

Motor Algorithmic Regulation System (MARS)

Northrop Grumman's noise reduction project for resolver encoders

Team Members:

Aaron Jarmusch <jarmusch@udel.edu>
Ethan Conway <ethanc@udel.edu>
Joel Huffman <joelhuff@udel.edu>
Jacob Stafford <rjstaff@udel.edu>

Parallel Computing
Computer Engineering
Software Experience
Control Systems

Executive Summary

For our senior design, we worked with Northrop Grumman to develop a noise reduction algorithm for their resolver encoder and resolver models. The project intended to create a better algorithm for their encoder than they currently have, thereby reducing the amount of noise-reducing material needed to be sent to space. We were constrained by the physical design of the model since we needed to know precisely how the resolver was being used with the other components of the mission. We were also constrained by cost and time. Using MATLAB and Simulink, we worked with Northrop Grumman to develop resolver encoder models they were unfamiliar with and yielded some positive results. Working with Northrop Grumman was challenging because of the constant communication back and forth that occurs over email instead of just having to communicate with our group members. We got a functioning physical model that could read out the resolver signals, which would be fed into our encoder designs. We also created three simulations of resolver encoders and presented them to Northrop Grumman. There was also an attempt to create a recurrent neural network encoder to explore its potential as an encoder. Our design is split between a physical copy of the version that mirrors Northrop Grummans' model, which will be sent on the Mars rover to space, and simulations of the encoder in Simulink to explore various encoder models.

Problem Statement

We worked with Northrop Grumman to design a better noise-reducing resolver encoder for their armatures which use resolvers, which will help improve the functionality of the motors of the Mars rover on NASA's upcoming Mars missions.

Objectives

The overall goal of this system is to create an algorithm for a three-phase brushless motor alongside Northrop Grumman that will help reduce noise for the system they are sending to Mars on the Mars rover. This, in turn, will help reduce the amount of signal noise-canceling material needed in the production of the motor, significantly reducing the engine's total weight. These algorithms were developed by building and testing several different MATLAB Simulink resolver encoder models. Another system goal would be to market this algorithm in a broader sense to other companies that rely on precision with brushless motors. This algorithm could also help them reduce noise and increase efficiency. Success would be achieving such an algorithm that works as intended and is better than what is currently being implemented. A failure would be not being able to complete the algorithm due to various factors such as communication with the third-party (Northrop Grumman) or lack of knowledge and time to develop a better algorithm than the one Northrop Grumman has in place at the moment. Some smaller objectives for this project are to establish a small physical model that can mirror the apparatus that Northrop Grumman uses. Still, on a much smaller scale, we can also begin developing a neural network for the algorithms.

Realistic Design Constraints

Our biggest engineering constraint was needing help to change the model. We didn't know the exact details of how the motor affects the other systems since we didn't remember those. Therefore we couldn't drastically change the configuration of the motor or resolver and will be working on software enhancements. This was set in place by our working Northrop Grumman. The following two most considerable constraints are time and cost. We had to implement a design by the end of May, within our senior design budget of ~\$200. These two constraints are set in place by the nature of the Senior Design course. Finally, we were bound by the physics of our model. Our motor could not spin at the same speeds as the Northrop Grumman's motor, thousands of cycles per second. The physical length of our model also bounds us. We could only run the motor for so long before the tread screw hits the end of the model.

Fall Goals

The goal of the fall semester was to build a model of the current system's approximate physical apparatus. We managed to get the motor and resolver in place with a trendline attached to the motor. We successfully ran the motor with the PWM signal going into the

ESC, which is used to spin the motor. We also created a virtual simulation of the resolver and resolver encoder in Simulink, then combined them for a final simulation apparatus. We also received and signed the NDA with Northrop Grumman, which allowed us to collaborate more on the Simulink models.

Spring Goals

The goal of the spring was to finish the system's model, read from the resolver, and create a resolver encoder. We found a better way to run the brushless motor, which gave us better control over the speed and direction the motor could spin. This was achieved with an Arduino that controlled the PWM signals that controlled the movement of the motor and an improved 3D-printed mount for the motor and resolver. Once that was complete, we completed work on 3 MATLAB Simulink models that tested the effectiveness of different resolver encoder algorithms. The three different resolver encoders that were tested were the PLL (phase-locked loop) resolver encoder, the Third-Order Rational Polynomial resolver encoder, and the S-Transform resolver encoder. All of the models were found satisfactory to Northrop Grumman, the Third-Order Rational Polynomial model, in particular, being the most adequate to them.

Prototype

Small-scale physical apparatus that mirrors what Northrop Grumman will send to space. The device includes a DC brushless motor, an Arduino, which acts as a PWM motor driver, a resolver, and a resolver encoder which reads the position of the trendline screw. An algorithm is implemented in the resolver encoder to reduce noise and digitalize the resolver signal. Northrop Grumman's primary goal is focused on the algorithm and simulation. However, our team also wanted to have a physical apparatus prototype that we could use to test the simulation and our various algorithms.

Approach

We researched every aspect of the project while waiting for the non-disclosure agreement and more involvement from Northrop Grumman, including the brushless motor, resolver, getting Simulink configured in MATLAB, and the different libraries needed inside MATLAB. We also researched ways to run the motor and landed on using the PWM in the ESC to control the brushless motor. Establishing an understanding of brushless motors and the systems that support them gave us a better idea of how to get started on the hardware aspect of the project. We also had to research using a brushless motor, sensor, and FPGA board to simulate the motor in real applications. We used MATLAB Simulink to determine the performance of various simulations we test and create. We also used an Arduino to build a program to run the motor, creating a baseline program based on a standard program for brushless motors. Once regular meeting times were established with Northrop Grumman, we worked closely with them to create and test our different Simulink models.

Fall Semester

WEEK OF	TASK
9/26/2022	<ul style="list-style-type: none"> Initial meeting with NG
10/09/2022	<ul style="list-style-type: none"> Submit proposal - Define our Fall Goals
10/16/2022	<ul style="list-style-type: none"> Algorithms Research - started waiting on NDA
10/30/2022	<ul style="list-style-type: none"> Start Designing motor holder w/ CAD
11/6/2022	<ul style="list-style-type: none"> Start Meeting with NG regularly to get more information
11/13/2022	<ul style="list-style-type: none"> Get Hardware for Replication
11/20/2022	<ul style="list-style-type: none"> 3D print updated part for physical model
11/27/2022	<ul style="list-style-type: none"> Receive and sign the NDA
12/04/2022	<ul style="list-style-type: none"> Baseline Algorithm Finished in Matlab
12/06/2022	<ul style="list-style-type: none"> Final Fall Presentation

Planned Spring Semester

WEEK OF	TASK
02/07/2023	<ul style="list-style-type: none"> Meet up, and see what was worked on over the winter Continue work on Matlab Simulink models. Continue biweekly meetings with NG
02/14/2023	<ul style="list-style-type: none"> Continue work on the physical model using Arduino Order more physical parts as needed
03/07/2023	<ul style="list-style-type: none"> Produce noise data files for Matlab testing
03/14/2023	<ul style="list-style-type: none"> Begin work on the neural network project
03/21/2023	<ul style="list-style-type: none"> 3D print updated parts for the physical model
04/04/2023	<ul style="list-style-type: none"> Register group for research day
04/11/2023	<ul style="list-style-type: none"> Finish all Simulink models before this date
04/18/2023	<ul style="list-style-type: none"> Complete research poster for printing
05/02/2023	<ul style="list-style-type: none"> Present research poster in iSuite for Research Day

05/16/2023	<ul style="list-style-type: none">• Present final project in iSuite Turn in the final report.
------------	---

Challenges

- Learning to work with MATLAB Simulink.
- Using Matlab to alter and optimize algorithms.
- Exporting our code into VHDL.
- Understanding Brushless Motors on a technical level.
- Working with a third party.
- Learning how to create a neural network

Design

Overall System

The overall system works with the algorithm and simulation in conjunction with the physical apparatus to test and demonstrate how the algorithm we create reduces noise in the system. This system emulates something that Northrop Grumman would put on a Mars rover that would be sent to space for various testing and space exploration.

