Final Report: Magnetic Fluid Control for Drug Therapy

ME-394: Capstone Senior Mechanical Engineering Design

Michael Colella, Skylar Eiskowitz, Jacob Maarek

Advisors: Prof. Yecko and Prof. Luchtenburg

The Cooper Union for the Advancement of Science and Art

Table of Contents

1 Abstract 2

2 Introduction 3

2.1 Motivation 3

2.2 Previous Work 4

2.3 Goal 7

3 Preliminary Design Work 8

3.1 Codes and Standards 8

3.2 Criteria and Constraints 8

3.3 Ferrofluid Selection 9

3.4 Electromagnet Selection 10

4 Theoretical Models 12

4.1 External Magnetic Field 12

4.1.1 Experimental Validation 14

5 Tracking Algorithm 17

5.1 Tracking a Permanent Magnet 19

5.2 Tracking a Ferrofluid Droplet 21

6 Controls Experimental Setup 22

6.1 Control Algorithm 24

6.2 OpenCV 24

7 Conclusion 27

8 Future Work 28

9 Appendix 29

9.1 Appendix A: Analytical Plot of Bx Field with Experimental Validation Code 29

9.2 Appendix B: Description of Control Algorithm 33

9.2.1 Linear Segment 31

9.2.2 Nonlinear Segment 32

9.2.3 Nonlinear Filter 35

9.3 Appendix C: Control Algorithm Code 37

9.4 Appendix D: Controls GUI Code 40

9.5 Appendix E: Localization GUI Code 46

9.6 Appendix F: Team Roles 51

1 Abstract

Chemotherapy drugs damage both healthy and unhealthy cells—only 0.1% of these drugs reach the targeted area. Magnetic drug therapy mitigates the adverse side effects of chemotherapy by coating magnetic particles with medicine and delivering them directly to tumors using external magnetic fields. However, these particles are difficult to track once injected in the body. Currently, strategies have been forced to track particles using visual means, limiting their use in humans. We have successfully controlled a magnetic particle using visual feedback, and have developed an algorithm that can accurately determine a magnetic particle's position in real time using an array of magnetometers.

2 Introduction

2.1 Motivation

Chemotherapy is one of the most commonly used treatments for cancer, but the drug damages both healthy and unhealthy cells because it is circulated throughout the entire body. Less than 0.1% of chemotherapy actually reaches unhealthy cells, meaning that 99.9% damages perfectly healthy cells.[1] This unfocused nature of drug administration directly results in adverse side effects: severe hair loss, nausea, fatigue, and mouth sores.

A developing method of treatment that can focus the cancer drug to a target area is called magnetic drug delivery. It is a process by which (a colloidal mixture of ferromagnetic nanoparticles, a surfactant, and an oil or water-based suspension fluid) is first coated with therapeutic drugs, then injected into the human body and "delivered" to its destination *in vivo* by manipulating an external magnetic field (*Figure 1*). This method accomplishes a focused chemotherapy treatment by holding the drug-coated ferrofluid in place, allowing the drug to enter tumorous regions without affecting healthy cells in other areas of the body.

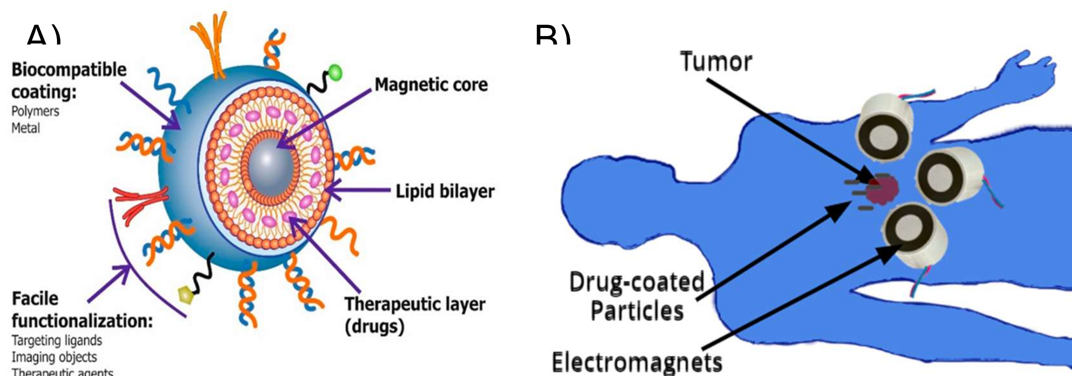


Figure 1: Magnetic drug delivery system. A) Functionalization of a ferrofluid droplet. B) Schematic of drug delivery setup

[1] Shapiro, B., Kulkarni, S., Nacev, A., Sarwar, A., Preciado, D., & Depireux, D. A. (2014). *Shaping magnetic fields to direct therapy to ears and eyes*. Annual Review of Biomedical Engineering, 16, 455-481.

Magnetic drug delivery has been shown to carry drugs to tumors in both shallow areas just beneath the skin and in hard-to-reach areas (e.g., the back of the eye or inner ear).[2] However, little effort has been put into realistically implementing the control system that moves the to a specified . Specifically, ideal testing conditions like the use of overhead cameras in clear containers merely validate the ability for the magnetic fluid to be controlled; these systems will not be able to operate in patients, where opaque body parts replace clear dishes. More realistic testing conditions will close the gaps in research, revolutionizing the way patients receive chemotherapy.

2.2 Previous Work

Researchers have shown that a single permanent magnet is capable of concentrating chemotherapy around normally inoperable tumors in the head, neck, and breast in a Phase 1 human clinical trial.[3,4] So far, research in magnetic drug therapy has heavily focused on the fabrication of magnetic carriers; however, there has been little progress in the magnetic systems that will control these carriers *in vivo*.

One primary challenge is the imaging of carriers and therapy in real-time once injected into the body. As a result, testing the magnetic drug delivery system on animals relies on euthanization to find out where the ferrofluid is located post-procedure. The problem with MRI as a real-time solution is that it is not possible to magnetically treat and image at the same time; the magnets would interfere with MRI operations. X-ray fluoroscopy is a viable method, but it is

---

[2] Shapiro, B. (2009) "*Towards Dynamic Control of Magnetic Fields to Focus Magnetic Carriers to Targets Deep inside the Body*". Journal of magnetism and magnetic materials 321.10: 1594.

[3] A. Sarwar, R. Lee, D. A. Depireux, and B. Shapiro, "Magnetic Injection of Nanoparticles Into Rat Inner Ears at a Human Head Working Distance," IEEE Transactions on Magnetics, vol. 49, no. 1, pp. 440-452, 2013.

[4] A.S. Lubbe et al., "Clinical experiences with magnetic drug targeting: a phase I study with 4'-epidoxorubicin in 14 patients with advanced solid tumors," Cancer Res., vol. 56, no. 20, pp. 4686-4693, 1996.

well-known that continuously receiving radiation increases the chance of developing cancer in the future.[5] One recent development that bypasses such problems is magnetic particle imaging (MPI), which is being developed for real-time sensing. The process exploits the non-linear magnetic response of the ferrofluid and creates a magnetic field node point within the imaging region using two external coils. Once this node is created, sensing coils interpret the magnetic response and infer the location of the ferrofluid.[6] Novel solutions like MPI provide the impetus for our solution: a cheaper, more accessible method to image carries in real time, as described in the next section.

Computationally, we consider the work of Dr. Benjamin Shapiro and collaborators at the University of Maryland. Shapiro discusses a mathematical model of how time-varying actuation can transport the ferrofluid to a desired setpoint with an implementation on COMSOL in a 2D simulation.[7] A model setup for this simulation is shown in *Figure 2*: a set of electromagnets is dispersed axisymmetrically around a ferrofluid control domain which continuously adjusts the external magnetic field to guide the ferrofluid to the desired setpoint. This paper as well as Probst's experimental setup[8] serve as an excellent basis for the creation of our controls experimental setup, discussed in *Section 6*.

[5] John D. Boice, Jr., Dale Preston, Faith G. Davis, and Richard R. Monson. Radiation Research (1991). *"Frequent Chest X-Ray Fluoroscopy and Breast Cancer Incidence among Tuberculosis Patients in Massachusetts."* 125:2, 214-222

[6] Ilbey, S. "*Real-Time Three-Dimensional Image Reconstruction Using Alternating Direction Method of Multipliers for Magnetic Particle Imaging". 2018.*

[7] Shapiro, B. (2009). "*Towards Dynamic Control of Magnetic Fields to Focus Magnetic Carriers to Targets Deep inside the Body*". Journal of Magnetism and Magnetic Materials, vol. 321, no. 10, pp. 1594–1599.

[8] Probst, R., et al. "*Planar Steering of a Single Ferrofluid Drop by Optimal Minimum Power Dynamic Feedback Control of Four Electromagnets at a Distance."* Journal of Magnetism and Magnetic Materials, vol. 323, no. 7, 2011, pp. 885–896., doi:10.1016/j.jmmm.2010.08.024.

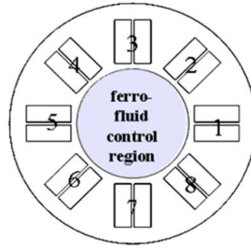


Figure 2: Shapiro's proposed experimental setup

Experimentally, we consider the work done for last year's Capstone Project at The Cooper Union on this subject matter. Four former mechanical engineering seniors created an experimental rig to control a ferrofluid droplet against a steady flow, shown in *Figure 3*.[9] The setup consists of a header tank and ball valve to control the flow rate of suspension fluid through the tubular control region, a set of electromagnets used to guide the ferrofluid towards a given setpoint, a servo-actuated timing belt system to move the magnets parallel to the axis of the tube, and a camera for sensing. Unlike Shapiro's setup, this rig accounts for the presence of a flow (albeit steady and not pulsatile), which will impact the development of the controls algorithm. However, the algorithm used is based in classical control theory; this is a poor choice for the naturally nonlinear dynamics of the system.

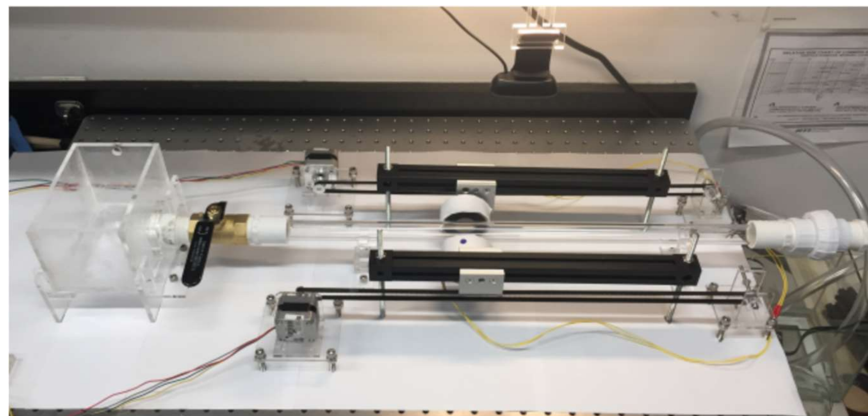


Figure 3: The Cooper Union 2017 Capstone Project experimental rig

[9] Faddoul, R., Iyengar, N., Kovalenko, A., Lacey, C., Yecko, P. (2018). *Magnetic Drug Delivery System* (Unpublished journal article). Cooper Union, New York, NY.

2.3 Goal

In this paper, we create an innovative method of passively imaging carriers in real time without using visual feedback. First, a vector magnetometer array and an algorithm are developed to track the position of a permanent magnet and a ferrofluid droplet in real time. Then, a control group is established with visual feedback to ensure proper functionality of a proposed nonlinear control algorithm. Future implementations of this method will integrate data obtained from the magnetometer sensing algorithm with the control algorithm.

3 Preliminary Design Work

A few considerations must be made regarding codes, standards, and constraints to design an algorithm and experimental setup. These help to immediately determine realistic bounds for our project. This section is dedicated to describing the decisions involved in creating our experimental setup.

3.1 Codes and Standards

The FDA outlines many regulations for drug delivery systems and magnetic imaging mechanisms. However, the scope of this project is not to make the materials for the drugs that will bind to ferrofluid, nor to make a product for an end user. The scope of the project is to prove the control of a ferrofluid droplet with passive magnetic imaging, which itself does not require as strong magnetic fields as MRI does. The exposure ceiling values for magnetic fields in humans is 8 Tesla,[10] and thus our scaled-down laboratory version of what the end product would be must take that limit into consideration. In its current state, the device is designed for the array of magnetometers and the controlling electromagnets to remain fixed, as the magnetic field does all the necessary work in pulling the ferrofluid to its desired destination. However, if scaled up, the magnetometers as well as the electromagnets may be movable devices. This product will eventually be stored in hospitals rooms, where registered technicians would be in charge of maintaining and utilizing such devices.

3.2 Criteria and Constraints

- The test rig should allow for testing in two dimensions with the versatility to extend to three dimensions

[10] AIHA NIR Committee. "Static Magnetic Field Quick Reference Sheet." *Aiha.org*, www.aiha.org/get-involved/VolunteerGroups/Documents/NONIONRAD-StaticMagneticFieldsQuickReferenceGuide.pdf.

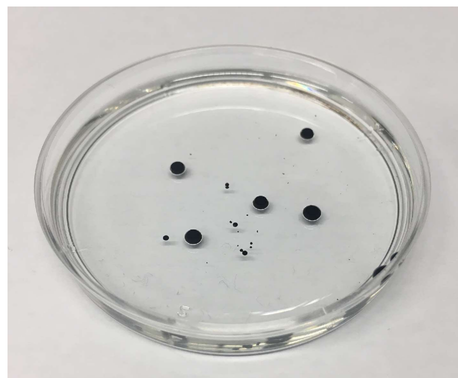
- A user should be able to easily set up and calibrate the device
- A user should be able to predefine a path for the droplet to follow to model the veins in a human
- The cost of the device must remain less than our budget of $1,500
- The experimental setup should be functional by the end of next semester

3.3 Ferrofluid Selection

A few idealizations are made to ensure that the control algorithm we implement functions properly. One of these idealizations is that the suspension fluid is sufficiently more viscous than the ferrofluid; this allows the ferrofluid to both naturally retain its shape with minimal diffusion and causes the ferrofluid to move at low speeds for stable control. Under these idealized conditions, the ferrofluid will form a distinct cluster within the surrounding fluid and does not break apart under the influence of a magnetic field.

Using information obtained from last year's Capstone Project, the ideal ferrofluid-suspension fluid mixture for our purposes is EMG 304 (a water-based ferrofluid) and vegetable oil. An image of this mixture is displayed in *Figure 4*, and the properties of the ferrofluid and of the vegetable oil are displayed in *Table 1*.

