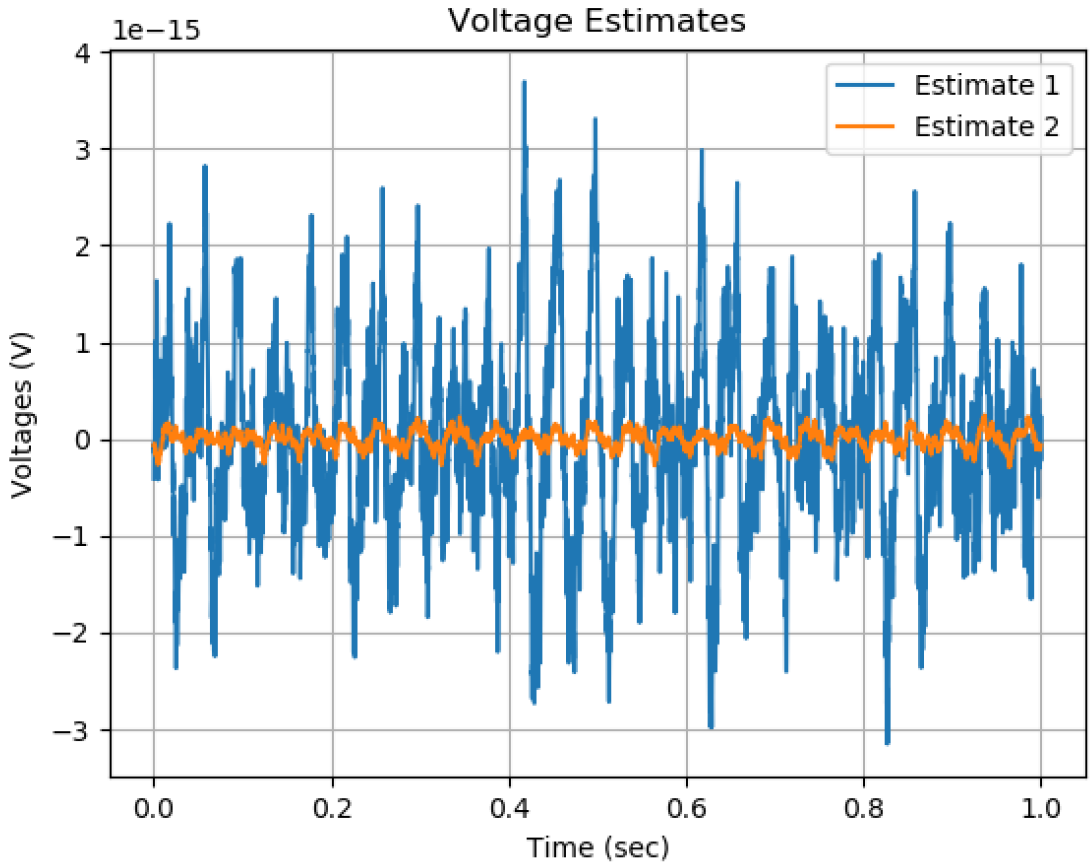

Figure 5-25: Voltage estimates were practically zero when the cables in the detector were removed.

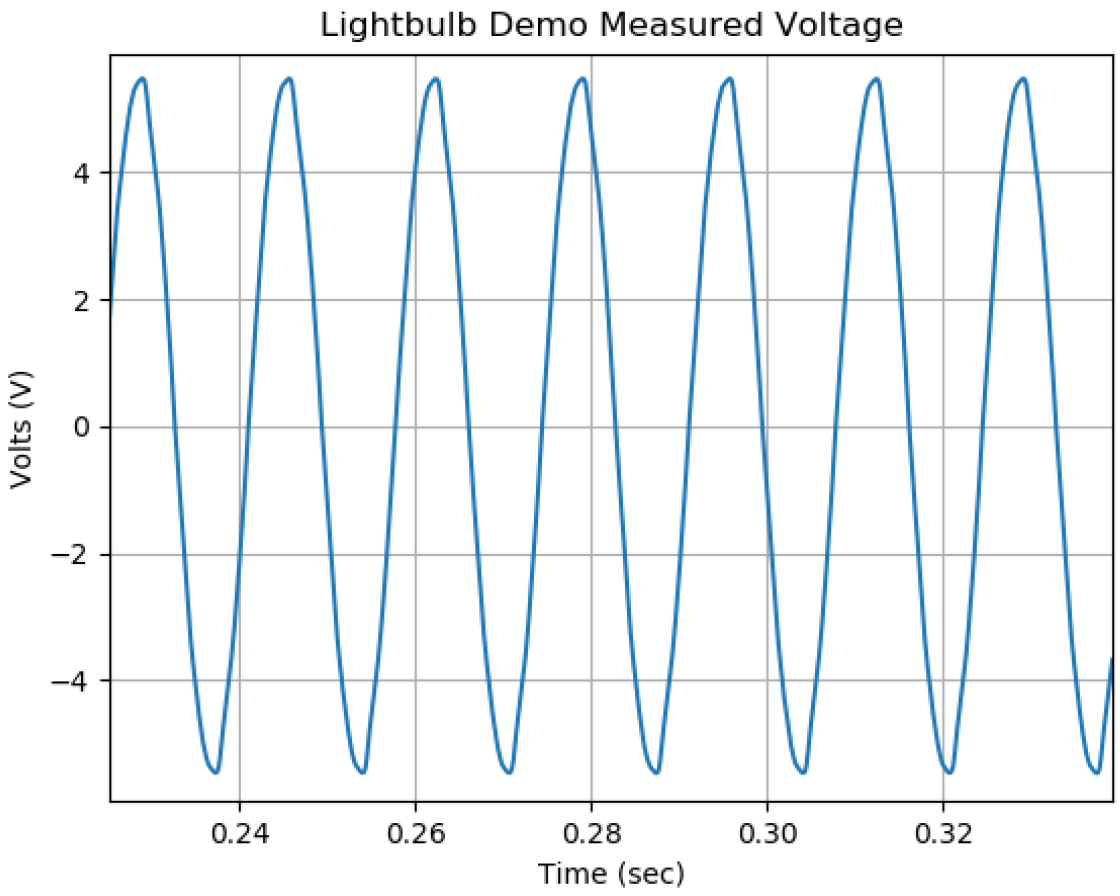

Figure 5-26: The lightbulb demo voltage.

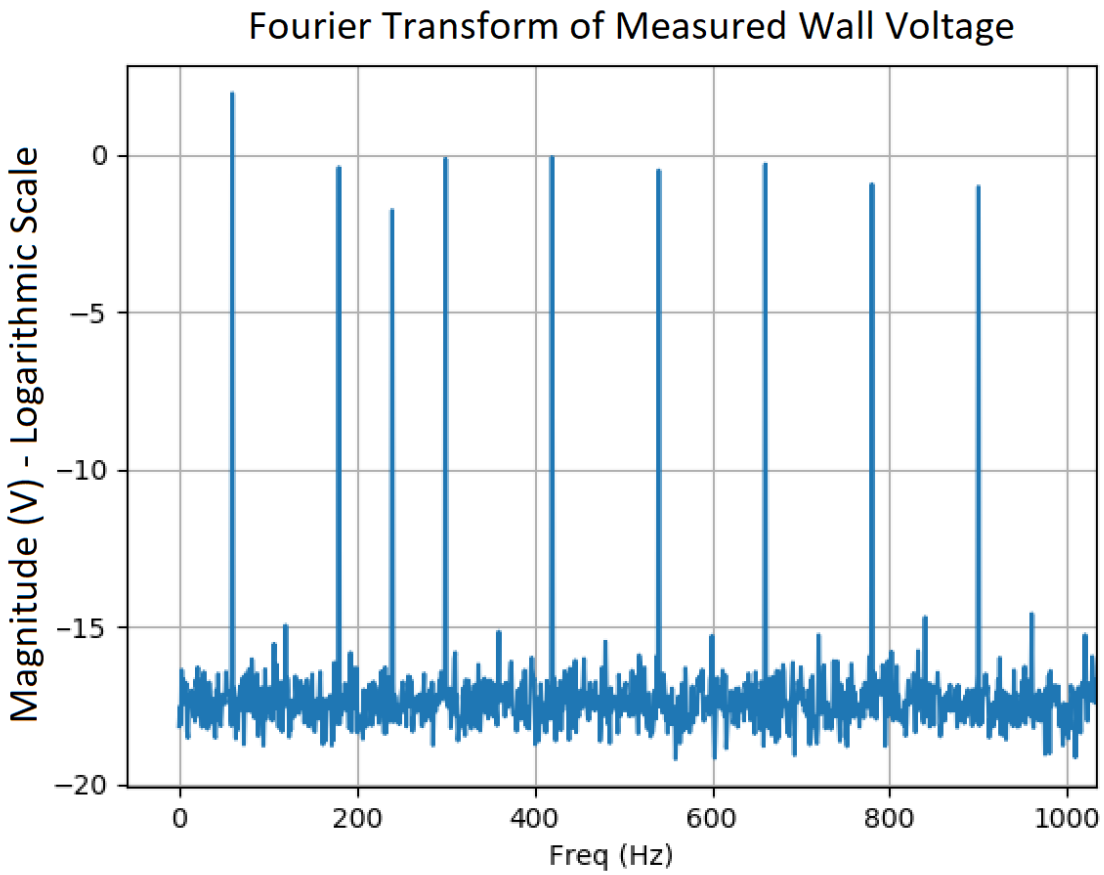

Figure 5-27: The Fourier transform of the light bulb voltage. The vertical scale is logarithmic to allow the higher level harmonics to be seen. The custom noise filter described in Section 4.7.1 with a threshold of 0.0008 was applied to the measurements before the logarithmic scale was applied.

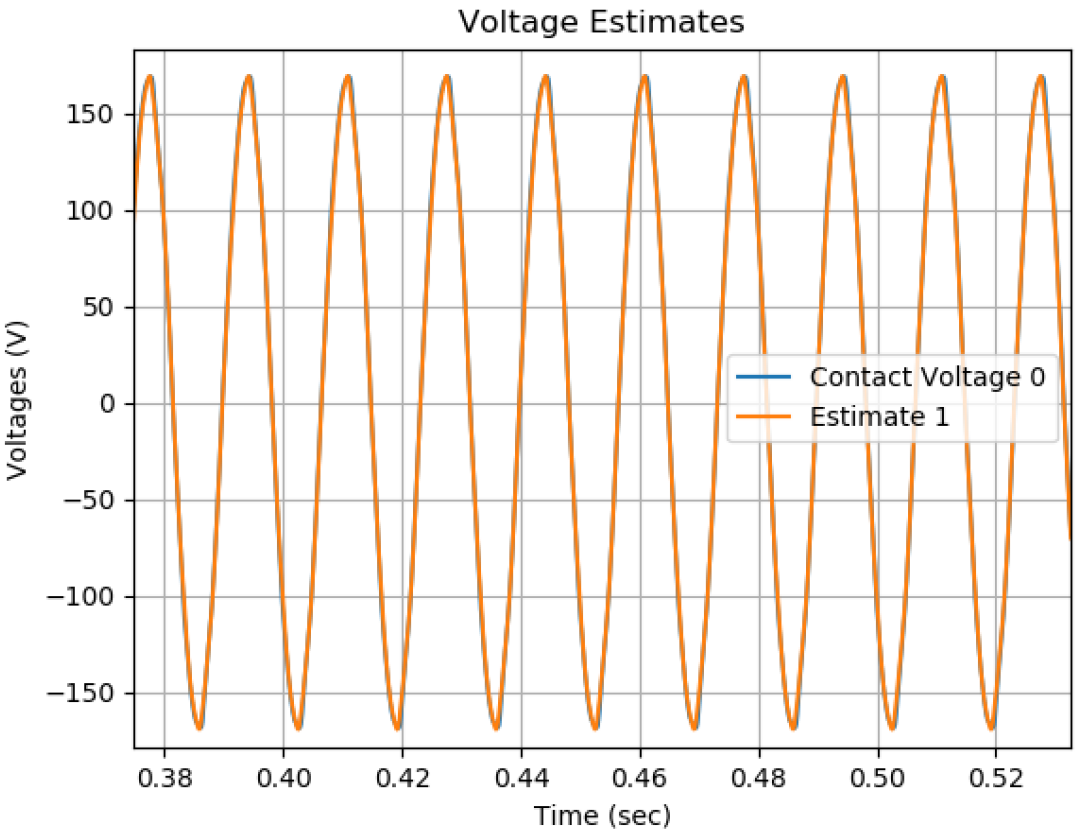

Figure 5-28: The lightbulb demo voltage in the case without interference.

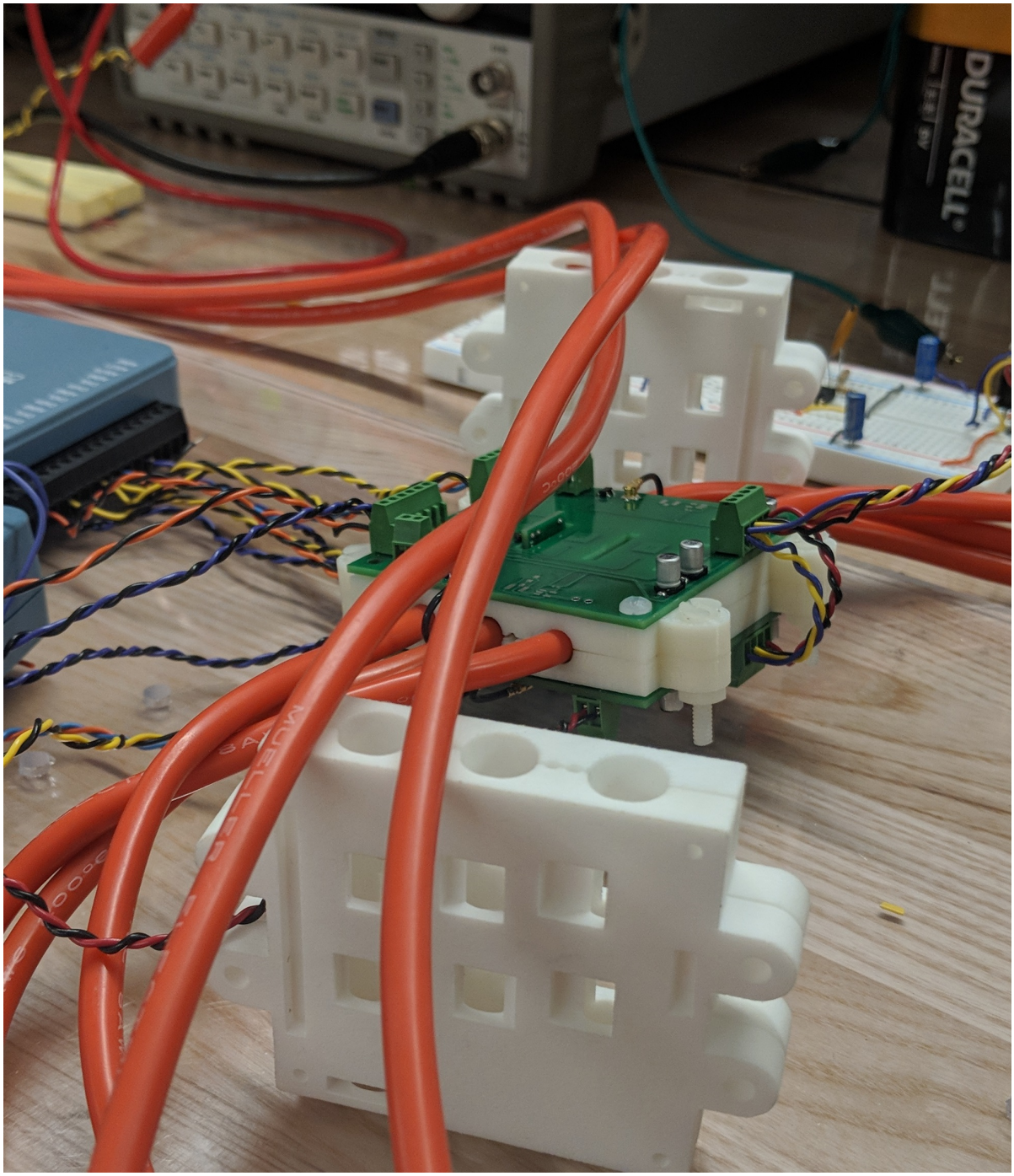

Figure 5-29: A pair of cables was placed 1.5 cm above the detector.

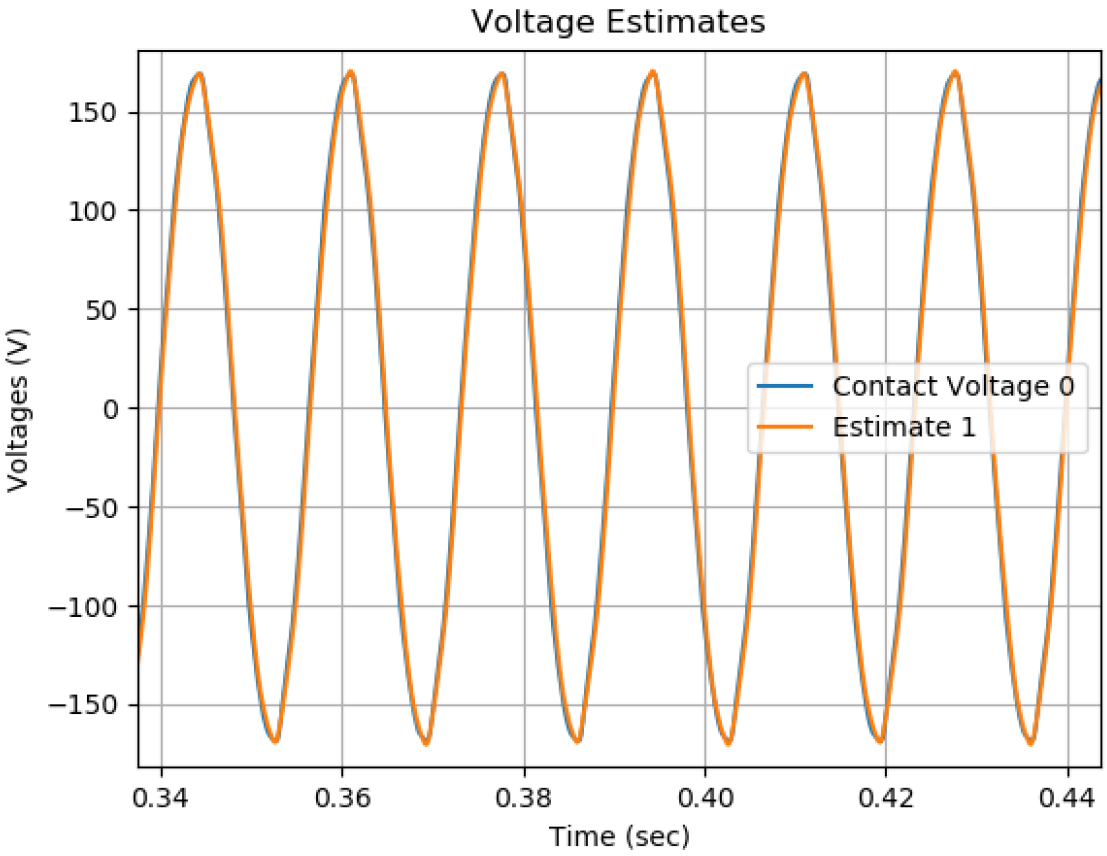

Figure 5-30: The light bulb demo voltage estimate in the presence of interference from two cables.

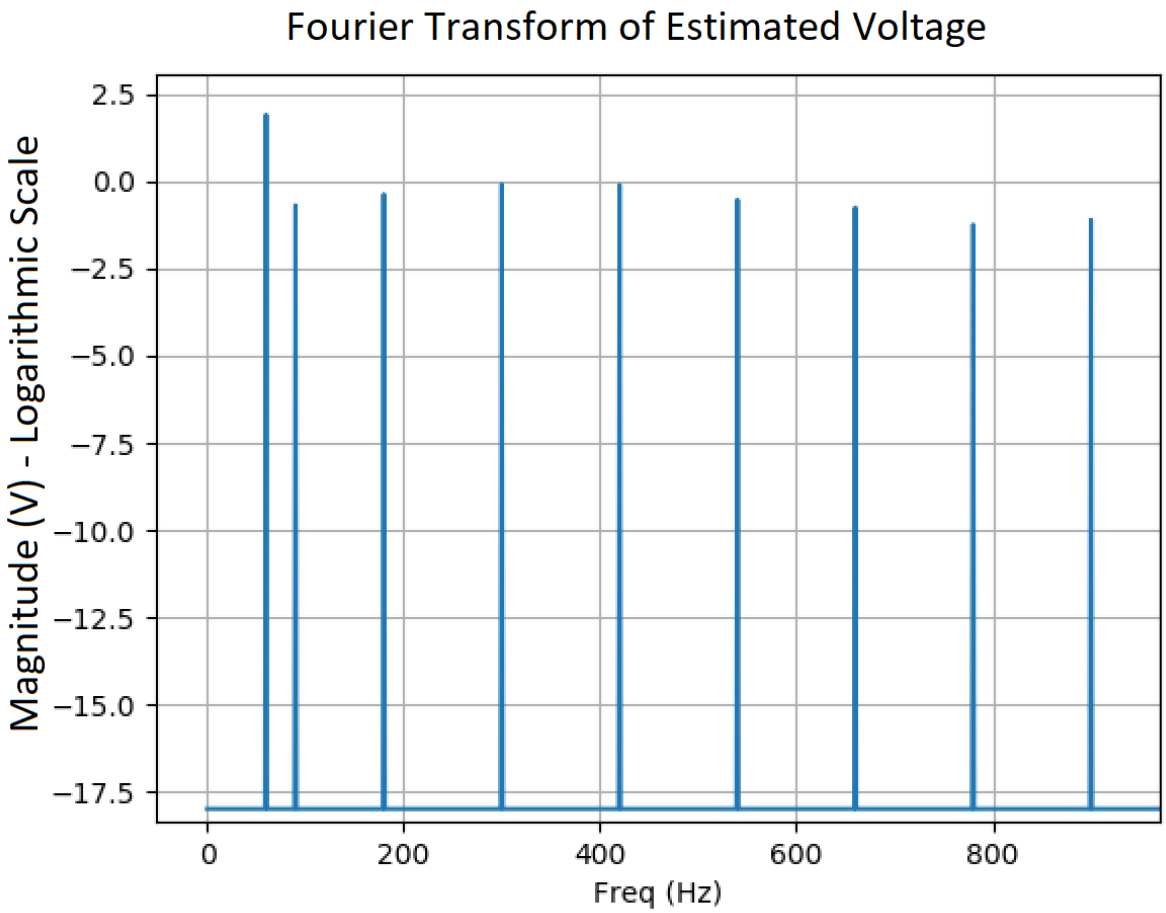

Figure 5-31: The Fourier transform of the estimated voltage when a pair of cables was placed over the detector. The vertical axis is logarithmic to allow for observation of small signals. The 90 Hz interference created by the external cables can be observed, but it is two orders of magnitude smaller than the main 60 Hz signal.

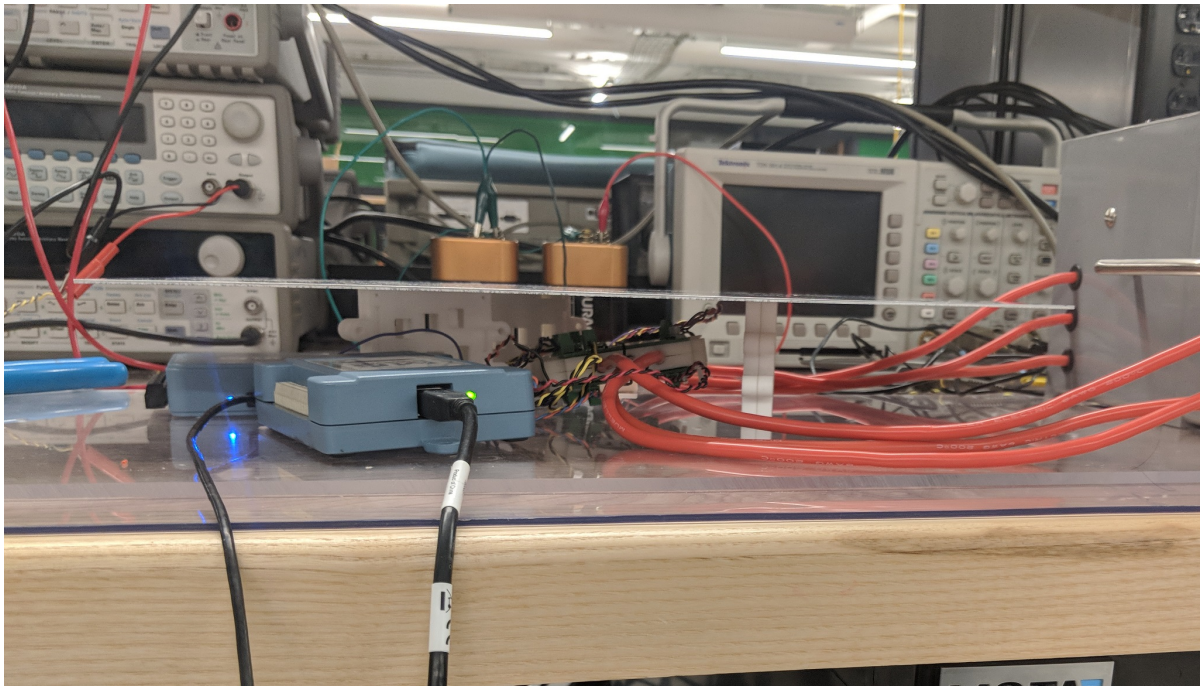

Figure 5-32: A plate was placed 1.5 cm above the detector.