Hardware

The hardware aspect of the system is a physical apparatus that we can use to test the algorithm. It consists of a brushless motor, resolver, Arduino to send PWM signals, electronic speed controller, two couplers, a threaded rod, a 3D-printed stand, a power supply, potentiometer, power supply, oscilloscope, and wave generator. The apparatus sends pulse width modulation signals from the Arduino to the electronic speed controller. This causes the brushless motor to spin and is controlled by the potentiometer. The motor can rotate in both directions and increases and decreases in speed depending on how far the potentiometer is turned. This will turn the threaded rod and cause a 3D-printed piece attached to the rod to move back and forth on one plane of motion. When the brushless motor spins, this causes the shaft of the resolver on the opposite side of the motor to spin along with it. The resolver then takes the input from the brushless motor and allows us to collect data from it. We use the waveform generator to send an excitation signal into the resolver. Then, the sine and cosine waves can be extracted to determine the position. These signals can be removed from the oscilloscope and sent into our various algorithms.

Physical Apparatus

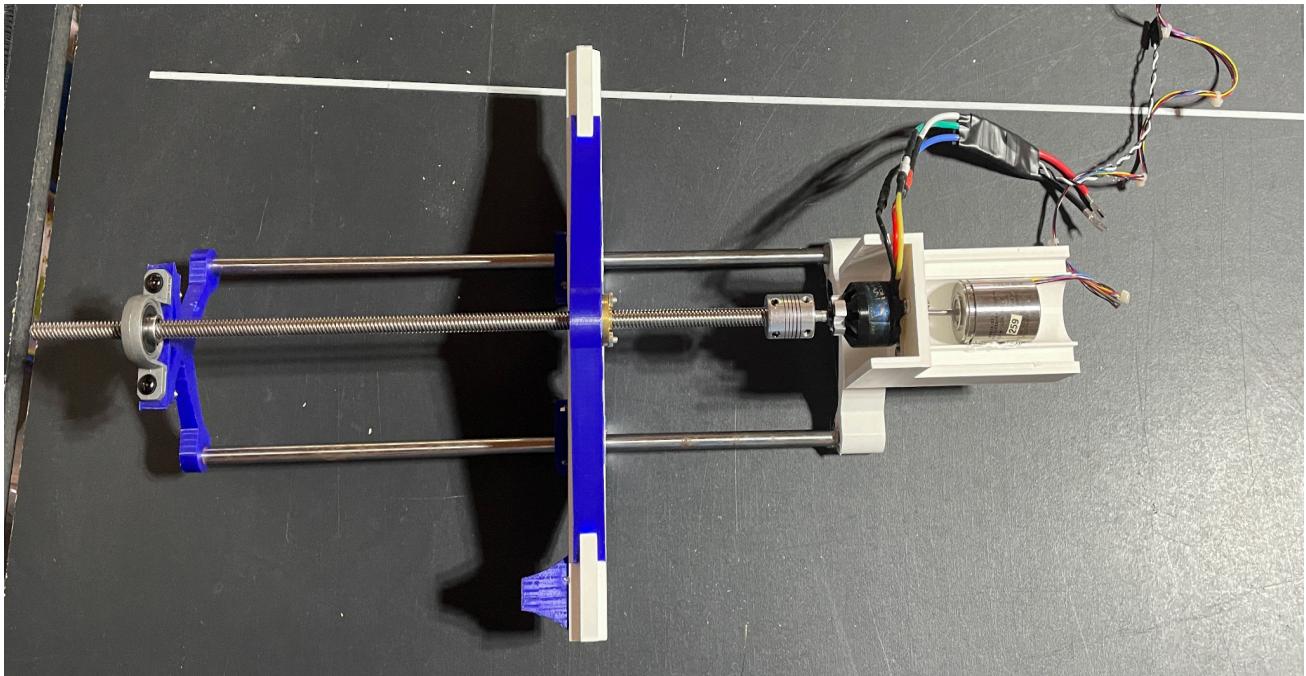


Figure 1: Top view of physical apparatus.

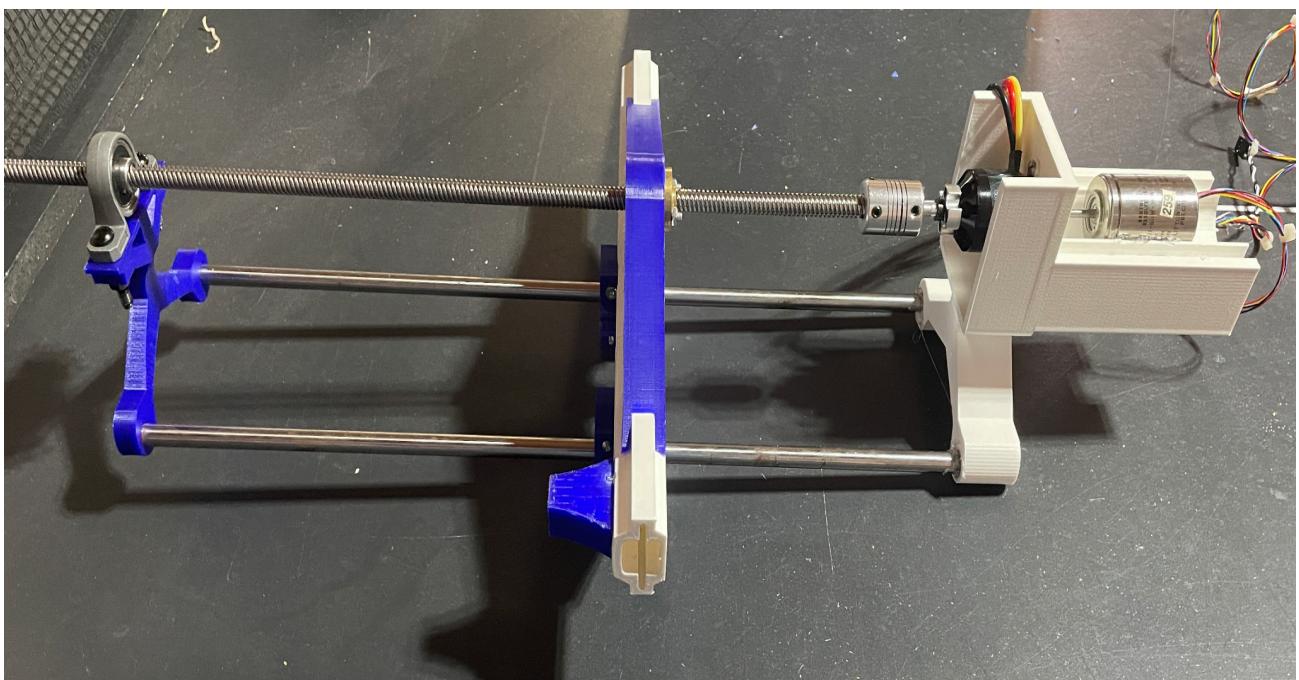


Figure 2: Side view of apparatus.