Figure 4: EMG 304 ferrofluid submerged in vegetable oil



	EMG 304 Ferrofluid	Vegetable Oil
Density	930 kg/m^3	1240 kg/m^3
Dynamic Viscosity	0.03 kg/m/s	0.04 kg/m/s
Magnetic Permeability	7.54 x 10^{-6} H/m	1.26 x 10^{-6} H/m

Table 1: Properties of ferrofluid and vegetable oil

3.4 Electromagnet Selection

The selection of the electromagnet is based on the nominal magnetic strength at the face of the electromagnet used by the Shapiro et al. setup, and reasonable design constraints such as cost and size. The magnetic field strength must be sufficiently high to pull a droplet while overcoming the countering viscous force for a long enough time to avoid overheating. Ferromagnetic particles experience a force that pulls them towards the highest magnetic field gradient based on the following equation:

$$\vec{F}_M = \frac{2\pi a^3}{3} \frac{\mu_0 \chi}{1+\chi/3} \nabla || \vec{H} ||^2$$

The magnetic force must be able to overcome the countering viscous force in all points where we want the ferrofluid to be controllable. Based on past research and simulations discussed in *Section 4*, we purchased four 16.4-ounce electromagnets from Solenoid City, with model number E-28-150. Such electromagnets have a high number of turns to output a magnetic field throughout a substantial control region that we determine based on these magnets (as shown in *Section 4.1*). They are small and inexpensive enough to fit in an experimental setup and within our budget. A picture and list of properties for the electromagnet are below:

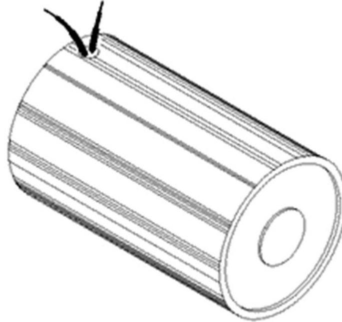


Figure 5: Selected electromagnet E-28-150 from Solenoid City

Properties of Electromagnet	
Voltage	31.6 V
Amperage	0.71 A
Resistance	41.5Ω
Number of turns	1687
Diameter	1.5”
Length	2.8”
Cost	$150

Table 2: Properties of Solenoid City tubular electromagnet, model number E-28-150

4 Theoretical Models

With the ferrofluid and electromagnets determined, an experimental setup for both controlling a magnetic object and localizing a magnetic object with a magnetometer array can effectively be made. However, to predict how the ferrofluid will move under the influence of the purchased electromagnets, it is necessary to model the magnetic field and perform a simulation. To start, the external magnetic field is characterized using Biot-Savart's law. This model will become even more important when the ferrofluid is introduced, which will be discussed in *Section 5.2*. One of the idealizations discussed in that section assumes that the ferrofluid acts as a magnetic dipole under the influence of a magnetic field, giving the shape of the induced magnetic field of the ferrofluid.

4.1 External Magnetic Field

We simulate the magnetic field in a 2 centimeter radius Petri dish surrounded by four electromagnets arranged axisymmetrically around the dish. In this study, the magnetic field at position *r* per unit of current generated by the electromagnets, $h_c(r)$, is determined from Biot-Savart's law, given by[11]:

$$h_c(r) = \frac{\mu_0 N}{4\pi l}\begin{bmatrix}\beta_1(\frac{r}{\rho}) - \beta_1(\frac{r}{\rho}+\nu)\\ \beta_2(\frac{r}{\rho}) - \beta_2(\frac{r}{\rho}+\nu)\end{bmatrix},$$

where μ_0 is the permeability of free space, *N* is the number of wire turns, *l* is the length of the electromagnet, ρ is the distance from *r* to the wire in the electromagnet, $\nu = \frac{l}{\rho}$, and β_1 and β_2 are given by:

[11] Probst, R., et al. "Planar Steering of a Single Ferrofluid Drop by Optimal Minimum Power Dynamic Feedback Control of Four Electromagnets at a Distance." *Journal of Magnetism and Magnetic Materials*, vol. 323, no. 7, 2011, pp. 885–896., doi:10.1016/j.jmmm.2010.08.024.

$$\beta_1(r) = \int_0^{2\pi} \frac{r_1(r_2\cos\theta - 1)d\theta}{((r_2-\cos\theta)^2+\sin^2\theta)\sqrt{r_1^2+(r_2-\cos\theta)^2+\sin^2\theta}}$$

and

$$\beta_2(r) = \int_0^{2\pi} \frac{\cos\theta d\theta}{\sqrt{r_1^2+(r_2-\cos\theta)^2+\sin^2\theta}}$$

A vector plot of the magnitude of the field created when all four electromagnets are turned on with an equal amount of current flowing through each one is shown in *Figure 6*.

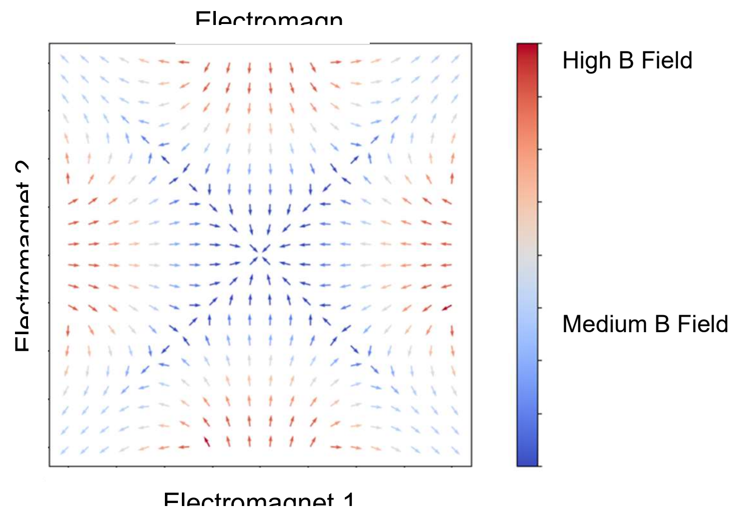


Figure 6: Vector plot of the magnetic field due to four electromagnets

Note how the above figure uses “Low B Field” and “High B Field” to denote areas of high and low magnetic field; without any form of physical validation, there is no guarantee that the magnetic field strength magnitude will actually be within the range given analytically based on the manufacturer's approximations, because they do not specify certain parameters. Hence, a validation scheme is needed; this will be discussed next.

4.1.1 *Experimental Validation*

To physically validate the above results, an experimental rig is developed and used to characterize the magnetic field. The setup consists of one electromagnet we analytically modeled in the previous section, a Lakeshore 410 Gaussmeter that measures between 10 μT and 2 T connected to a Hall probe used to measure the magnetic field along its axis (in the x direction), and a three-axis micromanipulator to hold the probe and accurately change its position in 3D space. As the probe moves around the electromagnet, it measures the magnetic field output in real-time; these data points are recorded for comparison to analytical results. A labeled schematic of the rig is shown in *Figure 7*.

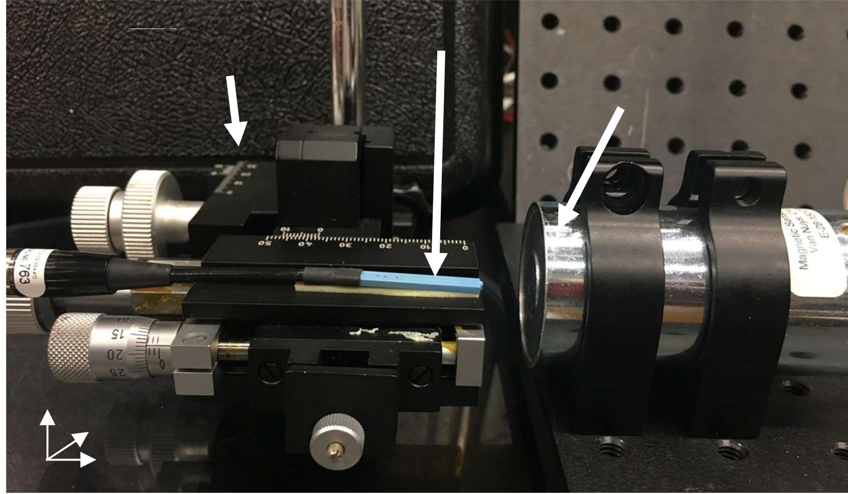


Figure 7: Magnetic field measurement experimental rig

The experimental data points serve to identify a free parameter in the magnetic field equation for the purchased electromagnets. The manufacturers do not specify the number of coils in the electromagnets and the strength of the ferrous core that amplifies the field. However, this combined constant simply scales the field. We optimize this constant using a least squares error optimizer that returns the value of this parameter that best fits the experimental points to the theoretical surface. This error is given by:

$$Error = \sum_{(x_i, y_i)=1}^{n} \left(B_{theory}(x_i, y_i) - B_{experimental}(x_i, y_i)\right)^2$$

The optimizer algorithm minimizes this error and returns the parameter in the theoretical model that achieves the minimum error.

An analytical surface plot of the magnetic field produced by a single electromagnet with this parameter is created and the experimentally measured points are overlaid, shown in *Figure 8*. The positions of the data points have error bars in the x and y directions for the instrumental uncertainty of the micromanipulator, and the magnitude of the magnetic field has uncertainty due to both the micromanipulator and the instrumental error in the Gaussmeter. The error bars represent the larger of instrumental and experimental errors. The code used to produce the plot in this figure is given in *Appendix A*.

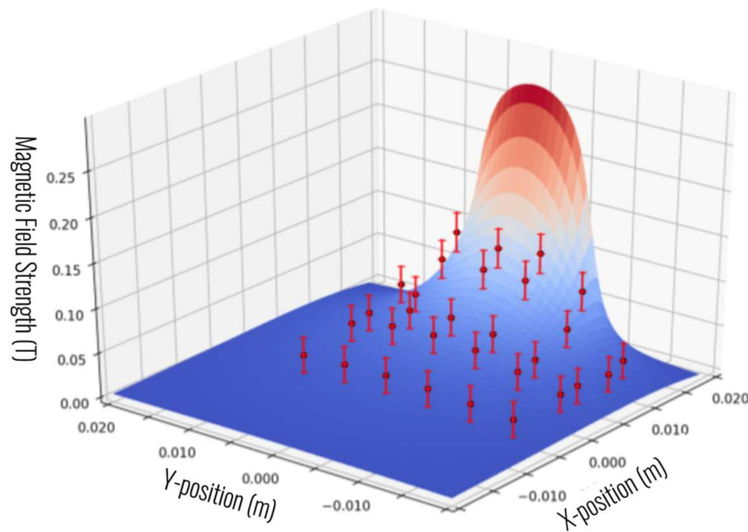


Figure 8: Analytical surface plot of B_x field with electromagnet centered at x = 0.02 m with experimental points overlaid

For further experiments and theoretical models, we simply back-calculate the number of turns in the electromagnet using the formula for the unknown parameter:

$$k = \frac{\mu_0 N}{4\pi l}$$

Solving for N and plugging in the optimal $k = 0.014$ H/m^2, we find that the electromagnets effectively have N = 1,687 turns each.

5 Tracking Algorithm

We implement a least squares error minimization optimizer (dubbed Sequential Least Squares Programming, or SLSQP), where we minimize the error between the recorded magnetic field and a simulated magnetic field within the control domain. This minimum error approximates the position of a magnetic particle within the region. Positional constraints are placed to keep the magnetic particle's location within the control domain, and magnetization constraints are placed to ensure that the orientation of the magnetic particle does not heavily impact its computed position. To aid in convergence, the optimizer initially starts at the center of the control domain, and the initial magnetization is randomized such that more accurate solutions are found faster. The next sections describe the physical setups where we get the recorded data as well as the theoretical models for the simulated data for both a permanent magnet and a ferrofluid droplet.

A GUI has been developed to easily display data recorded from the magnetometers and the calculated position on an XY plane, shown in *Figure 9*.

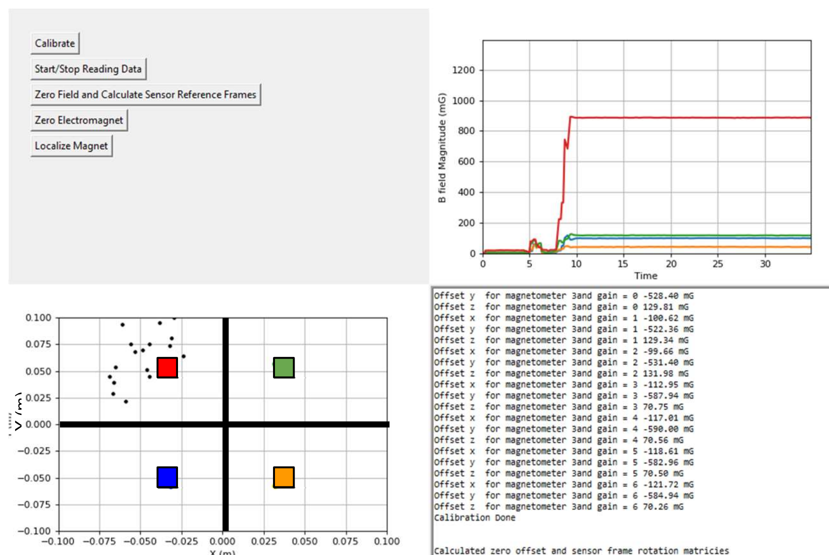


Figure 9: Position tracking GUI: A) Calibration, calculation, and localization buttons. B) Magnetometer recording output. C) Localization results

Slight manufacturing imperfections and residual magnetic fields caused by nearby magnetic objects (a term dubbed "hard-iron distortions") cause each magnetometer to read a different magnetic field value, even only under the influence of Earth's relatively constant magnetic field. Without a common magnetic field baseline to read off of for each magnetometer, localizing a magnetic particle would not be possible. To correct these distortions, a scaling factor is first determined through a calibration algorithm. Then, an offset factor is determined by rotating the entire setup on a turntable while the magnetometers output their maximum and minimum magnetic field values. Each recording is offset by a constant amount until the maxima and minima match. When this happens, all magnetometers read roughly the same value under Earth's magnetic field in any orientation, indicating that the magnetometers have been successfully calibrated. This procedure is carried out when the "Calibrate" button in *Figure 9* A) is pressed.To prevent any issues with wiring while the setup is rotating, a Bluetooth connection is established between the setup and the computer running the algorithm.