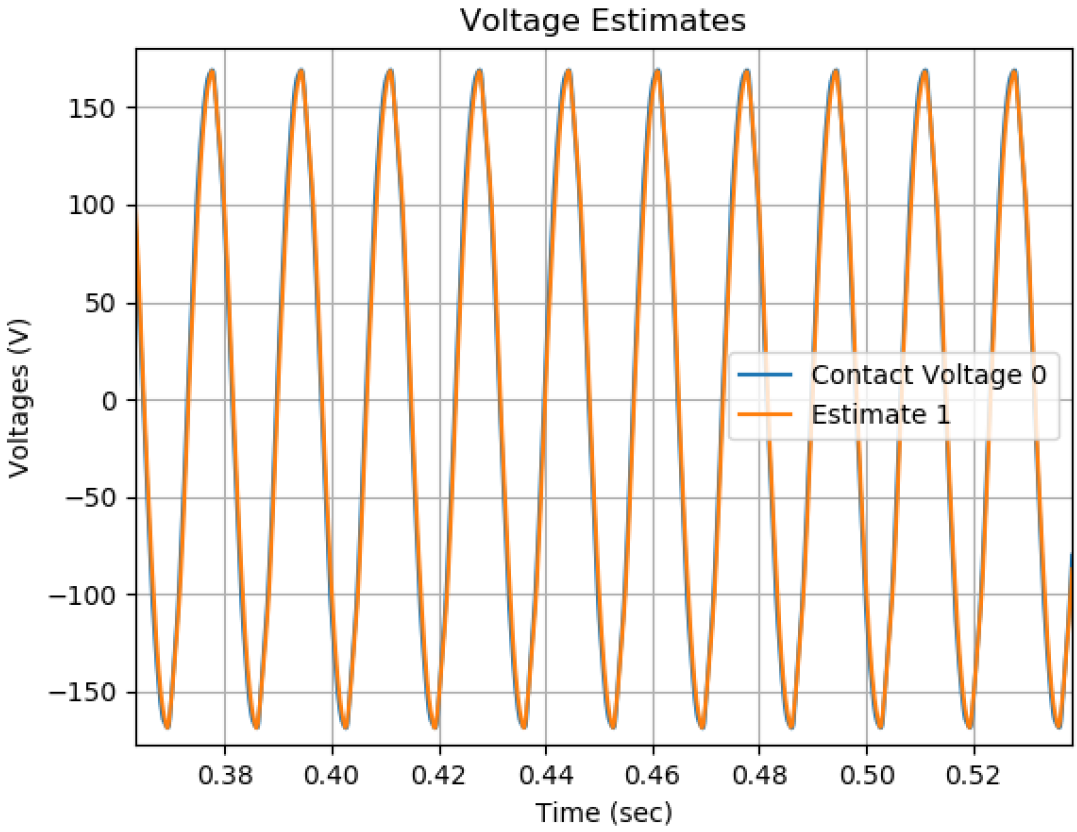

Figure 5-33: The voltage estimate when a plate was placed above the detector.

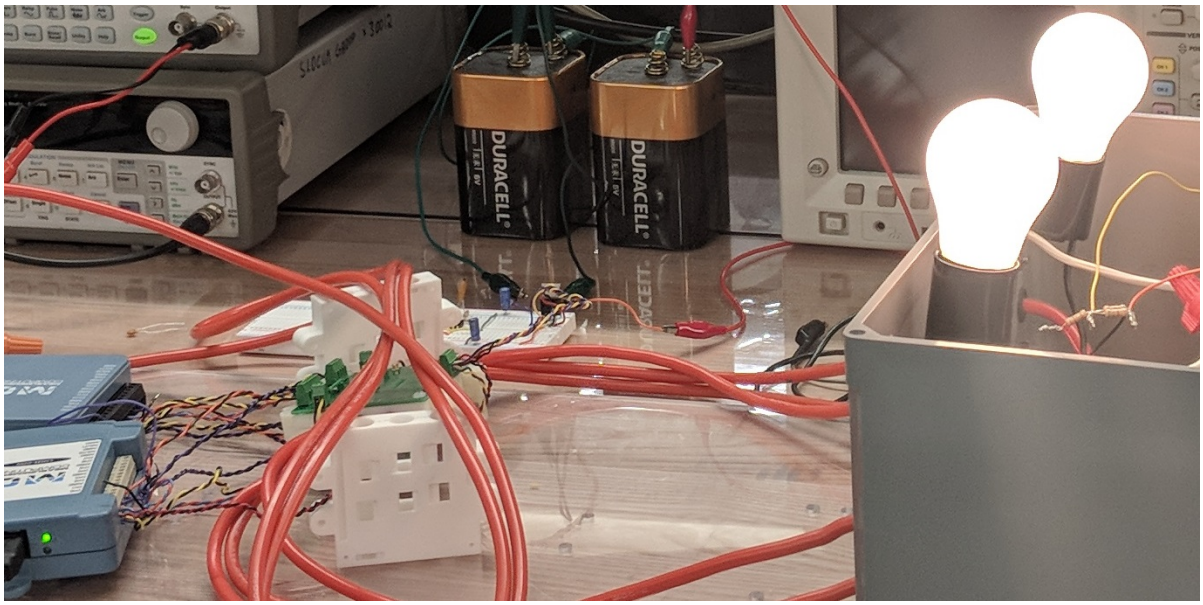

Figure 5-34: A bundle of six cables was placed 1.5 cm above the detector.

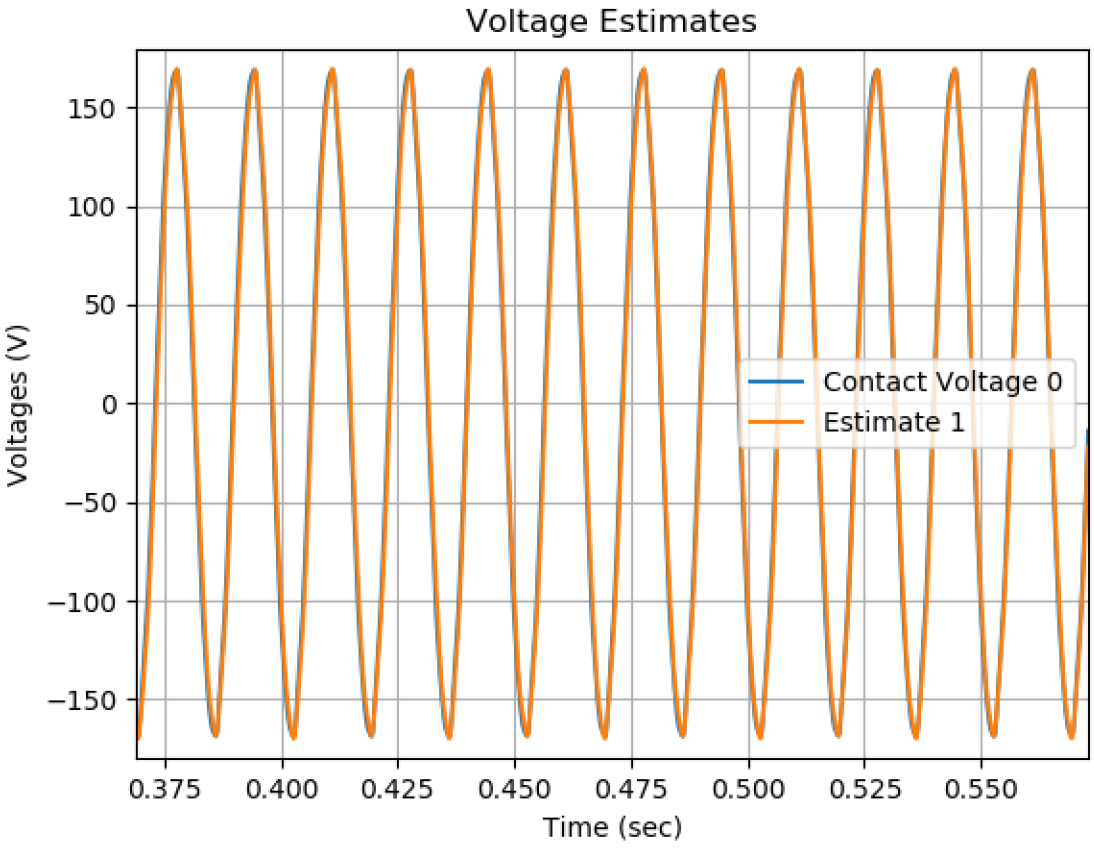

Figure 5-35: The estimate in the presence of six external cables.

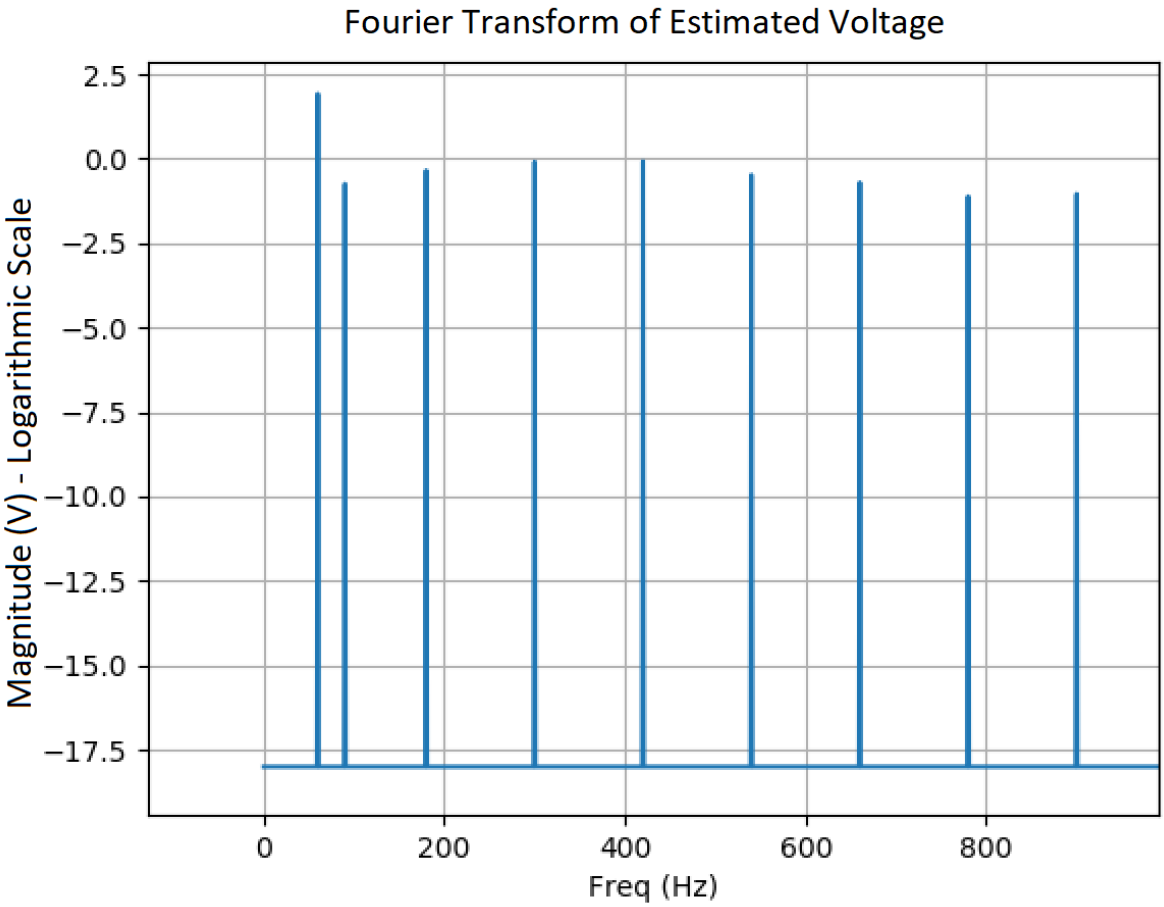

Figure 5-36: The Fourier transform of the estimated voltage when a bundle of six cables was placed over the detector. The vertical axis is logarithmic to allow for observation of small signals. The 90 Hz interference created by the external cables can be observed, but it is two orders of magnitude smaller than the main 60 Hz signal.

Chapter 6

Power Estimation

We used the voltage and current estimates to form a power estimate, which is simply the product of those two estimates. The power at a time sample n is given by

$$Power[n] = Voltage[n] * Current[n] \tag{6.1}$$

6.1 Power Estimates

6.1.1 Parallel Cables Test Bed

Figure 6-1 shows the estimated and measured power waveforms of the parallel cables test bed when 8.3 V 90 Hz voltages were applied to create a balanced three phase set of 1.67 A 90 Hz currents. The estimated power waveforms were obtained by multiplying the estimated currents across both resistors by the two line-to-line voltages estimated by the detector. However, in our setup, the measured current is calculated by measuring the voltage drop across the resistors and dividing by the resistance. Therefore, there was no phase lag between measured voltage and current. However, the two estimated currents and voltages were slightly out of phase and the error between the measured and estimated power waveforms was 2.57%. We have not yet closely investigated whether the phase lag really existed or was due to hardware limitations of the ADC. An area of future research could be to research the cause of

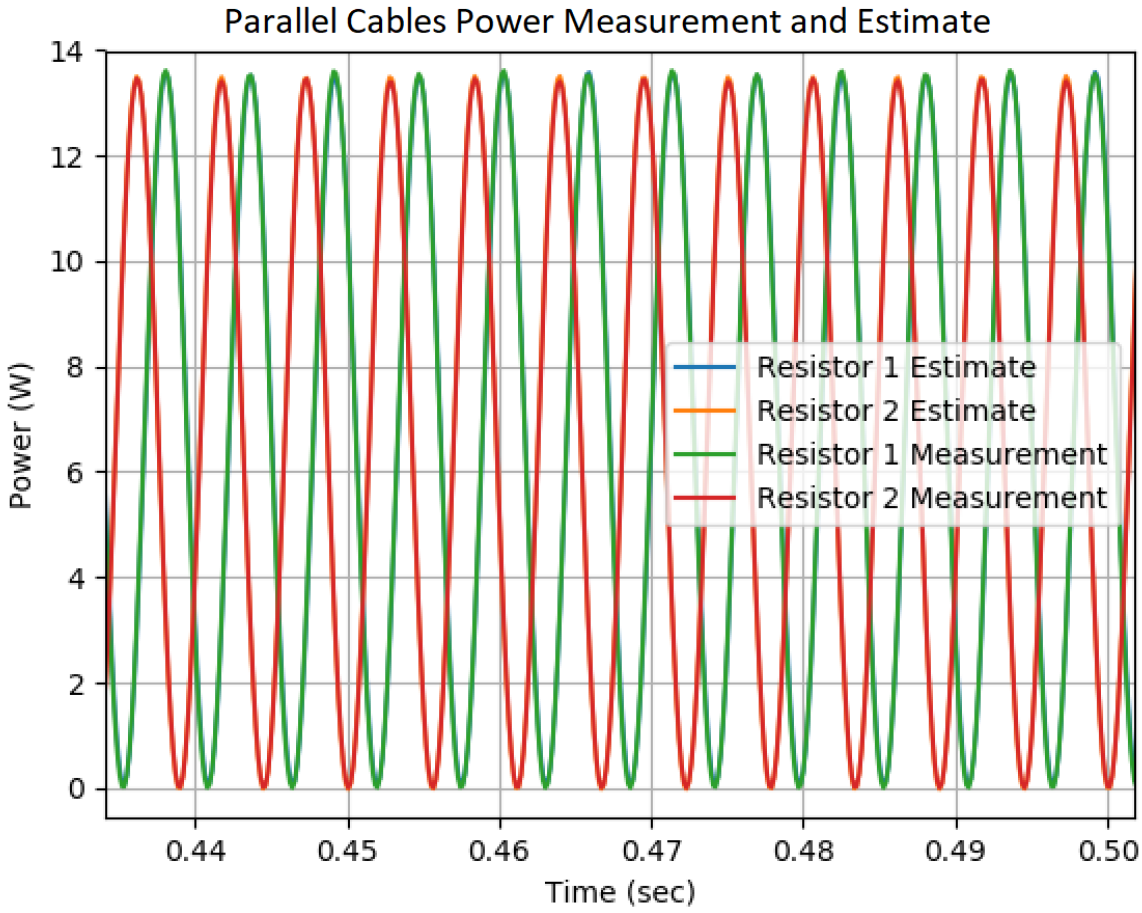

Figure 6-1: The estimated power waveforms in the parallel cables test bed superimposed over the measured power waveforms of the test bed.

this phase lag and to use different measurement methods to confirm these results.

6.1.2 Lightbulb Demo

We also measured and estimated the power in the lightbulb demo in which we connected a pair of 15 W lightbulbs. The measured and estimated power waveforms are shown in Figure 6-2. Since the light bulbs were both powered by the same 120 V RMS 60 Hz power line, their power waveforms are in phase. The estimated RMS values of the two power waveforms were 15.97 W and 16.17 W. The measured RMS values were 15.83 W and 16.38 W. The estimation error between the measured and estimated waveforms was 0.98%.

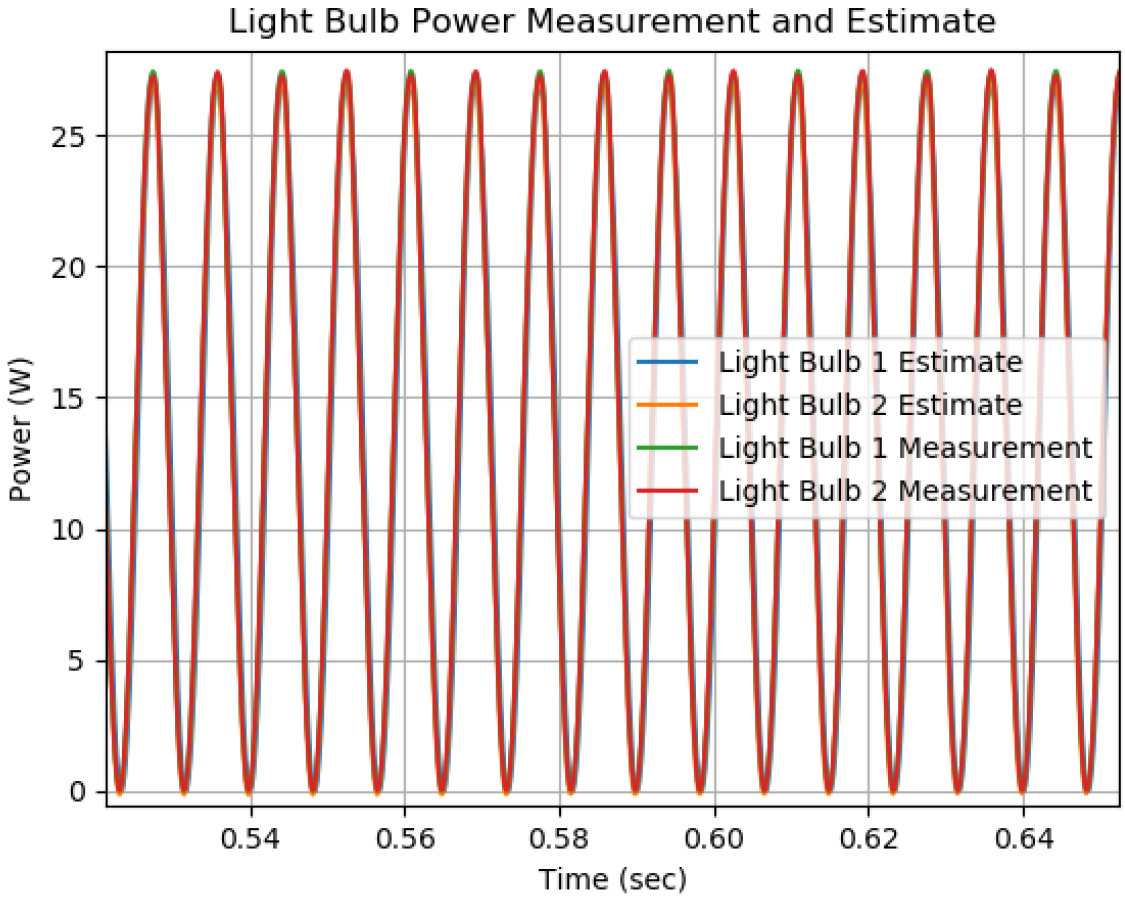

Figure 6-2: The estimated and measured power waveforms of the lightbulb demo superimposed.

6.2 Summary

This chapter provided a brief survey of power estimation using the detector, which is one of its main potential commercial applications. Measuring the power consumed by a machine is useful both for monitoring its health and for understanding the financial cost of its operation. To make the power estimates as accurate as possible, more research can be done into understanding whether the Measurement Computing ADC units were introducing phase lag into the estimated current and voltage waveforms.

Chapter 7

Summary, Conclusions, and Suggestions for Future Work

We were successful in creating a detector that could estimate current and voltage in a set of three cables with less than 1% error in the presence of external interference. The current estimate was formed using ten DRV425 fluxgate magnetic field sensors. These sensors are low-cost, highly accurate, resistant to changes in temperature, and capable of detecting a bandwidth of up to 47 KHz. The voltage sensors were built using copper tape and TLV2371 op-amps, which were capable of processing frequencies higher than 47 KHz. In bulk, the detection hardware can cost around $60. Since the detector estimates voltage and current in three cables at once, this represents a significant cost savings over, for example, currently available 200 A Hall effect sensors, which can cost $90 to estimate current in one cable. Furthermore, this detector is contactless and convenient to clip around a set of three cables. The power system being estimated does not need to be shut down for the detector to be installed. Lastly, since the estimated waveforms are digital, many different types of analysis can be performed on them, including Fourier transforms of the estimates and analyzing spectral components.

The technology we developed has the potential to be a novel and disruptive product in the field of industrial power monitoring. By using software algorithms to replace the role that heavy and costly hardware performs in other products, the current and

voltage detector can offer cost and accuracy advantages over competing products. The detector we developed met our thesis goals, and we believe it should continue to be developed into a more refined commercial product.

Several hardware and software design decisions made it possible for the detector to achieve a high level of performance. We chose the DRV425 sensor to form our current estimates due to its highly accurate readings of magnetic fields, which allowed us to produce estimates with lower error than if we used a different, less-accurate sensor. We arranged the ten magnetic field sensors in a configuration that placed the sensors as close as possible to the cables and also used the vertical spacing between and to the sides of the cables. This sensor layout, especially the use of vertically oriented sensors, reduced the error of our current estimates. We also tested a variety of different current estimators, namely: the Ordinary Least Squares Estimator, the Ampere's Law Estimator, the Non-Linear Estimator, the Spatial Harmonics Estimator, the Polynomial Estimator, the BLU Estimator, the Regression Estimator, and the Neural Net Estimator. The physics simulator we built was essential in comparing these estimators, as we were able to run simulations that allowed us to analyze how different estimators as well as different sensor placements and different numbers of sensors performed in the presence of many types of external interference. In the end we chose the BLU Estimator for current estimation, due to its consistently low error in the presence of external interference, its ability to produce estimates with zero error when there is no noise or interference, and its ability to perform estimates without requiring a specific number of sensors for a given number of external magnetic field sources.

Although we applied a series of currents through each cable to measure the values of the gain matrix used in the OLS and BLU current estimators, it is likely these values would not have to be measured for every detector manufactured. With good manufacturing tolerances, the matrix could be measured once for the array of sensors designed and then be used for all detectors, since the amount of estimation error introduced by minor sensor misplacement tends to be small.

We were able to perform accurate voltage estimates in the presence of interference

by shielding the signals in the voltage detection system. We placed an active shield around the electrode used to sense the cable voltage and used an op-amp to drive it to the same voltage as the sensing electrode. We placed a ground plane in the PCB board to shield the sensing electrode signal from the signals in the traces of the PCB board. We also placed the traces of each pair of sensing electrode outputs as far apart as possible from each other on the PCB board to avoid cross capacitance between them. These designs all contributed to the minimal level of parasitic capacitance and high level of accuracy in our voltage estimates.

We also developed a system to automatically calibrate the capacitance of the electrode used in the voltage detection system. We identified two key aspects critical to design of such a calibration system; namely, that the ground of the detection system has to remain isolated from the ground of the power system being measured to prevent the calibration signal from being shunted, and that the capacitance between the power system and the detector ground must also be included in a calibration model. In the prototype we used, we were able to estimate electrode capacitance with an error of 30% to 90%, but this error can be improved with better prototype manufacturing.

We conducted several test bed experiments to validate the performance of the current and voltage detector. We estimated currents and voltages without introducing interference, and we performed a second estimate with currents and voltages of a different amplitude to ensure the parameters of our estimators scaled linearly. We then introduced a pair of cables as a form of interference, followed by a large aluminum plate, and lastly, a bundle of six cables. These items are representative of sources of magnetic and electric field interference that are found in real electrical closets. Furthermore, we conducted these experiments in two different test beds. The parallel cable test beds allowed us to drive the currents at amplitudes and frequencies of our choosing. We presented experiments in which we drove the currents at 90 Hz, to be able to distinguish the error introduced by nearby cables and plates from the error introduced by ambient 60 Hz fields. The light bulb demo test bed allowed us to use the detector to estimate power coming from the wall, which contained harmonics that the detector estimates matched well. The test bed experiments provided an

opportunity to demonstrate the capabilities and versatility of the detector when used on real electrical systems.

There are several areas of the thesis work in which further research can be beneficial. One such area is the mathematical analysis of the OLS estimator. Specifically, it would be useful to analyze why the introduction of vertical sensors created a significant drop in the OLS estimate of currents. It would also be useful to analytically minimize (4.8) to determine the optimal placements of sensors using more mathematically rigorous techniques.

Of all the current estimation methods we tested, the BLU estimator yielded the best results. Another area of future research could be the development of additional probabilistic models of external cables that can produce a covariance matrix which can yield even better results. More irregular forms of interference can also be modeled, such as screwdrivers and small iron plates. These objects do not necessarily have to be modelled in a simulation and can instead be measured using hardware to develop an empirical covariance matrix.

Another possible area of future research is the modelling of the PCB traces that powered the magnetic field sensors and interfered with their readings. Although this problem was overcome by using hardware to measure the gain between cable currents and sensors, it would be useful to create a physics simulator sophisticated enough to generate gain and covariance matrices that can produce estimates with less than 1% error in hardware. Such a simulator would preclude the need to calibrate detectors before they can be used to detect current.

An important task for the detector to become a viable commercial product is to create a mechanism to place cables of different sizes in the center of the yoke channels. In our experiments, the yokes were 3D printed to fit snugly around the cables. However, in practice, it will not be possible to manufacture yokes for every possible cable size, since cables of the same gauge can vary in diameter by millimeters depending on the manufacturer. Therefore, some mechanism such as foam or screws are needed to hold the cable in place. It should be noted that spacing between the cable insulation and the walls of the yoke channels along which the electrodes are

taped will decrease the capacitance between the cable and the electrode.

The results of the automatic calibration can be improved by using better hardware. We recommend using a signal generator chip integrated into a PCB board, to reduce the amount of wires used to supply the calibration signal. Including a ground plane to shield the calibration signals on the PCB board from the cable would be ideal. The signal generator must provide signals as high as 50,000 Hz, but providing signals higher than 80,000 Hz, as well as using an op-amp that can output sinusoidal signals at frequencies above 80,000 Hz, should be sufficient to excite the ω^2 terms of the transfer functions presented in Chapter 5 and would allow for an estimate of C_1 to be performed using only one frequency sweep.