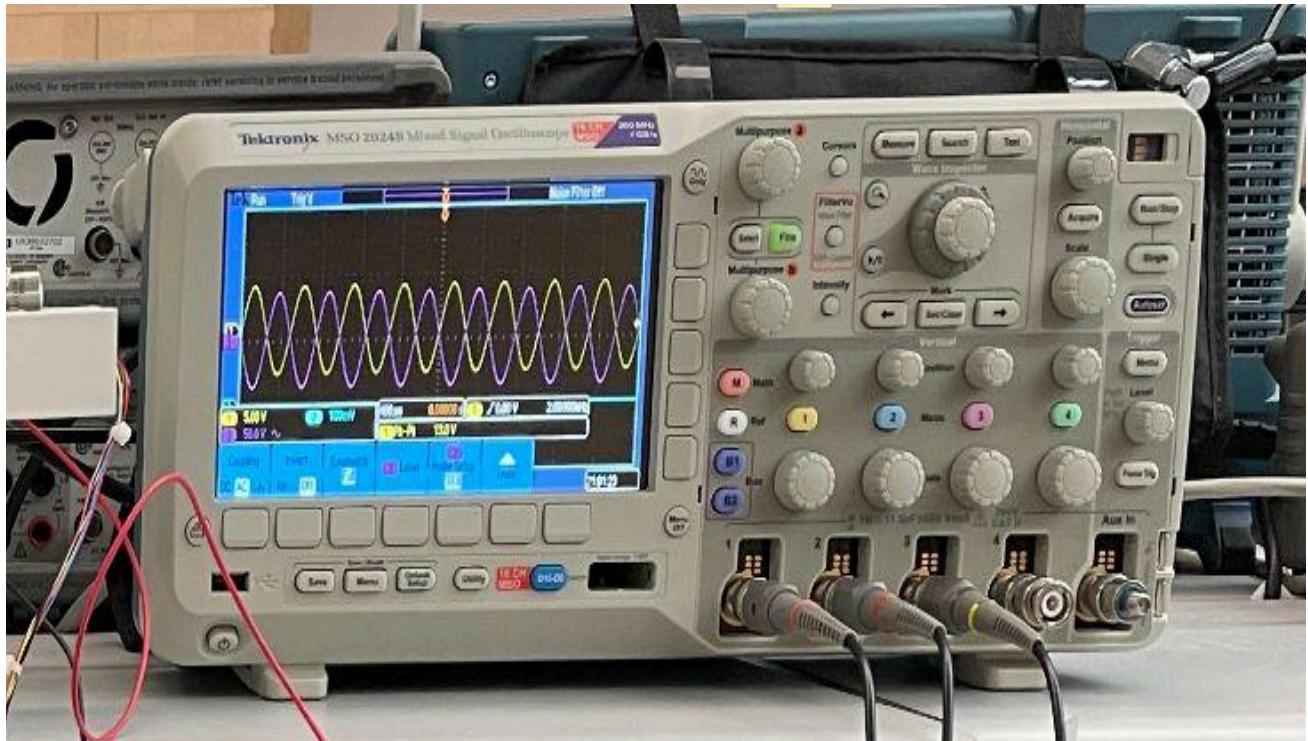


Figure 3: Oscilloscope to measure resolver. Sine (Yellow), Cosine (Purple).

CAD Design

The CAD design shown below is a modification of a design from a previous senior design group. This group used the apparatus to design a giant 3D printer. We are repurposing this design to emulate a moving platform on the Mars rover. On the left side of the invention, we measured and designed holes to secure the brushless motor onto the 3D-printed piece. Then, we extruded the other side to hold the resolver. There are also cutouts on the back, giving us access to the holes and the ability to screw the motor on with four nuts and bolts. We also widened the holes at the bottom, which hold two metal rods. These rods are the base of the apparatus, which causes it to be more sturdy and hold it up.

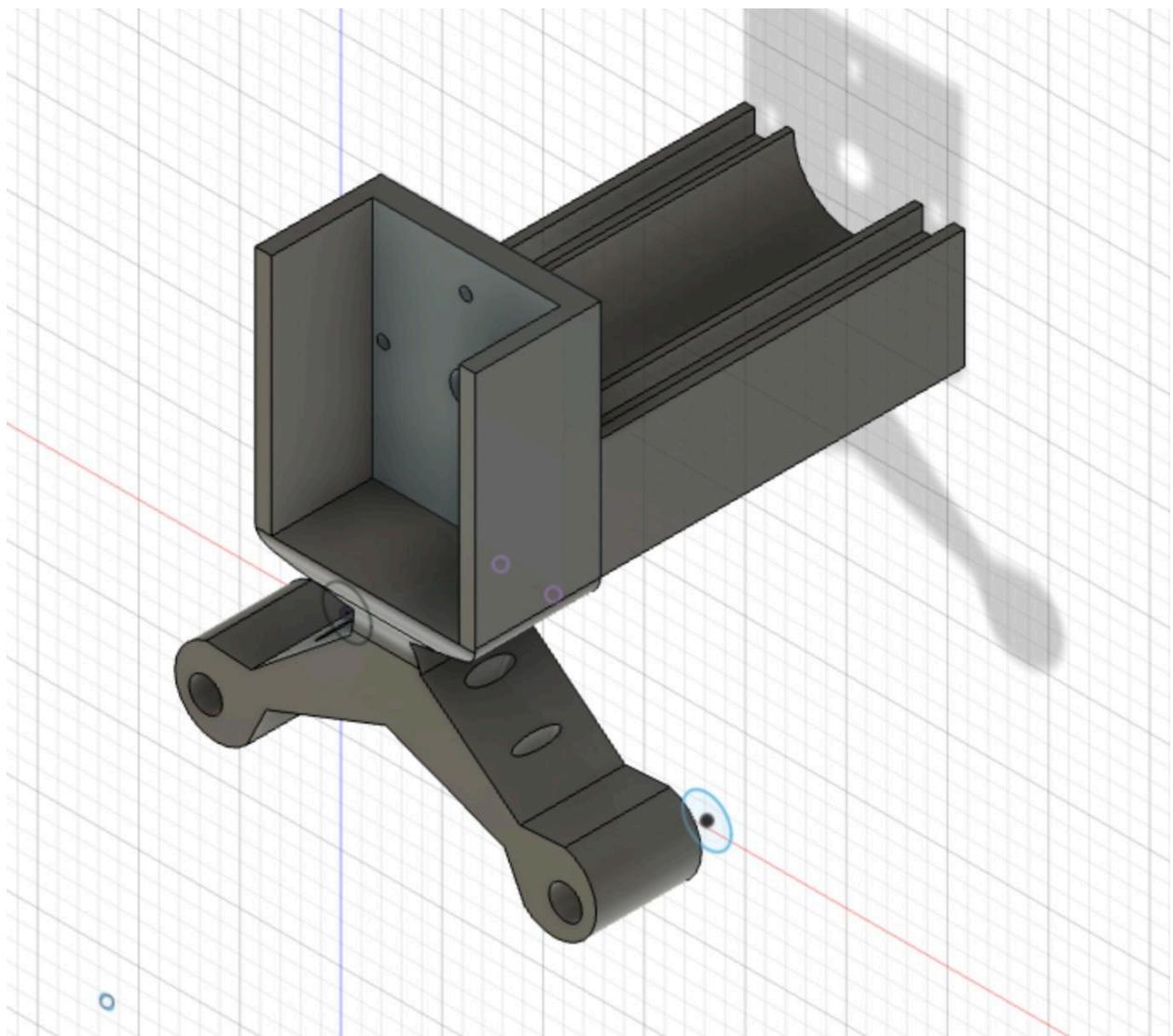


Figure 4: Main view of CAD design for the printed piece to hold motor and resolver.

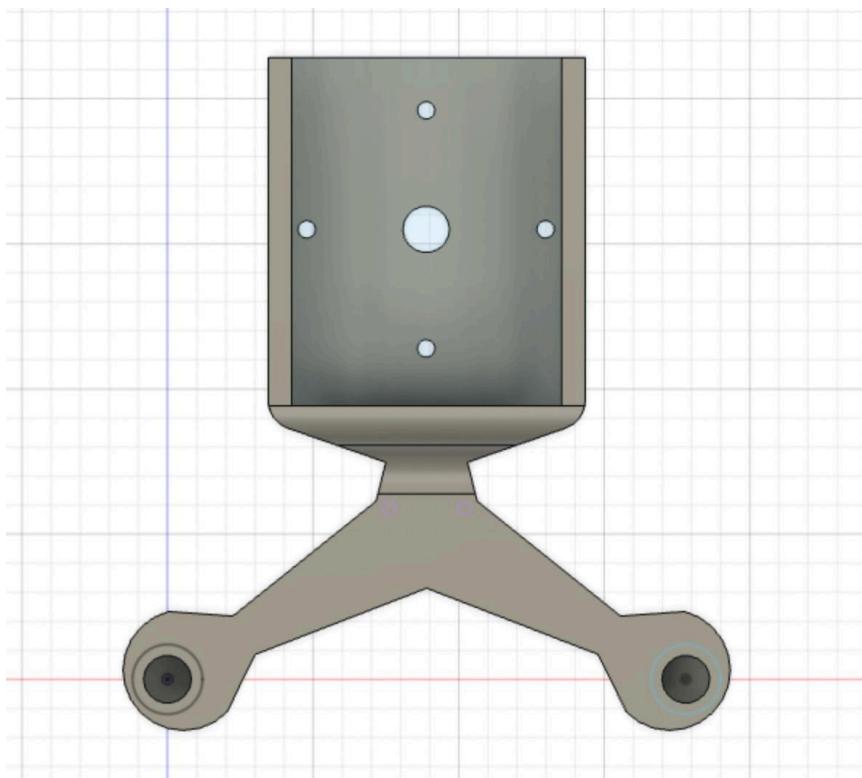


Figure 5: Front view of CAD design.