Next, for ease of calculation, Earth's magnetic field is uniformly subtracted from each magnetometer. Additionally, the magnetic field readings must all be based on the same reference frame to allow the recorded field to be compared to the field we simulate. A series of rotation matrices convert the magnetic field readings from the magnetometers' reference frames to a common reference frame. The "Zero Field and Calculate Sensor Reference Frames" button in *Figure 9* A) performs the operations described above. The "Zero Electromagnet" button in *Figure 9* A) subtracts the simulated magnetic field created by a nearby electromagnet when tracking a ferrofluid droplet, discussed in *Section 5.2.*

The "Start/Stop Reading Data" button in *Figure 9* A) reads the magnetic field measurements from each magnetometer and plots the field as a function of time in *Figure 9* B). The four colors on the plot represent the reading from the four magnetometers, and spikes in the data indicate that a magnetic particle is close to the corresponding magnetometer. The "Localize Magnet" button in *Figure 9* A) implements the localization algorithm described above to find the approximate position of the magnetic object projected onto the XY plane. The results of this algorithm are plotted as black markers in the scatter plot shown in *Figure 9* C), while the color-coded positions of the magnetometers are plotted as red, blue, green, and orange markers in the same scatter plot. In *Figure 9*, the permanent magnet is placed close to the magnetometer corresponding to the red color; the red line in *Figure 9* B) spikes and the black markers in *Figure 9* C) are concentrated close to the red marker.

5.1 Tracking a Permanent Magnet

To develop an effective framework for localizing a magnetic object in space without using a material as complicated as a ferrofluid droplet, a permanent magnet is first tracked. The experimental setup used to collect data is a planar array of magnetometers, shown in *Figure 10*. Four MAG 3110, digital 3-axis magnetometers are used on a SparkFun breakout board. These are low cost (approximately 25 USD) and relatively precise. The localization algorithm only requires three magnetometers to fully locate a magnetic object; four magnetometers are used in the current configuration for more accurate tracking. Not shown in *Figure 10* is a three-axis micromanipulator used to precisely locate the permanent magnet in three-dimensional space. These coordinates can be directly compared to the output of the tracking algorithm.

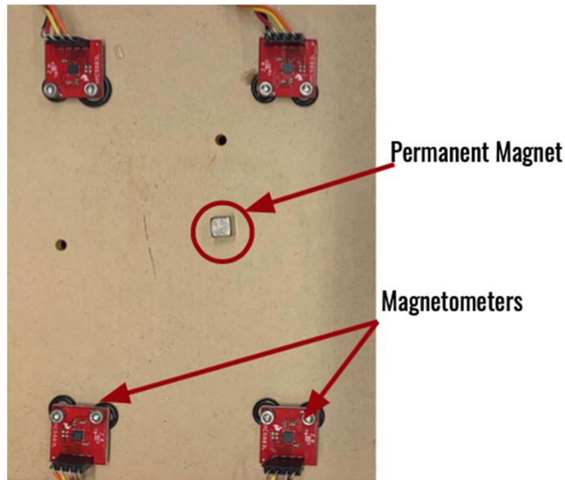


Figure 10:Permanent magnet localization setup

Next, we model this physical setup. Permanent magnets are ferromagnetic, which means they retain magnetic properties even without an external magnetic field. The permanent magnet can be approximated as a dipole for any distance greater than half the diameter of the magnet, providing an approximate analytical expression for the magnet's magnetic field. This expression is given by :

$$B_{simulated} = \frac{\mu_0}{4\pi}(\frac{3\hat{r}(\vec{m}\cdot\hat{r})}{r^3} - \frac{\vec{m}}{r^3})$$

The simulation of this field is shown in *Figure 11*. The magnetization of the permanent magnet is determined by a regression analysis that compares the magnetic field magnitude to its absolute position.

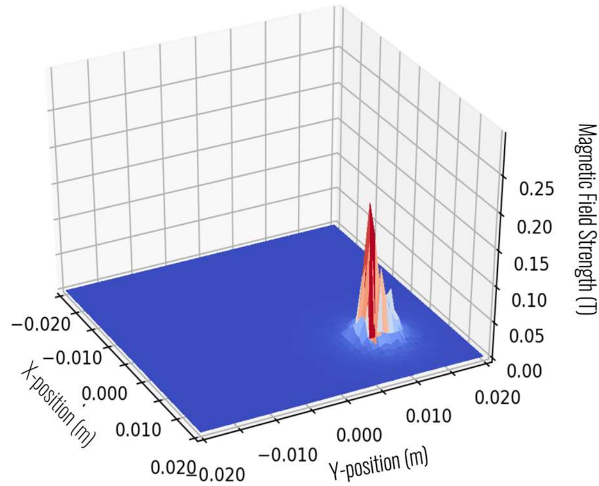


Figure 11: Simulated magnetic field due to a permanent magnet

---

5.2 Tracking a Ferrofluid Droplet

With a framework successfully established, the algorithm and experimental setup is extended to track a ferrofluid droplet. Ferrofluid is paramagnetic, meaning that it does not retain magnetic properties if it is not in the presence of an external magnetic field. The magnetic properties of a ferrofluid droplet are essential in the operation of magnetic drug therapy; it is the means by which the magnetic drugs are pulled to the desired location. Thus, in order to ensure magnetic properties in a ferrofluid droplet, an external magnet must be added to the setup. The experimental setup used to collect magnetometer data is shown in *Figure 12*.

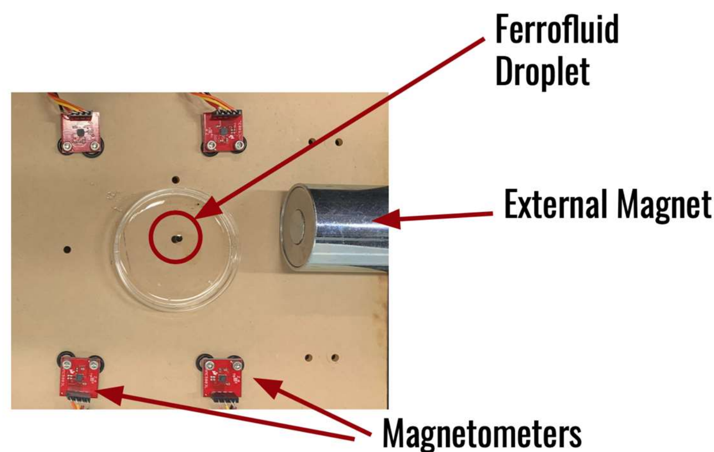


Figure 12:Ferrofluid localization setup

Similar to the permanent magnet, the ferrofluid droplet is approximated as a dipole; however, the magnetization factor is no longer a constant, but it is now a function of the external field. The magnetization increases until it reaches saturation. The simulation of both the external field and the ferrofluid droplet as a dipole with arbitrary magnetization is shown in *Figure 13*.

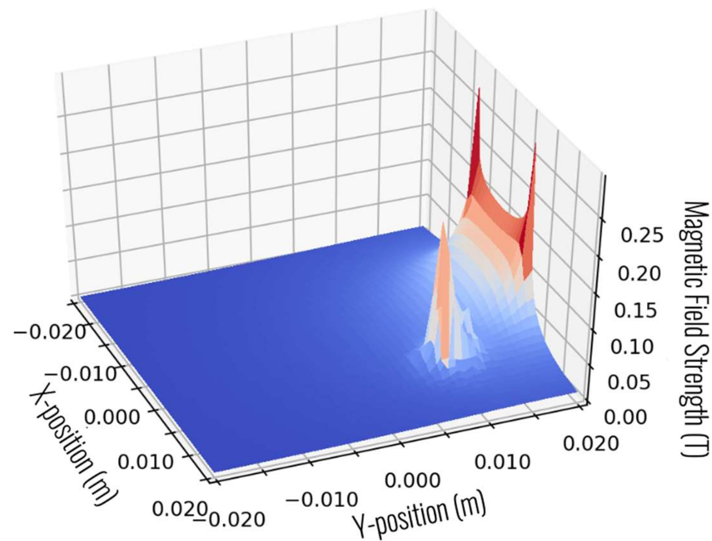


Figure 13: Simulated magnetic field due to external magnet and field induced by the ferrofluid droplet.

6 Controls Experimental Setup

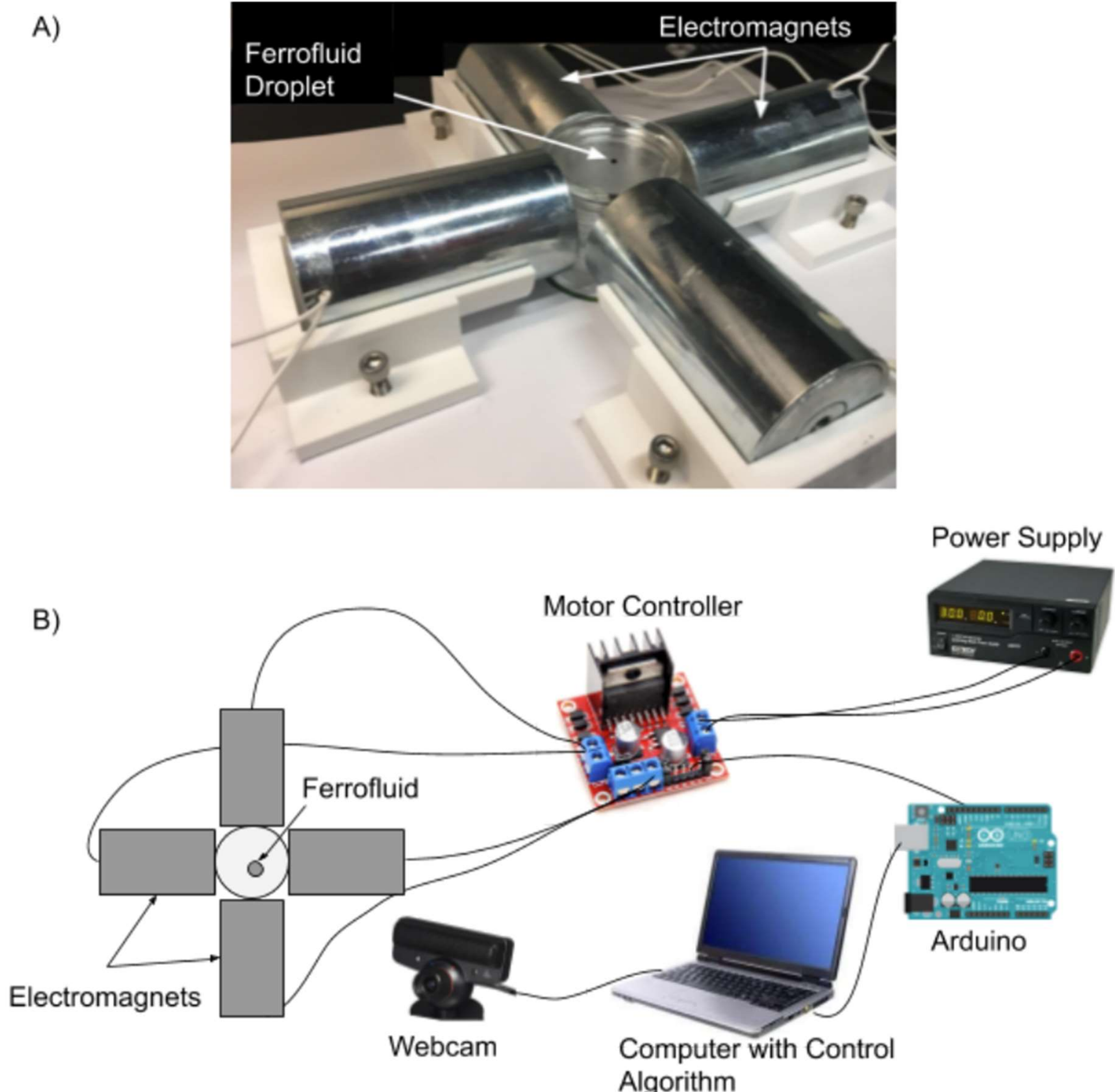


Figure 14: A) Experimental setup of four electromagnets surrounding Petri dish with ferrofluid droplet inside. B) Schematic of controls hardware setup.

To test the sensing mechanism, we construct an experimental setup for the control of a droplet. However, to use it as a control group, we use visual tracking, which will eventually be swapped out with the previously explained magnetometer array and compared to the visual tracking performance. The experimental setup is provided in *Figure 14.* The current setup is able to move a ferrofluid droplet in a control domain to a desired setpoint in 2D space. The control domain consists of a Petri dish inside which a ferrofluid droplet is placed, a set of four

electromagnets surrounding the Petri dish which creates a magnetic field to pull the ferrofluid in the desired direction, and an overhead camera to act as a temporary sensing mechanism before being replaced by a magnetometer array.

First, a webcam relays the current position of the ferrofluid to the control algorithm, which then outputs four voltage commands to the Arduino. Once the Arduino interprets the output of the controller, represented by an array of numbers between -255 and 255, it sends a pulse width modulated (PWM) signal to the L298N dual H-bridge motor controller. The motor controller is receiving a constant 30 Volts from the power supply, so it uses the PWM signal to control the effective voltage given to the electromagnets.

6.1 Control Algorithm

Based on the current position of the ferrofluid and a desired position, the control algorithm is able to successfully actuate each of the electromagnets to allow movement of a ferrofluid droplet to a setpoint. This algorithm takes advantage of the nonlinear nature of the magnetic fields to seek a solution that minimizes error, time spent, and power exerted on electromagnets simultaneously. To do so, the controller is split into three separate sections: a linear segment to scale the error between current and desired ferrofluid position using an experimentally-determined proportional gain, a nonlinear segment to determine the amount of voltage that must be sent to each electromagnet to obtain a certain magnetic field to pull the ferrofluid in the desired direction, and a nonlinear filter to ensure that the changes in voltage and magnetic field are smooth and natural for the ferrofluid as it travels along its prescribed path. Each section is discussed in detail in *Appendix B*.

6.2 OpenCV

The visual feedback system chosen for our control group consists of a Logitech C270 webcam and OpenCV: an open-source computer vision library available on Python 3.5. The

webcam has a resolution of 720p, which is more than enough resolution to track a particle moving at speeds on the order of 1 mm/s. OpenCV works by searching for a set of pixel colors based on hue, saturation, and value (dubbed the HSV color range) and filtering based on the concentration of similarly-colored pixels in the same area. In our case, we use OpenCV to locate the black ferrofluid droplet within the control domain of the Petri dish. However, if there are other black objects in view of the webcam but outside of the Petri dish, how can we guarantee that we select the ferrofluid alone? To answer this question, we created a script with a GUI that searches for the boundary of the Petri dish, ignores all images outside of that boundary, and searches for the largest black object within the control domain. A picture of this GUI is shown in *Figure 16*.