Lastly, more research can be done into individual machine monitoring using the voltage and current waveform estimates produced by our detector. For example, we could easily tell when the light bulb demo box was disconnected from a wall or when a light bulb was removed because the voltage and current readings would change dramatically. Similarly, a machine's states can be identified by analyzing the amount of current they are drawing and the voltage on the power lines. Online machine learning techniques can be used to learn a machine's behavior and normal operating characteristics. The accurate estimates produced by the detector will allow future research teams to perform many kinds of analysis on the collected digital waveforms without being concerned about the hardware and algorithms used to generate these estimates.

Appendix A

Source Code

This section lists the relevant Python scripts. To run these scripts, the following Python dependencies need to be installed:

- SciPy and Numpy.
- Matplotlib.
- Scikit-learn.
- Tensorflow.
- Mcculw.

The files are listed in the following order:

- simulator folder files, in alphabetical order.
- utilities folder files, in alphabetical order
- neuralnetworks folder files, in alphabetical order.
- All other files in alphabetical order.

`simulator/sensor_placer.py`

```
import numpy as np

#height and width of space between cable and sensors, in meters
h = 0.00525
w= 0.015
xof = .0075

def addtopbottom(loc_array,i,n):
    loc_array[i,:] = [xof+.0015*n,h,xof+.0015*(n+1),h]
    loc_array[i+1,:] = [xof+.0015*(n+1),-h,xof+.0015*n,-h]
    return loc_array

def addsideways(loc_array,i,n):
    loc_array[i,:] = [xof,.0015*n-h,xof,.0015*(n+1)-h]
    loc_array[i+1,:] = [xof+.015,.0015*n-h,xof+.015,.0015*(n+1)-h]
    loc_array[i+2,:] = [xof+.030,.0015*n-h,xof+.030,.0015*(n+1)-h]
    loc_array[i+3,:] = [xof+.045,.0015*n-h,xof+.045,.0015*(n+1)-h]
    return loc_array

def create_location_array():

    sorder = [3,1,5,0,6,2,4]
    torder = [4,2,7,1,8,0,9,3,6,5]

    cnt = 0
    tcnt = 0
    scnt = 0
    i = 0
    loc_array = np.zeros((88,4))

    firstgroup = []
    secondgroup = []
    thirdgroup = []


    while cnt < 88:
```

```
37          addtopbottom(loc_array,cnt,torder[tcnt])
38          firstgroup.append((cnt,-1))
39          firstgroup.append((cnt+1,-1))
40          cnt += 2
41          addtopbottom(loc_array,cnt,10+torder[tcnt])
42          secondgroup.append((cnt,-1))
43          secondgroup.append((cnt+1,-1))
44          cnt += 2
45          addtopbottom(loc_array,cnt,20+torder[tcnt])
46          thirdgroup.append((cnt,-1))
47          thirdgroup.append((cnt+1,-1))
48          cnt += 2
49
50          tcnt += 1
51
52          if i < 7:
53              addsideways(loc_array,cnt,sorder[scnt])
54              firstgroup.append((cnt,-1))
55              firstgroup.append((cnt+1,1))
56              secondgroup.append((cnt+1,-1))
57              secondgroup.append((cnt+2,1))
58              thirdgroup.append((cnt+2,-1))
59              thirdgroup.append((cnt+3,1))
60              cnt += 4
61              scnt +=1
62          i += 1
63
64      return loc_array,firstgroup,secondgroup,thirdgroup
65
66  def get_hardware_sensor_array():
67      x1 = .007
68      x2 = .022
69      x3 = .037
70      x5 = .052
71      sl = .0015
72
```

```
    yplane = .006

    inner_loc = .001
    outer_loc = 0

    loc_array = np.zeros((10,4))

    loc_array[0,:] = [.015,-yplane,.015+sl,-yplane]
    loc_array[1,:] = [x1,0,x1,-sl]
    loc_array[2,:] = [.030,-yplane,.030+sl,-yplane]
    loc_array[3,:] = [x2,inner_loc,x2,inner_loc-sl]
    loc_array[4,:] = [.045,-yplane,.045+sl,-yplane]
    loc_array[5,:] = [.045,yplane,.045+sl,yplane]
    loc_array[6,:] = [x5,0,x5,sl]
    loc_array[7,:] = [.030,yplane,.030+sl,yplane]
    loc_array[8,:] = [x3,-inner_loc,x3,-inner_loc+sl]
    loc_array[9,:] = [.015,yplane,.015+sl,yplane]



    return loc_array
```

simulator/sensors.py

```
import numpy as np

class Sensor:
  def __init__(self):
    self.location = None
    self.orientation = None

  def setLocation(self,location):
    self.location = np.array(location) #np 3-vector
    return self

  def setOrientation(self,orientation):
    self.orientation = np.array(orientation)
```

```
14      self.orientation = self.orientation/np.linalg.norm(self.
    orientation) #orientation vector always unity
15      return self
16
17    def detect(self,sources_array,n):
18      if self.location is None or self.orientation is None:
19        raise ValueError("Sensor not fully defined.")
20
21      total_field = 0.0
22      for each_source in sources_array:
23        total_field += np.dot(self.orientation,each_source.
    get_magnetic_field(self.location,n))
24      return total_field
25
26  class LISensor:
27    def __init__(self):
28      self.start = None
29      self.end = None
30
31    def setStart(self,start):
32      self.start = np.array(start) #np 3-vector
33      self._internalSetOrientation()
34      return self
35
36    def setEnd(self,end):
37      self.end = np.array(end)
38      self._internalSetOrientation()
39      return self
40
41    def _internalSetOrientation(self):
42      if self.start is None or self.end is None:
43        return
44      self.orientation = (self.end-self.start)
45      self.orientation = self.orientation/np.linalg.norm(self.
    orientation)
46
```

```
47   def detect(self,sources_array,n):
48     if self.start is None or self.end is None:
49       raise ValueError("Sensor not fully defined.")
50     loc = self.start+((self.end-self.start)/2.0)
51     total_field = 0.0
52     for each_source in sources_array:
53       total_field += np.dot(self.orientation,each_source.
     get_magnetic_field(loc,n))
54     return total_field
55
56
57     #fix this later, but for now we'll approximate
58     '''total_field = 0.0
59     for i in range(3):
60       loc = self.start+((i/2.0)*(self.end-self.start))
61       for each_source in sources_array:
62         total_field += np.dot(self.orientation,each_source.
     get_magnetic_field(loc,n))
63
64     return ((1/3)*total_field)  '''
```

simulator/sources.py

```
1 import numpy as np
2
3 class Source:
4     def __init__(self):
5         self.location = None
6
7 class ConstantField(Source):
8
9     def __init__(self):
10        self.field = None
11
12    def setField(self,field):
13        self.field = np.array(field)
14        return self
```

```
15
16      def get_magnetic_field(self,sensor_location,n):
17          if self.field is None:
18              raise ValueError("Field not set.")
19
20          if n >= self.field.shape[0]:
21              raise ValueError
22          return self.field[n,:]
23
24
25  class FiniteWire(Source):
26      def __init__(self):
27          self.start = None
28          self.current = None
29          self.finish = None
30          self.u_0 = 4*np.pi*10**-7
31
32      def setCurrent(self,current):
33          self.current = np.array(current)
34          return self
35
36      def get_line_projection(self,P,O,S):
37          a = (np.dot(S,O)-np.dot(O,P))/(np.dot(O,O))
38          return P+O*a
39
40      '''function not used anymore for this source'''
41      def isbetween(self,start,end,point):
42          return ((point[0]-start[0])*(point[0]-end[0]) <= 0) and ((
        point[1]-start[1])*(point[1]-end[1]) <= 0)  and ((point[2]-start
        [2])*(point[2]-end[2]) <= 0)
43
44      def k_fwire_integral(self,a,b,I,xs,xf):
45          return ((self.u_0*I*a)/(4*np.pi*(a**2+b**2)))*((xf/np.sqrt(a
        **2+b**2+xf**2))-(xs/np.sqrt(a**2+b**2+xs**2)))
46
47      def j_fwire_integral(self,a,b,I,xs,xf):
```

```
48          return -1*((self.u_0*I*b)/(4*np.pi*(a**2+b**2)))*((xf/np.
    sqrt(a**2+b**2+xf**2))-(xs/np.sqrt(a**2+b**2+xs**2)))
49
50      def to_wire_coor(self,angle,vec):
51          return np.matmul(np.array([[np.cos(angle),np.sin(angle)],[-
    np.sin(angle),np.cos(angle)]]),vec)
52
53      def to_universal_coor(self,angle,vec):
54          return np.matmul(np.array([[np.cos(angle),-np.sin(angle)],[
    np.sin(angle),np.cos(angle)]]),vec)
55
56      def get_magnetic_field(self,sensor_location,n):
57          raise ValueError("This is an abstract class")
58
59
60
61 #meant for the loop wires
62 class FiniteWireXY(FiniteWire):
63
64      def setStart(self,startp):
65          self.start = np.array(startp)
66          if self.finish is not None:
67              if self.finish[2] != self.start[2]:
68                  raise ValueError("Cannot have different Z direction.
    ")
69          return self
70
71      def setFinish(self,finishp):
72          self.finish = np.array(finishp)
73          if self.start is not None:
74              if self.finish[2] != self.start[2]:
75                  raise ValueError("Cannot have different Z direction.
    ")
76          return self
77
78      def get_magnetic_field(self,sensor_location,n):
```

```
79          if self.start is None or self.finish is None or self.current
     is None:
80              raise ValueError
81
82          if n >= self.current.shape[0]:
83              raise ValueError
84
85          I = np.linalg.norm(self.current[n,:])
86
87          #get positions in universal coordinates
88          perpendicular_point = self.get_line_projection(self.start,(
    self.finish-self.start)/np.linalg.norm(self.finish-self.start),
    sensor_location)
89          r = sensor_location-perpendicular_point
90          cangle = np.arctan2(r[1],r[0])-np.pi/2 #angle between
    universal and wire coordinate system
91          ustart = self.start - perpendicular_point
92          ufinish = self.finish - perpendicular_point
93
94          a = self.to_wire_coor(cangle,np.array([r[0],r[1]]))[1] #it
    is the y' component, the x' component should be 0
95          b = sensor_location[2]-self.start[2] #difference in z
    location
96
97          starti = self.to_wire_coor(cangle,np.array([ustart[0],ustart
    [1]]))[0] #it's the x' component, the y' component should be 0
98          finishi = self.to_wire_coor(cangle,np.array([ufinish[0],
    ufinish[1]]))[0] #it's the x' component
99
100         #perform biot-savart
101         kcomp = self.k_fwire_integral(a,b,I,starti,finishi)
102         jcomp = self.j_fwire_integral(a,b,I,starti,finishi)
103
104         #cconvert back to universal coordinate and return
105         originalxy = self.to_universal_coor(cangle,np.array([[0],[
    jcomp]]))
```

```
106         return [originalxy[0],originalxy[1],kcomp]
107
108
109 class FiniteWireXZ(FiniteWire):
110
111     def setStart(self,startp):
112         self.start = np.array(startp)
113         if self.finish is not None:
114             if self.finish[1] != self.start[1]:
115                 raise ValueError("Cannot have different Y direction.
    ")
116         return self
117
118     def setFinish(self,finishp):
119         self.finish = np.array(finishp)
120         if self.finish is not None:
121             if self.finish[1] != self.start[1]:
122                 raise ValueError("Cannot have different Y direction.
    ")
123         return self
124
125     def get_magnetic_field(self,sensor_location,n):
126         if self.start is None or self.finish is None or self.current
     is None:
127             raise ValueError
128
129         if n >= self.current.shape[0]:
130             raise ValueError
131
132         I = np.linalg.norm(self.current[n,:])
133
134         perpendicular_point = self.get_line_projection(self.start,(
    self.finish-self.start)/np.linalg.norm(self.finish-self.start),
    sensor_location)
135         r = sensor_location-perpendicular_point
136         cangle = np.arctan2(r[2],r[0])-np.pi/2 #this vector
```

```
    represents the y' direction.
137         ustart = self.start - perpendicular_point
138         ufinish = self.finish - perpendicular_point
139 
140         a = self.to_wire_coor(cangle,np.array([r[0],r[2]]))[1] #it
    is the y' component, the x' component should be 0
141         b = sensor_location[1]-self.start[1] #difference in y
    location
142 
143         starti = self.to_wire_coor(cangle,np.array([ustart[0],ustart
    [2]]))[0] #it's the x' component, the y' component should be 0
144         finishi = self.to_wire_coor(cangle,np.array([ufinish[0],
    ufinish[2]]))[0] #it's the x' component
145 
146         kcomp = self.k_fwire_integral(a,b,I,starti,finishi)
147         jcomp = self.j_fwire_integral(a,b,I,starti,finishi)
148 
149         originalxy = self.to_universal_coor(cangle,np.array([[0],[
    jcomp]]))
150         return [originalxy[0],kcomp,originalxy[1]]
151 
152 
153 class Wire(Source):
154 
155     def __init__(self):
156         self.orientation = None
157         self.current = None
158         self.location = None
159         self.u_0 = 4*np.pi*10**-7
160 
161     def setLocation(self,location):
162         self.location = np.array(location)
163         return self
164 
165     def setOrientation(self,orientation):
166         self.orientation = np.array(orientation)
```

```
167         return self
168 
169     def setCurrent(self,current):
170         self.current = np.array(current)
171         return self
172 
173 
174     def get_line_projection(self,P,O,S):
175         a = (np.dot(S,O)-np.dot(O,P))/(np.dot(O,O))
176         return P+O*a
177 
178     def get_magnetic_field(self,sensor_location,n):
179         if self.location is None or self.orientation is None or self
    .current is None:
180             raise ValueError
181 
182         if n >= self.current.shape[0]:
183             raise ValueError
184 
185         mag = self.current[n]
186 
187         perpendicular_point = self.get_line_projection(self.location
    ,self.orientation,sensor_location)
188 
189         r_hat = sensor_location-perpendicular_point
190 
191         u_hat = np.cross(self.orientation,r_hat)
192         u_hat = u_hat/np.linalg.norm(u_hat)
193 
194 
195 
196         return u_hat*((self.u_0*mag)/(2*np.pi*np.linalg.norm(r_hat))
    )
```

simulator/test.py

```
1 from sensors import Sensor
```

```
from sources import ConstantField,Wire, FiniteWireXZ, FiniteWireXY
import numpy as np

u_0 = 4*np.pi*10**-7
EPSILON = .00000000001

def is_close(test_value,target_value):
    return np.abs(test_value-target_value) < EPSILON

def perpendicular_sensor_tests():
    print('Testing perpendicular_sensor_tests...')
    sensor = Sensor().setLocation([0,0,0]).setOrientation([1,0,0])

    sources1 = [ConstantField().setField([[1,0,0]])]
    detected_field = sensor.detect(sources1,0)
    if not is_close(detected_field,1.0):
        print('[1,0,0] magnetic field detected incorrectly.')
        return False

    sources2 = [ConstantField().setField([[0,1,0]])]
    detected_field = sensor.detect(sources2,0)
    if not is_close(detected_field,0.0):
        print('[0,1,0] magnetic field detected incorrectly.')
        return False

    return True


def test_wire():
    print('Testing wire...')
    wire = Wire().setLocation([0,0,0]).setOrientation([0,0,1]).
   setCurrent([[1.0]])
    sensor = Sensor().setLocation([.1,.1,0]).setOrientation([1,0,0])
    detected_field = sensor.detect([wire],0)

    #calculate value manually
```

```
37      global u_0
38      b_field = u_0*1.0*np.cos(3*np.pi/4)/(2*np.pi*np.sqrt
    (.1**2+.1**2))
39      if not is_close(detected_field,b_field):
40          print('test_wire failed, Expected: ' + str(b_field) + '
    Actual: ' + str(detected_field))
41          return False
42
43      sensor = Sensor().setLocation([.5,.1,0]).setOrientation([1,0,0])
44      detected_field = sensor.detect([wire],0)
45      b_field = u_0*1.0*np.cos(np.arctan(.1/.5)+np.pi/2)/(2*np.pi*np.
    sqrt(.5**2+.1**2))
46
47      if not is_close(detected_field,b_field):
48          print('test_wire failed, Expected: ' + str(b_field) + '
    Actual: ' + str(detected_field))
49          return False
50
51      return True
52
53  def test_perpendicular_wire():
54      sensor = Sensor().setLocation([0,0,0]).setOrientation([1,0,0])
55      wire = Wire().setLocation([1,1,0]).setOrientation([1,1,0]).
    setCurrent([[100.0]])
56      detected_field = sensor.detect([wire],0)
57      print("Perp test, detected field: " + str(detected_field))
58      return True
59
60  def xz_finite_wire_test():
61      sensor = Sensor().setLocation([0,0,0]).setOrientation([1,0,0])
62      wire = FiniteWire().setStart([-1,2,0]).setFinish([1,2,4]).
    setCurrent([[10]])
63      detected_field = sensor.detect([wire],0)
64
65      b_field = 7.00
66
```

```
    if not is_close(detected_field,b_field):
        print('test_wire failed, Expected: ' + str(b_field) + '
    Actual: ' + str(detected_field))
        return False

    return True

def xy_finite_wire_test():
    sensor = Sensor().setLocation([0,0,0]).setOrientation([1,0,0])
    wire = FiniteWire().setStart([5,5,1]).setFinish([0,2,1]).
    setCurrent([[10]])
    detected_field = sensor.detect([wire],0)

    b_field = 7.00

    if not is_close(detected_field,b_field):
        print('test_wire failed, Expected: ' + str(b_field) + '
    Actual: ' + str(detected_field))
        return False

    return True

def run_test(test_to_run):
    if test_to_run():
        print('PASS')
    else:
        print('FAIL')

if __name__ == '__main__':
    run_test(perpendicular_sensor_tests)
    run_test(test_wire)
    run_test(test_perpendicular_wire)
    run_test(finite_wire_test)
```

utilities/data_loader.py

```
import numpy as np
```

```
2
3  class DataLoader:
4
5      def read_ch_data(self,filename):
6          with open(filename) as the_file:
7              content = the_file.readlines()
8              numch = len(content[0].strip().split(','))
9              samples = np.zeros((numch,len(content)))
10             for x in range(len(content)):
11                 valuestr = content[x].strip().split(',')
12                 for y in range(numch):
13                     samples[y][x] = float(valuestr[y])
14         return samples
15
16     #WARNING: this will overwrite what is in the file with no
       further warning.
17     def overwrite_ch_data(self,filename,samples):
18         with open(filename, 'w') as the_file:
19             for x in range(samples.shape[1]):
20                 for y in range(samples.shape[0]):
21                     the_file.write(str(samples[y][x]))
22                     if y == samples.shape[0]-1:
23                         the_file.write("\n")
24                     else:
25                         the_file.write(",")
26
27     def read_vec_data(self,filename):
28         with open(filename) as the_file:
29             content = the_file.readlines()
30             valuestr = content[0].strip().split(',')
31             samples = np.zeros(len(valuestr))
32             for x in range(len(valuestr)):
33                 samples[x] = float(valuestr[x])
34         return samples
35
36     def overwrite_vec_data(self,filename,samples):
```

```
        with open(filename, 'w') as the_file:
            for y in range(samples.shape[0]):
                the_file.write(str(samples[y]))
                if y == samples.shape[0]-1:
                    the_file.write("\n")
                else:
                    the_file.write(",")

    def load_matrix(self,filename):
        return self.read_ch_data(filename).T

    def overwrite_matrix(self,filename,samples):
        self.overwrite_ch_data(filename,samples.T)
```

utilities/ft_util.py

```
import numpy as np

class FTUtils:
    def __init__(self):
        pass

    def index2Hz(nyq_freq,i,n):
        if i >= n or i < 0:
            raise ValueError("Index out of array bounds")

        if i > n/2:
            #return negative frequency
            return (i-n)*((nyq_freq*2)/n)
        else:
            #return positive frequency
            return i*((nyq_freq*2)/n)

    #returns whether an index corresponds to the nyquist frequency
    def isNyq(nyq_freq,i,n):
        #odd N arrays don't have the nyquist frequency
        if n%2==1:
```

```
            return False
        return (i == n//2)

    #returns closest index for a corresponding hertz
    def hz2index(nyq_freq,freq,n):
        if freq > nyq_freq or freq < -nyq_freq:
            raise ValueError("Requested frequency out of bounds.")

        cindex = freq/((nyq_freq*2)/n)
        index = int(np.rint(cindex))

        if index < 0:
            index = n+index

        if index>n:
            index = n

        return index
```

neuralnetworks/model.py

```
import tensorflow as tf
from tensorflow.contrib.layers import flatten

'''These are different neural network architectures that were tested
    over the course of the thesis.
They were designed both my Alan and collaborators at HARTING.'''

num_samples = 100

n_hidden1 = 10

n_hidden2 = 7
n_hidden3 = 7
n_hidden4 = 7

n_hidden5 = 5
```

```
n_hidden6 = 5
n_hidden7 = 5

n_hidden8 = 4
n_hidden9 = 4
n_hidden10 = 4

n_outputs = 3

def leaky_relu(z, name=None):
    return tf.maximum(0.01 * z, z, name=name)

def init_weight(shape):
    w = tf.truncated_normal(shape=shape, mean = 0, stddev = 0.1)
    return tf.Variable(w)

def init_bias(shape):
    b = tf.zeros(shape)
    return tf.Variable(b)

def SignalSplitter(x):
    logits= tf.layers.dense(x,10)

    return logits

def AlanNet(x):
    logits= tf.layers.dense(x,3)

    return logits

def TestRegNet_2(x):
    mu = 0
    sigma = 0.1

    x = flatten(x)
    x = tf.reshape(x, [8,num_samples])
```

```
52
53      W_1 = tf.Variable(tf.truncated_normal(shape=(num_samples,
    num_samples), mean=mu, stddev=sigma))
54      b_1 = tf.Variable(tf.zeros([num_samples]))
55      layer_1 = tf.add(tf.matmul(x, W_1), b_1)
56
57      W_2 = tf.Variable(tf.truncated_normal(shape=(3,8), mean=mu,
    stddev=sigma))
58      b_2 = tf.Variable(tf.zeros([num_samples]))
59      layer_2 = tf.add(tf.matmul(W_2, layer_1), b_2)
60
61      W_3 = tf.Variable(tf.truncated_normal(shape=(3, 3), mean=mu,
    stddev=sigma))
62      b_3 = tf.Variable(tf.zeros([num_samples]))
63      layer_3 = tf.add(tf.matmul(W_3, layer_2), b_3)
64
65      return layer_3
66
67  def TestRegNet_4(x):
68      mu = 0
69      sigma = 0.1
70
71      x = flatten(x)
72      x = tf.reshape(x, [8,num_samples])
73
74      W_1 = tf.Variable(tf.truncated_normal(shape=(num_samples,
    num_samples), mean=mu, stddev=sigma))
75      b_1 = tf.Variable(tf.zeros([num_samples]))
76      layer_1 = tf.add(tf.matmul(x, W_1), b_1)
77
78      W_2 = tf.Variable(tf.truncated_normal(shape=(3,8), mean=mu,
    stddev=sigma))
79      b_2 = tf.Variable(tf.zeros([num_samples]))
80      layer_2 = tf.add(tf.matmul(W_2, layer_1), b_2)
81
82      W_3 = tf.Variable(tf.truncated_normal(shape=(3, 3), mean=mu,
```

```
    stddev=sigma))
    b_3 = tf.Variable(tf.zeros([num_samples]))
    layer_3 = tf.add(tf.matmul(W_3, layer_2), b_3)

    W_4 = tf.Variable(tf.truncated_normal(shape=(3, 3), mean=mu,
    stddev=sigma))
    b_4 = tf.Variable(tf.zeros([num_samples]))
    layer_4 = tf.add(tf.matmul(W_4, layer_3), b_4)

    W_5 = tf.Variable(tf.truncated_normal(shape=(3, 3), mean=mu,
    stddev=sigma))
    b_5 = tf.Variable(tf.zeros([num_samples]))
    layer_5 = tf.add(tf.matmul(W_5, layer_4), b_5)

    return layer_5

def TestRegNet(x):
    mu = 0
    sigma = 0.1

    x = flatten(x)
    x = tf.reshape(x, [8,num_samples])

    W_1 = tf.Variable(tf.truncated_normal(shape=(num_samples,
    num_samples), mean=mu, stddev=sigma))
    b_1 = tf.Variable(tf.zeros([num_samples]))
    layer_1 = tf.add(tf.matmul(x, W_1), b_1)

    W_2 = tf.Variable(tf.truncated_normal(shape=(3,8), mean=mu,
    stddev=sigma))
    b_2 = tf.Variable(tf.zeros([num_samples]))
    layer_2 = tf.add(tf.matmul(W_2, layer_1), b_2)

    return layer_2

def TestRegNet_3(x):
```

```
114     mu = 0
115     sigma = 0.1
116 
117     x = flatten(x)
118     x = tf.reshape(x, [8,num_samples])
119 
120     W_2 = tf.Variable(tf.truncated_normal(shape=(3,8), mean=mu,
    stddev=sigma))
121     b_2 = tf.Variable(tf.zeros([num_samples]))
122     layer_2 = tf.add(tf.matmul(W_2,x), b_2)
123 
124     return layer_2
125 
126 def diabolo_net(x):
127     x = flatten(x)
128     x = tf.reshape(x, [8, num_samples])
129 
130 # DIABOLO NETWORK
131     number_of_neurons_first_layer = num_samples
132     number_of_neurons_second_layer = 3
133     mu = 0
134     sigma = 0.1
135 
136     We1 = tf.Variable(tf.random_normal([num_samples,
    number_of_neurons_first_layer], dtype=tf.float32))
137     be1 = tf.Variable(tf.zeros([number_of_neurons_first_layer]))
138 
139     We2 = tf.Variable(tf.random_normal([
    number_of_neurons_first_layer, number_of_neurons_second_layer],
    dtype=tf.float32))
140     be2 = tf.Variable(tf.zeros([number_of_neurons_second_layer]))
141 
142     Wd1 = tf.Variable(tf.random_normal([
    number_of_neurons_second_layer, number_of_neurons_first_layer],
    dtype=tf.float32))
143     bd1 = tf.Variable(tf.zeros([number_of_neurons_first_layer]))
```

```
144
145     Wd2 = tf.Variable(tf.random_normal([
        number_of_neurons_first_layer, num_samples], dtype=tf.float32))
146     bd2 = tf.Variable(tf.zeros([num_samples]))
147
148     encoding = tf.nn.tanh(tf.matmul(x, We1) + be1)
149     encoding = tf.matmul(encoding, We2) + be2
150     decoding = tf.nn.tanh(tf.matmul(encoding, Wd1) + bd1)
151     decoded = tf.matmul(decoding, Wd2) + bd2
152
153 # ADDED BY ME
154     W_added = tf.Variable(tf.truncated_normal(shape=(3, 8), mean=mu,
        stddev=sigma))
155     b_added = tf.Variable(tf.zeros([num_samples]))
156     logits = tf.add(tf.matmul(W_added, decoded), b_added)
157
158     return logits
159
160
161 def RNN_1(x, n_outputs, n_neurons):
162     basic_cell = tf.contrib.rnn.BasicRNNCell(num_units=n_neurons)
163     outputs, states = tf.nn.dynamic_rnn(basic_cell, x, dtype=tf.
        float32)
164     logits = tf.layers.dense(states, n_outputs)
165
166     return logits
167
168 def RNN_predict(x, n_outputs, n_neurons):
169     cell = tf.contrib.rnn.OutputProjectionWrapper(tf.contrib.rnn.
        BasicRNNCell(num_units=n_neurons),output_size=n_outputs)
170     outputs, states = tf.nn.dynamic_rnn(cell, x, dtype=tf.float32)
171     #logits = tf.layers.dense(states, n_outputs)
172
173
174     return outputs
175
```

```
176 def LSTM(x, n_outputs, n_neurons):
177     n_layers = 2
178     #lstm_cells = [tf.contrib.rnn.BasicLSTMCell(num_units=n_neurons)
179     #              for layer in range(n_layers)]
180     #multi_cell = tf.contrib.rnn.MultiRNNCell(lstm_cells)
181     #outputs, states = tf.nn.dynamic_rnn(multi_cell, x, dtype=tf.
    float32)
182
183     basic_cell = tf.contrib.rnn.OutputProjectionWrapper(tf.contrib.
    rnn.BasicLSTMCell(n_neurons, forget_bias=1.0),output_size=
    n_outputs)
184     outputs, states = tf.nn.dynamic_rnn(basic_cell, x, dtype=tf.
    float32)
185     #logits = tf.layers.dense(n_outputs)
186
187     return outputs
```