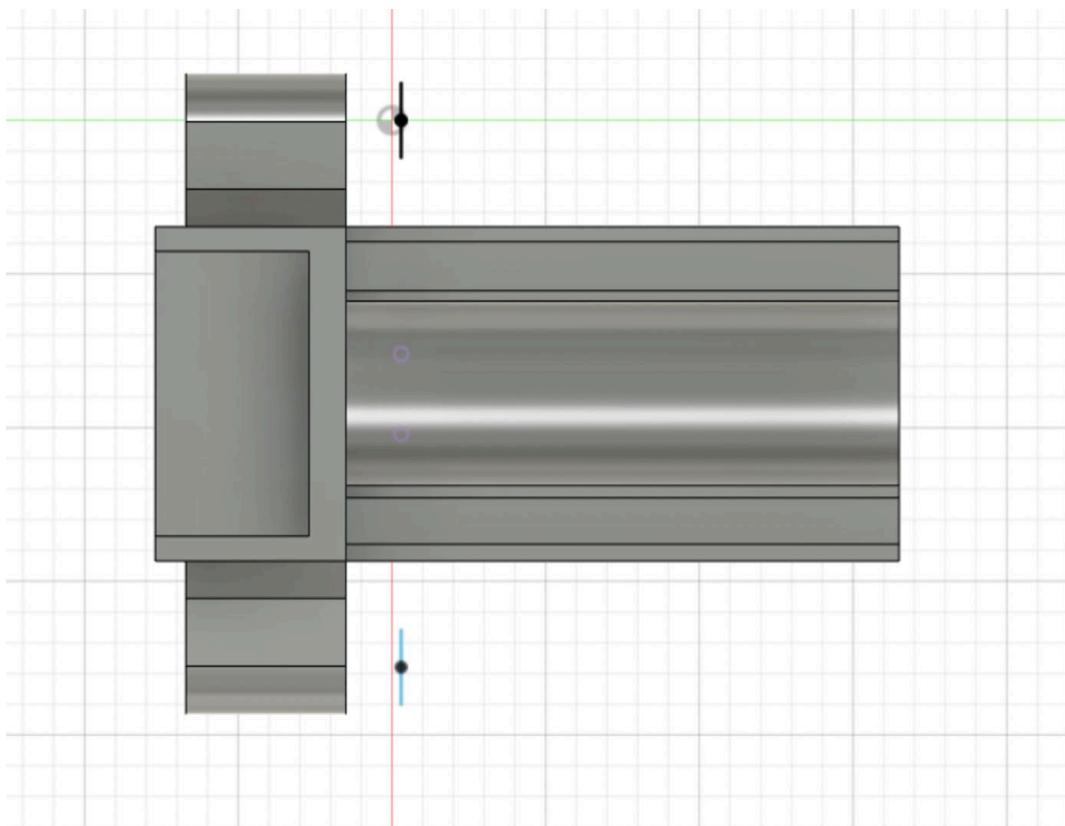
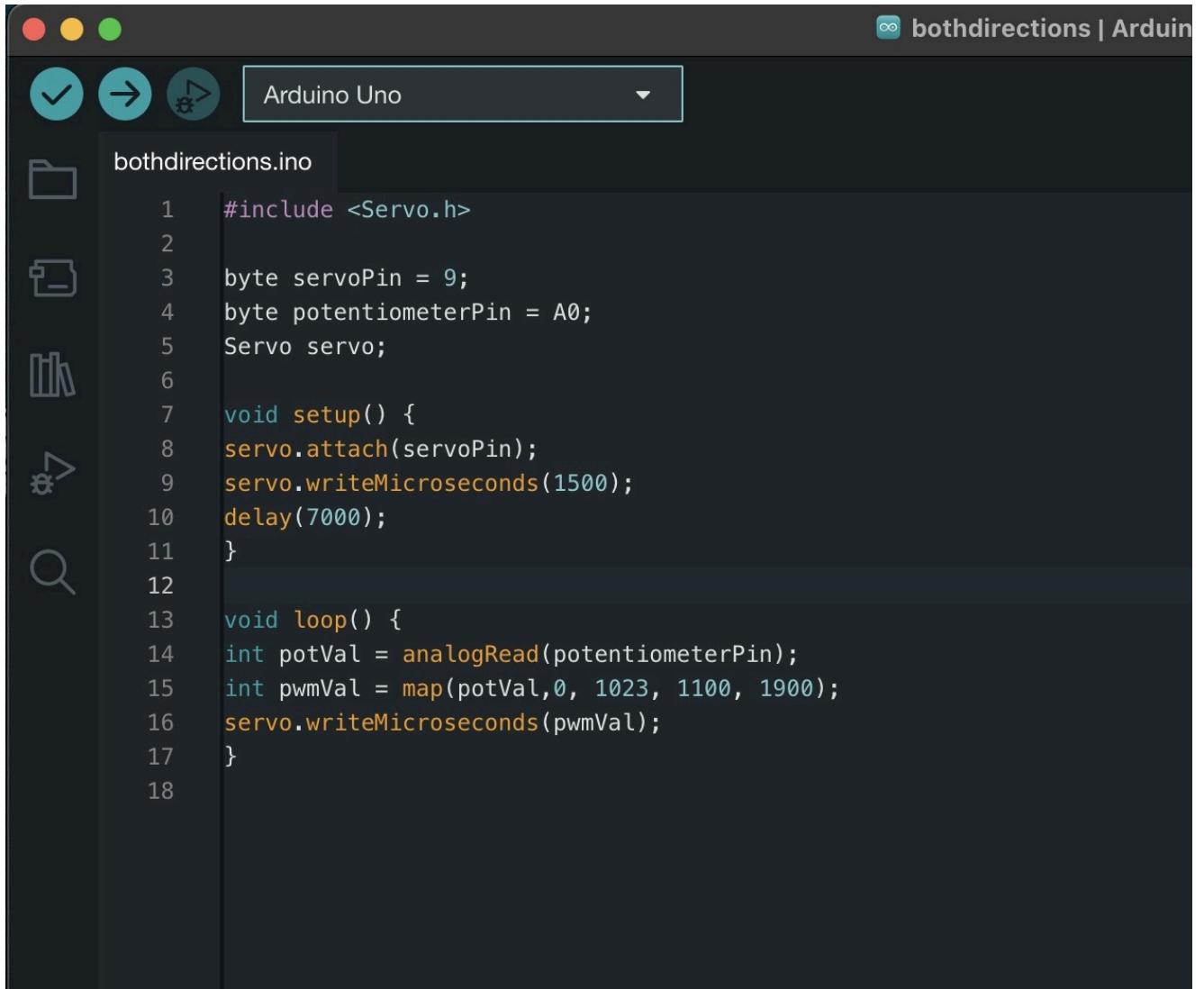


Figure 6: Top view of CAD design.

Arduino Code and Wiring

The code is relatively simple but gives us all the functionality we need to spin the motor in both directions. Using the Servo library from Arduino allows us to arm the motor, which is necessary for the motor to function. Once the potentiometer is at the center, the motor beeps to indicate that it is armed and ready to go. We can then turn the potentiometer clockwise and counterclockwise, causing the motor to spin in both directions.



The screenshot shows the Arduino IDE interface with the following details:

- Title Bar:** Shows the project name "bothdirections | Arduino".
- Tool Buttons:** Includes a checkmark, a right arrow, and a refresh symbol.
- Board Selection:** Set to "Arduino Uno".
- Code Editor:** Displays the "bothdirections.ino" file content. The code uses the Servo library to control a servo motor via a potentiometer input.

```
#include <Servo.h>
byte servoPin = 9;
byte potentiometerPin = A0;
Servo servo;
void setup() {
  servo.attach(servoPin);
  servo.writeMicroseconds(1500);
  delay(7000);
}
void loop() {
  int potVal = analogRead(potentiometerPin);
  int pwmVal = map(potVal, 0, 1023, 1100, 1900);
  servo.writeMicroseconds(pwmVal);
}
```

Figure 7: Code for Arduino to send PWM signals.