Figure 15: Dark green Petri dish boundary

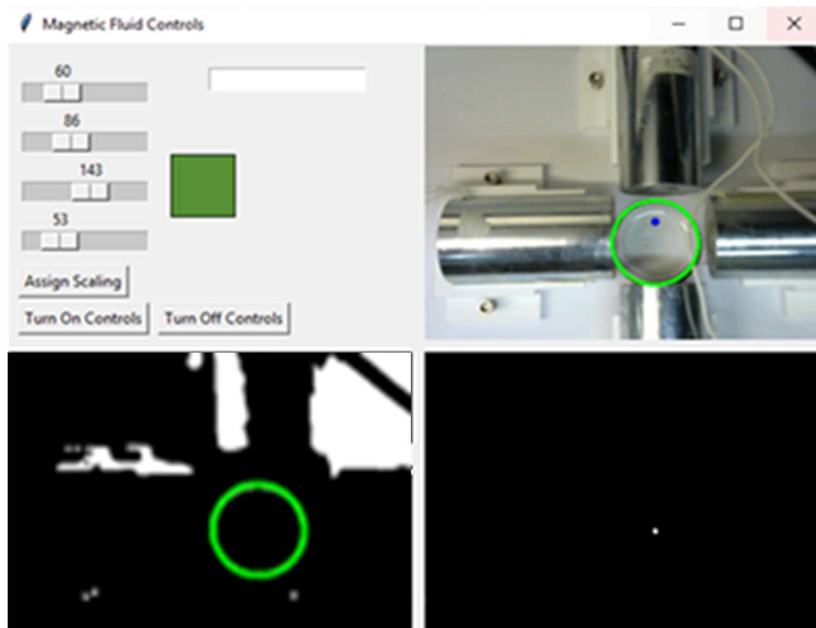


Figure 16: OpenCV GUI: A) HSC sliders and color output. B) Petri dish locator in black and white image C) Fixed Petri dish boundary in colored image D) Ferrofluid droplet locator in black and white image

The script is first used to locate the boundary of the Petri dish. To do so, a thick green circle is placed just below the Petri dish, shown in *Figure 15*, and OpenCV is used to localize the

circle in the HSV color space, as was described above. Hue, saturation, and value are represented by the second, third, and fourth respective sliders in *Figure 16 A).* The color represented by the three values we input on the sliders is displayed just to the right of the sliders. In the GUI, the Petri dish boundary is displayed as a bright green circle on an otherwise black and white image in *Figure 16 B)*, and once the "Assign Scaling" button is selected, this boundary is reproduced in a fully colorized version of the webcam view in *Figure 16 C)*.

With the bounds of the Petri dish determined, the script then ignores any images outside of the boundary. Thus, the control domain of the visual feedback system is effectively restricted to the Petri dish. Once this is done, the ferrofluid can be found by by adjusting the image sensitivity to black through the first slider in *Figure 16 A)*. The ferrofluid droplet appears as white in the black and white image in *Figure 16 D)*. When the user presses the "Turn On Controls" button, the control algorithm described in *Section 5.1* begins to run, and the control algorithm is deactivated when the "Turn Off Controls" button is pressed. The script containing this GUI is in *Appendix D*.

The OpenCV setup is solely created to serve as a control group for the magnetometer array. In the next section, we discuss the current experimental and analytical progress made using these magnetometers.

7 Conclusion

The sensing algorithm and setup we propose will allow for real time sensing of magnetic nanoparticles without use of visual feedback. This bridges the gap between the research done on magnetic drug therapy in idealized lab settings and the difficult conditions the drugs will have to be controlled through once inside a patient.

8 Future Work

To extend this work, data obtained from the magnetometer sensing algorithm will be integrated into the control algorithm. Then, the outcome of the controls can be evaluated, and the two different sensing methods will be compared. Further iterations of the algorithm will work to decrease the run time and increase accuracy. One iteration we propose in the magnetometer array is placing the magnetometers on different planes to restrict the localization solutions even more. Finally, the algorithm and controls can both be extended to three-dimensional space.

9 Appendix

9.1 Appendix A: Analytical Plot of B_x Field with Experimental Validation Code

```
import numpy as np
import scipy as sp
from scipy import integrate, LowLevelCallable
import os, ctypes

from matplotlib import animation
import matplotlib.pyplot as plt
import matplotlib.colors as colors
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.cm as cm
import time
from tqdm import tqdm

#Origin of Global Coordinates exists at center of Petri dish

# Define locations of center of front face of electomagnet: top, bottom, right, left
electromagnet_locs = np.array([[0.0,0.02], [0.0,-0.02], [0.02,0.0], [-0.02,0.0]])
mu_0 = (np.pi)*4e-7 #H/m
l = 0.0714 #m
a = 0.007 #m
N= 10000#turns of magnet
I = 1#A
k = mu_0*N/4/np.pi/l

nu = np.array([l/a,0])

Current_Vector = np.array([-1, 1, -1, 1], dtype = float)

lib = ctypes.CDLL(os.path.abspath('integrands.so'))
lib.integrand1.restype = ctypes.c_double
lib.integrand1.argtypes = (ctypes.c_int, ctypes.POINTER(ctypes.c_double), ctypes.c_void_p)
lib.integrand2.restype = ctypes.c_double
lib.integrand2.argtypes = (ctypes.c_int, ctypes.POINTER(ctypes.c_double), ctypes.c_void_p)

def B_point(Current_Vector,X,Y):

    def integrand1(theta, vec):
        return vec[0]/((vec[1]-np.cos(theta))**2+np.sin(theta)**2)*(vec[1]*np.cos(theta)-1)/(vec[0]**2+(vec[1]-
np.cos(theta))**2+np.sin(theta)**2)**(1/2)

    def integrand2(theta, vec):
        return (np.cos(theta))/(vec[0]**2+(vec[1]-np.cos(theta))**2+np.sin(theta)**2)**(1/2)

    local_vecs = np.zeros((4,2), dtype = float)
    # Convert Global X and Y into local x and y for all 4 electromagnets where x = distance from face, y = distance from outward normal vector
from center of electromagnet
    local_vecs[0,:] = [np.abs(Y-electromagnet_locs[0,1]), -(X-electromagnet_locs[0,0])]
    local_vecs[1,:] = [np.abs(Y-electromagnet_locs[1,1]), X-electromagnet_locs[1,0]]
    local_vecs[2,:] = [np.abs(X-electromagnet_locs[2,0]), Y-electromagnet_locs[2,1]]
    local_vecs[3,:] = [np.abs(X-electromagnet_locs[3,0]), -(Y-electromagnet_locs[3,1])]

    #Calculate B components
    B11 = np.zeros(4)
    B12 = np.zeros(4)
    B21 = np.zeros(4)
    B22 = np.zeros(4)

    for i in range(4):
        B11[i] = sp.integrate.quad(integrand1, 0, 2*np.pi,args=(local_vecs[i,:]/a))[0]
        B12[i] = sp.integrate.quad(integrand1, 0, 2*np.pi,args=(local_vecs[i,:]/a+nu))[0]
        B21[i] = sp.integrate.quad(integrand2, 0, 2*np.pi,args=(local_vecs[i,:]/a))[0]
        B22[i] = sp.integrate.quad(integrand2, 0, 2*np.pi,args=(local_vecs[i,:]/a+nu))[0]

    h_x = np.multiply(k*(B11-B12), Current_Vector)
    h_y = np.multiply(k*(B21-B22), Current_Vector)
```

```
    B_point_x = h_x[3]-h_x[2]+h_y[1]-h_y[0]
    B_point_y = -h_y[3]+h_y[2]+h_x[1]-h_x[0]

    return np.array([B_point_x, B_point_y])

def B_point_C_function(Current_Vector,X,Y):

    def integrand1(vec):
        c = (ctypes.c_float * len(vec))(*vec)
        user_data = ctypes.cast(ctypes.pointer(c), ctypes.c_void_p)
        return LowLevelCallable(lib.integrand1, user_data)

    def integrand2(vec):
        c = (ctypes.c_float * len(vec))(*vec)
        user_data = ctypes.cast(ctypes.pointer(c), ctypes.c_void_p)
        return LowLevelCallable(lib.integrand2, user_data)

    local_vecs = np.zeros((4,2), dtype = float)
    # Convert Global X and Y into local x and y for all 4 electromagnets where x = distance from face, y = distance from outward normal vector
from center of electromagnet
    local_vecs[0,:] = [np.abs(Y-electromagnet_locs[0,1]), -(X-electromagnet_locs[0,0])]
    local_vecs[1,:] = [np.abs(Y-electromagnet_locs[1,1]), X-electromagnet_locs[1,0]]
    local_vecs[2,:] = [np.abs(X-electromagnet_locs[2,0]), Y-electromagnet_locs[2,1]]
    local_vecs[3,:] = [np.abs(X-electromagnet_locs[3,0]), -(Y-electromagnet_locs[3,1])]

    #Calculate B components
    B11 = np.zeros(4)
    B12 = np.zeros(4)
    B21 = np.zeros(4)
    B22 = np.zeros(4)

    for i in range(4):
        B11[i] = sp.integrate.quad(integrand1(local_vecs[i,:]/a), 0, 2*np.pi)[0]
        B12[i] = sp.integrate.quad(integrand1(local_vecs[i,:]/a+nu), 0, 2*np.pi)[0]
        B21[i] = sp.integrate.quad(integrand2(local_vecs[i,:]/a), 0, 2*np.pi)[0]
        B22[i] = sp.integrate.quad(integrand2(local_vecs[i,:]/a+nu), 0, 2*np.pi)[0]

    h_x = np.multiply(k*(B11-B12), Current_Vector)
    h_y = np.multiply(k*(B21-B22), Current_Vector)

    B_point_x = h_x[3]-h_x[2]+h_y[1]-h_y[0]
    B_point_y = -h_y[3]+h_y[2]+h_x[1]-h_x[0]

    return B_point_x, B_point_y, [-h_y[0],-h_x[0]]

def Gradient_B_field(Current_Vector,X,Y, h):
    Gradient_X = (np.linalg.norm(B_point(Current_Vector,X+h/2,Y))**2- np.linalg.norm(B_point(Current_Vector,X-h/2,Y))**2)/h
    Gradient_Y = (np.linalg.norm(B_point(Current_Vector,X,Y+h/2))**2- np.linalg.norm(B_point(Current_Vector,X,Y-h/2))**2)/h

    return Gradient_X, Gradient_Y

STEPS = 20

X = np.linspace(-0.02,0.02,STEPS)
Y = np.linspace(-0.02,0.02,STEPS)

B_field_x = np.zeros((STEPS,STEPS))
B_field_y = np.zeros((STEPS,STEPS))
Gradient_X = np.zeros((STEPS,STEPS))
Gradient_Y = np.zeros((STEPS,STEPS))

xv, yv = np.meshgrid(X,Y)

time_original = time.clock()

for i in tqdm(range(STEPS)):
    for j in range(STEPS):
        B_field_x[i,j], B_field_y[i,j] = B_point(Current_Vector,xv[i,j],yv[i,j])[0:2]
        Gradient_X[i,j], Gradient_Y[i,j] = Gradient_B_field(Current_Vector,xv[i,j],yv[i,j], 10e-10)
```

```
#print((time.clock()-time_original)/100/100)

plt.close('all')
B_sort = np.sort(np.sqrt(np.square(Gradient_X)+np.square(Gradient_Y)), axis=None)
    #print(effsort)
colorbar_divisions = np.zeros((11))
for i in range(1,11):
    colorbar_divisions[i] = np.average(B_sort[int((i-1)*np.size(B_sort)/10):int(i*np.size(B_sort)/10)])
    colorbar_divisions[0] = np.min(B_sort)
    colorbar_divisions[10] = np.max(B_sort)

norm = colors.BoundaryNorm(boundaries=colorbar_divisions, ncolors=8)
fig1 = plt.figure()
#fig2 = plt.figure()
fig2= plt.figure()

plt.gca().set_aspect('equal', adjustable='box')
color= np.sqrt(np.square(Gradient_X)+np.square(Gradient_Y))
norm.autoscale(color)

ax1 = fig1.add_subplot(111, projection='3d')
ax2 = fig2.add_subplot(111, projection='3d')
ax1.quiver(X,Y,Gradient_X/np.sqrt(np.square(Gradient_X)+np.square(Gradient_Y)),
Gradient_Y/np.sqrt(np.square(Gradient_X)+np.square(Gradient_Y)), norm(color).data, cmap = 'coolwarm')
cbar = plt.colorbar()
plt.quiver(X,Y,Gradient_X/np.sqrt(np.square(Gradient_X)+np.square(Gradient_Y)),
B_field_y/np.sqrt(np.square(B_field_x)+np.square(B_field_y)), norm(color).data, cmap = 'coolwarm')
cbar.set_ticks(9)
cbar.ax.set_yticklabels(colorbar_divisions)
#ax1.plot_surface(xv, yv, B_field_x, color='r')
ax1.plot_surface(xv, yv, B_field_x, cmap = 'coolwarm', edgecolors='k')
plt.title('X Component of B field for Equal Voltage in all Magnets')
ax1.set_facecolor((0,0,0,0))
#ax2.plot_surface(xv, yv, B_field_y, color='b')

#plt.show()

# rotate the axes and update
"""
fig = plt.figure()
ax1 = Axes3D(fig)

def init():
    ax1.plot_surface(xv, yv, B_field_x, color='r')
    return fig,

def animate(i):
    ax1.view_init(elev=10., azim=i)
    return fig,

anim = animation.FuncAnimation(fig, animate, init_func=init,
                               frames=360, interval=20, blit=True)
anim.save('basic_animation.mp4')
"""
```

9.2 Appendix B: Description of Control Algorithm

9.2.1 Linear Segment

First, the linear segment of the controller inputs the desired ferrofluid position and the actual ferrofluid position and outputs an error proportional to the magnetic force the ferrofluid experiences. The expression for the error *x(t)* is given by:

$$x(t) = k_p(r_d(t) - r(t)) \quad ,$$

where k_p is the proportional gain of the linear controller, $r_d(t)$ is the desired position of the ferrofluid as a function of time, and $r(t)$ is the current position of the ferrofluid as a function of time. The k_p of this controller was determined experimentally to be around 0.49, though this number is subject to change based on the concavity of the curve chosen for the ferrofluid to move along (e.g. if the path is along a straight line or along a curve). Alternatively, this error is shown to be equivalent to[12]:

$$x(t) = \nabla \|H(r)y(t)\|^2 = g(r, y),$$

where $y(t)$ is the state vector describing the amount of voltage in each electromagnet as a function of time, and $H(r)$ is:

$$H(r) = \frac{1}{R}\begin{bmatrix} h_1(r) & h_2(r) & \dots & h_n(r) \end{bmatrix},$$

where R is the resistance in each electromagnet and $h_i(r)$ for $i = 1, 2, ..., n$ is the ith electromagnet's theoretical magnetic field strength (derived in *Section 4.1* of the report) in a system of n electromagnets.