neuralnetworks/my_pre_data.py

```
1 from tensorflow.examples.tutorials.mnist import input_data
2 import numpy as np
3
4 num_samples_total = 10000
5 num_samples = 1
6
7 num_train = 18#135
8 num_val = 5#34
9 num_test = 5
10
11 path = "../data_gen/no_interference/"
12 #path = "../readings/training_sets/t_set_with_and_without_pext/"
13
14 def pre_data():
15     X_train_batches = []
16     y_train_batches = []
17     X_validation_batches = []
18     y_validation_batches = []
```

```
    X_test_batches = []
    y_test_batches = []

    """
    Load Trainings Data
    """
    print('Load trainings data: ...')

    for num in range(num_train):
        X_train = np.loadtxt(path + "x/" + str(num + 1) + ".txt",
delimiter=",")
        y_train = np.loadtxt(path + "y/" + str(num + 1) + ".txt",
delimiter=",")
        for i in range(int(num_samples_total / num_samples)):
            X_train_batches.append(X_train[i * num_samples:(i + 1) *
 num_samples].T)
            y_train_batches.append(y_train[i * num_samples:(i + 1) *
 num_samples].T)

    X_train_batches = np.array(X_train_batches)
    y_train_batches = np.array(y_train_batches)

    """
    Load Validation Data
    """
    print('Load validation data: ...')

    for v_num in range(num_val):
        X_validation = np.loadtxt(path + "x/" + str(v_num +
num_train) + ".txt", delimiter=",")
        y_validation = np.loadtxt(path + "y/" + str(v_num +
num_train) + ".txt", delimiter=",")
        for i in range(int(num_samples_total / num_samples)):
            X_validation_batches.append(X_validation[i * num_samples
:(i + 1) * num_samples].T)
            y_validation_batches.append(y_validation[i * num_samples
```

```
    :(i + 1) * num_samples].T)
48   X_validation_batches = np.array(X_validation_batches)
49   y_validation_batches = np.array(y_validation_batches)
50
51   """
52   Load Test Data
53   """
54   print('Load test data: ...')
55
56   X_test = np.loadtxt(path + "x/test.txt", delimiter=",")
57   y_test = np.loadtxt(path + "y/test.txt", delimiter=",")
58   X_test_batches.append(X_test.T)
59   y_test_batches.append(y_test.T)
60   X_test_batches = np.array(X_test_batches, dtype=np.float32)
61   y_test_batches = np.array(y_test_batches, dtype=np.float32)
62
63   """
64   Do shapes of data fit?
65   """
66
67   assert (len(X_train_batches) == len(y_train_batches))
68   assert (len(X_validation_batches) == len(y_validation_batches))
69   assert (len(X_test_batches) == len(y_test_batches))
70
71   print("Input Shape: {}".format(X_train_batches[0].shape))
72   print("Output Shape: {}".format(y_train_batches[0].shape))
73   print("Training Set:   {} samples".format(len(X_train_batches)))
74   print("Validation Set: {} samples".format(len(
    X_validation_batches)))
75   print("Test Set:       {} samples".format(len(X_test_batches)))
76
77   return X_train_batches, y_train_batches, X_validation_batches,
    y_validation_batches, X_test_batches, y_test_batches
```

neuralnetworks/train_and_evaluate.py

```
1 from __future__ import absolute_import
```

```
2 from __future__ import division
3 from __future__ import print_function
4
5 from sklearn.utils import shuffle
6 import my_pre_data
7 import tensorflow as tf
8 from tensorflow.contrib.layers import flatten
9
10 import numpy as np
11
12 import model
13 from model import TestRegNet, TestRegNet_2, TestRegNet_3,
      TestRegNet_4,RNN_predict, RNN_1, LSTM, diabolo_net, AlanNet
14
15 import matplotlib.pyplot as plt
16
17 from datetime import datetime
18 import data_loader
19
20 ######################################################
21 #Used for Tensorboard                                #
22 #now = datetime.utcnow().strftime("%Y%m%d%H%M%S")    #
23 #root_logdir = "tf_logs"                             #
24 #logdir = "{}/run-{}/".format(root_logdir, now)      #
25 ######################################################
26
27 # LOAD DATA FOR FULL-CONNECTED-NN-TRAINING:
28 num_samples = 1
29 n_inputs = 10
30 n_output=20
31 #n_outputs = 3
32 n_neurons = 100
33
34 X_train,y_train,X_validation,y_validation,X_test,y_test =
      my_pre_data.pre_data()
35
```

```
print(X_train.shape)
print(y_train.shape)
print(X_validation.shape)
print(y_validation.shape)
print('\n')

X_train = X_train.reshape(-1, num_samples*n_inputs)
y_train = y_train.reshape(-1, num_samples*n_outputs)
X_validation = X_validation.reshape(-1, num_samples*n_inputs)
y_validation = y_validation.reshape(-1, num_samples*n_outputs)
X_test = X_test.reshape(-1,num_samples*n_inputs)
y_test = y_test.reshape(-1,num_samples*n_outputs)

print(X_train.shape)
print(y_train.shape)
print(X_validation.shape)
print(y_validation.shape)
print(X_test.shape)
#X_test = X_test.reshape(-1, num_samples*n_inputs)
#y_test = y_test.reshape(-1, num_samples*n_outputs)
# LOAD DATA FOR RNN-TRAINING:
#X_train,y_train,X_validation,y_validation,X_test,y_test = pre_data.
    pre_data_RNN()
print("Data loaded")


X_train, y_train = shuffle(X_train, y_train)
EPOCHS = 500
BATCH_SIZE = 1000

# PLACEHOLDERS FOR FULL-CONNECTED-NN:
x = tf.placeholder(tf.float32, (None, n_inputs*num_samples))
y = tf.placeholder(tf.float32, (None, n_outputs*num_samples))
# PLACEHOLDERS FOR RNN:
#x = tf.placeholder(tf.float32, (None, num_samples, n_inputs))
#y = tf.placeholder(tf.float32, (None, num_samples,n_outputs))
```

```
71
72 #floor = tf.ones([num_samples,n_outputs], tf.float32)*.001
73
74 #the 3 below is the number of different metrics I'm saving, not the
       number of cables
75 dnnresults = np.zeros((3,500))
76
77
78 rate = 0.1
79
80 # REMEMBER TO ADAPT LOGITS, IF USING ANOTHER MODEL
81 #logits= TestRegNet_4(x)
82 logits= AlanNet(x)
83 #logits = LSTM(x, n_outputs, n_neurons)
84
85 #accuracy_operation = tf.reduce_mean(tf.square(logits-y), name="mse
       ")
86 #training_operation = tf.train.GradientDescentOptimizer(rate).
       minimize(accuracy_operation)
87 #training_operation = tf.train.AdamOptimizer(rate).minimize(
       accuracy_operation)
88
89 #
       ##############################################################################
90 # TESTING DIFFERENT ERROR FUNCTIONS:
91 #y_abs = flatten(y)
92 #y_abs = tf.reshape(y_abs, [3, num_samples])
93 accuracy_operation = tf.reduce_mean(tf.square(logits-y), name="mse")
94 amp_error = tf.reduce_mean(tf.abs(tf.div((logits-y),tf.maximum(tf.
       abs(y),tf.ones([n_outputs],tf.float32)*.0001))))
95 #accuracy_operation = tf.reduce_sum(tf.square(tf.div((logits-y),tf.
       maximum(tf.abs(y_abs),tf.ones([n_outputs, num_samples], tf.
       float32)*.001))), name="mse")#
96 training_operation = tf.train.AdamOptimizer(rate).minimize(
       accuracy_operation)
```

```
                                #
#
    ################################################################################


#
    ###########################################################################

#Used for Tensorboard
    #
#mse_summary = tf.summary.scalar('MSE', accuracy_operation)
    #
#file_writer = tf.summary.FileWriter(logdir, tf.get_default_graph())
    #
#
    ###########################################################################


saver = tf.train.Saver()

def evaluate(X_data, y_data):
    num_examples = len(X_data)
    total_accuracy = 0
    sess = tf.get_default_session()
    for offset in range(0, num_examples, BATCH_SIZE):
        batch_x, batch_y = X_data[offset:offset+BATCH_SIZE], y_data[
    offset:offset+BATCH_SIZE]
        accuracy = sess.run(accuracy_operation, feed_dict={x:
    batch_x, y: batch_y})
        total_accuracy += (accuracy * len(batch_x))
    return total_accuracy / num_examples

def evaluate_amp(X_data, y_data):
    num_examples = len(X_data)
    total_accuracy = 0
    sess = tf.get_default_session()
```

```
121     for offset in range(0, num_examples, BATCH_SIZE):
122         batch_x, batch_y = X_data[offset:offset+BATCH_SIZE], y_data[
    offset:offset+BATCH_SIZE]
123         accuracy = sess.run(amp_error, feed_dict={x: batch_x, y:
    batch_y})
124         total_accuracy += (accuracy * len(batch_x))
125     return total_accuracy / num_examples
126 
127 with tf.Session() as sess:
128     sess.run(tf.global_variables_initializer())
129     num_examples = len(X_train)
130     print("Training...")
131     #print()
132     for i in range(EPOCHS):
133 
134         acc_cnt = 0
135         acc_total = 0
136 
137         X_train, y_train = shuffle(X_train, y_train)
138         for offset in range(0, num_examples, BATCH_SIZE):
139             end = offset + BATCH_SIZE
140             batch_x, batch_y = X_train[offset:end], y_train[offset:
    end]
141             #print("Batch x shape: " + str(batch_x.shape))
142             sess.run(training_operation, feed_dict={x: batch_x, y:
    batch_y})
143 
144 
145             if offset%100000==0:
146                 #justmax = sess.run(amp_error,feed_dict={x: batch_x,
     y: batch_y})
147                 #print("justmax: {0}".format(justmax))
148                 accuracy = sess.run(accuracy_operation, feed_dict={x
    : batch_x, y: batch_y})
149                 #print("EPOCH {0}, offset {1}, accuracy: {2}".format
    (i,offset,accuracy))
```

```
150                 acc_total += accuracy
151                 acc_cnt += 1
152
153
154             #
    ##########################################################################

155             #Used for Tensorboard
                  #
156             #summary_str = mse_summary.eval(feed_dict={x: batch_x, y
    : batch_y})#
157             #step = i * BATCH_SIZE + offset
                  #
158             #file_writer.add_summary(summary_str, step)
                  #
159             #
    ##########################################################################

160         validation_accuracy = evaluate(X_validation, y_validation)
161         validation_amp_error = evaluate_amp(X_validation,
    y_validation)
162         print("EPOCH {} ...".format(i+1))
163         print("Training Square Error (Amps^2) = {:.7f}".format(
    acc_total/acc_cnt))
164         print("Validation Square Error (Amps^2) = {:.7f}".format(
    validation_accuracy))
165         print("Validation Average Error (Amps) = {:.7f}".format(
    validation_amp_error))
166
167         dnnresults[0,i] = acc_total/acc_cnt
168         dnnresults[1,i] = validation_accuracy
169         dnnresults[2,i] = validation_amp_error
170
171         if i%50==0:
172             saver.save(sess, './alans_test.ckpt')
173             print("Model saved")
```

```
174
175     print("test yields: " + str(sess.run(logits, feed_dict={x:
    X_test, y: y_test})))
176
177     data_loader.overwrite_ch_data("theresults.txt",dnnresults)
178     #pred = sess.run(logits, feed_dict={x: X_test})
179
180     ################################################
181     # FOR PLOTTING RNN-RESULTS:                    #
182     #y_test_plt = np.transpose(y_test, (0,2,1))    #
183     #pred = np.transpose(pred, (0, 2, 1))          #
184     ################################################
185
186     ################################################
187     # PLOT FULL-CONNECTED-NN-RESULTS:             #
188     '''plt.subplot(1,2,1)                          #
189     plt.plot(y_test[0][0], label="Current 0")   #
190     plt.plot(y_test[0][1], label="Current 1")   #
191     plt.plot(y_test[0][2], label="Current 2")   #
192     plt.legend()                                #
193     plt.title('"Original Data"')                #
194     plt.subplot(1,2,2)                          #
195     plt.plot(pred[0], label="Current 0")        #
196     plt.plot(pred[1], label="Current 1")        #
197     plt.plot(pred[2], label="Current 2")        #
198     plt.title("Predicted Data")                 #
199     plt.legend()                                #
200     plt.show()                                  #'''
201     ################################################
202
203     ################################################
204     # PLOT RNN-RESULTS:                           #
205     # plt.subplot(1,2,1)                          #
206     # plt.plot(y_test_plt[0][0], label="Current 0")#
207     # plt.plot(y_test_plt[0][1], label="Current 1")#
208     # plt.plot(y_test_plt[0][2], label="Current 2")#
```

```
    # plt.legend()                                      #
    # plt.title('"Original Data"')                      #
    # plt.subplot(1,2,2)                                #
    # plt.plot(pred[0][0], label="Current 0")           #
    # plt.plot(pred[0][1], label="Current 1")           #
    # plt.plot(pred[0][2], label="Current 2")           #
    # plt.title("Predicted Data")                       #
    # plt.legend()                                      #
    # plt.show()                                        #
    ####################################################

#########################
#Used for Tensorboard #
#file_writer.close()  #
#########################

'''with tf.Session() as sess:
    saver.restore(sess, './alans_test.ckpt')
    test_accuracy = evaluate(X_test, y_test)
    print("Test Accuracy = {:.3f}".format(test_accuracy))'''
```

averages_plot.py

```
import numpy as np
from scipy.fftpack import fft, ifft
from scipy.stats import linregress
import matplotlib.pyplot as plt
from sys import argv
import os
from utilities.data_loader import DataLoader

'''
Searchers for folders named 'channel1', 'channel2', and 'channel3'.
    Calculates the gain matrix and offset vector using those readings
    .
'''

```

```
def get_channel_slopes(chdir):
    matrix = np.zeros((10,3))
    offset = np.zeros((10,1))

    data_loader = DataLoader()

    for i in range(3):
        cnt = 0
        channeldir = chdir+"/channel"+str(i+1)
        testfiles = os.listdir(channeldir) #get all files

        x_array = np.zeros((1,len(testfiles)))
        y_array = np.zeros((16,len(testfiles)))
        print("x_array shape is: {}".format(x_array.shape))

        for x in range(len(testfiles)):
            currentnumber = float(testfiles[x].split('.')[0])/1000
            x_array[0,x] = currentnumber
            print("Current number: " + str(currentnumber))
            samples = data_loader.read_ch_data(channeldir+"/"+
    testfiles[x])
            y_array[:,x] = np.average(samples,axis=1)

        for n in [3,4,5,9,10,11,12,13,14,15]:

            plt.figure(n)
            plt.scatter(x_array,y_array[n,:])
            slope, intercept, r_value, p_value, std_err = linregress
    (x_array,y_array[n,:])
            smoothx = np.linspace(np.min(x_array),np.max(x_array)
    ,50)
            smoothy = slope*smoothx+intercept
            plt.plot(smoothx,smoothy,'r-',label="Cable " + str(i))
            plt.title('DRV425 Readings - Channel ' + str(n))
            plt.xlabel('Current (A)')
            plt.ylabel('Voltage (V)')
```

```
            plt.show()

            matrix[cnt,i] = slope
            offset[cnt,0] = (offset[cnt,0]*i+intercept)/(i+1)
            cnt +=1
    return matrix,offset

if __name__ == '__main__':
    matrix,offset = get_channel_slopes(argv[1])
    print(matrix)
    print(offset)
```

correlation.py

```
import numpy as np
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import PolynomialFeatures
import data_loader
from sys import argv
import sensor_placer
from sensors import Sensor, LISensor
from sources import ConstantField,Wire,FiniteWireXZ
import matplotlib.pyplot as plt

'''Calculates the covariance matrix used with the BLU Estimator.
   Uses probabilistic model that only
models one external cable per realization'''

h = 0.00525

xyi = np.zeros((100000,3))

startx = -.02
endx=.065
starty= -.03
endy=.03
starti=-10
```

```
endi=10

numx=29
numy=29
numi=9

cntr = 0

def isInBox(x,y):
    if gety(y) > .006 or gety(y) < -.006:
        return True
    elif getx(x) < 0 or getx(x) > .045:
        return True
    else:
        return False


def geti(i):
    return (i/(numi-1))*(endi-starti)+starti
def getx(x):
    return (x/(numx-1))*(endx-startx)+startx
def gety(y):
    return (y/(numy-1))*(endy-starty)+starty

for i in range(numi):
    print(geti(i))

for x in range(numx):
    for y in range(numy):
        if isInBox(x,y):
            for i in range(numi):
                xyi[cntr][0] = getx(x)
                xyi[cntr][1] = gety(y)
                xyi[cntr][2] = geti(i)
                cntr+= 1
                if cntr%1000==0:
```

```
59                      print('filling {0}'.format(cntr))
60
61 print("cntr is: {0}".format(cntr))
62
63 #CREATE SENSORS
64 loc_array,firstgroup,secondgroup,thirdgroup = sensor_placer.
       create_location_array()
65 sensor_array = []
66 for i in range(loc_array.shape[0]):
67     sensor_array.append(LISensor().setStart([loc_array[i,0],
       loc_array[i,1],0]).setEnd([loc_array[i,2],loc_array[i,3],0]))
68
69 Eb0 = 0
70 Eb1 = 0
71 Eb0b0 = 0
72 Eb1b1 = 0
73 Eb0b1 = 0
74 currents = 0
75
76 num_sensors = 10
77
78 covariance_matrix = np.zeros((num_sensors,num_sensors))
79
80 b_temp = np.zeros(num_sensors)
81 b_avg = np.zeros(num_sensors)
82
83 for i in range(cntr):
84     wire = Wire().setLocation([xyi[i][0],xyi[i][1],0.0]).
       setOrientation([0,0,1]).setCurrent([xyi[i][2]])
85     for k in range(num_sensors):
86         b_temp[k] = sensor_array[k].detect([wire],0)
87     for k in range(num_sensors):
88         b_avg[k] += b_temp[k]
89     for k in range(num_sensors):
90         for j in range(num_sensors):
91             covariance_matrix[k][j] += b_temp[k]*b_temp[j]
```

```
    if i %1000==0:
        print('simulating {0}'.format(i))

print(b_avg/cntr)
print(covariance_matrix/cntr)

covariance_matrix = covariance_matrix/cntr

data_loader.overwrite_ch_data('data_gen/covariancematrix2.txt',
    covariance_matrix)
```