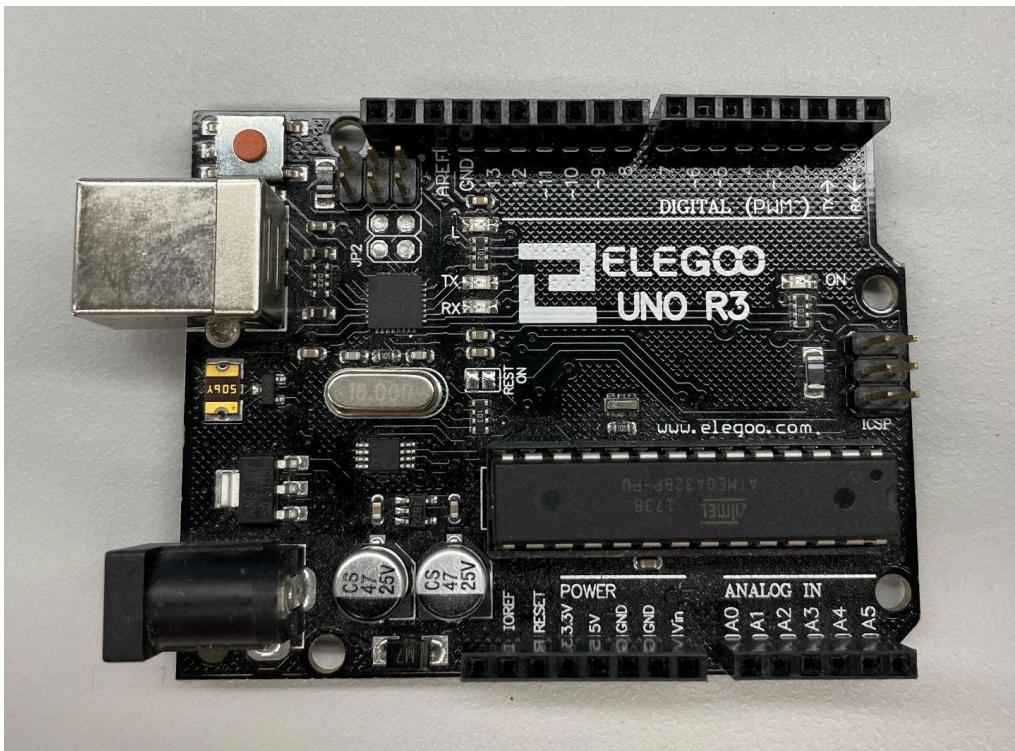


Figure 8: Arduino UNO board used.

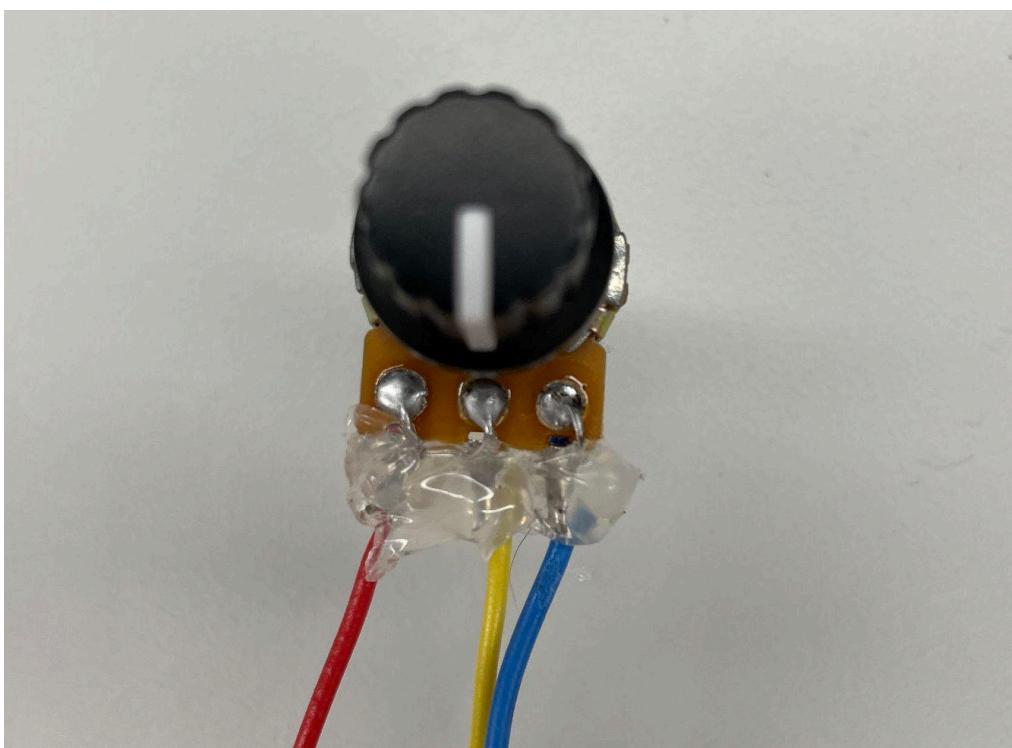


Figure 9: Potentiometer used to control the brushless motor with Arduino.

Shown below is the wiring of the potentiometer to the Arduino board. The red is connected to five volts, and the blue is connected to the ground. The yellow wire is connected to the analog 0 input. This allows us to get an analog reading from 0 to 1023 and transfer this into a PWM value to send to the ESC and control the brushless motor.

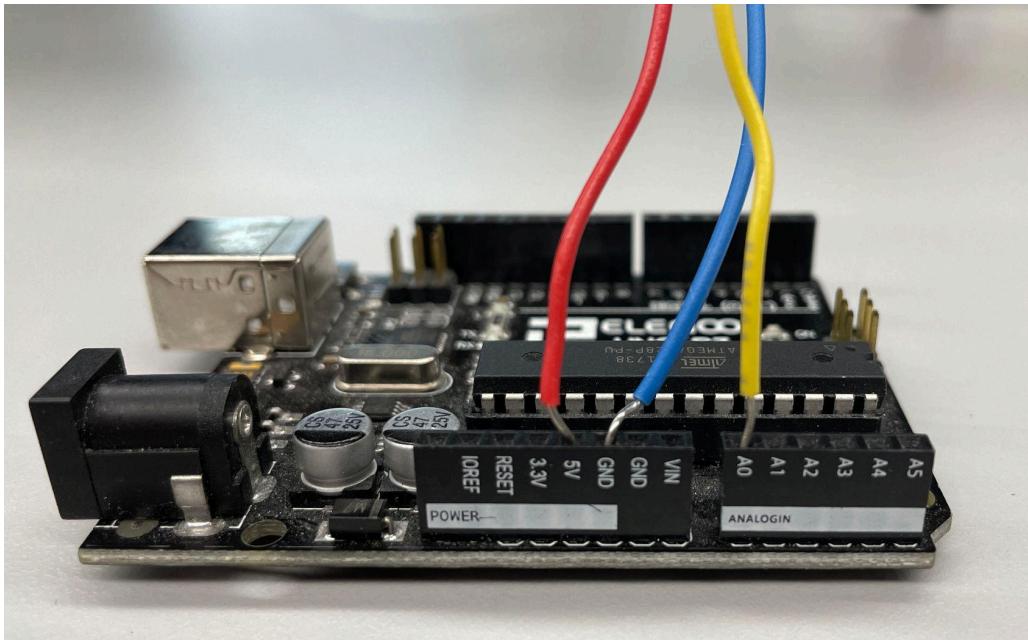


Figure 10: Potentiometer hooked up to Arduino. Red (5v), Blue (Gnd), Yellow (A0)

Software

The project's software consists of MATLAB Simulink simulations and an attempted RNN. The simulations all use the exact simulation of a resolver, which was developed in collaboration with Northrop Grumman. After the resolver, the resolver encoder is varied across simulations, as that was the main focus of our project with Northrop Grumman. By using Simulink, we can easily transmit our simulation to Northrop Grumman, who has a similar setup with which they can exploit onto their systems. We can also vary the signals from realistic motor velocities to do-right impossible velocities. It was possible to add noise into the Matlab, but Northrop Grumman needed to be more focused on us exploring how various noises affected the systems and took the simulations at the baseline of being able to read the sine and cosine waves into position and velocity.