9.2.2 Nonlinear Segment

The nonlinear segment of the controller takes the error from the linear segment of the controller as an input, and outputs a requested voltage for each electromagnet. Thus, the voltage output y(t) must be put in terms of ferrofluid position r and error x. That is,

$$y = g^{-1}(r, x)$$

[12] Probst, R. et al. "Planar Steering of a Single Ferrofluid Drop by Optimal Minimum Power Dynamic Feedback Control of Four Electromagnets at a Distance." *Journal of magnetism and magnetic materials* 323.7 (2011): 885–896. *PMC*. Web. 30 Sept. 2018.

To start, however, we will first investigate how *g(r,y)* = *x* can most efficiently be written to allow for the simplest inversion process.. Letting (ρ, ϕ) be the polar representation for the parameter vector, the above equality can be written elegantly as:

$$F(\phi)H_e(r)y = \rho e + \frac{1}{\rho}Ex,$$

where *e* is the first basis vector in $\Re^4$. That is:

$$e = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Next, *E* is given by:

$$E = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ \frac{1}{2} & 0 \\ 0 & \frac{1}{2} \end{bmatrix}$$

Additionally, $F(\phi)$ is given by:

$$F(\phi) = \begin{bmatrix} \cos(\phi) & sin(\phi) & 0 & 0 & 0 \\ -\sin(\phi) & \cos(\phi) & 0 & 0 & 0 \\ 0 & 0 & \cos(\phi) & \sin(\phi) & 0 \\ 0 & 0 & 0 & \cos(\phi) & \sin(\phi) \end{bmatrix}$$

That is, $F(\phi)$ is the rotation matrix in 2D space. Finally, $H_e(r)$ is given by:

$$H_e(r) = \begin{bmatrix} H(r) \\ \frac{\partial H(r)}{\partial r_1} \\ \frac{\partial \begin{bmatrix} 0 & 1 \end{bmatrix} H(r)}{\partial r_2} \end{bmatrix}$$

A unique solution to this equation when solving for *y* can be obtained when both sides are multiplied by the inverse of $F(\phi)H_e(r)$. Solving for *y* yields:

$$y(r, x, \rho, \phi) = \rho(r, x, \phi)b_1 + \frac{b_2 x}{\rho(r, x, \phi)},$$

where b_1 is given by:

$$b_1 = \left[F(\phi)H_e(r)\right]^{-1} e$$

and b_2 is given by:

$$b_2 = \left[F(\phi)H_e(r)\right]^{-1} E$$

Among all the solutions given by this expression, we would like to obtain the one that minimizes the quadratic cost function:

$$J = y^T W(r) y,$$

where $W(r)$ is the weight matrix of the cost function. This cost function would allow the obtained solution to minimize both the power consumed by the electromagnets and the the time for the ferrofluid to move to its desired setpoint. The weight matrix helps accomplish this goal by correcting the magnetic field more quickly and more dramatically when the ferrofluid is close to one of the electromagnets; a further improvement to this controls algorithm is required to prevent the ferrofluid from getting attracted too quickly to the electromagnet since its speed rises dramatically close to a steeper change in magnetic field. The expression for the weight matrix is given by:

$$W(r) = \epsilon I_n + \frac{H_e^T(r)H_e(r)}{\|H_e^T(r)H_e(r)\|}$$

where ϵ is a small positive number and I_n is the n x n identity matrix.

To minimize the cost function J, we substitute our solution for y into J, fix variables r and x, and minimize J with respect to ρ and ϕ. This minimization can occur in two steps: minimizing J with respect to ρ with ϕ fixed, then minimizing J with respect to ϕ. Fixing ϕ and minimizing with respect to ρ yields the following solution to the minimized ρ (denoted as ρ^*):

$$\rho^*(r, x, \phi) = \left(\frac{x^T b_2^T W(r) b_2 x}{b_1^T W(r) b_1}\right)^{\frac{1}{4}}$$

Now, ρ^* is plugged back into our solution for *y* to obtain a solution that is optimized with respect to ρ (denoted as y^*_ρ):

$$y^*_\rho = \rho^*(r,x,\phi)b_1 + \frac{b_2 x}{\rho^*(r,x,\phi)}$$

Finally, we minimize the cost function with respect to ϕ. This fully optimized cost function (denoted as J^*_ρ) is given by:

$$J^*_\rho(r,x,\phi) = (y^*_\rho(r,x,\phi))^T W(r) y^*_\rho(r,x,\phi)$$

By solving the optimization problem for minimum ϕ (denoted as ϕ^*):

$$\phi^*(r,x) = \min_{\phi \in [0,\pi)} J^*_\rho(r,x,\phi),$$

the inverse mapping $y = g^{-1}(r,x)$ can finally be obtained as:

$$g^{-1}(r,x) = y^*_\rho(r,x,\phi^*(r,x))$$

This extraordinarily long and complicated procedure runs surprisingly quickly in Python, yielding accurate solutions within a single second per time step. However, the results obtained between time steps are often spatially discontinuous, leading to sharp errors as the ferrofluid moves along its trajectory. In an effort to mitigate these errors, a nonlinear filter is created to both smooth out the data obtained and to maintain the equality of *g(r,y)* and *x*. This filter is discussed in *Section 5.1.3*.

9.2.3 Nonlinear Filter

As mentioned in the previous section, the nonlinear filter is necessary to simultaneously smooth consecutive data points and to satisfy the condition that *x = g(r,y)*. To do so, the initial unsmoothed output of the nonlinear segment of the controller (denoted $\tilde{z}_k$ to avoid confusion with the previous section's results) is weighted by a constant factor $\lambda \in (0,1)$ and is optimized

to minimize the amount of error. This nonlinear filter can be represented by the recursive equation:

$$\tilde{y}_k = \psi((1-\lambda)\tilde{y}_{k-1} + \lambda\tilde{z}_k, r_k, \tilde{x}_k)$$

with initial condition:

$$\tilde{y}_0 = \tilde{z}_0,$$

where $\tilde{y}_k$ is the smoothed final output of the controller at iteration *k*, r_k is the position of the ferrofluid at iteration *k*, $\tilde{x}_k$ is the smoothed error between the desired and actual position of the ferrofluid at iteration *k*, and ψ is a function defined by the optimization problem:

$$\psi(w, r, x) = \min_{\xi \in \left\{ y \in \Re^n | \nabla \| H(r) y) \|^2 = x \right\}} \|\xi - w\|$$

Thus, the recursive equation updates $\tilde{y}_k$ with the closest vector ξ^* to $(1-\lambda)\tilde{y}_{k-1} + \lambda\tilde{z}_k$ that also satisfies $g(r_k, \xi^*) = \tilde{x}_k$ at iteration *k*.

To compute this nonlinear map, an optimization problem fairly similar to the one posed in the previous section is posed. That is, the cost function *J*, given by:

$$J = \left\| \rho b_1(r, \phi) + \frac{1}{\rho} b_2(r, \phi) x - w \right\|^2,$$

is minimized with respect to ρ and ϕ. Similarly to the previous section, we can minimize *J* with respect to ρ with ϕ fixed, then minimize *J* with respect to ϕ. First, the optimized ρ (denoted as ρ^*) is given as one of the roots to the polynomial equation:

$$b_1^T b_1 {\rho^*}^4 - b_1^T w {\rho^*}^3 + x^T b_2^T w \rho^* - x^T b_2^T b_2 x = 0$$

This optimized ρ can be plugged back into the cost function expression to get the optimized cost function (denoted as J_ρ^*); this expression is given by:

$$J_\rho^* = b_1^T b_1 {\rho^*}^2 - 2 b_1^T w \rho - 2 x^T b_2^T w {\rho^*}^{-1} + x^T b_2^T b_2 x {\rho^*}^{-2} + 2 b_1^T b_2 x + w^T w$$

Finally, we minimize the cost function with respect to ϕ (denoted as ϕ^*). This expression is given by:

$$\phi^*(r,x,w) = \min_{\phi\in[0,\pi)} J^*_\rho(r,x,w,\phi)$$

With these optimal values obtained, ψ can be calculated as:

$$\psi(w,r,x) = \rho^*(w,r,x)b_1(r,\phi^*(w,r,x)) + \frac{1}{\rho^*(w,r,x)}b_2(r,\phi^*(w,r,x))x$$

With the ψ function defined, it is straightforward to apply the recursive relation defined above to ψ. All of the equations provided above are applied to our control algorithm, and are used to move the ferrofluid in space under time and power constraints as prescribed by the weight matrix. This code is reproduced in *Appendix B.*