data_cruncher.py

```
from __future__ import absolute_import, division, print_function

from builtins import *  # @UnusedWildImport
from mcculw import ul
from mcculw.ul import ULError

import time

from examples.props.ai import AnalogInputProps
from mcculw.enums import ScanOptions, FunctionType, Status,
    AnalogInputMode

from signalp import Signalp
from scipy.fftpack import fft, ifft
import numpy as np
import math
import data_loader


from examples.console import util
from examples.props.ao import AnalogOutputProps

from daq_readers_instant import ReaderPoolInstant,
    USB205ReaderInstant,USB231ReaderInstant
```

```
23
24 import ft_display
25
26 from mcculw.enums import InterfaceType
27
28 import socket
29 import sys
30 import threading
31
32 from matplotlib.backends.backend_tkagg import (FigureCanvasTkAgg,
       NavigationToolbar2Tk)
33 # Implement the default Matplotlib key bindings.
34 from matplotlib.backend_bases import key_press_handler
35 from matplotlib.figure import Figure
36
37 '''Backend program to read data from ADC and process it
38 This code may need to be updated to use the VoltageEstimator and
       CurrentEstimator object, rather than the old signalp file.
39 '''
40
41 class StringHolder(object):
42     def __init__(self):
43         self.data = ""
44
45 class ThreadingExample(object):
46
47     def __init__(self, stringholder, interval=.1):
48
49         self.interval = interval
50         self.stringholder = stringholder
51
52         thread = threading.Thread(target=self.run, args=())
53         thread.daemon = True                             # Daemonize
       thread
54         thread.start()
55
```

```
56          self.calibration_requested = False
57          self.vcalibration_requested = False
58          self.sync_requested = False
59          self.sync_code = "111111111111"
60
61      def run(self):
62
63          # Create a UDP socket
64          sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
65
66          # Bind the socket to the port
67          server_address = ('localhost', 10000)
68          print('starting up on {} port {}'.format(*server_address))
69          sock.bind(server_address)
70          """ Method that runs forever """
71          while True:
72
73              data, address = sock.recvfrom(4096)
74
75              if str(data) == "b'cal'":
76                  self.calibration_requested = True
77                  sent = sock.sendto(bytes("",'utf-8'), address)
78              elif str(data) == "b'vcal'":
79                  self.vcalibration_requested = True
80                  sent = sock.sendto(bytes("",'utf-8'), address)
81              elif str(data)[:6] == "b'sync":
82                  self.sync_code = str(data)[6:18]
83                  self.sync_requested = True
84                  sent = sock.sendto(bytes("",'utf-8'), address)
85              else:
86                  sent = sock.sendto(bytes(self.stringholder.data,'utf
    -8'), address)
87
88  def config_one_device(board_num):
89      devices = ul.get_daq_device_inventory(InterfaceType.ANY)
90      if len(devices) > 0:
```

```
91          device = devices[0]
92              # Print a messsage describing the device found
93          print("Found device: " + device.product_name + " (" + device
    .unique_id + ")\n")
94              # Add the device to the UL.
95          ul.create_daq_device(board_num, device)
96          return [device]
97      else:
98          return None
99
100 def config_two_devices(board_num,board_num1):
101     devices = ul.get_daq_device_inventory(InterfaceType.ANY)
102     # Check if any devices were found
103
104     if len(devices) > 0:
105         device = devices[0]
106         # Print a messsage describing the device found
107         print("Found device: " + device.product_name + " (" + device
    .unique_id + ")\n")
108         # Add the device to the UL.
109         ul.create_daq_device(board_num, device)
110         device1 = devices[1]
111         # Print a messsage describing the device found
112         print("Found device: " + device1.product_name +" (" +
    device1.unique_id + ")\n")
113         # Add the device to the UL.
114         ul.create_daq_device(board_num1, device1)
115         return devices
116
117     return None
118
119 def process_voltage(volsamples,sf,N):
120
121     #Estimate Voltage
122     avg1 = np.average(volsamples)
123     volsamples = volsamples- avg1
```

```
124
125     volest= np.zeros(len(volsamples))
126     volest[0] = volsamples[0]
127     for i in range(1,len(volsamples)):
128         volest[i] = volsamples[i] + volest[i-1]
129
130     avg2 = np.average(volest)
131     volest = volest - avg2
132
133     fvol = fft(volest)
134     for i in range(ft_display.hz2index(sf/2,40,N)):
135         fvol[i] = 0
136     for i in reversed(range(len(volsamples)-ft_display.hz2index(sf
    /2,40,N),len(volsamples))):
137         fvol[i] = 0
138
139     #get FT magnitude
140     fvol_abs = (1/N)*np.abs(fvol)
141     v_maxind = np.argmax(fvol_abs)
142     v_maxhz = ft_display.index2Hz(sf/2,v_maxind,N)
143
144     return v_maxhz,fvol_abs[v_maxind],ifft(fvol).real
145
146 def get_frequency_axis(nyq_freq,N):
147     frequencies = np.zeros(N)
148
149     if N%2 == 0:
150         ssp = nyq_freq/float(N/2)
151         frequencies[0:int(N/2)] = np.linspace(0.0,nyq_freq-ssp,N/2)
152         frequencies[int(N/2):N] = np.linspace(-nyq_freq,-ssp,N/2)
153     else:
154         ssp = nyq_freq/float((N-1)/2)
155         frequencies[0:int(N/2+1)] = np.linspace(0.0,nyq_freq,N/2+1)
156         frequencies[int(N/2+1):N] = np.linspace(-nyq_freq,-ssp,N/2)
157
158     return frequencies
```

```
159
160 if __name__ == '__main__':
161
162     #setup boards and everything
163     board_nums = [0,1]
164     board_nums = board_nums
165
166     stringholder = StringHolder()
167
168     background_thread = ThreadingExample(stringholder)
169
170     ul.ignore_instacal()
171     devices = config_two_devices(board_nums[0],board_nums[1])
172     if devices is None:
173         print("Could not find both devices.")
174         exit()
175
176     pool = ReaderPoolInstant()
177     for x in range(len(devices)):
178         if devices[x].product_name[0:7] == "USB-205":
179             pool.add_reader(USB205ReaderInstant(board_nums[x]))
180         elif devices[x].product_name[0:7] == "USB-231":
181             pool.add_reader(USB231ReaderInstant(board_nums[x]))
182
183     sf = 1000
184     N= 2000
185
186     frequencies = get_frequency_axis(sf/2,N)
187
188     processor = Signalp(3,12)
189     temp_matrix = data_loader.read_ch_data('matrices/matrix21/matrix
    .txt').T
190     print(temp_matrix)
191     mymatrix = np.concatenate((temp_matrix,np.array
    ([[1,1,1,1,1,1,-1,-1,-1,-1,-1,-1]]).T),axis=1)
192     processor.set_matrix(mymatrix)
```

```
193
194     offsets0 = np.zeros((8,N))
195     offsets1 = np.zeros((8,N))
196
197     running = False
198     calibrating = False
199     calibrate_index = 0
200     calibrate_size = 100
201     calibratemat = np.zeros((7,calibrate_size))
202     calibrated_yet = False
203
204     vol_gain = 447
205     vol_min = 0.0
206
207     pool.setup_buffers(2,sf)
208     pool.start_background()
209
210     sensors_to_use = [1,1,1,1,1,1,1,1,1,1,1,1]
211
212     #prepare variables needed for time shift
213     usb231map = [0,2,4,6,1,3,5,7]
214     cT = (1.0/sf)/8
215     daq_delay = 0.0
216     linear_shift0 = np.zeros((8,N),dtype=np.complex_)
217     linear_shift1 = np.zeros((8,N),dtype=np.complex_)
218     for x in range(8):
219         linear_shift0[x] = np.exp(frequencies*2*np.pi*-cT*x*1j)
220     for x in range(8):
221         linear_shift1[x] = np.exp(frequencies*2*np.pi*(-cT+daq_delay
    )*x*1j)
222
223     process_cntr = 0
224
225     while True:
226
227         if background_thread.calibration_requested:
```

```
228             print('calibrating')
229             background_thread.calibration_requested = False
230             saverages = np.average(pool.readers[0].samples,axis=1)
231             offsets = np.ones((4,2000))
232             for i in range(4):
233                 print('saverge' + str(i) + ' ' + str(saverages[i]))
234                 offsets[i,:] = offsets[i,:]*saverages[i]
235             calibrated_yet = True
236         elif background_thread.vcalibration_requested:
237             background_thread.vcalibration_requested = False
238             fvol = integrate_voltage(pool.readers[0].samples[0,:])
239             sigvol_abs=(1/N)*np.abs(fft(pool.readers[0].samples
    [0,:]))
240             fvol_abs = (1/N)*np.abs(fvol)
241             vol_gain = 170/(2*fvol_abs[v_maxind])
242             vol_min = fvol_abs[v_maxind]
243         elif background_thread.sync_requested:
244
245             background_thread.sync_requested = False
246             for i in range(len(background_thread.sync_code)):
247                 if background_thread.sync_code[i] == "1":
248                     sensors_to_use[i] = 1
249                 else:
250                     sensors_to_use[i] = 0
251
252             if sum(sensors_to_use) < 4:
253                 print("Cannot calculate estimate with so few sensors
    !")
254             else:
255                 #strip down matrix:
256                 for i in range(12):
257                     if sensors_to_use[i] == 1:
258                         thematrix = mymatrix[None,i,:]
259                         break
260                 for j in range(i+1,12):
261                     if sensors_to_use[j] == 1:
```

```
                        thematrix = np.concatenate((thematrix,
mymatrix[None,j,:]),axis=0)
                print(thematrix)

            '''anglediff = np.angle(fft1)[max1]-np.angle(fft2)[max1]
            if anglediff < -np.pi:
                anglediff += 2*np.pi
            elif anglediff > np.pi:
                anglediff -= 2*np.pi
            return (anglediff)/(2*np.pi*index2Hz(500,max1,10000))'''
        else:

            pool.scan_all()
            process_cntr += 1

            if process_cntr >= 5:
                process_cntr = 0
                print('processing data')

                '''Take FT'''
                #TODO: subtract bias?
                ftc0 = fft(pool.readers[0].samples-offsets0,axis=1)
                ftc1 = fft(pool.readers[1].samples-offsets1,axis=1)

                '''Time shift signals'''
                for i in range(8):
                    ftc0[x] = ftc0[x]*linear_shift0[x]
                for i in range(8):
                    ftc1[x] = ftc1[x]*linear_shift1[usb231map[x]]

                '''Estimate Voltage 1'''
                electrode_diff1 = pool.readers[1].samples[0,:]-pool.
readers[1].samples[1,:]

                estVolFreq1, estVolMag1, estVol1 = process_voltage(
electrode_diff1,sf,N)
```

```
295
296             abs_electrodeft1=(1/N)*np.abs(fft(electrode_diff1))
297             electrodeFreqInd1 = np.argmax(abs_electrodeft1)
298             electrodeFreq1 = ft_display.index2Hz(sf/2,
    electrodeFreqInd1,N)
299
300             '''Estimate Voltage 2'''
301             electrode_diff2 = pool.readers[0].samples[0,:]-pool.
    readers[1].samples[1,:]
302
303             estVolFreq2, estVolMag2, estVol2 = process_voltage(
    electrode_diff2,sf,N)
304
305             abs_electrodeft2=(1/N)*np.abs(fft(electrode_diff2))
306             electrodeFreqInd2 = np.argmax(abs_electrodeft2)
307             electrodeFreq2 = ft_display.index2Hz(sf/2,
    electrodeFreqInd2,N)
308
309             #Prepare sensor readings
310             magftc = np.concatenate((ftc0[2:,:],ftc1[2:,:]),axis
    =0)
311             abs_magftc = (1/N)*np.abs(magftc)
312
313             #NOTE: using freq of channel 4 to get max (wait why
    ?)
314             ftc_maxind = np.argmax(abs_magftc[4,:])
315             ftc_maxhz = ft_display.index2Hz(sf/2,ftc_maxind,N)
316
317             currents_ls = np.zeros((4,N))
318             curr_est,_,_ = processor.ls_estimate(abs_magftc)
319             currents_ls[0] = ifft(curr_est[0])
320             currents_ls[1] = ifft(curr_est[1])
321             currents_ls[2] = ifft(curr_est[2])
322
323             stringholder.data = ""
324
```

```
325                 #Add Electrode Difference 1
326                 stringholder.data += '{:.3f}'.format(2*
    abs_electrodeft1[electrodeFreqInd1]) +'V @ ' + str(electrodeFreq1
    ) + 'Hz_'
327                 #Add Electrode Difference 2
328                 stringholder.data += '{:.3f}'.format(2*
    abs_electrodeft2[electrodeFreqInd2]) +'V @ ' + str(electrodeFreq2
    ) + 'Hz_'
329
330                 #Add Magnetic Field Values
331                 for i in range(12):
332                     stringholder.data += '{:.3f}'.format(1000*2*
    abs_magftc[i][ftc_maxind]/4.88) + 'uT @ ' + str(ftc_maxhz) + 'Hz_
    '
333
334                 #Add 3 Currents
335                 curr0mag = 2*(1/N)*np.abs(curr_est[0][ftc_maxind])
336                 curr1mag = 2*(1/N)*np.abs(curr_est[1][ftc_maxind])
337                 curr2mag = 2*(1/N)*np.abs(curr_est[1][ftc_maxind])
338                 stringholder.data += '{:.3f}'.format(curr0mag) + 'A
    @ ' + str(ftc_maxhz) + 'Hz_'
339                 stringholder.data += '{:.3f}'.format(curr1mag) + 'A
    @' + str(ftc_maxhz) + 'Hz_'
340                 stringholder.data += '{:.3f}'.format(curr2mag) + 'A
    @' + str(ftc_maxhz) + 'Hz_'
341
342                 #Earth's magnetic field
343                 stringholder.data += '{:.3f}'.format(curr_est[3][0].
    real/4.88) + ' mT_'
344
345                 #Add first voltage difference
346                 if 2*estVolMag1 > vol_min:
347                     estVol1 = estVol1*(169.7/(2*estVolMag1))
348                     stringholder.data += '{:.1f}'.format(vol_gain*2*
    estVolMag1) + 'V @' + str(estVolFreq1) + 'Hz_'
349                 else:
```

```
                    estVol1 = np.zeros(N)
                    stringholder.data += '{:.3f}'.format(0) + ' V_'

                #Add second voltage difference
                if 2*estVolMag2 > vol_min:
                    estVol2 = estVol2*(169.7/(2*estVolMag2))
                    stringholder.data += '{:.1f}'.format(vol_gain*2*
    estVolMag2) + 'V @' + str(estVolFreq2) + 'Hz_'
                else:
                    estVol2 = np.zeros(N)
                    stringholder.data += '{:.3f}'.format(0) + ' V_'

                for i in range(3):
                    stringholder.data += np.array2string(currents_ls
    [0,:], formatter={'float_kind':lambda x: "%.2f" % x}) + "_"

        time.sleep(.1)
```

data_displayer.py

```
from __future__ import absolute_import, division, print_function


from tkinter.ttk import Combobox  # @UnresolvedImport

import tkinter as tk
from scipy.fftpack import fft, ifft
import numpy as np
import math
import utilities.data_loader

import datetime

''''To improve performance, read the following stack over flow
    question (has not yet been implemented):
https://stackoverflow.com/questions/11874767/how-do-i-plot-in-real-
    time-in-a-while-loop-using-matplotlib
```

```
'''

'''
Main GUI application to display estimated currents and voltages in
    real time.
'''
import threading
import time

import socket
import sys

import custom_gui.tksimpledialog

from matplotlib.backends.backend_tkagg import (FigureCanvasTkAgg,
    NavigationToolbar2Tk)
# Implement the default Matplotlib key bindings.
from matplotlib.backend_bases import key_press_handler
from matplotlib.figure import Figure


class ThreadingExample(object):

    def __init__(self, interval=.1):

        self.interval = interval
        thread = threading.Thread(target=self.run, args=())
        thread.daemon = True
        thread.start()
        self.received_data = ""
        self.cmd = "get"

    def run(self):
        # Create a UDP socket
        sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

```

```
50          server_address = ('localhost', 10000)
51
52          while True:
53              if self.cmd == 'cal':
54                  sent = sock.sendto(bytes('cal','utf-8'),
    server_address)
55                  self.cmd = 'get'
56              elif self.cmd == 'vcal':
57                  sent = sock.sendto(bytes('vcal','utf-8'),
    server_address)
58                  self.cmd = 'get'
59              elif self.cmd[0:4] == 'sync':
60                  sent = sock.sendto(bytes(self.cmd,'utf-8'),
    server_address)
61                  self.cmd = 'get'
62              else:
63                  #print('sending get...')
64                  sent = sock.sendto(bytes('get','utf-8'),
    server_address)
65
66              data, server = sock.recvfrom(4096)
67              self.received_data = str(data)
68              time.sleep(.1)
69
70  class MyDialog(custom_gui.tksimpledialog.Dialog):
71
72      def body(self, master):
73          self.sensor_readings = tk.StringVar()
74          self.sensor_readings.set('hide')
75
76          self.estimates = tk.StringVar()
77          self.estimates.set('hide')
78
79          tk.Label(master, text="Sensor Displays:").grid(row=0)
80          tk.Radiobutton(master,text="Hide",padx = 20,variable=self.
    sensor_readings,value='hide').grid(row=1)
```

```
81          tk.Radiobutton(master,text="Average",padx = 20,variable=self
    .sensor_readings,value='avg').grid(row=2)
82          tk.Radiobutton(master,text="Time Function",padx = 20,
    variable=self.sensor_readings,value='time').grid(row=3)
83          tk.Radiobutton(master,text="FT Display",padx = 20,variable=
    self.sensor_readings,value='ft').grid(row=4)
84
85          tk.Label(master, text="Estimate Displays:").grid(row=5)
86          tk.Radiobutton(master,text="Hide",padx = 20,variable=self.
    estimates,value='hide').grid(row=6)
87          tk.Radiobutton(master,text="Average",padx = 20,variable=self
    .estimates,value='avg').grid(row=7)
88          tk.Radiobutton(master,text="Time Function",padx = 20,
    variable=self.estimates,value='time').grid(row=8)
89          tk.Radiobutton(master,text="FT Display",padx = 20,variable=
    self.estimates,value='ft').grid(row=9)
90
91          return None
92
93      def apply(self):
94          pass
95
96  class VIn01(object):
97      def __init__(self, master):
98          self.master = master
99
100         self.background_thread = ThreadingExample()
101
102         self.create_widgets()
103
104     def update_value(self):
105         ocntr = 0
106         outputs = self.background_thread.received_data[2:].split('_'
    )
107
108         for i in range(2):
```

```
109             if ocntr < len(outputs):
110                 self.volSensorReadings[i]["text"] = outputs[ocntr]
111                 ocntr+=1
112 
113         for i in range(12):
114             if ocntr < len(outputs):
115                 self.magSensorReadings[i]["text"] = outputs[ocntr]
116                 ocntr+=1
117 
118         for i in range(3):
119             if ocntr < len(outputs):
120                 self.currentLabels[i]["text"] = outputs[ocntr]
121                 ocntr+=1
122 
123         if ocntr < len(outputs):
124             self.resLabel["text"] = "Ambient Magnetic Field: " +
    outputs[ocntr]
125             ocntr+=1
126 
127         for i in range(2):
128             if ocntr < len(outputs):
129                 self.voltage_est_labels[i]["text"] = outputs[ocntr]
130                 ocntr+=1
131 
132         for i in range(3):
133             if self.estimate_display == 'ft':
134                 self.currentLines[i].set_ydata(np.fromstring(outputs
    [ocntr][1:-1],dtype=float,sep=' ',count=2000))
135                 self.currentFigures[i].canvas.draw()
136                 self.currentFigures[i].canvas.flush_events()
137             ocntr+=1
138 
139         if self.running:
140             self.master.after(500, self.update_value)
141 
142 
```

```
143     def stop(self):
144         self.running = False
145         self.start_button["command"] = self.start
146         self.start_button["text"] = "Start"
147 
148     def start(self):
149         self.running = True
150 
151 
152         self.start_button["command"] = self.stop
153         self.start_button["text"] = "Stop"
154         self.update_value()
155 
156     def get_channel_num(self):
157         return 0
158 
159     def validate_channel_entry(self, p):
160         if p == '':
161             return True
162         try:
163             value = int(p)
164             if(value < 0 or value > self.ai_props.num_ai_chans - 1):
165                 return False
166         except ValueError:
167             return False
168         return True
169 
170     def set_bias(self):
171         self.background_thread.cmd = 'cal'
172 
173     def set_vbias(self):
174         self.background_thread.cmd = 'vcal'
175 
176     def set_sync(self):
177         state_string = ""
178         for i in range(12):
```

```
179             state_string += str(self.intvars[i].get())
180         self.background_thread.cmd = 'sync' + state_string
181 
182     def open_display_settings(self):
183         d = MyDialog(self.master)
184         self.sensor_reading_display = str(d.sensor_readings.get())
185         self.estimate_display = str(d.estimates.get())
186         self.update_displays()
187 
188     def update_displays(self):
189         for i in range(2):
190             self.volSensorReadings[i].grid_remove()
191             self.volCanvases[i].get_tk_widget().grid_remove()
192         for i in range(12):
193             self.magSensorReadings[i].grid_remove()
194             self.magCanvases[i].get_tk_widget().grid_remove()
195 
196         for i in range(3):
197             self.currentLabels[i].grid_remove()
198             self.currentCanvases[i].get_tk_widget().grid_remove()
199 
200         for i in range(2):
201             self.voltage_est_labels[i].grid_remove()
202             self.voltage_est_canvases[i].get_tk_widget().grid_remove
    ()
203 
204         if self.sensor_reading_display == 'ft' or self.
    sensor_reading_display == 'time':
205             for i in range(2):
206                 self.volCanvases[i].get_tk_widget().grid()
207             for i in range(12):
208                 self.magCanvases[i].get_tk_widget().grid()
209         elif self.sensor_reading_display == 'avg':
210             for i in range(2):
211                 self.volSensorReadings[i].grid()
212             for i in range(12):
```

```
213                 self.magSensorReadings[i].grid()
214
215         if self.estimate_display == 'ft' or self.estimate_display ==
    'time':
216             for i in range(3):
217                 self.currentCanvases[i].get_tk_widget().grid()
218             for i in range(2):
219                 self.voltage_est_canvases[i].get_tk_widget().grid()
220         elif self.estimate_display == 'avg':
221             for i in range(3):
222                 self.currentLabels[i].grid()
223             for i in range(2):
224                 self.voltage_est_labels[i].grid()
225
226     def test_function(self):
227         pass
228
229     def create_widgets(self):
230         self.sensor_reading_display = 'avg'
231         self.estimate_display = 'avg'
232
233         button_frame = tk.Frame(self.master)
234         button_frame.pack(fill=tk.X)
235
236         self.start_button = tk.Button(button_frame)
237         self.start_button["text"] = "Start"
238         self.start_button["command"] = self.start
239         self.start_button.grid(row=0, column=0, padx=3, pady=3)
240
241         self.calibrate_button = tk.Button(button_frame)
242         self.calibrate_button["text"] = "Calibrate"
243         self.calibrate_button["command"] = self.set_bias
244         self.calibrate_button.grid(row=0, column=1, padx=3, pady=3)
245
246         self.vcalibrate_button = tk.Button(button_frame)
247         self.vcalibrate_button["text"] = "Vol Cal"
```

```
248         self.vcalibrate_button["command"] = self.set_vbias
249         self.vcalibrate_button.grid(row=0, column=2, padx=3, pady=3)
250 
251         self.sync_button = tk.Button(button_frame)
252         self.sync_button["text"] = "Sync"
253         self.sync_button["command"] = self.set_sync
254         self.sync_button.grid(row=0, column=3, padx=3, pady=3)
255 
256         quit_button = tk.Button(button_frame)
257         quit_button["text"] = "Quit"
258         quit_button["command"] = self.master.destroy
259         quit_button.grid(row=0, column=4, padx=3, pady=3)
260 
261         self.doptions_button = tk.Button(button_frame)
262         self.doptions_button["text"] = "Display Options"
263         self.doptions_button["command"] = self.open_display_settings
264         self.doptions_button.grid(row=0, column=5, padx=3, pady=3)
265 
266         self.test_button = tk.Button(button_frame)
267         self.test_button["text"] = "Test"
268         self.test_button["command"] = self.test_function
269         self.test_button.grid(row=0, column=6, padx=3, pady=3)
270 
271         checkbox_frame = tk.Frame(self.master)
272         checkbox_frame.pack(fill=tk.X)
273 
274         self.intvars = []
275         for i in range(12):
276             self.intvars.append(tk.IntVar())
277             self.intvars[i].set(1)
278             tk.Checkbutton(checkbox_frame, text="Sensor " + str(i),
    variable=self.intvars[i]).grid(row=0, column=i)
279 
280         sensor_readings1 = tk.Frame(self.master)
281         sensor_readings1.pack(anchor=tk.CENTER,pady=20)
282 
```

```
283         sensor_readings2 = tk.Frame(self.master)
284         sensor_readings2.pack(anchor=tk.CENTER,pady=20)
285 
286         sensor_readings3 = tk.Frame(self.master)
287         sensor_readings3.pack(anchor=tk.CENTER,pady=20)
288 
289         title_size = 24
290         readings_size = 28
291         cur_width = 400
292 
293         self.magSensorReadings = []
294         self.magCanvases = []
295         self.magFigures = []
296         self.magLines = []
297 
298 
299         self.volSensorReadings = []
300         self.volCanvases = []
301         self.volFigures = []
302         self.volLines=[]
303 
304         for i in range(2):
305             tk.Label(sensor_readings1,text=("Vol. Probe "+ str(i)),
    font=("Helvetica", 14,"bold"),width=10).grid(row=0,column=i,padx
    =30)
306 
307             vol = tk.Label(sensor_readings1,text="---",font=("
    Helvetica", 16))
308             vol.grid(row=1,column=i,padx=30)
309             self.volSensorReadings.append(vol)
310 
311             fig = Figure(figsize=(2, 1.5), dpi=100)
312             self.volFigures.append(fig)
313             self.t = np.arange(0, 3, .01)
314             y = 2*(np.sin(2*np.pi*self.t))
315             line1, = fig.add_subplot(111).plot(self.t, y)
```

```
316             self.volLines.append(line1)
317 
318             canvas = FigureCanvasTkAgg(fig, master=sensor_readings1)
      # A tk.DrawingArea.
319             canvas.draw()
320             canvas.get_tk_widget().grid(row=2,column=i)
321             self.volCanvases.append(canvas)
322 
323 
324         for i in range(6):
325             tk.Label(sensor_readings2,text=("Mag. Field "+ str(i)),
    font=("Helvetica", 14,"bold"),width=10).grid(row=0,column=i+1,
    padx=30)
326 
327             vol = tk.Label(sensor_readings2,text="---",font=("
    Helvetica", 16))
328             vol.grid(row=1,column=i+1,padx=30)
329             self.magSensorReadings.append(vol)
330 
331             fig = Figure(figsize=(2, 1.5), dpi=100)
332             self.magFigures.append(fig)
333             self.t = np.arange(0, 3, .01)
334             y = 2*(np.sin(2*np.pi*self.t))
335             line1, = fig.add_subplot(111).plot(self.t, y)
336             self.magLines.append(line1)
337 
338             canvas = FigureCanvasTkAgg(fig, master=sensor_readings2)
      # A tk.DrawingArea.
339             canvas.draw()
340             canvas.get_tk_widget().grid(row=2,column=i+1)
341             self.magCanvases.append(canvas)
342 
343         for i in range(6):
344             tk.Label(sensor_readings3,text=("Mag. Field "+ str(i)),
    font=("Helvetica", 14,"bold"),width=10).grid(row=0,column=i+1,
    padx=30)
```

```
345
346             vol = tk.Label(sensor_readings3,text="---",font=("
    Helvetica", 16))
347             vol.grid(row=1,column=i+1,padx=30)
348             self.magSensorReadings.append(vol)
349
350             fig = Figure(figsize=(2, 1.5), dpi=100)
351             self.magFigures.append(fig)
352             self.t = np.arange(0, 3, .01)
353             y = 2*(np.sin(2*np.pi*self.t))
354             line1, = fig.add_subplot(111).plot(self.t, y)
355             self.magLines.append(line1)
356
357             canvas = FigureCanvasTkAgg(fig, master=sensor_readings3)
      # A tk.DrawingArea.
358             canvas.draw()
359             canvas.get_tk_widget().grid(row=2,column=i+1)
360             self.magCanvases.append(canvas)
361
362         currents = tk.Frame(self.master,pady=20)
363         currents.pack(anchor=tk.CENTER,expand=True)
364         currents.grid_columnconfigure(0, minsize=cur_width)
365         currents.grid_columnconfigure(1, minsize=cur_width)
366         currents.grid_columnconfigure(2, minsize=cur_width)
367
368         self.currentLabels = []
369         self.currentCanvases = []
370         self.currentFigures = []
371         self.currentLines = []
372
373         for i in range(3):
374             tk.Label(currents,text=("Current "+ str(i)),font=("
    Helvetica", title_size,"bold")).grid(row=0,column=i)
375
376             label = tk.Label(currents,text="---",font=("Helvetica",
    readings_size))
```

```
377             label.grid(row=1,column=i)
378             self.currentLabels.append(label)
379 
380             fig = Figure(figsize=(5, 4), dpi=100)
381             self.currentFigures.append(fig)
382             self.t = np.linspace(.001,2.0,num=2000)
383             y = self.t*10
384             line1, = fig.add_subplot(111).plot(self.t, y)
385             self.currentLines.append(line1)
386 
387             canvas = FigureCanvasTkAgg(fig, master=currents)  # A tk
    .DrawingArea.
388             canvas.draw()
389             canvas.get_tk_widget().grid(row=2,column=i)
390             self.currentCanvases.append(canvas)
391 
392         earth_frame = tk.Frame(self.master,pady=10)
393         earth_frame.pack(fill=tk.X,anchor=tk.CENTER)
394 
395         self.resLabel = tk.Label(earth_frame,text=("Ambient Magnetic
     Field:"),font=("Helvetica", 20))
396         self.resLabel.grid(row=1,column=1)
397         self.resLabel.grid_remove()
398 
399         voltage_est_frame = tk.Frame(self.master,pady=20)
400         voltage_est_frame.pack(anchor=tk.CENTER, expand=True)
401 
402         self.voltage_est_labels = []
403         self.voltage_est_canvases = []
404         self.voltage_est_lines = []
405         self.voltage_est_figures = []
406 
407         for i in range(2):
408             tk.Label(voltage_est_frame,text=("Voltage" + str(i)),
    font=("Helvetica", title_size,"bold")).grid(row=0,column=i)
409             label = tk.Label(voltage_est_frame,text="---",font=("
```

```
Helvetica", readings_size))
410             label.grid(row=1,column=i)
411             self.voltage_est_labels.append(label)
412 
413             fig = Figure(figsize=(2, 1.5), dpi=100)
414             self.voltage_est_figures.append(fig)
415             self.t = np.arange(0, 3, .01)
416             y = 2*(np.sin(2*np.pi*self.t))
417             line1, = fig.add_subplot(111).plot(self.t, y)
418             self.voltage_est_lines.append(line1)
419 
420             canvas = FigureCanvasTkAgg(fig, master=voltage_est_frame
    )  # A tk.DrawingArea.
421             canvas.draw()
422             canvas.get_tk_widget().grid(row=2,column=i)
423             self.voltage_est_canvases.append(canvas)
424 
425         power_est_frame = tk.Frame(self.master,pady=20)
426         power_est_frame.pack(anchor=tk.CENTER,expand=True)
427 
428         power_est_frame.grid_columnconfigure(0, minsize=400)
429         power_est_frame.grid_columnconfigure(1, minsize=400)
430 
431         tk.Label(power_est_frame,text=("Left Bulb Power"),font=("
    Helvetica", title_size,"bold")).grid(row=0,column=0)
432 
433         tk.Label(power_est_frame,text=("Right Bulb Power"),font=("
    Helvetica", title_size,"bold")).grid(row=0,column=1)
434 
435         self.lbpLabel = tk.Label(power_est_frame,text=("---"),font=(
    "Helvetica", readings_size))
436         self.lbpLabel.grid(row=1,column=0)
437 
438         self.rbpLabel = tk.Label(power_est_frame,text=("---"),font=(
    "Helvetica", readings_size))
439         self.rbpLabel.grid(row=1,column=1)
```

```
440
441         padFrame = tk.Frame(self.master, width = 25, height = 25,
    padx = 15, pady = 15)
442         padFrame.pack(fill = tk.X, expand = True, side = tk.BOTTOM)
443
444         self.update_displays()
445
446 if __name__ == "__main__":
447     VIn01(master=tk.Tk()).master.mainloop()
```