To summarize what the goal of the encoder is, it is to give out data such that we can find the position, either the position itself or the velocity or arctan, which is just the position modded, so concerning time, we can find the position. Each resolver encoder tries to achieve this using different mathematical properties. We shall detail each one below and include screenshots of each encoder and the resolver model. In the fall report, we stated we could not obtain specific libraries for Matlab or Simulink, we have since corrected this by getting the licensing from the school.

The Simulink of the Resolver across both semesters:

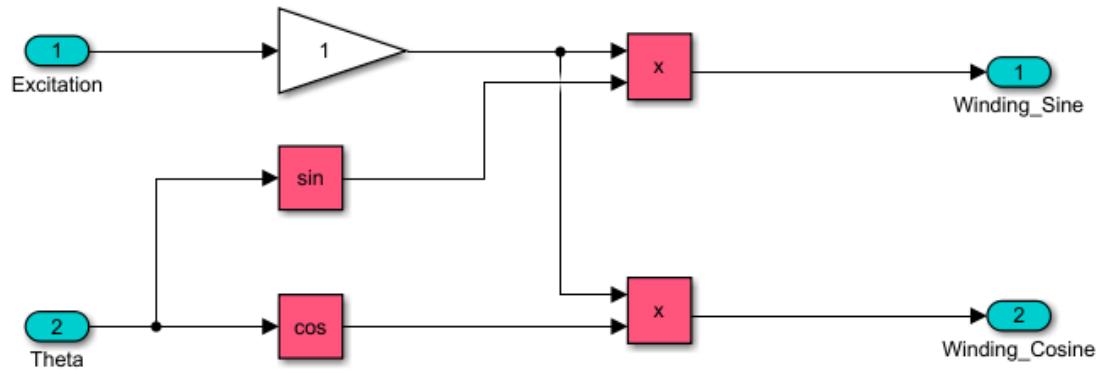


Figure 8: Screenshot of the simulink resolver we had in the fall.

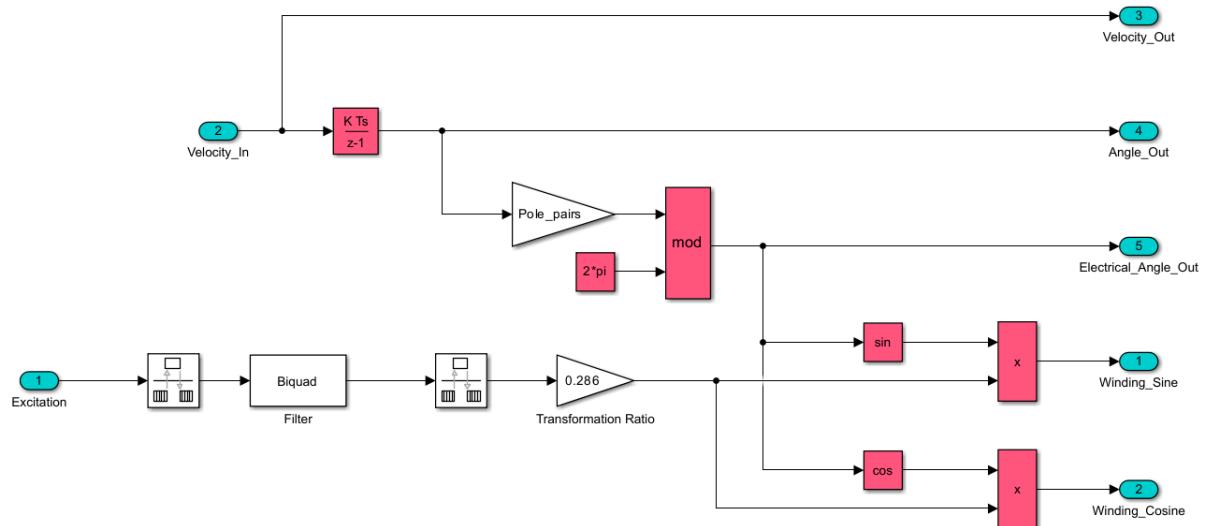


Figure 9: Screenshot of the simulink resolver encoder we made by the end of the spring.

As the figures above show, we have worked with Northrop Grumman to further develop a simulation of resolvers after the end of the fall semester. The newer model has more outputs, allowing us to gauge the encoder's performance by comparing it to the simulations' truths or signals we could see in real life. The newer model also allows for more realistic resolver signals, as will be noted in future figures.

The Simulations of Resolver Encoders across both semesters:

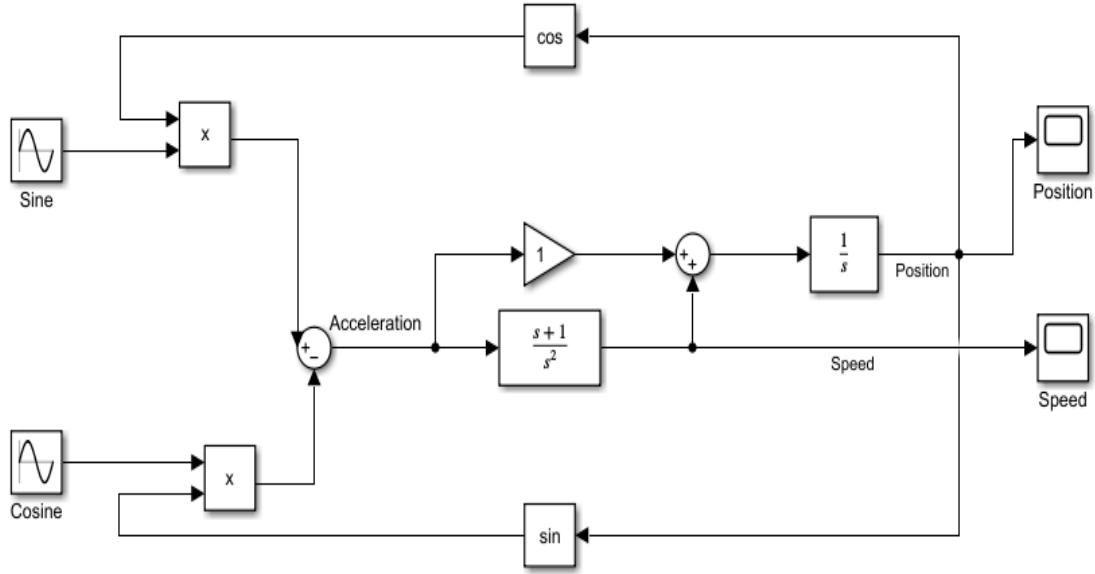


Figure 10: Screenshot of the PLL simulink resolver encoder we had in the fall.

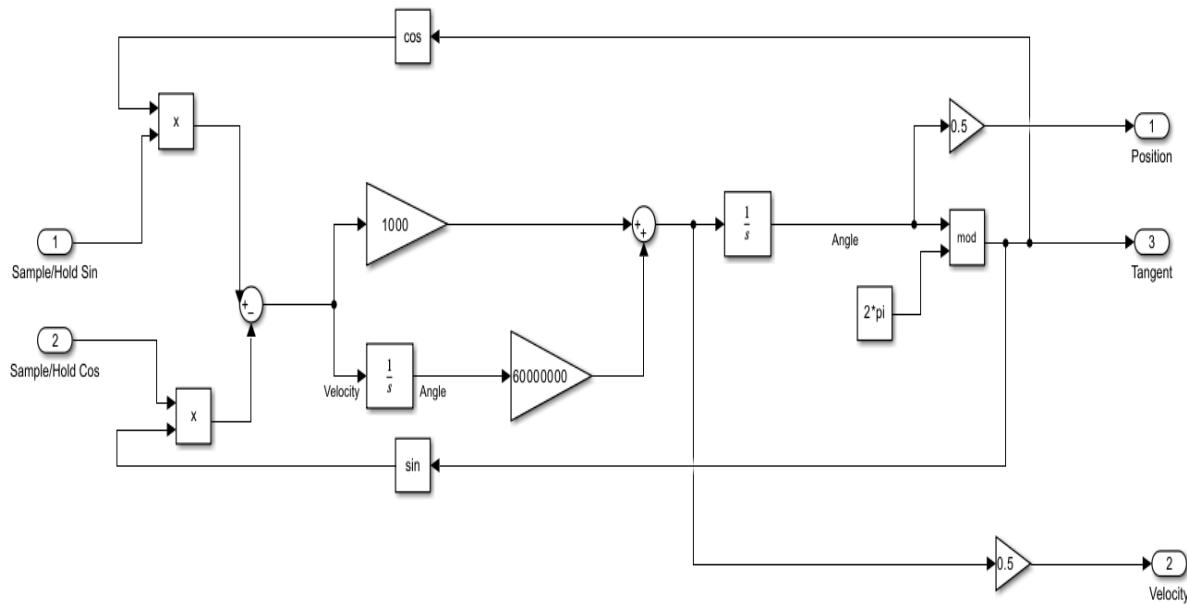


Figure 11: Screenshot of the PLL simulink resolver encoder we made by the end of the spring.