9.3 Appendix C: Control Algorithm Code

```
import numpy as np
import scipy as sp
import scipy.integrate
import scipy.optimize
import time

class RandomDisplacementBounds(object):
    """random displacement with bounds"""
    def __init__(self, xmin, xmax, stepsize=np.pi/2):
        self.xmin = xmin
        self.xmax = xmax
        self.stepsize = stepsize

    def __call__(self, x):
        """take a random step but ensure the new position is within the bounds"""
        return np.clip( x + np.random.uniform(-self.stepsize, self.stepsize, np.shape(x)), self.xmin, self.xmax )

#Origin of Global Coordinates exists at center of Petri dish

# Define locations of center of front face of electomagnet: top, bottom, right, left
electromagnet_locs = np.array([[0.0,0.02], [0.0,-0.02], [0.02,0.0], [-0.02,0.0]])
mu_0 = (np.pi)*4e-7 #H/m
l = 0.0714 #m
a = 0.007 #m
N= 1570*.6#turns of magnet This is way off
I = 1.#A
k = mu_0*N/4/np.pi/l
R = 100. #Ohms Internal Resistance in wire

nu = np.array([l/a,0])

Current_Vector = np.array([1, 1, 1, 1], dtype = float)

def B_point(Current_Array, Particle_Position):

    def integrand1(theta, vec):
        return vec[0]/((vec[1]-np.cos(theta))**2+np.sin(theta)**2)*(vec[1]*np.cos(theta)-1)/(vec[0]**2+(vec[1]-
np.cos(theta))**2+np.sin(theta)**2)**(1/2)
```

```
    def integrand2(theta, vec):
        return (np.cos(theta))/(vec[0]**2+(vec[1]-np.cos(theta))**2+np.sin(theta)**2)**(1/2)

    local_vecs = np.zeros((4,2), dtype = float)
    # Convert Global X and Y into local x and y for all 4 electromagnets where x = distance from face, y = distance from outward normal vector
from center of electromagnet
    local_vecs[0,:] = [np.abs(Particle_Position[1]-electromagnet_locs[0,1]), -(Particle_Position[0]-electromagnet_locs[0,0])]
    local_vecs[1,:] = [np.abs(Particle_Position[1]-electromagnet_locs[1,1]), Particle_Position[0]-electromagnet_locs[1,0]]
    local_vecs[2,:] = [np.abs(Particle_Position[0]-electromagnet_locs[2,0]), Particle_Position[1]-electromagnet_locs[2,1]]
    local_vecs[3,:] = [np.abs(Particle_Position[0]-electromagnet_locs[3,0]), -(Particle_Position[1]-electromagnet_locs[3,1])]

    #Calculate B components
    B11 = np.zeros(4)
    B12 = np.zeros(4)
    B21 = np.zeros(4)
    B22 = np.zeros(4)

    for i in range(4):
        B11[i] = sp.integrate.quad(integrand1, 0, 2*np.pi,args=(local_vecs[i,:]/a))[0]
        B12[i] = sp.integrate.quad(integrand1, 0, 2*np.pi,args=(local_vecs[i,:]/a+nu))[0]
        B21[i] = sp.integrate.quad(integrand2, 0, 2*np.pi,args=(local_vecs[i,:]/a))[0]
        B22[i] = sp.integrate.quad(integrand2, 0, 2*np.pi,args=(local_vecs[i,:]/a+nu))[0]

    h_x = np.multiply(k*(B11-B12), Current_Array)
    h_y = np.multiply(k*(B21-B22), Current_Array)

    B_point_x = h_x[3]-h_x[2]+h_y[1]-h_y[0]
    B_point_y = h_y[3]-h_y[2]+h_x[1]-h_x[0]

    return np.array([[-h_y[0], h_y[1], -h_x[2], h_x[3], B_point_x], [-h_x[0], h_x[1], -h_y[2], h_y[3], B_point_y]])


def Gradient_B_field(Current_Vector, Particle_Position, h):

    Gradient_X = (B_point(Current_Vector,Particle_Position+np.array([h/2, 0])) - B_point(Current_Vector,Particle_Position-np.array([h/2, 0])))/h
    Gradient_Y = (B_point(Current_Vector,Particle_Position+np.array([0, h/2])) - B_point(Current_Vector,Particle_Position-np.array([0, h/2])))/h

    return Gradient_X, Gradient_Y

def Linear_Controller(Particle_Position, Desired_Position, kp):
    return kp*(Desired_Position-Particle_Position)

def Non_Linear_Controller(Current_Array, Particle_Position, h, X, x_0):
    omega = np.identity(4)
    Hr=1/R*np.matmul(B_point(Current_Array, Particle_Position)[:,0:4],omega)
    Her = np.vstack((Hr, Gradient_B_field(Current_Vector, Particle_Position, h)[0][:,0:4], np.matmul(np.array([0.,1.]),
Gradient_B_field(Current_Vector, Particle_Position, h)[1][:,0:4])))

    def y_star(phi, X, Her):
        e = np.transpose(np.array([1,0,0,0]))
        F = np.array([[np.cos(phi),np.sin(phi),0,0,0],
                [-np.sin(phi),np.cos(phi),0,0,0],
                [0,0,np.cos(phi),np.sin(phi),0],
                 [0,0,0,np.cos(phi),np.sin(phi)]], dtype = float)
        b1 =np.linalg.inv(F@Her)@e
        E = np.transpose(np.concatenate((np.zeros((2,2)),.5*np.identity(2)), axis=1))
        B2 =np.matmul(np.linalg.inv( np.matmul(F ,Her)),E)
        eps = 0.01
        w = eps*np.identity(4)+ np.matmul((np.transpose(Her)),Her)/(np.linalg.norm(np.matmul((np.transpose(Her)),Her),ord=None))
        rho_star = (np.transpose(X)@np.transpose(B2)@w@B2@X/(np.transpose(b1)@w@b1))**(1/4)
        return rho_star*b1+B2@X/rho_star, w


    def J_star(phi, X, Her):
        return np.transpose(y_star(phi, X, Her)[0])@y_star(phi, X, Her)[1]@y_star(phi, X, Her)[0]

    J_star_zero = J_star(0, X, Her)
    J_star_pi = J_star (np.pi, X, Her)
```

```
    def callback(x, f, accepted): #Stop after finding 2 minima
        if len(number_minima) == 0 and f< J_star_zero and f < J_star_pi:
            number_minima.append(x)

        elif len(number_minima) == 1  and f< J_star_zero and f < J_star_pi:
            if abs(x - number_minima[0]) >= 0.000001:
                return True

    number_minima = []
    take_step = RandomDisplacementBounds(0, np.pi)
    minimizer_kwargs = dict(method="L-BFGS-B", args = (X, Her), bounds = ((0, np.pi),))
    phi_star = sp.optimize.basinhopping(J_star, x_0, minimizer_kwargs=minimizer_kwargs, take_step=take_step, callback = callback).x
    print(phi_star)
    return y_star(phi_star, X, Her)[0], Her, phi_star

def Filter(X, Her, Yk_1, Z, phi_star):

  def psi(X, Her, w, phi_star):

    def B_vals(phi, X, Her):
        e = np.transpose(np.array([1,0,0,0]))
        F = np.array([[np.cos(phi),np.sin(phi),0,0,0],
                [-np.sin(phi),np.cos(phi),0,0,0],
                [0,0,np.cos(phi),np.sin(phi),0],
                 [0,0,0,np.cos(phi),np.sin(phi)]], dtype = float)
        b1 =np.linalg.inv(F@Her)@e
        E = np.transpose(np.concatenate((np.zeros((2,2)),.5*np.identity(2)), axis=1))
        B2 =np.matmul(np.linalg.inv(np.matmul(F ,Her)),E)
        return b1, B2

    def Jqp_star(phi, X, Her, w):
        ro_zero = np.real(np.roots([np.transpose(B_vals(phi, X, Her)[0])@B_vals(phi, X, Her)[0],-np.transpose(B_vals(phi, X, Her)[0])@w, 0,
np.transpose(X)@np.transpose(B_vals(phi, X, Her)[1])@w, -np.transpose(X)@np.transpose(B_vals(phi, X, Her)[1])@B_vals(phi, X,
Her)[1]@X])[0])
        return np.transpose(B_vals(phi, X, Her)[0])@B_vals(phi, X, Her)[0]*ro_zero**2-2*np.transpose(B_vals(phi, X, Her)[0])@w*ro_zero-
2*np.transpose(X)@np.transpose(B_vals(phi, X, Her)[1])@w/ro_zero+np.transpose(X)@np.transpose(B_vals(phi, X, Her)[1])@B_vals(phi, X,
Her)[1]@X/ro_zero**2+2*np.transpose(B_vals(phi, X, Her)[0])@B_vals(phi, X, Her)[1]@X+np.transpose(w)@np.identity(4)@w

    J_star_zero = Jqp_star(0, X, Her, w)
    J_star_pi = Jqp_star(np.pi, X, Her, w)

    def callback(x, f, accepted): #Stop after finding 2 minima
        if len(number_minima) == 0 and f< J_star_zero and f < J_star_pi:
            number_minima.append(x)

        elif len(number_minima) == 1  and f< J_star_zero and f < J_star_pi:
            if abs(x - number_minima[0]) >= 0.000001:
                return True

    number_minima = []
    take_step = RandomDisplacementBounds(0, np.pi)
    minimizer_kwargs = dict(method="L-BFGS-B", args = (X, Her, w), bounds = ((0, np.pi),))
    phi_zero = sp.optimize.basinhopping(Jqp_star, phi_star, minimizer_kwargs=minimizer_kwargs, take_step=take_step, callback = callback).x
    ro_zero = np.real(np.roots([np.transpose(B_vals(phi_zero, X, Her)[0])@B_vals(phi_zero, X, Her)[0],-np.transpose(B_vals(phi_zero, X,
Her)[0])@w, 0, np.transpose(X)@np.transpose(B_vals(phi_zero, X, Her)[1])@w, -np.transpose(X)@np.transpose(B_vals(phi_zero, X,
Her)[1])@B_vals(phi_zero, X, Her)[1]@X])[0])

    return ro_zero*B_vals(phi_zero,X, Her)[0]+B_vals(phi_zero,X, Her)[1]@X/ro_zero

  return psi(X, Her, (1-lamda)*Yk_1+lamda*Z, phi_star)

#Simulation
Particle_Position = np.array([0.01918367,-0.00204])
Desired_Position = np.array([0.001, 0.001])
Blob_Volume = .1e-6 #.1 mL in m^3
Blob_Radius = (Blob_Volume*3/4/np.pi)**(1/3)
Glucose_water_visc = 8.9e-4 #Pa s
#kp = 3*0.12*Glucose_water_visc/mu_0/Blob_Radius**2/((mu_ferrofluid-mu_0)/(mu_ferrofluid+2*mu_0))
lamda = 0.1
```

```
kp = .49
x_0 = np.pi/2
original_time = time.clock()
X_out = Linear_Controller(Particle_Position, Desired_Position, kp)
Z, Her, phi_star = Non_Linear_Controller(Current_Vector, Particle_Position, 1e-10, X_out, x_0)
Yk_1 = Z + np.array([-1,1,-1,1])

Y = Filter(X_out, Her, Yk_1, Z, phi_star)

print(Z)
print(Yk_1)
print(Y)
print(time.clock()-original_time)
```