display_ft.py

```
1 import numpy as np
2 from scipy.fftpack import fft, ifft
3 import matplotlib.pyplot as plt
4 from sys import argv
5 from utilities.data_loader import DataLoader
6 import utilities.filter_util
7 from utilities.ft_util import FTUtils
8 from preprocessor import PreProcessor
9
10 '''Displays Fourier Transform of selected channel, or multiple hard-
    coded channels'''
11
12 def display_ft(x,sf,start_freq,end_freq,show=True,title="Fourier
    Transform",channel = 0,mylabel = ""):
13     nyquist_rate = sf/2.0
14     N = len(x)
15     start = FTUtils.hz2index(nyquist_rate,start_freq,len(x))
16     end = FTUtils.hz2index(nyquist_rate,end_freq,len(x))
17
18     y = fft(x)
19     magnitude_array = (1.0/len(x))*np.abs(y[start:end])
20     xf = np.linspace(FTUtils.index2Hz(nyquist_rate,start,N), FTUtils
    .index2Hz(nyquist_rate,end-1,N), end-start)
21     plt.title(title+' - Sampled at ' + str(int(sf)) + ' Hz Channel '
     + str(channel) )
```

```
22      plt.xlabel('Freq (Hz)')
23      plt.ylabel('Magnitude')
24
25      plt.plot(xf, magnitude_array,label='Channel ' + str(channel))
26      plt.grid()
27      if show:
28          plt.show()
29
30      angle_array = np.angle(y[start:end])
31      for x in range(angle_array.shape[0]):
32          if magnitude_array[x] < .1:
33              angle_array[x] = 0
34
35  def display_8ch_ft(samples,sf,start_freq,end_freq):
36      for i in range(8):
37          plt.figure(i)
38          display_ft(samples[i,:],sf,start_freq,end_freq,show=False,
    channel=i)
39      plt.show()
40
41  if __name__ == '__main__':
42      data_loader = DataLoader()
43      processor = PreProcessor()
44
45      samples = data_loader.read_ch_data(argv[1])
46
47      sf = float(argv[2])
48
49      samples[:8,:] = processor.shift_signals_USB205(samples[:8,:],sf)
50
51      ch = int(argv[3])
52
53      if len(argv) == 3:
54          display_8ch_ft(samples,sf,10.0,sf/2-10.0)
55      else:
56          display_ft(samples[ch,:],float(argv[2]),0.0,sf/2-10.0,show=
```

```
    True,title="Fourier Transform",channel=ch)
```

display_heatmap.py

```
import tkinter as tk
import simulator.sensor_placer
from simulator.sources import Wire
from simulator.sensors import LISensor
import numpy as np
import random
from estimate_current import CurrentEstimator

class Example(tk.Frame):
    '''Illustrate how to drag items on a Tkinter canvas'''

    def __init__(self, parent,location_array,group1,group2,group3):
        tk.Frame.__init__(self, parent)

        self.location_array = location_array

        self.estimator = CurrentEstimator()
        self.group1 = group1
        self.group2 = group2
        self.group3 = group3


        self.canvas_left = 100
        self.canvas_zero = 200
        self.cwidth = 800
        self.cheight = self.canvas_zero*2
        # create a canvas
        self.canvas = tk.Canvas(width=self.cwidth, height=self.
    cheight)
        self.canvas.pack(fill="both", expand=True)

        # this data is used to keep track of an
        # item being dragged
```

```
33          self._drag_data = {"x": 0, "item": None}
34          self._ydrag_data = {"y": 0, "item": None}
35
36          #create wires
37          color="black"
38          w1 = .015
39          w2 = w1+.015
40          w3 = w2+.015
41          self.canvas.create_oval(self.toCanvasX(w1)-25, self.
    toCanvasY(0)-25, self.toCanvasX(w1)+25, self.toCanvasY(0)+25,
    outline=color)
42          self.canvas.create_oval(self.toCanvasX(w2)-25, self.
    toCanvasY(0)-25, self.toCanvasX(w2)+25, self.toCanvasY(0)+25,
    outline=color)
43          self.canvas.create_oval(self.toCanvasX(w3)-25, self.
    toCanvasY(0)-25, self.toCanvasX(w3)+25, self.toCanvasY(0)+25,
    outline=color)
44          self.wires_ref = []
45          self.wires_ref.append(Wire().setLocation([w1,0,0.0]).
    setOrientation([0,0,1]).setCurrent([1]))
46          self.wires_ref.append(Wire().setLocation([w2,0,0.0]).
    setOrientation([0,0,1]).setCurrent([1]))
47          self.wires_ref.append(Wire().setLocation([w3,0,0.0]).
    setOrientation([0,0,1]).setCurrent([1]))
48
49          # create a couple of movable objects
50          for i in range(6):
51              self._create_token(self.location_array[i,:], "black",i)
52          for i in range(6,10):
53              self._create_ytoken(self.location_array[i,:],"black",i)
54
55          # add bindings for clicking, dragging and releasing over
56          # any object with the "token" tag
57          self.canvas.tag_bind("token", "<ButtonPress-1>", self.
    on_token_press)
58          self.canvas.tag_bind("token", "<ButtonRelease-1>", self.
```

```
    on_token_release)
59          self.canvas.tag_bind("token", "<B1-Motion>", self.
    on_token_motion)
60
61          self.canvas.tag_bind("ytoken", "<ButtonPress-1>", self.
    on_ytoken_press)
62          self.canvas.tag_bind("ytoken", "<ButtonRelease-1>", self.
    on_ytoken_release)
63          self.canvas.tag_bind("ytoken", "<B1-Motion>", self.
    on_ytoken_motion)
64
65      def _create_ytoken(self,coord,color,sensor_index):
66          self.canvas.create_rectangle(self.toCanvasX(coord[0])-5,
    self.toCanvasY(coord[1]), self.toCanvasX(coord[2])+5, self.
    toCanvasY(coord[3]), fill="white", outline=color, tags=("ytoken",
    str(sensor_index)))
67
68      def _create_token(self, coord, color,sensor_index):
69          '''Create a token at the given coordinate in the given color
    '''
70          self.canvas.create_rectangle(self.toCanvasX(coord[0]), self.
    toCanvasY(coord[1])-5, self.toCanvasX(coord[2]), self.toCanvasY(
    coord[3])+5, fill="white", outline=color, tags=("token",str(
    sensor_index)))
71
72      def on_token_press(self, event):
73          self._drag_data["item"] = self.canvas.find_closest(event.x,
    event.y)[0]
74          self._drag_data["x"] = event.x
75          self._drag_data["startx"] = event.x
76
77      def on_token_release(self, event):
78          self._drag_data["x"] = 0
79          delta_x = event.x - self._drag_data["startx"]
80          i = int(self.canvas.itemcget(self._drag_data["item"],"tags")
    .split(" ")[1])
```

```
81          self.location_array[i,0] = self.location_array[i,0] +
    delta_x/10000
82          self.location_array[i,2] = self.location_array[i,2] +
    delta_x/10000
83          self._drag_data["item"] = None
84
85      def on_token_motion(self, event):
86          delta_x = event.x - self._drag_data["x"]
87          self.canvas.move(self._drag_data["item"], delta_x, 0)
88          self._drag_data["x"] = event.x
89
90      def on_ytoken_press(self, event):
91          self._ydrag_data["item"] = self.canvas.find_closest(event.x,
     event.y)[0]
92          self._ydrag_data["y"] = event.y
93          self._ydrag_data["starty"] = event.y
94
95      def on_ytoken_release(self, event):
96          self._ydrag_data["y"] = 0
97          delta_y = event.y - self._ydrag_data["starty"]
98          i = int(self.canvas.itemcget(self._ydrag_data["item"],"tags"
    ).split(" ")[1])
99          self.location_array[i,1] = self.location_array[i,1] +
    delta_y/10000
100         self.location_array[i,3] = self.location_array[i,3] +
    delta_y/10000
101         self._ydrag_data["item"] = None
102
103     def on_ytoken_motion(self, event):
104         delta_y = event.y - self._ydrag_data["y"]
105         self.canvas.move(self._ydrag_data["item"], 0, delta_y)
106         self._ydrag_data["y"] = event.y
107
108     def _from_rgb(self,rgb):
109         """translates an rgb tuple of int to a tkinter friendly
    color code
```

```
110         """
111         return "#%02x%02x%02x" % rgb
112 
113     def toCanvasX(self,x):
114         return x*10000 + self.canvas_left
115 
116     def toSimX(self,x):
117         return (x - self.canvas_left)/10000.0
118 
119     def toCanvasY(self,y):
120         cny = y*10000 + self.canvas_zero
121         return self.cheight - cny
122 
123     def toSimY(self,y):
124         return (y - self.cheight + self.canvas_zero)/10000.0
125 
126     def run_sim(self):
127         sensor_array = []
128         num_sensors = 10
129         for i in range(num_sensors):
130             sensor_array.append(LISensor().setStart([self.
    location_array[i,0],self.location_array[i,1],0]).setEnd([self.
    location_array[i,2],self.location_array[i,3],0]))
131 
132         self.matrix = np.zeros((num_sensors,3))
133         for i in range(3):
134             for j in range(num_sensors):
135                 self.matrix[j][i] = sensor_array[j].detect([self.
    wires_ref[i]],0)
136 
137         print(self.matrix)
138 
139         self.estimator.setMatrix(self.matrix)
140         self.estimator.setSensorLocations(self.location_array,self.
    group1,self.group2,self.group3)
141         self.estimator.setEstimatorType('lap')
```

```
142         self.estimator.setM(3)
143 
144         maxscore = 0
145         maxloc = 0
146 
147         numx = 100.0
148         numy = 50.0
149 
150         print("Boundaries: cx < {} or cy < {} or cy > {} or cx > {}"
    .format(self.toSimX(-.01),self.toSimY(.015),self.toSimY(-.015),
    self.toSimX(.0575)))
151 
152         w=self.cwidth/numx
153         h =self.cheight/numy
154         for i in range(int(numx)):
155             for j in range(int(numy)):
156                 cx = i*self.cwidth/numx + w/2
157                 cy = j*self.cheight/numy + h/2
158 
159                 if (cx < self.toCanvasX(0.0025)) or (cy < self.
    toCanvasY(.01)) or (cy>self.toCanvasY(-.01)) or (cx>self.
    toCanvasX(.0575)):
160                     self.wires = []
161 
162                     self.wires.append(Wire().setLocation([self.
    toSimX(cx),self.toSimY(cy),0.0]).setOrientation([0,0,1]).
    setCurrent([1]))
163                     bfields = np.zeros((num_sensors,1))
164                     for k in range(num_sensors):
165                         bfields[k][0] = sensor_array[k].detect(self.
    wires,0)
166                     #del self.wires_ref[-1]
167                     est = self.estimator.getEstimate(bfields)
168                     score = np.mean(np.abs(est[:3,:]))
169                     g = int(score*1000)
170                     if score > maxscore:
```

```
171                         maxscore = score
172                         maxloc = (self.toSimX(cx),self.toSimY(cy))
173 
174                     if g > 255:
175                         g=255
176                     g=255-g
177                     self.canvas.create_rectangle(cx-w/2, cy-h/2, cx+
    w/2, cy+h/2,width=0.0, fill=self._from_rgb((255,g,g)))
178 
179             #for i in range(255):
180             #    self.canvas.create_rectangle(self.cwidth-30, i*1,
    self.cwidth, i*1+1, fill=self._from_rgb((255,i,i)), width=0.0)
181         print("Max score was {} at {}".format(maxscore,maxloc))
182 
183 def run_sim():
184     customcanvas.run_sim()
185 
186 if __name__ == "__main__":
187     root = tk.Tk()
188 
189 
190     button_frame = tk.Frame(root)
191     button_frame.pack(fill=tk.X)
192 
193     start_button = tk.Button(button_frame)
194     start_button["text"] = "Start"
195     start_button["command"] = run_sim
196     start_button.grid(row=0, column=0, padx=3, pady=3)
197     location_array,g1,g2,g3 = simulator.sensor_placer.
    create_location_array()
198 
199     customcanvas = Example(root,location_array,g1,g2,g3)
200     customcanvas.pack(fill="both", expand=True)
201     root.mainloop()
```

estimate_current.py

```
import numpy as np
from scipy.fftpack import fft, ifft
import matplotlib.pyplot as plt
from sys import argv
import utilities.data_loader
import time_shift
import os
from simulator.sensors import Sensor, LISensor
from simulator.sources import ConstantField,Wire,FiniteWireXZ
import simulator.sensor_placer
from sklearn.preprocessing import PolynomialFeatures

class CurrentEstimator:
    def __init__(self):
        self._estimate = self.olsEstimate
        self.PDEG = 1

    def setMatrix(self,matrix):
        self.A = matrix

    def setCovarianceMatrix(self,matrix):
        self.S = matrix

    def setRegressionModel(self,intercept,model,degree):
        self.reg_intercept = intercept
        self.reg_model = model
        self.PDEG = degree

    #Sensor Location array information is needed for both
   LaplaceEstimate and Ampere Estimate
    def setSensorLocations(self,location_array,group1,group2,group3)
   :
        maxM = location_array.shape[0]//2

        self.harmonics_matrix = np.zeros((location_array.shape[0],
   maxM*2))
```

```
34          for s in range(location_array.shape[0]):
35              x = (location_array[s,0]+location_array[s,2])/2
36              y = (location_array[s,1]+location_array[s,3])/2
37
38              phi = np.arctan2(y,x)
39              r = np.sqrt(x**2+y**2)
40
41              theta = np.arctan2(location_array[s,3]-location_array[s
    ,1],location_array[s,2]-location_array[s,0])
42              srn = np.cos(theta)*np.cos(phi)+np.sin(theta)*np.sin(phi
    )
43              spn = -np.cos(theta)*np.sin(phi)+np.sin(theta)*np.cos(
    phi)
44
45              for m in range(1,maxM+1):
46                  self.harmonics_matrix[s,m-1] = m*(r**(m-1))*(srn*np.
    cos(m*phi)-spn*np.sin(m*phi))
47                  self.harmonics_matrix[s,maxM+m-1] = m*(r**(m-1))*(
    srn*np.sin(m*phi)+spn*np.cos(m*phi))
48          self.groups = [group1,group2,group3]
49
50
51      def setM(self,M):
52          if M*2+3 > self.harmonics_matrix.shape[0]:
53              raise ValueError("M too high")
54
55          if self.harmonics_matrix is None:
56              raise ValueError("Harmonics Matrix not set yet.")
57
58          num_sensors = self.A.shape[0]
59
60          maxM = self.harmonics_matrix.shape[1]//2
61          self.supermatrix = np.concatenate((self.A,self.
    harmonics_matrix[:num_sensors,0,None]),axis=1)
62          self.supermatrix = np.concatenate((self.supermatrix,self.
    harmonics_matrix[:num_sensors,maxM,None]),axis=1)
```

```
63
64          for i in range(1,M):
65              self.supermatrix = np.concatenate((self.supermatrix,self
    .harmonics_matrix[:num_sensors,i,None]),axis=1)
66              self.supermatrix = np.concatenate((self.supermatrix,self
    .harmonics_matrix[:num_sensors,maxM+i,None]),axis=1)
67
68
69      def olsEstimate(self,b):
70          x,r,rank,s = np.linalg.lstsq(self.A,b)
71          return x
72
73      def bluEstimate(self,b):
74          invS = np.linalg.inv(self.S)
75          firstterm = np.linalg.inv(np.matmul(np.matmul(self.A.T,invS)
    ,self.A))
76          return np.matmul(np.matmul(np.matmul(firstterm,self.A.T),
    invS),b)
77
78      def ampereEstimate(self,b):
79          x = np.zeros((3,1))
80          for i in range(3):
81              atotal = 0
82              btotal = 0
83              group = self.groups[i]
84              for j in range(len(group)):
85                  sensor_index = group[j][0]
86                  sensor_direction = group[j][1]
87                  if sensor_index < self.A.shape[0]:
88                      btotal += b[sensor_index,0]*sensor_direction
89                      atotal += self.A[sensor_index,i]*
    sensor_direction
90              x[i,0] = btotal/atotal
91          return x
92
93      def regressionEstimate(self,b):
```

```
94          poly_features = PolynomialFeatures(degree=self.PDEG,
    include_bias=False)
95          readings_POLY = poly_features.fit_transform(b.T)
96
97          intercept_matrix = np.ones((3,b.shape[1]))
98          for i in range(len(self.reg_intercept)):
99              intercept_matrix[i,:] = intercept_matrix[i,:]*self.
    reg_intercept[i]
100
101         x = np.matmul(self.reg_model,readings_POLY.T) +
    intercept_matrix
102
103         return x
104
105     def polynomialEstimate(self,b):
106         pass
107
108     def laplaceEstimate(self,b):
109         x,r,rank,s = np.linalg.lstsq(self.supermatrix,b,rcond
    =10**-20)
110         return x
111
112     def getEstimate(self,readings):
113         return self._estimate(readings)
114
115     def setEstimatorType(self,est_type):
116         if est_type == "blu":
117             self._estimate = self.bluEstimate
118         elif est_type == "ols":
119             self._estimate = self.olsEstimate
120         elif est_type == "amp":
121             self._estimate = self.ampereEstimate
122         elif est_type == "reg":
123             self._estimate = self.regressionEstimate
124         elif est_type == "pol":
125             self._estimate = self.polynomialEstimate
```

```
126         elif est_type == "lap":
127             self._estimate = self.laplaceEstimate
128         else:
129             raise ValueError("Unknown estimator type {}".format(
    est_type))
130 
131 def get_theoretical_matrix(sensors,wires):
132     th_matrix = np.zeros((len(sensors),3))
133     for x in range(len(sensors)):
134         for y in range(3):
135             th_matrix[x][y] = sensors[x].detect([wires[y]],0)
136     return th_matrix
137 
138 if __name__ == '__main__':
139 
140     loc_array,firstgroup,secondgroup,thirdgroup = simulator.
    sensor_placer.create_location_array()
141 
142     sensor_array = []
143     for i in range(10):
144         sensor_array.append(LISensor().setStart([loc_array[i,0],
    loc_array[i,1],0]).setEnd([loc_array[i,2],loc_array[i,3],0]))
145 
146     curs = [-.7,1,-.3]
147 
148     wires_ref = []
149     wires_ref.append(Wire().setLocation([.0075,0,0.0]).
    setOrientation([0,0,1]).setCurrent([1]))
150     wires_ref.append(Wire().setLocation([.0225,0,0.0]).
    setOrientation([0,0,1]).setCurrent([1]))
151     wires_ref.append(Wire().setLocation([.0375,0,0.0]).
    setOrientation([0,0,1]).setCurrent([1]))
152 
153     matrix = get_theoretical_matrix(sensor_array,wires_ref)
154 
155     estimator = CurrentEstimator()
```

```
156     estimator.setMatrix(matrix)
157     estimator.setSensorLocations(loc_array,firstgroup,secondgroup,
    thirdgroup)
```

estimate_voltage.py

```
1  import numpy as np
2  from scipy.fftpack import fft, ifft
3  import matplotlib.pyplot as plt
4  from sys import argv
5
6  from utilities.ft_util import FTUtils
7  import display_ft
8  import utilities.data_loader
9
10 '''full voltage estimation procedure:
11 -observe ft of detecting electrode
12 -choose frequency that is not present in output, and put that
13 into input
14 -observe output at that frequency and calculate C
15 -use C calculation to estimate voltage
16
17 '''
18 class VoltageEstimator:
19     def __init__(self):
20         pass
21
22     def setRCValues(self,R_1,C_1):
23         self.R_1 = R_1
24         self.C_1 = C_1
25
26     def calibrate(self,signal,cal_freq,nyquist_rate,cal_input):
27         cal_mag = (1/len(signal))*np.abs(y[FTUtils.hz2index(
    nyquist_rate,cal_freq,len(signal))])
28         self.C_1 = ((cal_mag*2)/cal_input)/(R_1*2*np.pi*500)*16.0
29
30     def estimate_voltage(self,signal,nyq):
```

```
        hz_array = np.zeros(signal.shape[0])
        for i in range(signal.shape[0]):
            hz_i = FTUtils.index2Hz(nyq,i,signal.shape[0])
            hz_array[i] = hz_i

        print("frequency array:")
        print(hz_array)

        transfer_function = 1j*2*np.pi*hz_array*self.R_1*self.C_1
        filter_function = 1/transfer_function

        for i in range(FTUtils.hz2index(nyq,20,signal.shape[0])):
            filter_function[i] = 0
            #print(i)
        for i in range(FTUtils.hz2index(nyq,-20,signal.shape[0])+1,
    signal.shape[0]):
            filter_function[i] = 0



        to_return = ifft(fft(signal)*filter_function).real
        return  to_return

    def integrate(self,data):
        avg1 = np.average(data)
        data = data - avg1
        mysum = 0
        for i in range(len(data)):
            mysum += data[i]
            data[i] = mysum

        avg2 = np.average(data)
        data = data - avg2

        return data*(1/(self.R_1*self.C_1))

```

```
66
67 if __name__ == '__main__':
68
69     cal_freq = 500
70     cal_input = 3.2
71     cal_channel = 0
72     print("Calibration File: {}".format(argv[1]))
73     samples = data_loader.read_ch_data(argv[1])[cal_channel,:]
74     sf = float(argv[3])
75
76     y = fft(samples)
77     numsamples = len(samples)
78     nyquist_rate = sf/2.0
79     cal_mag = (1/numsamples)*np.abs(y[FTUtils.hz2index(nyquist_rate,
       cal_freq,numsamples)])
80
81     R_1 = 470000
82     C_1 = 1e-12
83     C_1 = ((cal_mag*2)/cal_input)/(R_1*2*np.pi*500)*16.0
84
85     print("Calculating effective capacitance to be: {}".format(C_1))
86
87     estimator = VoltageEstimator()
88     estimator.setRCValues(R_1,C_1)
89     myestimate = estimator.estimate_voltage(samples1,sf/2.0)
90     plt.plot(time,myestimate,label="Voltage Estimate")
91     plt.title("Voltage Estimate")
92     plt.ylabel("Voltage (V)")
93     plt.xlabel("Time (s)")
94     plt.show()
```

fit_gaussians.py

```
1 import numpy as np
2 from scipy.fftpack import fft, ifft
3 from scipy.stats import linregress
4 from scipy.optimize import curve_fit
```

```
import matplotlib.pyplot as plt
from sys import argv
import os
from utilities.data_loader import DataLoader

def gauss(x, *p):
    A, mu, sigma = p
    return A*np.exp(-(x-mu)**2/(2.*sigma**2))

samples = DataLoader().read_ch_data(argv[1])
for x in range(8,16):
    unique_values, counts = np.unique(samples[x,:],return_counts=
    True)
    unique_values = unique_values - np.mean(samples[x,:])
    print(unique_values)
    print(counts)

    if len(unique_values) > 2:

        coeff, var_matrix = curve_fit(gauss,unique_values,counts
    ,[4000,0,.0028])
        print("Height: " + str(coeff[0]) +  " Mean: " + str(coeff
    [1]) + " Std "+str(coeff[2]) + "\n")
        plt.figure(x)
        xaxis = np.linspace(unique_values[0],unique_values[-1],100)
        plt.plot(xaxis,gauss(xaxis,*coeff),'r-')
        plt.bar(unique_values, counts,.0002)
        plt.title('Channel ' + str(x))
        plt.xlabel('Deviation from Mean (V)')
        plt.ylabel('Num. Readings')
        plt.show()
    else:
        print('Not more than 2 unique values')
```