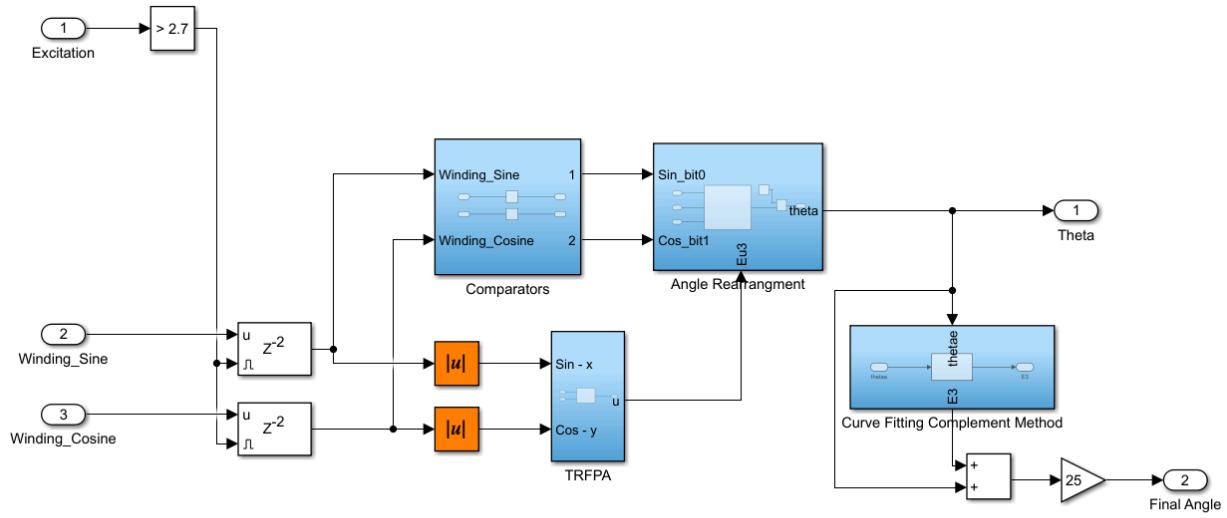


Figure 12: Screenshot of the third-order ration polynomial simulink resolver encoder we made by the end of the spring.

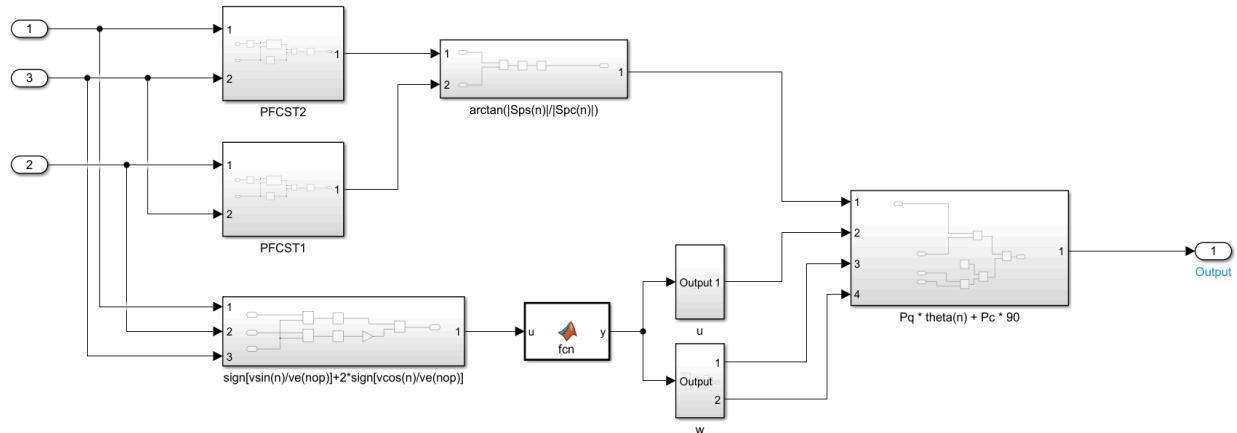


Figure 13: Screenshot of the S-Transform simulink resolver encoder we made by the end of the spring.

We have worked over the semester to develop resolver encoders further, as that was Northrop Grumman's primary goal for us. The PLL uses PI control logic to adjust velocity with input and feedback. The significant change is to adjust the feedback by modding 2π the feedback.

The second encoder simulations use different math to achieve the goal of the encoder. The third-order ration polynomial tries to approximate the arctangent function with a polynomial of the third order. After filtering them, it does this by comparing the cosine and sine signals and some curve fitting to achieve better results than the PLL.

The last encoder uses Fourier and Inverse Fourier to realize an arctan. The problem is, however, that the Fourier is extensively expensive on computation power and time, making the last encoder impractical compared to the first two.

Full Apparatus Simulation of Both Semesters

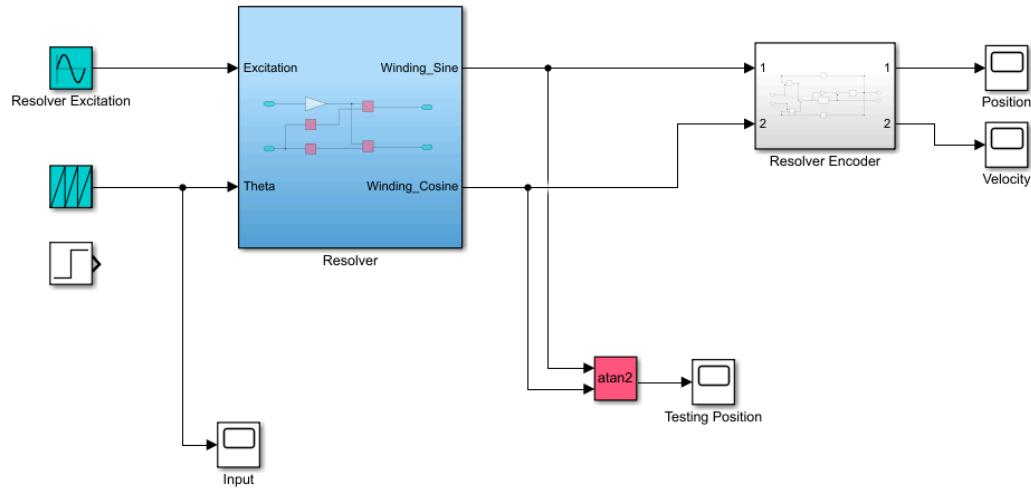


Figure 14: Screenshot of the fall simulation with the resolver and resolver encoder.