9.4 Appendix D: Controls GUI Code

```
import tkinter as tk
import cv2
import PIL.Image, PIL.ImageTk
import numpy as np
import scipy as sp
import scipy.integrate
import scipy.optimize
import serial
import struct
import traceback
import time

class App:
  def __init__(self, window, window_title, video_source=0):
    self.window = window
    self.window.title(window_title)
    self.video_source = video_source

    # open video source (by default this will try to open the computer webcam)
    self.vid = MyVideoCapture(self.video_source)
    self.mask_inv = self.vid

    # Create a canvas that can fit the above video source size
    self.canvas = tk.Canvas(window, width = self.vid.width, height = self.vid.height)
    self.black_scale = tk.Scale(window, from_=0, to=255, orient=tk.HORIZONTAL)
    self.black_scale_window = self.canvas.create_window(10, 10, anchor=tk.NW, window=self.black_scale)
    self.red_scale = tk.Scale(window, from_=0, to=255, orient=tk.HORIZONTAL)
    self.red_scale_window = self.canvas.create_window(10, 50, anchor=tk.NW, window=self.red_scale)
    self.green_scale = tk.Scale(window, from_=0, to=255, orient=tk.HORIZONTAL)
    self.green_scale_window = self.canvas.create_window(10, 90, anchor=tk.NW, window=self.green_scale)
    self.blue_scale = tk.Scale(window, from_=0, to=255, orient=tk.HORIZONTAL)
    self.blue_scale_window = self.canvas.create_window(10, 130, anchor=tk.NW, window=self.blue_scale)
    colorval = "#%02x%02x%02x" % (self.red_scale.get(), self.green_scale.get(), self.blue_scale.get())
    self.color_rectangle = self.canvas.create_rectangle(130, 90, 180, 140, fill=colorval)
    self.assign_scaling = tk.Button(window, text="Assign Scaling", command = self.Assign_Scaling)
    self.assign_scaling_window = self.canvas.create_window(10, 180, anchor=tk.NW, window=self.assign_scaling)
    self.on_controls = tk.Button(window, text="Turn On Controls", command = self.Controls)
    self.on_controls_window = self.canvas.create_window(10, 210, anchor=tk.NW, window=self.on_controls)
    self.off_controls = tk.Button(window, text="Turn Off Controls", command = self.Off_Controls)
    self.off_controls_window = self.canvas.create_window(120, 210, anchor=tk.NW, window=self.off_controls)
    self.kp_entry = tk.Entry(window)
    self.off_controls_window = self.canvas.create_window(160, 20, anchor=tk.NW, window=self.kp_entry)
    self.canvas.pack()
    self.mask_pass = np.ones((int(0.5*self.vid.height), int(0.5*self.vid.width), 3), dtype = np.uint8)*255
    self.Controller = Controls(1, 0, 0)
    self.On_Controls = False
    # After it is called once, the update method will be automatically called every delay milliseconds
    self.delay = 15
    self.update()

    self.window.mainloop()

  def Controls(self):
    print("Turn On Controls")
```

```
        #self.Controller = Controls(self.radius, self.x_assigned, self.y_assigned)
        self.Controller.centroid = np.array([self.x_assigned, self.y_assigned])
        self.Controller.ppr = self.Controller.Petri_Dish_Radius/self.radius
        self.Controller.phi_star = np.pi/2
        try:
            del self.Controller.Yk_1
        except:
            pass
        try:
            self.Controller.kp = float(self.kp_entry.get())
            print(self.Controller.kp)
        except:
            pass
        self.On_Controls = True

    def Off_Controls(self):
        #self.Controller = Controls(self.radius, self.x_assigned, self.y_assigned)
        print("Turn Off Controls")
        self.Controller.set_zero
        self.On_Controls = False

    def Assign_Scaling(self):
        self.x_assigned = self.x
        self.y_assigned = self.y
        self.radius = (self.MA + self.ma)/4
        self.mask_pass = np.zeros((int(0.5*self.vid.height), int(0.5*self.vid.width), 3), dtype = np.uint8)
        self.mask_pass=cv2.ellipse(self.mask_pass, center=(int(self.x_assigned), int(self.y_assigned)), axes=(int(self.MA/2)-2,int(self.ma/2)-2),
angle=self.angle, startAngle=0, endAngle=360, color=(255,255,255), thickness=-1)
        #cv2.imshow('mask', self.mask_pass)

        print(self.x_assigned, self.y_assigned, self.radius)

    def update(self):
        # Get a frame from the video source
        precision = 45
        ret1, frame, mask1, self.cX, self.cY = self.vid.get_frame(self.black_scale.get(), self.mask_pass)
        #print(np.shape(frame))
        #print(np.shape(mask_pass))
        ret2, mask2, self.x, self.y, self.MA, self.ma, self.angle = self.vid.get_circle(self.red_scale.get()+precision, self.red_scale.get()-precision,
self.green_scale.get()+precision, self.green_scale.get()-precision, self.blue_scale.get()+precision, self.blue_scale.get()-precision)
        colorval = "#%02x%02x%02x" % (self.red_scale.get(), self.green_scale.get(), self.blue_scale.get())
        self.canvas.itemconfig(self.color_rectangle, fill=colorval)

        try:
            cv2.ellipse(frame, (int(self.x_assigned), int(self.y_assigned)), (int(self.radius), int(self.radius)), 0, 0, 360, (0, 255, 0), 2)
        except:
            pass

        if ret1 and ret2:
            self.photo1 = PIL.ImageTk.PhotoImage(image = PIL.Image.fromarray(frame))
            self.photo2 = PIL.ImageTk.PhotoImage(image = PIL.Image.fromarray(mask1))
            self.photo3 = PIL.ImageTk.PhotoImage(image = PIL.Image.fromarray(mask2))
            self.canvas.create_image(0.5*self.vid.width+10, 0, image = self.photo1, anchor = tk.NW)
            self.canvas.create_image(0.5*self.vid.width+10, 0.5*self.vid.height+10, image = self.photo2, anchor = tk.NW)
            self.canvas.create_image(0, 0.5*self.vid.height+10, image = self.photo3, anchor = tk.NW)

        if self.On_Controls == True:
            try:
                clock_temp = time.clock()
                self.Controller.__call__(np.array([self.cX, self.cY]))
                print("Solving time", time.clock()-clock_temp)
                print("")
            except Exception as exc:
                print("Error thrown")
                print (traceback.format_exc())
                print (exc)
                print("")

        self.window.after(self.delay, self.update)
```

```
class MyVideoCapture:
    def __init__(self, video_source=0):
        # Open the video source
        self.vid = cv2.VideoCapture(video_source)
        if not self.vid.isOpened():
            raise ValueError("Unable to open video source", video_source)
         # Get video source width and height
        self.width = self.vid.get(cv2.CAP_PROP_FRAME_WIDTH)
        self.height = self.vid.get(cv2.CAP_PROP_FRAME_HEIGHT)

    def get_frame(self, threshold, mask_pass):
        if self.vid.isOpened():
            ret, frame = self.vid.read()
            if ret:
                # Return a boolean success flag and the current frame converted to BGR
                frame = cv2.resize(frame,None,fx=0.5,fy=0.5)
                frame2 = np.bitwise_and(frame,mask_pass)
                img2gray = cv2.cvtColor(frame2,cv2.COLOR_BGR2GRAY)
                mask_inv =  cv2.inRange(img2gray, 1, threshold)
                _, contours, hierarchy = cv2.findContours(mask_inv, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
                cX = 0
                cY = 0
                try:
                    size = []
                    for i in range(len(contours)):
                        size.append(cv2.contourArea(contours[i]))
                        maxpos = size.index(max(size))
                        M = cv2.moments(contours[maxpos])
                        cX = int(M["m10"] / M["m00"])
                        cY = int(M["m01"] / M["m00"])
                        cv2.circle(frame, (cX, cY), 3, (255, 0, 0), -1)
                except:
                    pass
                return (ret, cv2.cvtColor(frame, cv2.COLOR_BGR2RGB), mask_inv, cX, cY)

            else:
                return (ret, None)
        else:
            return (ret, None)

    def get_circle(self, thresholdR_high, thresholdR_low, thresholdG_high, thresholdG_low, thresholdB_high, thresholdB_low):
        def second_largest(numbers):
            first, second = 0, 0
            for n in range(len(numbers)):
                if numbers[n] > first:
                    first, second = numbers[n], first
                elif first > numbers[n] > second:
                   second = numbers[n]
            return second

        if self.vid.isOpened():
            ret, frame = self.vid.read()
            if ret:
                # Return a boolean success flag and the current frame converted to BGR
                frame = cv2.resize(frame,None,fx=0.5,fy=0.5)
                mask = cv2.inRange(frame, (thresholdB_low, thresholdG_low, thresholdR_low), (thresholdB_high, thresholdG_high,
thresholdR_high))
                mask = cv2.medianBlur(mask, 5)
                _, contours, hierarchy = cv2.findContours(mask, cv2.RETR_LIST, cv2.CHAIN_APPROX_SIMPLE)
                mask = cv2.cvtColor(mask, cv2.COLOR_GRAY2BGR)
                x = 0
                y = 0
                MA = 0
                ma = 0
                angle = 0
                # ensure at least some circles were found
                if contours is not None:
                    #convert the (x, y) coordinates and radius of the circles to integers
                    area = np.zeros(len(contours))
                    #print(len(contours))
```

```
                for i in range(len(contours)):
                    area[i] = cv2.contourArea(contours[i])
                #print("Areas:", area)
                if len(area) >= 2:
                    area_2 = np.where(area == second_largest(area))
                    #print(area_2[0][0])
                    try:
                        Inner_Ellipse = cv2.fitEllipse(contours[area_2[0][0]])
                        (x,y),(MA,ma),angle = Inner_Ellipse
                    #print(Inner_Ellipse)
                        cv2.ellipse(mask, Inner_Ellipse, (0, 255, 0), 3)
                    except:
                        pass
            return (ret, mask, x, y, MA, ma, angle)

        else:
            return (ret, None)
    else:
        return (ret, None)

  # Release the video source when the object is destroyed
  def __del__(self):
    if self.vid.isOpened():
        self.vid.release()

class RandomDisplacementBounds(object):
    """random displacement with bounds"""
    def __init__(self, xmin, xmax, stepsize=np.pi/2):
        self.xmin = xmin
        self.xmax = xmax
        self.stepsize = stepsize

    def __call__(self, x):
        """take a random step but ensure the new position is within the bounds"""
        return np.clip( x + np.random.uniform(-self.stepsize, self.stepsize, np.shape(x)), self.xmin, self.xmax )

class Controls:
    def __init__(self, radius, centroid_x, centroid_y):

        self.ser1 = serial.Serial('COM6', 9600)

        #Hyperparameters
        self.lamda = 1.
        self.kp = .0025
        self.Voltage_Constraint = 30
        self.Petri_Dish_Radius = 0.02

        #Constants
        self.centroid = np.array([centroid_x, centroid_y])
        self.ppr = self.Petri_Dish_Radius/radius
        self.electromagnet_locs = np.array([[0.0,self.Petri_Dish_Radius], [0.0,-self.Petri_Dish_Radius], [self.Petri_Dish_Radius,0.0], [-
self.Petri_Dish_Radius,0.0]])
        self.mu_0 = (np.pi)*4e-7 #H/m
        self.l = 0.0714 #m
        self.a = 0.007 #m
        self.N= 1570*.6#turns of magnet This is way off
        self.I = 1.#A
        self.k = self.mu_0*self.N/4/np.pi/self.l
        #self.R = np.array([42., 42., 108., 108.]) #Ohms Internal Resistance in wire
        self.R = 42.
        self.nu = np.array([self.l/self.a,0])

        #Working Variables
        self.Voltage_Array = np.array([1, 1, 1, 1], dtype = float)
        self.Desired_Position = np.array([0.001, 0.001])
        self.phi_star = np.pi/2

    def B_point(self, Current_Array, Particle_Position):

        def integrand1(theta, vec):
```

```
            return vec[0]/((vec[1]-np.cos(theta))**2+np.sin(theta)**2)*(vec[1]*np.cos(theta)-1)/(vec[0]**2+(vec[1]-
np.cos(theta))**2+np.sin(theta)**2)**(1/2)

        def integrand2(theta, vec):
            return (np.cos(theta))/(vec[0]**2+(vec[1]-np.cos(theta))**2+np.sin(theta)**2)**(1/2)

        local_vecs = np.zeros((4,2), dtype = float)
        # Convert Global X and Y into local x and y for all 4 electromagnets where x = distance from face, y = distance from outward normal vector
from center of electromagnet
        local_vecs[0,:] = [np.abs(Particle_Position[1]-self.electromagnet_locs[0,1]), -(Particle_Position[0]-self.electromagnet_locs[0,0])]
        local_vecs[1,:] = [np.abs(Particle_Position[1]-self.electromagnet_locs[1,1]), Particle_Position[0]-self.electromagnet_locs[1,0]]
        local_vecs[2,:] = [np.abs(Particle_Position[0]-self.electromagnet_locs[2,0]), Particle_Position[1]-self.electromagnet_locs[2,1]]
        local_vecs[3,:] = [np.abs(Particle_Position[0]-self.electromagnet_locs[3,0]), -(Particle_Position[1]-self.electromagnet_locs[3,1])]

        #Calculate B components
        B11 = np.zeros(4)
        B12 = np.zeros(4)
        B21 = np.zeros(4)
        B22 = np.zeros(4)

        for i in range(4):
            B11[i] = sp.integrate.quad(integrand1, 0, 2*np.pi,args=(local_vecs[i,:]/self.a))[0]
            B12[i] = sp.integrate.quad(integrand1, 0, 2*np.pi,args=(local_vecs[i,:]/self.a+self.nu))[0]
            B21[i] = sp.integrate.quad(integrand2, 0, 2*np.pi,args=(local_vecs[i,:]/self.a))[0]
            B22[i] = sp.integrate.quad(integrand2, 0, 2*np.pi,args=(local_vecs[i,:]/self.a+self.nu))[0]

        h_x = np.multiply(self.k*(B11-B12), Current_Array)
        h_y = np.multiply(self.k*(B21-B22), Current_Array)

        B_point_x = h_x[3]-h_x[2]+h_y[1]-h_y[0]
        B_point_y = h_y[3]-h_y[2]+h_x[1]-h_x[0]

        return np.array([[-h_y[0], h_y[1], -h_x[2], h_x[3], B_point_x], [-h_x[0], h_x[1], -h_y[2], h_y[3], B_point_y]])

    def Gradient_B_field(self, Current_Vector, Particle_Position, h):

        Gradient_X = (self.B_point(Current_Vector,Particle_Position+np.array([h/2, 0])) - self.B_point(Current_Vector,Particle_Position-
np.array([h/2, 0])))/h
        Gradient_Y = (self.B_point(Current_Vector,Particle_Position+np.array([0, h/2])) - self.B_point(Current_Vector,Particle_Position-
np.array([0, h/2])))/h

        return Gradient_X[0]+Gradient_Y[0], Gradient_X[1]+Gradient_Y[1]

    def Linear_Controller(self, Particle_Position, Desired_Position, kp):
        return kp*(Desired_Position-Particle_Position)

    def Non_Linear_Controller(self, Voltage_Array, Particle_Position, h, X, x_0):
        omega = np.identity(4)
        Hr=1/self.R*np.matmul(self.B_point(Voltage_Array, Particle_Position)[:,0:4],omega)
        Her = np.vstack((Hr, self.Gradient_B_field(Voltage_Array/self.R, Particle_Position, h)[0][:,0:4], np.matmul(np.array([0.,1.]),
self.Gradient_B_field(Voltage_Array/self.R, Particle_Position, h)[1][:,0:4])))

        def y_star(phi, X, Her):
            e = np.transpose(np.array([1,0,0,0]))
            F = np.array([[np.cos(phi),np.sin(phi),0,0,0],
                  [-np.sin(phi),np.cos(phi),0,0,0],
                  [0,0,np.cos(phi),np.sin(phi),0],
                   [0,0,0,np.cos(phi),np.sin(phi)]], dtype = float)
            b1 =np.linalg.inv(F@Her)@e
            E = np.transpose(np.concatenate((np.zeros((2,2)),.5*np.identity(2)), axis=1))
            B2 =np.matmul(np.linalg.inv( np.matmul(F ,Her)),E)
            eps = 0.01
            w = eps*np.identity(4)+ np.matmul((np.transpose(Her)),Her)/(np.linalg.norm(np.matmul((np.transpose(Her)),Her),ord=None))
            rho_star = (np.transpose(X)@np.transpose(B2)@w@B2@X/(np.transpose(b1)@w@b1))**(1/4)
            return rho_star*b1+B2@X/rho_star, w

        def J_star(phi, X, Her):
            return np.transpose(y_star(phi, X, Her)[0])@y_star(phi, X, Her)[1]@y_star(phi, X, Her)[0]

        J_star_zero = J_star(0, X, Her)
```

```
        J_star_pi = J_star (np.pi, X, Her)

        def callback(x, f, accepted): #Stop after finding 2 minima
            if len(number_minima) == 0 and f< J_star_zero and f < J_star_pi:
                number_minima.append(x)

            elif len(number_minima) == 1  and f< J_star_zero and f < J_star_pi:
                if abs(x - number_minima[0]) >= 0.000001:
                    return True
        number_minima = []
        take_step = RandomDisplacementBounds(0, np.pi)
        minimizer_kwargs = dict(method="L-BFGS-B", args = (X, Her), bounds = ((0, np.pi),))
        phi_star = sp.optimize.basinhopping(J_star, x_0, minimizer_kwargs=minimizer_kwargs, take_step=take_step, callback = callback).x
        #print(phi_star)
        return y_star(phi_star, X, Her)[0], Her, phi_star

    def Filter(self, X, Her, Yk_1, Z, phi_star, lamda):

        def psi(X, Her, w, phi_star):

            def B_vals(phi, X, Her):
                e = np.transpose(np.array([1,0,0,0]))
                F = np.array([[np.cos(phi),np.sin(phi),0,0,0],
                              [-np.sin(phi),np.cos(phi),0,0,0],
                              [0,0,np.cos(phi),np.sin(phi),0],
                              [0,0,0,np.cos(phi),np.sin(phi)]], dtype = float)
                b1 =np.linalg.inv(F@Her)@e
                E = np.transpose(np.concatenate((np.zeros((2,2)),.5*np.identity(2)), axis=1))
                B2 =np.matmul(np.linalg.inv(np.matmul(F ,Her)),E)
                return b1, B2

            def Jqp_star(phi, X, Her, w):
                ro_zero = np.real(np.roots([np.transpose(B_vals(phi, X, Her)[0])@B_vals(phi, X, Her)[0],-np.transpose(B_vals(phi, X, Her)[0])@w,
0, np.transpose(X)@np.transpose(B_vals(phi, X, Her)[1])@w, -np.transpose(X)@np.transpose(B_vals(phi, X, Her)[1])@B_vals(phi, X,
Her)[1]@X])[0])
                return np.transpose(B_vals(phi, X, Her)[0])@B_vals(phi, X, Her)[0]*ro_zero**2-2*np.transpose(B_vals(phi, X,
Her)[0])@w*ro_zero-2*np.transpose(X)@np.transpose(B_vals(phi, X, Her)[1])@w/ro_zero+np.transpose(X)@np.transpose(B_vals(phi, X,
Her)[1])@B_vals(phi, X, Her)[1]@X/ro_zero**2+2*np.transpose(B_vals(phi, X, Her)[0])@B_vals(phi, X,
Her)[1]@X+np.transpose(w)@np.identity(4)@w

            J_star_zero = Jqp_star(0, X, Her, w)
            J_star_pi = Jqp_star(np.pi, X, Her, w)

            def callback(x, f, accepted): #Stop after finding 2 minima
                if len(number_minima) == 0 and f< J_star_zero and f < J_star_pi:
                    number_minima.append(x)

                elif len(number_minima) == 1  and f< J_star_zero and f < J_star_pi:
                    if abs(x - number_minima[0]) >= 0.000001:
                        return True

            number_minima = []
            take_step = RandomDisplacementBounds(0, np.pi)
            minimizer_kwargs = dict(method="L-BFGS-B", args = (X, Her, w), bounds = ((0, np.pi),))
            phi_zero = sp.optimize.basinhopping(Jqp_star, phi_star, minimizer_kwargs=minimizer_kwargs, take_step=take_step, callback =
callback).x
            ro_zero = np.real(np.roots([np.transpose(B_vals(phi_zero, X, Her)[0])@B_vals(phi_zero, X, Her)[0],-np.transpose(B_vals(phi_zero, X,
Her)[0])@w, 0, np.transpose(X)@np.transpose(B_vals(phi_zero, X, Her)[1])@w, -np.transpose(X)@np.transpose(B_vals(phi_zero, X,
Her)[1])@B_vals(phi_zero, X, Her)[1]@X])[0])

            return ro_zero*B_vals(phi_zero,X, Her)[0]+B_vals(phi_zero,X, Her)[1]@X/ro_zero
        return psi(X, Her, (1-lamda)*Yk_1+lamda*Z, phi_star)

    def __call__(self, Particle_Pixel_location):
        try:
            self.Particle_Position = (Particle_Pixel_location-self.centroid)*np.array([1,-1])*self.ppr
            X_out = self.Linear_Controller(self.Particle_Position, self.Desired_Position, self.kp)
            try:
                Z, Her, phi_star_temp = self.Non_Linear_Controller(self.Voltage_Array, self.Particle_Position, 1e-10, X_out, self.phi_star)
                Yk_1_temp = self.Filter(X_out, Her, self.Yk_1, Z, self.phi_star, self.lamda)
```

```
                self.phi_star = phi_star_temp
                self.Yk_1 = Yk_1_temp
            except:
                pass

            print("Particle Position", self.Particle_Position)
            print("Output Voltage Unconstrained", self.Yk_1)
            Y_constrained = self.Yk_1*np.min([self.Voltage_Constraint,
np.max(abs(self.Yk_1))])/np.max(abs(self.Yk_1))/self.Voltage_Constraint*255
            self.Voltage_Array = Y_constrained
            values = Y_constrained.astype(int)
            print("Outputs to Arduino", values)
            signs = np.zeros((4), dtype = np.uint8)
            for i in range(len(values)):
                values[i] = int(values[i])
                if values[i] >= 0:
                    signs[i] = 1
                else:
                    pass
            valuesToWrite = struct.pack("BBBBBBBB", np.uint8(abs(values[0])), np.uint8(signs[0]), np.uint8(abs(values[1])), np.uint8(signs[1]),
np.uint8(abs(values[2])), np.uint8(signs[2]), np.uint8(abs(values[3])), np.uint8(signs[3]))
            self.ser1.write(valuesToWrite)

        except:
            print("First Call")
            self.Particle_Position = (Particle_Pixel_location-self.centroid)*np.array([1,-1])*self.ppr
            X_out = self.Linear_Controller(self.Particle_Position, self.Desired_Position, self.kp)
            self.Yk_1, Her, self.phi_star = self.Non_Linear_Controller(self.Voltage_Array, self.Particle_Position, 1e-10, X_out, self.phi_star)
            print("Particle Position", self.Particle_Position)
            print("Output Voltage Unconstrained", self.Yk_1)
            Y_constrained = self.Yk_1*np.min([self.Voltage_Constraint,
np.max(abs(self.Yk_1))])/np.max(abs(self.Yk_1))/self.Voltage_Constraint*255
            self.Voltage_Array = Y_constrained
            values = Y_constrained.astype(int)
            print("Outputs to Arduino", values)
            signs = np.zeros((4), dtype = np.uint8)
            for i in range(len(values)):
                values[i] = int(values[i])
                if values[i] >= 0:
                    signs[i] = 1
                else:
                    pass
            valuesToWrite = struct.pack("BBBBBBBB", np.uint8(abs(values[0])), np.uint8(signs[0]), np.uint8(abs(values[1])), np.uint8(signs[1]),
np.uint8(abs(values[2])), np.uint8(signs[2]), np.uint8(abs(values[3])), np.uint8(signs[3]))
            self.ser1.write(valuesToWrite)

    def set_zero(self):
        values = np.array([0,0,0,0])
        signs = np.zeros((4), dtype = np.uint8)
        for i in range(len(values)):
            values[i] = int(values[i])
            if values[i] >= 0:
                signs[i] = 1
            else:
                pass
        valuesToWrite = struct.pack("BBBBBBBB", np.uint8(abs(values[0])), np.uint8(signs[0]), np.uint8(abs(values[1])), np.uint8(signs[1]),
np.uint8(abs(values[2])), np.uint8(signs[2]), np.uint8(abs(values[3])), np.uint8(signs[3]))
        self.ser1.write(valuesToWrite)

    def __del__(self):
        self.set_zero

 # Create a window and pass it to the Application object
App(tk.Tk(), "Magnetic Fluid Controls")
```