linear_fitting.py

```
import numpy as np
```

```
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import PolynomialFeatures
import data_loader
from sys import argv
import sensor_placer
from sensors import Sensor, LISensor
from sources import ConstantField,Wire,FiniteWireXZ
import matplotlib.pyplot as plt

PDEG = 1
tot_sensors=26

length = int((26-4)/2)
axisnsensors = np.linspace(6,26,length)
print(axisnsensors)
errort0 = np.zeros(length)
errort1 = np.zeros(length)
errort2 = np.zeros(length)
errort3 = np.zeros(length)
cntr = 0

def get_random_range(m,width,offset=0):
    return width*np.random.rand(m, 1) - (width/2) + offset

#GENERATE TRAINING PARAMETERS
num_train = 1000
X_1 = get_random_range(num_train,10)#x1.reshape((dimen,1))
X_2 =  get_random_range(num_train,10)#x2.reshape((dimen,1))
X_3 = get_random_range(num_train,10)#x3.reshape((dimen,1))
X_4 = get_random_range(num_train,10)#x3.reshape((dimen,1))
X_5 = get_random_range(num_train,.065,offset=.065/2-.01)#x3.reshape
   ((dimen,1))
X_6 = get_random_range(num_train,.04)#x3.reshape((dimen,1))

X_internal = np.concatenate((X_1,X_2,X_3),axis=1)
X_external = np.concatenate((X_4,X_5,X_6),axis=1)
```

```
37
38 poly_features = PolynomialFeatures(degree=PDEG, include_bias=False)
39 x_external_POLY = poly_features.fit_transform(X_external)
40 X = np.concatenate((X_internal,x_external_POLY),axis=1)
41
42 lin_reg = LinearRegression()
43
44 #CREATE SENSORS
45 loc_array,firstgroup,secondgroup,thirdgroup = sensor_placer.
       create_location_array()
46 sensor_array = []
47 for i in range(loc_array.shape[0]):
48     sensor_array.append(LISensor().setStart([loc_array[i,0],
       loc_array[i,1],0]).setEnd([loc_array[i,2],loc_array[i,3],0]))
49
50 #PLACE WIRES
51 wires = []
52 wires.append(Wire().setLocation([.0075,0,0.0]).setOrientation
       ([0,0,1]).setCurrent(X_1))
53 wires.append(Wire().setLocation([.0225,0,0.0]).setOrientation
       ([0,0,1]).setCurrent(X_2))
54 wires.append(Wire().setLocation([.0375,0,0.0]).setOrientation
       ([0,0,1]).setCurrent(X_3))
55
56 curs = [-.7,1,-.3]
57
58 h = sensor_placer.h
59
60 wires_ref = []
61 wires_ref.append(Wire().setLocation([.0075,0,0.0]).setOrientation
       ([0,0,1]).setCurrent([1]))
62 wires_ref.append(Wire().setLocation([.0225,0,0.0]).setOrientation
       ([0,0,1]).setCurrent([1]))
63 wires_ref.append(Wire().setLocation([.0375,0,0.0]).setOrientation
       ([0,0,1]).setCurrent([1]))
64
```

```
wires_zero = []
wires_zero.append(Wire().setLocation([.0075,0,0.0]).setOrientation([0,0,1]).setCurrent([curs[0]]))
wires_zero.append(Wire().setLocation([.0225,0,0.0]).setOrientation([0,0,1]).setCurrent([curs[1]]))
wires_zero.append(Wire().setLocation([.0375,0,0.0]).setOrientation([0,0,1]).setCurrent([curs[2]]))

wires_one = []
wires_one.append(Wire().setLocation([.0075,0,0.0]).setOrientation([0,0,1]).setCurrent([curs[0]]))
wires_one.append(Wire().setLocation([.0225,0,0.0]).setOrientation([0,0,1]).setCurrent([curs[1]]))
wires_one.append(Wire().setLocation([.0375,0,0.0]).setOrientation([0,0,1]).setCurrent([curs[2]]))
wires_one.append(Wire().setLocation([.0225,3*h,0.0]).setOrientation([0,0,1]).setCurrent([[1]]))

wires_ttwo = []
wires_ttwo.append(Wire().setLocation([.0075,0,0.0]).setOrientation([0,0,1]).setCurrent([curs[0]]))
wires_ttwo.append(Wire().setLocation([.0225,0,0.0]).setOrientation([0,0,1]).setCurrent([curs[1]]))
wires_ttwo.append(Wire().setLocation([.0375,0,0.0]).setOrientation([0,0,1]).setCurrent([curs[2]]))
wires_ttwo.append(Wire().setLocation([.0075,4*h,0.0]).setOrientation([0,0,1]).setCurrent([-1*curs[0]]))
wires_ttwo.append(Wire().setLocation([.0225,4*h,0.0]).setOrientation([0,0,1]).setCurrent([-1*curs[1]]))
wires_ttwo.append(Wire().setLocation([.0375,4*h,0.0]).setOrientation([0,0,1]).setCurrent([-1*curs[2]]))

wires_tthree = []
wires_tthree.append(Wire().setLocation([.0075,0,0.0]).setOrientation([0,0,1]).setCurrent([curs[0]]))
wires_tthree.append(Wire().setLocation([.0225,0,0.0]).setOrientation
```

```
    ([0,0,1]).setCurrent([curs[1]]))
wires_tthree.append(Wire().setLocation([.0375,0,0.0]).setOrientation
    ([0,0,1]).setCurrent([curs[2]]))
wires_tthree.append(Wire().setLocation([.0075,3*h,0.0]).
    setOrientation([0,0,1]).setCurrent([curs[0]]))
wires_tthree.append(Wire().setLocation([.0225,3*h,0.0]).
    setOrientation([0,0,1]).setCurrent([curs[1]]))
wires_tthree.append(Wire().setLocation([.0375,3*h,0.0]).
    setOrientation([0,0,1]).setCurrent([curs[2]]))
wires_tthree.append(Wire().setLocation([.006,-3*h,0.0]).
    setOrientation([0,0,1]).setCurrent([curs[0]]))
wires_tthree.append(Wire().setLocation([.030,-3*h,0.0]).
    setOrientation([0,0,1]).setCurrent([curs[1]]))
wires_tthree.append(Wire().setLocation([.04,-3*h,0.0]).
    setOrientation([0,0,1]).setCurrent([curs[2]]))


for s in range(6,tot_sensors+1):
    if s%2==0:
        print("now doing number of sensors: {0}".format(s))
        #Define 1000x10 matrix
        readings = np.zeros((s,num_train))
        for i in range(s):
            print("now doing sensor {0}".format(i))
            for j in range(num_train):
                wires.append(Wire().setLocation([X_5[j][0],X_6[j
    ][0],0.0]).setOrientation([0,0,1]).setCurrent(X_4))
                readings[i][j] = sensor_array[i].detect(wires,j)
                wires.pop()

        lin_reg = LinearRegression()
        lin_reg.fit(X, readings.T)

        max_axis = lin_reg.coef_.max(axis=0)

        argsortcoef = np.argsort(np.abs(max_axis[3:]))
```

```
114         sparse_coef = argsortcoef[-(s-4):]
115         sparse_coef = np.concatenate((np.array([0,1,2]),sparse_coef
    +3))
116 
117         matrix = lin_reg.coef_[:,sparse_coef]
118 
119         #Test Case 1
120         readings = np.zeros((s,1))
121         for i in range(s):
122             readings[i][0] = sensor_array[i].detect(wires_zero,0)
123         x,_,_,_ = np.linalg.lstsq(matrix,readings)
124         x=x.T
125         errort0[cntr] = np.mean(np.abs(np.array([[(x[0][0]-curs[0])/
    curs[0],(x[0][1]-curs[1])/curs[1],(x[0][2]-curs[2])/curs[2]]]).T)
    )
126 
127         #test 2
128         for i in range(s):
129             readings[i][0] = sensor_array[i].detect(wires_one,0)
130         x,_,_,_ = np.linalg.lstsq(matrix,readings)
131         x=x.T
132         errort1[cntr] = np.mean(np.abs(np.array([[(x[0][0]-curs[0])/
    curs[0],(x[0][1]-curs[1])/curs[1],(x[0][2]-curs[2])/curs[2]]]).T)
    )
133 
134         #test 3
135         for i in range(s):
136             readings[i][0] = sensor_array[i].detect(wires_ttwo,0)
137         x,_,_,_ = np.linalg.lstsq(matrix,readings)
138         x=x.T
139         errort2[cntr] = np.mean(np.abs(np.array([[(x[0][0]-curs[0])/
    curs[0],(x[0][1]-curs[1])/curs[1],(x[0][2]-curs[2])/curs[2]]]).T)
    )
140 
141         #test 4
142         for i in range(s):
```

```
            readings[i][0] = sensor_array[i].detect(wires_tthree,0)
        x,_,_,_ = np.linalg.lstsq(matrix,readings)
        x=x.T
        errort3[cntr] = np.mean(np.abs(np.array([[(x[0][0]-curs[0])/
    curs[0],(x[0][1]-curs[1])/curs[1],(x[0][2]-curs[2])/curs[2]]]).T)
    )

        cntr += 1

plt.title("LS Error")
plt.ylabel("Error")
plt.xlabel("Number of Sensors")
plt.plot(axisnsensors,errort0,label="No Interference")
plt.plot(axisnsensors,errort2,label="Plate")
plt.plot(axisnsensors,errort1,label="Ext Wire")
plt.plot(axisnsensors,errort3,label="Six Ext. Wires")
plt.legend()
plt.grid()
plt.show()
```

minimze_fun.py

```
import numpy as np
from scipy.fftpack import fft, ifft
from scipy import signal
from scipy import signal
from scipy.signal import blackman
from scipy.signal import get_window
from scipy.signal import filter_design as fd
from scipy.optimize import minimize
import matplotlib.pyplot as plt
import data_loader
import time_shift
import math
from signalp import Signalp

'''Script used with non-linear estimator used to curve fit non-
```

```
    linear model to readings'''
16
17 #creates an 8-vector represnting magnetic fields from synthetic data
18 def create_synthetic_data(mymatrix,coeffs,location):
19     i1 = 1.2
20     i2 = 0
21     i3 = 0
22     u = .001
23     xw = .030
24     yw = .050
25     iw = -1.2
26
27     breadings = np.zeros(8)
28
29     for i in range(8):
30         breadings[i] = mymatrix[i][0]*i1+mymatrix[i][1]*i2+mymatrix[
    i][2]*i3+mymatrix[i][3]*u+coeffs[i]*iw*(yw-location[i][1])/((xw-
    location[i][0])**2+(yw-location[i][1])**2)
31
32     return breadings
33
34 def get_real_data(processor):
35     offset_samples = data_loader.read_8ch_data('readings\\July18data
    \\ret_ext_wire\\0mag.txt')
36     averages = np.average(offset_samples,axis=1)
37     offsets = np.zeros((8,1))
38     for i in range(8):
39         offsets[i][0] = averages[i]
40
41     samples = data_loader.read_8ch_data('readings\\July18data\\
    ret_ext_wire\\magyn25x220c859_zigzag.txt')
42     samples = processor.shift_signals_USB231(samples,5000)
43     samples = processor.noise_filter(samples,0.00004)
44
45     samples = np.average(samples,axis=1)
46     for i in range(8):
```

```
47          samples[i] = samples[i]-offsets[i][0]
48      return samples
49
50  location = np.zeros((8,3))
51  location[0,:] = [.009,0,1]
52  location[1,:] = [.018,0,1]
53  location[2,:] = [.019,0,1]
54  location[3,:] = [.029,0,1]
55  location[4,:] = [.028,0.02,-1]
56  location[5,:] = [.019,0.02,-1]
57  location[6,:] = [.018,0.02,-1]
58  location[7,:] = [.008,0.02,-1]
59
60  matrix = data_loader.load_matrix('matrices/matrix9/matrix.txt')
61  processor = Signalp(3,8)
62  matrix = processor.augment_8x3_matrix(matrix)
63  processor.set_matrix(matrix)
64
65  coeffs =
        [0.00097,0.00097,0.00097,0.00097,-0.00097,-0.00097,-0.00097,-0.00097]


66
67  samples = create_synthetic_data(matrix,coeffs,location)
68
69  def fun(x,*args):
70      #use matrix multiplication to model internal currents and
        uniform field
71      mag = np.matmul(matrix,np.transpose(np.array([x[0:4]])))
72
73      #model external cable
74      for i in range(8):
75          mag[i][0] += coeffs[i]*x[6]*(x[5]-location[i][1])/((x[4]-
        location[i][0])**2+(x[5]-location[i][1])**2)
76
77      #return squared error score
78      tot = 0
```

```
    for i in range(8):
        tot += (samples[i]-mag[i][0])**2
    return tot

def fun_grad(x,*args):
    gradient = [0,0,0,0,0,0,0]
    for i in range(4):
        comp = 0
        for j in range(8):
            comp += 2*matrix[j][i]*(matrix[j][0]*x[0]+matrix[j][1]*x[1]+matrix[j][2]*x[2]+matrix[j][3]*x[3]+coeffs[i]*x[6]*(x[5]-location[i][1])/((x[4]-location[i][0])**2+(x[5]-location[i][1])**2)-samples[j])
        gradient[i] = comp

    #do x (x[4])
    comp4 = 0
    for j in range(8):
        comp4 += 2*(matrix[j][0]*x[0]+matrix[j][1]*x[1]+matrix[j][2]*x[2]+matrix[j][3]*x[3]+coeffs[i]*x[6]*(x[5]-location[i][1])/((x[4]-location[i][0])**2+(x[5]-location[i][1])**2)-samples[j])*(-1*coeffs[i]*x[6]*(x[4]-location[i][0])*(x[5]-location[i][1])/((x[4]-location[i][0])**2+(x[5]-location[i][1])**2)**2)
    gradient[4] = comp4

    #do y (x[5])
    comp5 = 0
    for j in range(8):
        comp5 += 2*(matrix[j][0]*x[0]+matrix[j][1]*x[1]+matrix[j][2]*x[2]+matrix[j][3]*x[3]+coeffs[i]*x[6]*(x[5]-location[i][1])/((x[4]-location[i][0])**2+(x[5]-location[i][1])**2)-samples[j])*(coeffs[i]*x[6]/((x[4]-location[i][0])**2+(x[5]-location[i][1])**2)-2*coeffs[i]*((x[5]-location[i][1])**2)*x[6]/((x[4]-location[i][0])**2+(x[5]-location[i][1])**2)**2)
    gradient[5] = comp5

```

```
    #do current (x[6])
    comp6 = 0
    for j in range(8):
        comp6 += 2*(matrix[j][0]*x[0]+matrix[j][1]*x[1]+matrix[j
    ][2]*x[2]+matrix[j][3]*x[3]+coeffs[i]*x[6]*(x[5]-location[i][1])
    /((x[4]-location[i][0])**2+(x[5]-location[i][1])**2)-samples[j])
    *(coeffs[i]*(x[5]-location[i][1])/((x[4]-location[i][0])**2+(x
    [5]-location[i][1])**2))
    gradient[6] = comp6
    print("GRADIENT" + str(gradient))
    return np.array(gradient)


res = minimize(fun,[1.0,0,0,0.001,.05,.05,1.0],method = 'BFGS',jac=
    fun_grad)
print('MINIZER ANSWER:')
print(res)
```

multi_correlation.py

```
import numpy as np
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import PolynomialFeatures
from utilities.data_loader import DataLoader
from sys import argv
import simulator.sensor_placer
from simulator.sensors import Sensor, LISensor
from simulator.sources import ConstantField,Wire,FiniteWireXZ
import matplotlib.pyplot as plt

'''Script used to generate covariance matrix with second
    probabilistic model, where all
currents in current grid are active in each random realization'''

h = 0.00525

xy = np.zeros((1000,3))
```

```
numsamples = 20000

startx = -.02
endx=.065
starty= -.03
endy=.03
starti=-10
endi=10

numx=7
numy=5
numi=9

cntr = 0

def get_random_range(m,width,offset=0):
    return width*np.random.rand(m, 1) - (width/2) + offset

def isInBox(x,y):
    if gety(y) > .006 or gety(y) < -.006:
        return True
    elif getx(x) < 0 or getx(x) > .045:
        return True
    else:
        return False


def geti(i):
    return (i/(numi-1))*(endi-starti)+starti
def getx(x):
    return (x/(numx-1))*(endx-startx)+startx
def gety(y):
    return (y/(numy-1))*(endy-starty)+starty

for i in range(numi):
    print(geti(i))
```

```
53
54 for x in range(numx):
55     for y in range(numy):
56         if isInBox(x,y):
57             xy[cntr][0] = getx(x)
58             xy[cntr][1] = gety(y)
59             cntr+= 1
60             if cntr%1000==0:
61                 print('filling {0}'.format(cntr))
62
63 print("cntr is: {0}".format(cntr))
64
65 #CREATE SENSORS
66 #loc_array,firstgroup,secondgroup,thirdgroup = sensor_placer.
    create_location_array()
67 loc_array = simulator.sensor_placer.get_hardware_sensor_array()
68 sensor_array = []
69 for i in range(loc_array.shape[0]):
70     sensor_array.append(LISensor().setStart([loc_array[i,0],
    loc_array[i,1],0]).setEnd([loc_array[i,2],loc_array[i,3],0]))
71
72 Eb0 = 0
73 Eb1 = 0
74 Eb0b0 = 0
75 Eb1b1 = 0
76 Eb0b1 = 0
77 currents = 0
78
79 num_sensors = 10
80
81 covariance_matrix = np.zeros((num_sensors,num_sensors))
82
83 b_temp = np.zeros(num_sensors)
84 b_avg = np.zeros(num_sensors)
85
86 wires = []
```

```
for n in range(cntr):
    wires.append(Wire().setLocation([xy[n][0],xy[n][1],0.0]).
    setOrientation([0,0,1]).setCurrent(get_random_range(numsamples
    ,10)))

for i in range(numsamples):

    for k in range(num_sensors):
        b_temp[k] = sensor_array[k].detect(wires,i)
    for k in range(num_sensors):
        b_avg[k] += b_temp[k]
    for k in range(num_sensors):
        for j in range(num_sensors):
            covariance_matrix[k][j] += b_temp[k]*b_temp[j]
    if i %100==0:
        print('simulating {0}'.format(i))

print(b_avg/numsamples)
print(covariance_matrix/numsamples)

covariance_matrix = covariance_matrix/numsamples

DataLoader().overwrite_ch_data('data_gen/hardware_multi_covariance3.
    txt',covariance_matrix)
```

preprocesor.py

```
import numpy as np
from scipy.fftpack import fft, ifft
import matplotlib.pyplot as plt
from sys import argv
import utilities.data_loader
import time_shift
import os
import utilities.ft_util

class PreProcessor:
```

```
    def __init__(self):
        pass

    def antiantialiasingfilter(self,signal,sf):
        R = 4925#4973
        C = .01e-6
        freq = np.zeros(signal.shape[0])
        myfilter = np.zeros(signal.shape[0])

        for i in range(myfilter.shape[0]):
            freq[i] = utilities.ft_util.FTUtils.index2Hz(sf/2,i,
myfilter.shape[0])

        myfilter = (1+1j*2*np.pi*freq*2*R*C-(2*np.pi*freq*R*C)**2)

        return ifft(fft(signal)*myfilter)

    '''sets self.frequencies, used for things involving frequency,
such as white noise removal'''
    def set_frequency_axis(self,nyq_freq,N):
        self.frequencies = np.zeros(N)

        if N%2 == 0:
            ssp = nyq_freq/float(N/2)
            self.frequencies[0:int(N/2)] = np.linspace(0.0,nyq_freq-
ssp,N/2)
            self.frequencies[int(N/2):N] = np.linspace(-nyq_freq,-
ssp,N/2)
        else:
            ssp = nyq_freq/float((N-1)/2)
            self.frequencies[0:int(N/2+1)] = np.linspace(0.0,
nyq_freq,N/2+1)
            self.frequencies[int(N/2+1):N] = np.linspace(-nyq_freq,-
ssp,N/2)

    def shift_signals_USB205(self,samples,sf,extra=0):
```

```
41          num_ch = samples.shape[0]
42          cT = (1.0/sf)/num_ch
43          N = samples.shape[1]
44          for x in range(num_ch):
45              #print('doing channel...')
46              samples[x,:] = time_shift.shift_signal(samples[x,:],sf
    /2,N,-cT*x+extra)
47          return samples
48
49      def shift_signals_USB231(self,samples,sf,extra=0):
50          num_ch = samples.shape[0]
51          cT = (1.0/sf)/num_ch
52          N = samples.shape[1]
53          mymap = [0,2,4,6,1,3,5,7]
54          for x in range(num_ch):
55              samples[mymap[x],:] = time_shift.shift_signal(samples[
    mymap[x],:],sf/2,N,(-cT*x)+extra)
56          return samples
57
58      #returns time in seconds that signal2 should be shifted by to
    get in line with signal 1
59      def sync(self,signal1,signal2,sf,N):
60          fft1 = fft(signal1)
61          fft2 = fft(signal2)
62          max1 = np.argmax(np.abs(fft(signal1)[10:]))+10
63          print("Frequency used for sync is {}: ".format(utilities.
    ft_util.FTUtils.index2Hz(sf/2,max1,N)))
64          anglediff = np.angle(fft1)[max1]-np.angle(fft2)[max1]
65          if anglediff < -np.pi:
66              anglediff += 2*np.pi
67          elif anglediff > np.pi:
68              anglediff -= 2*np.pi
69          return (anglediff)/(2*np.pi*utilities.ft_util.FTUtils.
    index2Hz(sf/2,max1,N))
70
71      def correlate(self,signal1,signal2):
```

```
72          spots = 500
73          scores = np.zeros(spots)
74
75          for i in range(spots):
76              if i % 100 == 0:
77                  print('at iteration ' + str(i))
78              scores[i] = (1/(signal2.shape[0]-i))*np.sum((signal1[i
    :]*signal2[:(signal2.shape[0]-i)]))
79
80          return scores
81
82      def shift_signals_USB231_diff(self,samples,sf):
83          num_ch = samples.shape[0]
84          cT = (1.0/sf)/4
85          N = samples.shape[1]
86          for x in [1,2,3]:
87              samples[x,:] = time_shift.shift_signal(samples[x,:],sf
    /2,N,-cT*x)
88          return samples
89
90      '''eliminate fourier components not common to all channels'''
91      def common_comp_filter(self,samples,threshold):
92          numch = samples.shape[0]
93          numsamples = samples.shape[1]
94          ft_samples = np.zeros((numch,numsamples))
95          ftransform = np.zeros((numch,numsamples),dtype=np.complex_)
96          ft_agree = np.ones(numsamples)
97          for x in range(numch):
98              ftransform[x,:] = fft(samples[x,:])
99              ft_samples[x,:] = (1/numsamples)*np.abs(ftransform[x,:])
100         for n in range(1,numsamples):
101             frequency_has_component = (ft_samples[0][n] > threshold)
102             for x in range(1,numch):
103                 if frequency_has_component:
104                     if ft_samples[x][n] <= threshold:
105                         ft_agree[n] = 0
```

```
                        break
                else:
                    if ft_samples[x][n] > threshold:
                        ft_agree[n] = 0
                        break

        for x in range(numch):
            ft_samples[x,:] = ftransform[x,:]*ft_agree
            samples[x,:] = ifft(ft_samples[x,:])

        return samples

    def noise_filter(self,samples,threshold):
        #eliminate unique fourier components
        numch = samples.shape[0]
        numsamples = samples.shape[1]

        ft_samples = np.zeros((numch,numsamples))
        ftransform = np.zeros((numch,numsamples),dtype=np.complex_)

        ft_agree = np.ones(numsamples)
        for x in range(numch):
            ftransform[x,:] = fft(samples[x,:])
            ft_samples[x,:] = (1/numsamples)*np.abs(ftransform[x,:])


        for n in range(3,numsamples):
            for x in range(numch):
                if ft_samples[x,n] <= threshold:
                    if n==60:
                        print("Found less than threshold in channel
{}".format(x))
                    ft_agree[n] = 0
                    #ftransform[x,n] = ftransform[x,n]*0

        for x in range(numch):
```

```
141             ftransform[x,:] = ftransform[x,:]*ft_agree
142             samples[x,:] = ifft(ftransform[x,:])
143 
144         return samples
145 
146     def weiner_noise_filter(self,samples,noisepower):
147         for i in range(8):
148             samples[i,:] = wiener(samples[i,:],mysize=3,noise=
    noisepower)
149         return samples
150 
151     '''Performs time shift and noise removal'''
152     def shift_and_clean(self,samples,threshold,sf):
153         num_ch = samples.shape[0]
154         cT = (1.0/sf)/4
155         N = samples.shape[1]
156         for x in [1,2,3]:
157             linear_shift = np.exp(frequencies*2*np.pi*-cT*x*1j)
158             ftc[x-1] = ftc[x-1]*linear_shift
159             ftc[x-1] = ifft(ftc[x-1])
160 
161         numch = samples.shape[0]
162         numsamples = samples.shape[1]
163 
164         ft_mag = np.zeros((numch,numsamples))
165         ftransform = np.zeros((numch,numsamples),dtype=np.complex_)
166 
167         max_freq = np.zeros((numch,2))
168 
169         for x in range(numch):
170             ftransform[x,:] = fft(samples[x,:])
171             ft_mag[x,:] = (1/numsamples)*np.abs(ftransform[x,:])
172 
173         cT = (1.0/sf)/4
174 
175         for n in range(numsamples):
```

```
            for x in range(numch):
                if ft_mag[x,n] > max_freq[x][0]:
                    max_freq[x][0] = ft_mag[x,n]
                    max_freq[x][1] = n
                ftransform[x,n] = ftransform[x,n]*np.exp(self.
    frequencies[n]*2*np.pi*-cT*x*1j)

        return ifft(ftransform),ftransform,max_freq
```

run_estimators.py

```
import numpy as np
from scipy.fftpack import fft, ifft
import matplotlib.pyplot as plt
from sys import argv
import os

import time_shift
import display_ft

from utilities.data_loader import DataLoader
from estimate_current import CurrentEstimator
from estimate_voltage import VoltageEstimator
from preprocessor import PreProcessor

def getRMS(samples,sf,start_time,end_time):
    start = int(start_time*sf)
    end = int(end_time*sf)
    return np.sqrt((1/(end-start))*np.sum(samples[start:end]*samples
    [start:end]))*.99

dataLoader = DataLoader()
currentEstimator = CurrentEstimator()
voltageEstimator = VoltageEstimator()
preProcessor = PreProcessor()

sf = float(argv[3])
```

```
26
27 samples = dataLoader.read_ch_data(argv[1])
28 realsamples = dataLoader.read_ch_data(argv[2])
29 matrix = dataLoader.load_matrix('matrices/matrix28/matrix.txt')
30 col1 = np.array([[1,0,1,0,1,1,0,1,1]]).T #missing 0
31 col2 = np.array([[0,-1,0,-1,0,0,1,0,0]]).T #missing 1
32 matrix = np.concatenate((matrix,col1,col2),axis=1)
33 offset = dataLoader.load_matrix('matrices/matrix28/offset.txt')
34 currentEstimator.setCovarianceMatrix(DataLoader().read_ch_data('
      data_gen/realhardware_multi_covariance3.txt'))
35 currentEstimator.setEstimatorType('blu')
36
37 currentEstimator.setMatrix(matrix)
38
39 t=preProcessor.sync(samples[0,:],samples[8,:],sf,samples.shape[1])
40 print("timeshift is {}".format(t))
41 samples[:8,:] = preProcessor.shift_signals_USB205(samples[:8,:],sf)
42 realsamples[:8,:] = preProcessor.shift_signals_USB205(realsamples
      [:8,:],sf)
43 samples[8:16,:]=preProcessor.shift_signals_USB231(samples[8:16,:],sf
      ,t)
44
45 voltage_ch = [0,1,2]
46 real_vol_ch = [0,1,2]
47 real_cur_ch = [2,4]
48 res1 = 1.08
49 res2 = 1.08
50 current_ch = [3,4,5,9,10,11,12,13,15]
51
52 volsamples = samples[voltage_ch,:]
53 volsamples[0,:] = volsamples[0,:]-volsamples[1,:]
54 volsamples[2,:] = volsamples[2,:]-volsamples[1,:]
55 #display_ft.display_ft(volsamples[2,:],sf,0.0,sf/2-10.0,show=True)
56 volsamples = preProcessor.noise_filter(volsamples,.005)
57
58 #real_voltage = realsamples[real_vol_ch,:]
```

```
#real_voltage = preProcessor.noise_filter(real_voltage,0)

time_axis = np.linspace(1/sf,(1/sf)*samples.shape[1],samples.shape
    [1])

real_currents = realsamples[real_cur_ch,:]
real_currents = preProcessor.noise_filter(real_currents,.0015)

cursamples = samples[current_ch,:]
cursamples = cursamples-offset

for i in range(9):
  cursamples[i,:] = cursamples[i,:] - np.mean(cursamples,axis=1)[i]

cursamples = preProcessor.noise_filter(cursamples,.00008)

#optionally set locations in current estimator
currentEstimate = currentEstimator.getEstimate(cursamples)

plt.title('Current Estimates')
plt.xlabel('Time (sec)')
plt.ylabel('Current (A)')
ccurrent1= (real_currents[0,:])/res1

ccurrent3 = (real_currents[1,:])/res2
ccurrent2 = -(ccurrent1+ccurrent3)

toterr = 0
for i in range(len(ccurrent1)):
  err = np.abs(currentEstimate[0,i]-ccurrent1[i])+np.abs(
    currentEstimate[1,i]-ccurrent2[i])+np.abs(currentEstimate[2,i]-
    ccurrent3[i])
  toterr += err/(np.abs(ccurrent1[i])+np.abs(ccurrent2[i])+np.abs(
    ccurrent3[i]))

print("Percent error is: {}".format(toterr/len(ccurrent1)))
```

```
91
92 #plt.plot(time_axis, ccurrent1, label = 'Contact Current 0')
93 #plt.plot(time_axis, ccurrent3, label = 'Contact Current 1')
94 #plt.plot(time_axis, ccurrent2, label = 'Contact Current 2')
95 plt.plot(time_axis, currentEstimate[0], label = 'Current '+str(0))
96 plt.plot(time_axis, currentEstimate[1], label = 'Estimate '+str(1))
97 plt.plot(time_axis, currentEstimate[2], label = 'Current '+str(2))
98
99 #plt.show()
100
101 #use voltage estimator
102 voltageEstimator.setRCValues(10000000,3.6e-12)
103 vol1 = voltageEstimator.estimate_voltage(volsamples[0,:],sf/2)
104 voltageEstimator.setRCValues(1000000,4.12e-12)
105 vol2 = voltageEstimator.estimate_voltage(volsamples[2,:],sf/2)
106
107 #rvoltage0 = real_voltage[0,:]-real_voltage[1,:]
108 #rvoltage1 = real_voltage[2,:]-real_voltage[1,:]
109
110 #plt.figure()
111 plt.title('Power Estimate')
112 plt.xlabel('Time (sec)')
113 plt.ylabel('Power (W)')
114
115 #plt.plot(time_axis,rvoltage0,label = 'Contact Voltage 0')
116 #plt.plot(time_axis,rvoltage1,label = 'Contact Voltage 1')
117 plt.plot(time_axis, vol1, label = 'Voltage 1')
118 plt.plot(time_axis, vol2, label = 'Voltage 2')
119
120 #err = 0
121 #mycntr = 0
122 #for i in range(real_voltage.shape[1]):
123 # ierr = np.abs(rvoltage0[i]-vol1[i])+np.abs(rvoltage1[i]-vol2[i])
124 # err += ierr/(np.abs(vol1[i])+np.abs(vol2[i]))
125
126 #print("Error is {}".format(err/real_voltage.shape[1]))
```

```
127
128 power1 = np.zeros(vol1.shape[0])
129 for i in range(vol1.shape[0]):
130   power1[i]=vol1[i]*currentEstimate[0][i]
131
132 power2 = np.zeros(vol1.shape[0])
133 for i in range(vol1.shape[0]):
134   power2[i]=vol2[i]*currentEstimate[2][i]
135
136 print(getRMS(power1,6250,0,1.0))
137 print(getRMS(power2,6250,0,1.0))
138
139 #plt.plot(time_axis, power1, label = 'Resistor 1 Power')
140 #plt.plot(time_axis, power2, label = 'Resistor 2 Power')
141 plt.legend()
142 plt.grid()
143 plt.show()
```

run_simulation.py

```
1 from simulator.sensors import Sensor, LISensor
2 from simulator.sources import ConstantField,Wire,FiniteWireXZ
3 import simulator.sensor_placer
4 import numpy as np
5 from scipy.spatial import distance
6 import itertools
7 import math
8 import matplotlib.pyplot as plt
9 from utilities.data_loader import DataLoader
10 from estimate_current import CurrentEstimator
11 from sklearn.linear_model import LinearRegression
12 from sklearn.preprocessing import PolynomialFeatures
13
14
15 i_c = 0
16 h = 0.00525 #half the height of the HARTING Han-C connector, in
    meters
```

```
w= 0.015

def get_theoretical_matrix(sensors,wires):
    th_matrix = np.zeros((len(sensors),3))
    for x in range(len(sensors)):
        for y in range(3):
            th_matrix[x][y] = sensors[x].detect([wires[y]],0)
    return th_matrix

def getReferenceWires():
    wires_ref = []
    wires_ref.append(Wire().setLocation([.015,0,0.0]).setOrientation
    ([0,0,1]).setCurrent([1]))
    wires_ref.append(Wire().setLocation([.03,0,0.0]).setOrientation
    ([0,0,1]).setCurrent([1]))
    wires_ref.append(Wire().setLocation([.045,0,0.0]).setOrientation
    ([0,0,1]).setCurrent([1]))
    return wires_ref

curs = [-.7,1,-.3]
def getFourTestCases():
    wires = []
    wires.append(Wire().setLocation([.015,0,0.0]).setOrientation
    ([0,0,1]).setCurrent([curs[0]]))
    wires.append(Wire().setLocation([.03,0,0.0]).setOrientation
    ([0,0,1]).setCurrent([curs[1]]))
    wires.append(Wire().setLocation([.045,0,0.0]).setOrientation
    ([0,0,1]).setCurrent([curs[2]]))

    wires_one = []
    wires_one.append(Wire().setLocation([.015,0,0.0]).setOrientation
    ([0,0,1]).setCurrent([curs[0]]))
    wires_one.append(Wire().setLocation([.03,0,0.0]).setOrientation
    ([0,0,1]).setCurrent([curs[1]]))
    wires_one.append(Wire().setLocation([.045,0,0.0]).setOrientation
    ([0,0,1]).setCurrent([curs[2]]))
```

```
44      wires_one.append(Wire().setLocation([.03,3*h,0.0]).
    setOrientation([0,0,1]).setCurrent([[1]]))
45
46      wires_ttwo = []
47      wires_ttwo.append(Wire().setLocation([.015,0,0.0]).
    setOrientation([0,0,1]).setCurrent([curs[0]]))
48      wires_ttwo.append(Wire().setLocation([.03,0,0.0]).setOrientation
    ([0,0,1]).setCurrent([curs[1]]))
49      wires_ttwo.append(Wire().setLocation([.045,0,0.0]).
    setOrientation([0,0,1]).setCurrent([curs[2]]))
50      wires_ttwo.append(Wire().setLocation([.015,4*h,0.0]).
    setOrientation([0,0,1]).setCurrent([-1*curs[0]]))
51      wires_ttwo.append(Wire().setLocation([.03,4*h,0.0]).
    setOrientation([0,0,1]).setCurrent([-1*curs[1]]))
52      wires_ttwo.append(Wire().setLocation([.045,4*h,0.0]).
    setOrientation([0,0,1]).setCurrent([-1*curs[2]]))
53
54      wires_tthree = []
55      wires_tthree.append(Wire().setLocation([.015,0,0.0]).
    setOrientation([0,0,1]).setCurrent([curs[0]]))
56      wires_tthree.append(Wire().setLocation([.03,0,0.0]).
    setOrientation([0,0,1]).setCurrent([curs[1]]))
57      wires_tthree.append(Wire().setLocation([.045,0,0.0]).
    setOrientation([0,0,1]).setCurrent([curs[2]]))
58      wires_tthree.append(Wire().setLocation([.015,3*h,0.0]).
    setOrientation([0,0,1]).setCurrent([curs[0]]))
59      wires_tthree.append(Wire().setLocation([.03,3*h,0.0]).
    setOrientation([0,0,1]).setCurrent([curs[1]]))
60      wires_tthree.append(Wire().setLocation([.045,3*h,0.0]).
    setOrientation([0,0,1]).setCurrent([curs[2]]))
61      wires_tthree.append(Wire().setLocation([.006,-3*h,0.0]).
    setOrientation([0,0,1]).setCurrent([curs[0]]))
62      wires_tthree.append(Wire().setLocation([.030,-3*h,0.0]).
    setOrientation([0,0,1]).setCurrent([curs[1]]))
63      wires_tthree.append(Wire().setLocation([.04,-3*h,0.0]).
    setOrientation([0,0,1]).setCurrent([curs[2]]))
```

```
64
65      return wires,wires_one,wires_ttwo,wires_tthree
66
67  def getGeneralTestCases():
68      num_samples = 500*21
69
70      readings = np.zeros((10,num_samples))
71      X = np.zeros((num_samples,3))
72
73      dataLoader = DataLoader()
74
75      for j in range(21):
76          readings[:,j*500:(j+1)*500] = dataLoader.read_ch_data("
    data_gen/varied_ext_with_vertical/x/"+str(j)+".txt")[:10,:500]
77          X[j*500:(j+1)*500,:] = dataLoader.read_ch_data("data_gen/
    varied_ext_with_vertical/y/"+str(j)+".txt")[:3,:500].T
78
79      return readings,X
80
81  PDEG = 1
82  def sim_estimator_per_sensors(estimator,sensor_array,estimator_type=
    "blu"):
83      estimator.setEstimatorType(estimator_type)
84
85      tot_sensors=20
86      axisnsensors = []
87      errort0 = []
88      errort1 = []
89      errort2 = []
90      errort3 = []
91
92      wires,wires_one,wires_ttwo,wires_tthree=getFourTestCases()
93
94      A = estimator.A
95      S = estimator.S
96
```

```
    genreadings,X = getGeneralTestCases()

    for s in range(6,tot_sensors+1):
        if s%2==0:

            poly_features = PolynomialFeatures(degree=PDEG,
include_bias=False)
            regreadings = genreadings[:s,:]
            readings_POLY = poly_features.fit_transform(regreadings.
T)
            lin_reg = LinearRegression()
            lin_reg.fit(readings_POLY, X)

            estimator.setMatrix(A[:s,:])
            estimator.setM((s-4)//2)
            estimator.setCovarianceMatrix(S[:s,:s])
            estimator.setRegressionModel(lin_reg.intercept_,lin_reg.
coef_,PDEG)

            readings = np.zeros((s,1))
            for j in range(s):
                readings[j][0] = sensor_array[j].detect(wires,0)
            x = estimator.getEstimate(readings)
            errort0.append(np.mean(np.abs(np.array([[(x[0][0]-curs
[0])/curs[0],(x[1][0]-curs[1])/curs[1],(x[2][0]-curs[2])/curs
[2]]]).T)))

            for j in range(s):
                readings[j][0] = sensor_array[j].detect(wires_one,0)
            x = estimator.getEstimate(readings)
            errort1.append(np.mean(np.abs(np.array([[(x[0][0]-curs
[0])/curs[0],(x[1][0]-curs[1])/curs[1],(x[2][0]-curs[2])/curs
[2]]]).T)))

            for j in range(s):
                readings[j][0] = sensor_array[j].detect(wires_ttwo
```

```
    ,0)
126             x = estimator.getEstimate(readings)
127             errort2.append(np.mean(np.abs(np.array([[(x[0][0]-curs
    [0])/curs[0],(x[1][0]-curs[1])/curs[1],(x[2][0]-curs[2])/curs
    [2]]]).T)))
128
129             for j in range(s):
130                 readings[j][0] = sensor_array[j].detect(wires_tthree
    ,0)
131             x = estimator.getEstimate(readings)
132             errort3.append(np.mean(np.abs(np.array([[(x[0][0]-curs
    [0])/curs[0],(x[1][0]-curs[1])/curs[1],(x[2][0]-curs[2])/curs
    [2]]]).T)))
133
134             axisnsensors.append(s)
135
136
137
138     print("No external wire: {}".format(errort0))
139     print("External Wire: {}".format(errort1))
140     print("Plate: {}".format(errort2))
141     print("Six Ext. Wires: {}".format(errort3))
142
143     plt.title("Harmonics Estimator")
144     plt.ylabel("Error")
145     plt.xlabel("Number of Sensors")
146     plt.plot(axisnsensors,errort0,label="No Interference")
147     plt.plot(axisnsensors,errort2,label="Plate")
148     plt.plot(axisnsensors,errort1,label="Ext Wire")
149     plt.plot(axisnsensors,errort3,label="Six Ext. Wires")
150     plt.legend()
151     plt.grid()
152     plt.show()
153
154 def show_histogram_analysis(estimator,sensor_array,estimator_type="
    blu"):
```

```
155     readings,X = getGeneralTestCases()
156     est = np.zeros(X.shape)
157 
158     for i in range(X.shape[0]):
159         est[i,:] = estimator.bluEstimate(readings[:,i])
160 
161     all_percents = []
162     for i in range(X.shape[0]):
163         for j in range(X.shape[1]):
164             if X[i,j] > .5:
165                 all_percents.append(100*np.abs((X[i,j]-est[i,j])/X[i
    ,j]))
166     print("Mean percent error is: {}".format(sum(all_percents)/len(
    all_percents)))
167     plt.hist(np.array(all_percents),bins=100)
168     plt.show()
169 
170 
171 if __name__ == '__main__':
172 
173     wires_ref = getReferenceWires()
174 
175     field = ConstantField().setField([[1,0,0]])
176 
177     loc_array = simulator.sensor_placer.get_hardware_sensor_array()
178     sensor_array = []
179     for i in range(10):
180         sensor_array.append(LISensor().setStart([loc_array[i,0],
    loc_array[i,1],0]).setEnd([loc_array[i,2],loc_array[i,3],0]))
181     matrix = get_theoretical_matrix(sensor_array,wires_ref)
182     readings = np.zeros((10,1))
183     for i in range(10):
184         readings[i,0] = sensor_array[i].detect([field],0)
185 
186 
187     print(matrix*1000*4.88)
```

```
    exit()



    loc_array,firstgroup,secondgroup,thirdgroup = simulator.
sensor_placer.create_location_array()

    sensor_array = []
    for i in range(88):
        sensor_array.append(LISensor().setStart([loc_array[i,0],
loc_array[i,1],0]).setEnd([loc_array[i,2],loc_array[i,3],0]))

    wires_ref = getReferenceWires()
    matrix = get_theoretical_matrix(sensor_array,wires_ref)

    estimator = CurrentEstimator()
    estimator.setMatrix(matrix)
    estimator.setCovarianceMatrix(DataLoader().read_ch_data('
data_gen/multicovariancematrix.txt'))
    estimator.setSensorLocations(loc_array,firstgroup,secondgroup,
thirdgroup)

    #show_histogram_analysis(estimator,sensor_array,estimator_type="
blu")
    sim_estimator_per_sensors(estimator,sensor_array,estimator_type=
"lap")
```

time_shift.py

```
import numpy as np
from scipy.fftpack import fft, ifft
from scipy import signal
from scipy.signal import blackman
from scipy.signal import get_window
from scipy.signal import filter_design as fd
import matplotlib.pyplot as plt

```

```
'''
API for shifting signals in the time domain by applying a linear phase filter.
This is used by several other scripts.
'''

def shift_signal(x,nyq_freq,N,t):
    frequencies = np.zeros(N)

    if N%2 == 0:
        ssp = nyq_freq/float(N/2)
        frequencies[0:int(N/2)] = np.linspace(0.0,nyq_freq-ssp,N/2)
        frequencies[int(N/2):N] = np.linspace(-nyq_freq,-ssp,N/2)
    else:
        ssp = nyq_freq/float((N-1)/2)
        frequencies[0:int(N/2+1)] = np.linspace(0.0,nyq_freq,N/2+1)
        frequencies[int(N/2+1):N] = np.linspace(-nyq_freq,-ssp,N/2)

    linear_shift = np.exp(frequencies*2*np.pi*t*1j)
    ft = fft(x)
    interpolated_shift = ifft(ft*linear_shift)

    #print("right before np.imag()...")
    icomponents = np.imag(interpolated_shift)
    rcomponents = np.real(interpolated_shift)
    #print("Note, largest real component is: " + str(np.abs(np.max(rcomponents))))
    #print("Note, largest imaginary component is: " + str(np.abs(np.max(icomponents))))
    return np.real(interpolated_shift)

def cross_correlate(x,y,nyq_freq,times):
    results = np.zeros(len(times))
    for i in range(len(times)):
        y_shifted = shift_signal(y,nyq_freq,len(y),times[i])
        product = x*y_shifted
```

```
        results[i] = np.sum(product)
    return results
```

use_readers2.py

```
from __future__ import absolute_import, division, print_function

import math
import time

from builtins import *  # @UnusedWildImport

from mcculw import ul
from mcculw.enums import ScanOptions, FunctionType, Status,
    AnalogInputMode
from examples.console import util
from examples.props.ao import AnalogOutputProps
from mcculw.ul import ULError

import numpy as np
import datetime

from sys import argv

from utilities.data_loader import DataLoader

use_device_detection = True

from mcculw.enums import InterfaceType

from daqlib.daq_readers import ReaderPool,USB205Reader,USB231Reader
from daqlib.daq_writers import USB231Writer
import current_display

'''
Script to concurrently read data from two DAQs and write data as
    well.
```

```
31
32 parameters:
33     -file to write first DAQ data to.
34     -file to write second DAQ data to.
35 '''
36
37 def config_two_devices(board_num):
38     devices = ul.get_daq_device_inventory(InterfaceType.ANY)
39     # Check if any devices were found
40     cntr = 0
41
42     for eachdevice in devices:
43         print("Found device: " + eachdevice.product_name + " (" +
    eachdevice.unique_id + ")\n")
44         ul.create_daq_device(board_num[cntr],eachdevice)
45         cntr += 1
46
47     if len(devices) > 0:
48         return devices
49     else:
50         return None
51
52 cntr = 0
53 def channel0_write(x,chan):
54     return 0
55     #return 10.0*np.cos(2*2080*np.pi*x)
56
57 def channel1_write(x,chan):
58     return 0.0
59
60 def run_example(filenames):
61     board_nums = [0,1]
62
63     ul.ignore_instacal()
64     devices = config_two_devices(board_nums)
65     if devices is None:
```

```
66          print("Could not find two devices.")
67          return
68
69      writer = None
70
71      pool = ReaderPool()
72      for x in range(len(devices)):
73          if devices[x].product_name[0:7] == "USB-205":
74              pool.add_reader(USB205Reader(board_nums[x],
    isDifferential=False))
75          elif devices[x].product_name[0:7] == "USB-231":
76              pool.add_reader(USB231Reader(board_nums[x],
    isDifferential=False))
77              writer = USB231Writer(board_nums[x])
78
79      #writer.prepare_write([channel0_write,channel1_write],2,5000)
80
81      pool.setup_buffers(1,6250)
82
83      #writer.start_write()
84      #time.sleep(.2)
85      samples_array = pool.scan_all() #this call blocks
86      #writer.wait_until_done_writing()
87
88      ul.release_daq_device(board_nums[0])
89      if len(devices) > 1:
90          ul.release_daq_device(board_nums[1])
91
92      print("samples array 0 : {}".format(samples_array[0].shape))
93      print("samples array 1 : {}".format(samples_array[1].shape))
94
95      #Code below used when auto-generating files with their current
    values
96      #current = (np.mean(samples_array[0][0,:])-np.mean(samples_array
    [0][1,:]))/5.1
97      #print(np.mean(samples_array[0][0,:]))
```

```
    #print(np.mean(samples_array[0][1,:]))
    #print("current is {}".format(current))
    #DataLoader().overwrite_ch_data(filenames[0]+"/"+str(int(current*1000))+".txt",np.concatenate((samples_array[0],samples_array[1]),axis=0))
    #exit()

    if len(devices) > 1:
        DataLoader().overwrite_ch_data(filenames[0],np.concatenate((samples_array[0],samples_array[1]),axis=0))
    else:
        DataLoader().overwrite_ch_data(filenames[0],samples_array[0])

if __name__ == '__main__':
    run_example([argv[1]])
```

voltage_display.py

```
import numpy as np
from scipy.fftpack import fft, ifft
import matplotlib.pyplot as plt
from sys import argv
import utilities.data_loader
import time_shift
import os
from preprocessor import PreProcessor

'''
Displays data from 8-channel voltages readings graphically.

Parameters:
    -filename
    -sampling frequency
    -Optional: start time
    -Optional: end time
'''
```

```
19
20 def display_ch_data(start,end,sf,samples):
21     length = end-start
22     T = 1.0/sf
23     times = np.linspace(start*T,(end-1)*T, length)
24
25     plt.title('ADC Voltage')
26     plt.xlabel('Time (sec)')
27     plt.ylabel('Current (A)')
28     plt.plot(times, samples[start:end])
29     plt.grid()
30
31     plt.show()
32
33 def display_all_data(start,end,sf,samples):
34     length = end-start
35     T = 1.0/sf
36     times = np.linspace(start*T,(end-1)*T, length)
37
38     for i in [12]:
39         plt.title('Voltage Readings channel' + str(i))
40         plt.xlabel('Time (sec)')
41         plt.ylabel('Volts (V)')
42         plt.plot(times, samples[i,start:end])
43         plt.grid()
44
45     plt.show()
46
47
48 if __name__ == '__main__':
49     filename = argv[1]
50     sf = float(argv[2])
51
52     samples = utilities.data_loader.DataLoader().read_ch_data(argv
    [1])
53
```

```
    start = 0
    end = samples.shape[1]

    print("Averages: " + str(np.mean(samples,axis=1)))

    preProcessor = PreProcessor()

    #samples[1:3,:] = preProcessor.noise_filter(samples
    [4:5,:],.0001)
    #samples=samples/1.08

    if len(argv) > 3:
        ch = int(argv[3])
        display_ch_data(start,end,sf,samples[ch,:])
    else:
        display_all_data(start,end,sf,samples)
```

voltage_freq_response_fit.py

```
from scipy.optimize import curve_fit
import numpy as np
import matplotlib.pyplot as plt

'''Program used to curve fit readings of electrode voltage to
   transfer functions modelling
voltage detection system'''

#Transfer function including omega squared term
def simplefunc(x,a,b,c):
  return x/np.sqrt((a-c*(x**2))**2+(b*x)**2)

#Transfer function without omega squared term
def simplefunc2(x,a,b):
  return x/np.sqrt((a)**2+(b*x)**2)


f=np.array([150,330,550,880,1230,1791,3367,9928,20648,26518,33595])
```

```
w=f*2*np.pi
gain=np.array
    ([.034552,.071968,.114061,.16934,.2211,.2935,.42098,.54857,.5771,.5812,.5826])


plt.plot(w/(2*np.pi),gain,label="Measurements")

popt,pcov = curve_fit(simplefunc2,w,gain,p0=[300000,1.7],bounds
    =(0,1000000000))#,0],bounds=(0,[1000000,100000000,1000000000000])
    )

testvoutsimple =simplefunc2(w,popt[0],popt[1])
plt.plot(w/(2*np.pi),testvoutsimple,label="Curve Fit")
plt.xlabel("Frequency (Hz)")
plt.ylabel("Gain")
plt.title("Electrode Frequency Response")
plt.legend()
plt.grid()
plt.show()

for i in range(len(testvoutsimple)):
  print("Freq" + str(f[i])+ " vol: " + str(testvoutsimple[i]))

print("b/a: {}".format(popt[1]/popt[0]))

print(popt)
print(popt[0])


print("Error is: {0}".format((1/len(vout)*np.sum((testvoutsimple-
    vout)**2))))
```

Bibliography

[1] Martín Abadi. TensorFlow: learning functions at scale. 2016.

[2] Mark D. Plumbley Andrew J.R. Simpson, Gerard Roma. Deep Karaoke: Extracting Vocals from Musical Mixtures Using a Convolutional Deep Neural Network. 2015.

[3] Measurement Computing. Low Cost DAQ Devices. `https://www.mccdaq.com/data-acquisition/low-cost-daq/`, 2019. [Online; accessed 23-August-2019].

[4] Dataforth Corporation. Harmonics and Utility Costs, Application Note. 2018.

[5] J. Wang Z. Zhang D. Li, L. Di Rienzo. Current measurement in the time domain based on the inversion of magnetic field data. 2011.

[6] John Donnal David Lawrence and Steven Leeb. Non-contact Measurement of Line Voltage. 2016.

[7] Steven Leeb David Lawrence, John S. Donnal. Current and Voltage Reconstruction From Non-Contact Field Measurements. 2016.

[8] Jimmy Ba Diederik P. Kingma. Adam: A Method for Stochastic Optimization. 2014.

[9] Luca Di Rienzo Dongwei Li. Robustness analysis of magnetic sensor arrays for current sensing. 2011.

[10] M. Norgren F. Ghasemifard, M. Johansson. Current reconstruction from magnetic field using spherical harmonic expansion to reduce impact of disturbance fields. 2016.

[11] Roberto Ottoboni Gabriele D'Antona, Luca Di Rienzo. Processing Magnetic Sensor Array Data for AC Current Measurement in Multiconductor Systems. 10 2001.

[12] Kun-Long Chen Wilsun Xu Guangchao Geng, Juncheng Wang. Contactless Current Measurement for Enclosed Multiconductor Systems Based on Sensor Array. 2017.

[13] Huayi Liu Hao Yu, Zhen Qian and Jiaqi Qu. Circular array of magnetic sensors for current measurement: Analysis for error caused by position of conductor. 02 2018.

[14] HARTING. Current Measurement Technique. 2016.

[15] Kittikhun Thongpull Hiroshi Nakahara. Design and Analysis of Non-Invasive Capacitive Coupling Voltage Sensor. 2018.

[16] Alan S. Edelstein James Lenz. Magnetic Sensors and Their Applications. 2017.

[17] Steven B. Leeb John Donnal. Non-Contact Power Meter. 2013.

[18] Lawrence A. Jones and Jeffrey H. Lang. A State Observer for the Permanent-Magnet Synchronous Motor. 08 1989.

[19] M.E. HONARMAND Adham SHARIFI Jamshid TALEBI Mahnaz YOUHANNAEI, Hossein MOKHTARI. Non-contact Measurement of Line Voltage. 06 2015.

[20] Enrique M. Spinelli Marcelo A. Haberman. A Non-Contact Voltage Measurement System for Power-Line Voltage Waveforms. 2019.

[21] Enrique Mario Spinelli Marcelo Alejandro Haberman. Noncontact AC Voltage Measurements: Error and Noise Analysis. 2018.

[22] Martin Norgren. Explicit reconstruction of line-currents and their positions in a two-dimensional parallel conductor structure. 2013.

[23] Andrey Chirtsov Pavel Ripka. Influence of External Current on Yokeless Electric Current Transducers. 2017.

[24] L. Di Rienzo and Z. Zhang. Spatial Harmonic Expansion for Use With Magnetic Sensor Arrays. 01 2010.

[25] Alexander Itzke Robert Weigel Roland Weiss, Rory Makuch. Crosstalk in Circular Arrays of Magnetic Sensors for Current Measurement. 2017.

[26] Ivelina N. Cholakova ; Tihomir B. Takov ; Radostin Ts. Tsankov ; Nicolas Simonne. Temperature influence on Hall effect sensors characteristics. 2012.

[27] Kittikhun Thongpull Pornchai Phukpattarnont Kanadit Chetpattananondh Sotara Ren, Hiroshi Nakahara. A development of capacitive voltage sensor for non-intrusive energy meter. 2018.

[28] W.A. Zisman. A NEW METHOD OF MEASURING CONTACT POTENTIAL DIFFERENCES IN METALS. 1932.