9.5 Appendix E: Localization GUI Code

```
import serial
import time
import sys
```

```
import tkinter as tk

from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
from matplotlib.figure import Figure
from matplotlib import pyplot as plt
import threading
import numpy as np
import matplotlib.animation as animation
import multiprocessing
import scipy as sp
import scipy.optimize


class GUI:

    def __init__(self, window, window_title, video_source=0):
        self.window = window
        self.continuePlotting = False

        # create a Frame for the Text and Scrollbar
        txt_frm = tk.Frame(self.window, width=500, height=325)
        txt_frm.place(x=500, y = 325)
        # ensure a consistent GUI size
        txt_frm.grid_propagate(False)
        # implement stretchability
        txt_frm.grid_rowconfigure(0, weight=1)
        txt_frm.grid_columnconfigure(0, weight=1)

        # create a Text widget
        self.txt = tk.Text(txt_frm, borderwidth=3, relief="sunken")
        self.txt.config(font=("consolas", 8), undo=True, wrap='word')
        self.txt.grid(row=0, column=0, sticky="nsew", padx=2, pady=2)

        # create a Scrollbar and associate it with txt
        scrollb = tk.Scrollbar(txt_frm, command=self.txt.yview)
        scrollb.grid(row=0, column=1, sticky='nsew')
        self.txt['yscrollcommand'] = scrollb.set

        #create Buttons
        self.calibrate_button = tk.Button(self.window, text = "Calibrate", command = self.start_calibrate)
        self.calibrate_button.place(x=30, y= 30)
        self.data_button = tk.Button(self.window, text = "Start/Stop Reading Data", command = self.data_toggle)
        self.data_button.place(x=30, y= 60)
        self.zero_button = tk.Button(self.window, text = "Zero Field and Calculate Sensor Reference Frames", command = self.zero_field)
        self.zero_button.place(x=30, y= 90)
        self.zero_button = tk.Button(self.window, text = "Localize Magnet", command = self.start_localization)
        self.zero_button.place(x=30, y= 120)


        self.fig = Figure(figsize=(500/80, 325/80), dpi=80, facecolor='w', edgecolor='k')


        self.ax = self.fig.add_subplot(111)
        self.ax.set_xlabel("Time")
        self.ax.set_ylabel("B field Magnitude (mG)")
        self.ax.grid()
        self.line0, = self.ax.plot([0], [0], lw=2)
        self.line1, = self.ax.plot([0], [0], lw=2)
        self.line2, = self.ax.plot([0], [0], lw=2)
        self.line3, = self.ax.plot([0], [0], lw=2)
        self.ax.set_ylim(0, 1000)

        self.fig2 = Figure(figsize=(500/80, 325/80), dpi=80, facecolor='w', edgecolor='k')
        self.ax2 = self.fig2.add_subplot(111)
        self.ax2.set_xlabel("X (m)")
        self.ax2.set_ylabel("Y (m)")
        self.ax2.grid()
        self.line_local, = self.ax2.plot([0], [0], lw=2)
        self.ax2.set_xlim(-0.035, 0.035)
        self.ax2.set_ylim(-0.055, 0.055)
```

```
        #self.line, = self.ax.plot([],[])
        #self.ax.set_ylim(0, 1)
        #self.xdata, self.ydata = [0]*100, [0]*100

        self.graph = FigureCanvasTkAgg(self.fig, master=self.window)
        self.graph.draw()
        self.graph.get_tk_widget().place(x=500, y = 0)

        self.graph2 = FigureCanvasTkAgg(self.fig2, master=self.window)
        self.graph2.draw()
        self.graph2.get_tk_widget().place(x=0, y = 325)

        self.scaled_reading_array = []
        self.mag_vec = np.zeros((4,3), float)
        self.mags = np.zeros((4), float)
        self.reading_mags = []
        self.data_toggle_val = 0
        self.localize_toggle = 0
        self.zero_reading_array = np.zeros(12)
        self.zero_offset = np.zeros(3)
        self.rotation_matricies = [np.identity(3),np.identity(3),np.identity(3),np.identity(3)]

        self.r_sensor1 = np.array([[-0.03175,-0.05715,0]])
        self.r_sensor2 = np.array([[0.03175,-0.05715,0]])
        self.r_sensor3 = np.array([[0.03175,0.05715,0]])
        self.r_sensor4 = np.array([[-0.03175,0.05715,0]])
        self.r_sensor = np.concatenate((self.r_sensor1,self.r_sensor2,self.r_sensor3,self.r_sensor4), axis=0)
        self.mu_0 = (np.pi)*4e-7
        self.x0 = np.array([0,0,0,0.1,0.1,0.1])
        self.x = np.array([0,0,0], dtype = float)



        self.window.mainloop()

    def B_perm(self, q,r_sensor): #q is 1 by 6 vector with first 3 components being r vector and next 3 being m vector
            r_rel = q[:3]-r_sensor
            B_theory = np.transpose(np.transpose(self.mu_0/4/np.pi*(3*np.transpose(np.transpose(r_rel)/np.linalg.norm(r_rel, axis = 
1)*(np.sum(np.multiply(np.ones((4,3))*q[3:6], np.transpose(np.transpose(r_rel)/np.linalg.norm(r_rel, axis = 1))), axis = 1)))-
np.ones((4,3))*q[3:6]))/(np.linalg.norm(r_rel, axis = 1)**3))
            return B_theory

    def error(self, r, exper, r_sensor):
        theory=self.B_perm(r,r_sensor)
        return np.sum(np.linalg.norm(exper, axis = 1)*np.linalg.norm((exper - theory), axis = 1)) #with weighting
        #return np.sum(np.linalg.norm((exper - theory), axis = 1)) #without weighting

    def start_localization(self):
        if self.localize_toggle == 0:
            self.localize_toggle = 1
            threading.Thread(target=self.localize).start()
            self.x_hist = [0]*25
            self.y_hist = [0]*25
            self.z_hist = [0]*25

            self.ani2 = animation.FuncAnimation(self.fig2, self.plot_localize, self.localize, interval=1, blit=True)
            multiprocessing.Process(target = self.ani2._start()).start()
            #self.line_local, =
        else:
            self.localize_toggle = 0
            self.ani2.event_source.stop()

    def localize(self):
        while self.localize_toggle == 1:
            #bnds = ([-0.03175, 0.03175], [-0.05715, 0.05715], [-0.02, 0.15], [-np.inf, np.inf], [-np.inf, np.inf], [-np.inf, np.inf])
            bnds = ([-0.03175, -0.05715, -0.02, -np.inf, -np.inf, -np.inf], [0.03175, 0.05715, 0.15, np.inf, np.inf, np.inf])
            sol = sp.optimize.least_squares(self.error, self.x0, args=(self.mag_vec, self.r_sensor), method='trf', bounds=bnds, ftol = 1e-6)
            if sol.success == True:
                self.x = sol.x[0:3]
```

```
            yield sol.x[0], sol.x[1], sol.x[2]

    def plot_localize(self, data):
        x_val, y_val, z_val = data
        del self.x_hist[0]
        del self.y_hist[0]
        del self.z_hist[0]
        self.x_hist.append(x_val)
        self.y_hist.append(y_val)
        self.z_hist.append(z_val)
        self.line_local.set_data(self.x_hist, self.y_hist)
        return self.line_local,

            #self.x0 = sol.x


    def start_calibrate(self):
        threading.Thread(target=self.calibrate).start()

    def calibrate(self):
       ser.write("a".encode())
       line = ''
       while (1):
           buffer = ser.readline().decode()
           if '\n' in buffer:
               line = line + buffer
               #self.txt.insert('end', line)
               #self.window.update()
               self.txt.insert('end', line)
               #print(line)
               if 'Calibration Done' in line:
                   #print("True")
                   break
               line = ''
           else:
               line = line+buffer
       self.txt.insert('end', '\n\n')

    def zero_field(self):
        num_readings = 30;
        ser.reset_output_buffer()
        self.zero_reading_array = np.zeros(12)
        for ii in range(num_readings):
            line = ''
            ser.write("b".encode())
            while line == '':
                line = ser.readline().decode()
            self.zero_reading_array += np.array(list(map(float, line[:len(line)-2].split(','))))
        self.zero_reading_array = self.zero_reading_array/num_readings

        for jj in range(1,4):

            v = np.cross(self.zero_reading_array[3*(jj):3*(jj+1)]/np.linalg.norm(self.zero_reading_array[3*(jj):3*(jj+1)]),
self.zero_reading_array[:3]/np.linalg.norm(self.zero_reading_array[:3]))
            s = np.linalg.norm(v)
            c = np.dot(self.zero_reading_array[3*(jj):3*(jj+1)]/np.linalg.norm(self.zero_reading_array[3*(jj):3*(jj+1)]),
self.zero_reading_array[:3]/np.linalg.norm(self.zero_reading_array[:3]))
            skew_mat = np.array([[0,-v[2],v[1]], [v[2],0,-v[0]], [-v[1],v[0],0]])
            self.rotation_matricies[jj] = np.identity(3) + skew_mat + np.matmul(skew_mat,skew_mat)*(1-c)/s**2
            self.zero_offset = self.zero_reading_array[0:3]

        self.txt.insert('end', 'Calculated zero offset and sensor frame rotation matricies \n\n')


    def run(self, data):
        t, mags = data
        #print(mags)
        del self.time[0]
        del self.mag0[0]
        del self.mag1[0]
```

```
        del self.mag2[0]
        del self.mag3[0]
        self.time.append(time.clock()-self.start_time)
        self.mag0.append(mags[0])
        self.mag1.append(mags[1])
        self.mag2.append(mags[2])
        self.mag3.append(mags[3])
        self.line0.set_data(self.time, self.mag0)
        self.line1.set_data(self.time, self.mag1)
        self.line2.set_data(self.time, self.mag2)
        self.line3.set_data(self.time, self.mag3)
        self.ax.set_xlim(min(self.time), max(self.time))
        self.ax.set_ylim(0, max(self.mag0+self.mag1+self.mag2+self.mag3)+500)
        return self.line0, self.line1, self.line2, self.line3,


    def data_gen(self):
        t = 0
        line = ''
        self.mags = [0., 0., 0., 0.]
        #print("blah")
        while self.data_toggle_val == 1:
            try:
                ser.write("b".encode())
                t+=1
                #time.sleep(0.2)
                line = ser.readline().decode()
                #print(line)

                self.scaled_reading_array = list(map(float, line[:len(line)-2].split(',')))
                for ii in range(4):
                    self.mag_vec[ii] = np.matmul(self.rotation_matricies[ii], np.transpose(np.array(self.scaled_reading_array[3*ii:3*(ii+1)]))) -
self.zero_offset
                    self.mags[ii] = np.linalg.norm(self.mag_vec[ii])

            except:
                #print(e)
                #traceback.print_exc()
                pass
            yield t, self.mags


    def data_toggle(self):

      if self.data_toggle_val == 0:
        self.data_toggle_val = 1
        self.clock = time.clock()
        self.clock = 0
        self.mag0 = [0]*200
        self.mag1 = [0]*200
        self.mag2 = [0]*200
        self.mag3 = [0]*200
        self.time = [0]*200
        self.start_time = time.clock()
        self.k = 1
        self.xdata, self.ydata = [0]*100, [0]*100
        ser.reset_input_buffer()
        self.ani = animation.FuncAnimation(self.fig, self.run, self.data_gen, interval=1, blit=False)
        multiprocessing.Process(target = self.ani._start()).start()


      else:
        self.data_toggle_val = 0
        #self.line = ''
        ser.reset_output_buffer()
        self.ani.event_source.stop()
```

```
ser = serial.Serial('COM9', 9600, timeout = .1)
ser.close()
ser.open()
root = tk.Tk()
root.geometry("1000x650") #You want the size of the app to be 500x500
root.resizable(0, 0) #Don't allow resizing in the x or y direction
GUI(root, "Localization GUI")
ser.reset_output_buffer()
ser.close()
sys.stdout.flush()
#root.destroy()
```

9.6 Appendix F: Team Roles

For such a comprehensive project, valuable team roles must be given to each team member to ensure that all deliverables are given on time and to the highest quality possible. On our team, Jacob and Mike had created and ensured the timely completion of all scripts (a few of which are given in *Appendix A-D*), Skylar had performed extensive background research and had helped with all technical deliverables, and Mike had created the controls experimental setup and helped to debug any issues that arose during the design procedure. Jacob and Mike coded the localization algorithm, including the GUI and bluetooth connection, while Skylar implemented the hardware and wiring of the magnetometer array. All class assignments were completed together with ample communication among members and with our advisors.