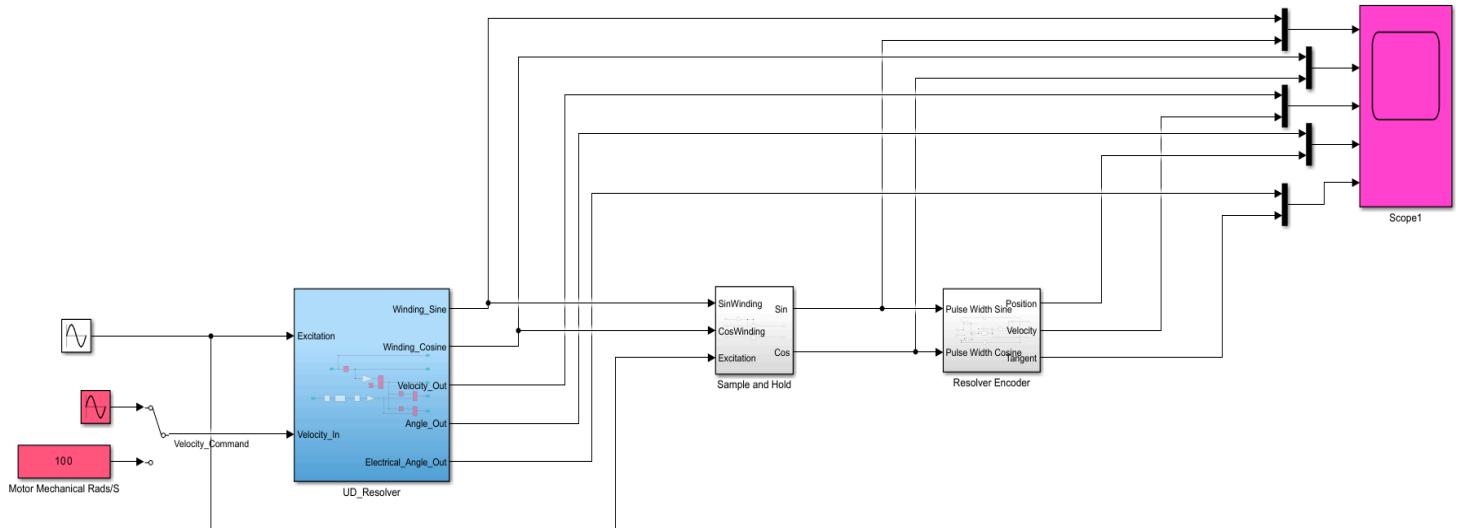


Figure 15: Screenshot of the PLL simulation in the spring. Note the Sample and Hold before the encoder.

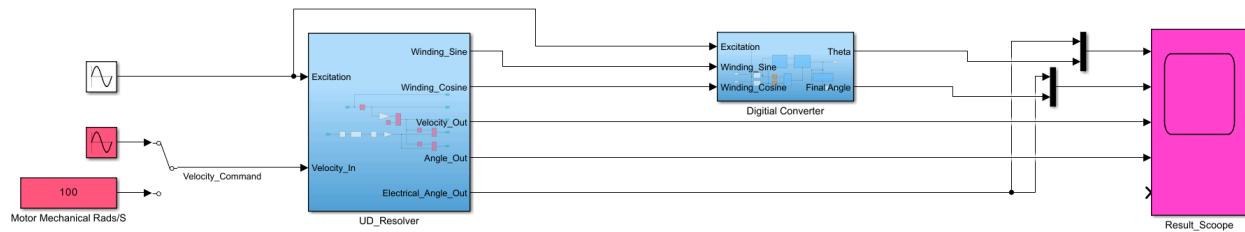


Figure 18: Screenshot of the full Third-Order Rational Polynomial simulation.

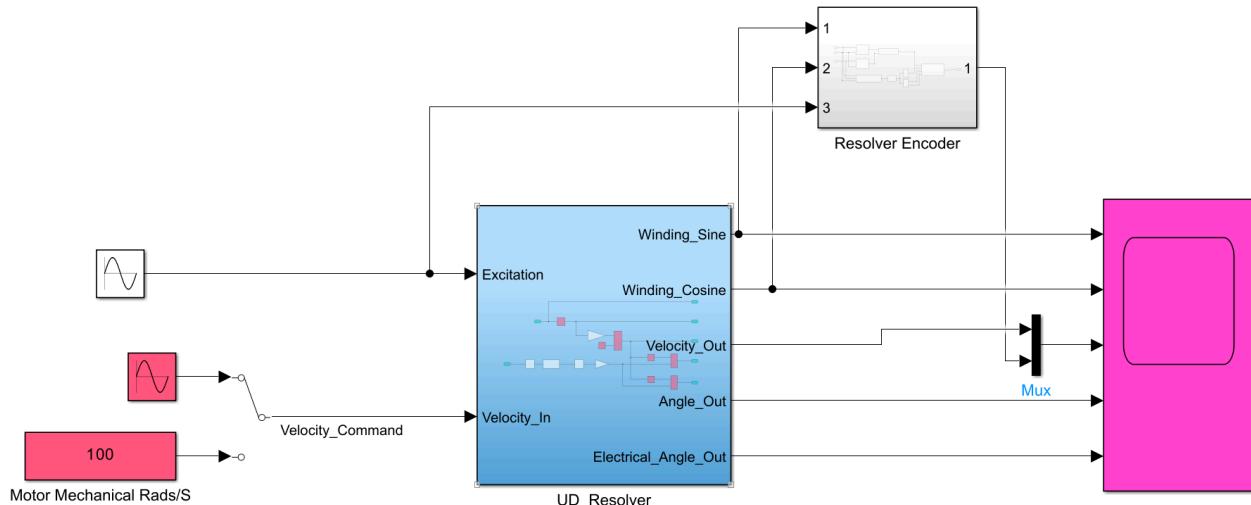


Figure 17: Screenshot of the full S-Transform simulation in Simulink.

As figures above show the simulation of change by their encoders, each using the same resolver simulation as a control. The difference in the central part of the encoders is explored under the close-ups of the encoders in the previous section. It should be noted that the PLL uses a sample and hold before feeding into the encoder, though the third-order ration polynomial does similar filtering of the education within its encoder. It should be noted that the purple box is a scope to read the output signals, with some mux to overlay two signals in the same graph. A trick that came out of our talks with Northrop Grumman.

Output Performance of Simulations

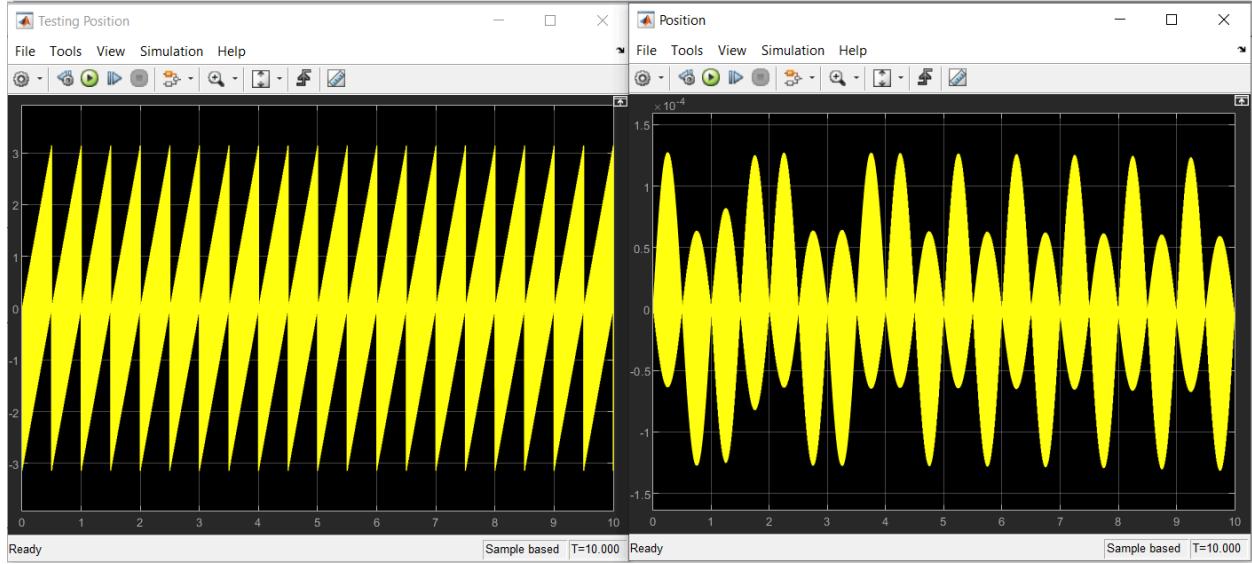


Figure 18: Screenshot of the simulation output in the fall when it is fed a tangent input. On the left is the output of the resolver, and on the right is the output of the encoder. As you can see, we had some work ahead of us.

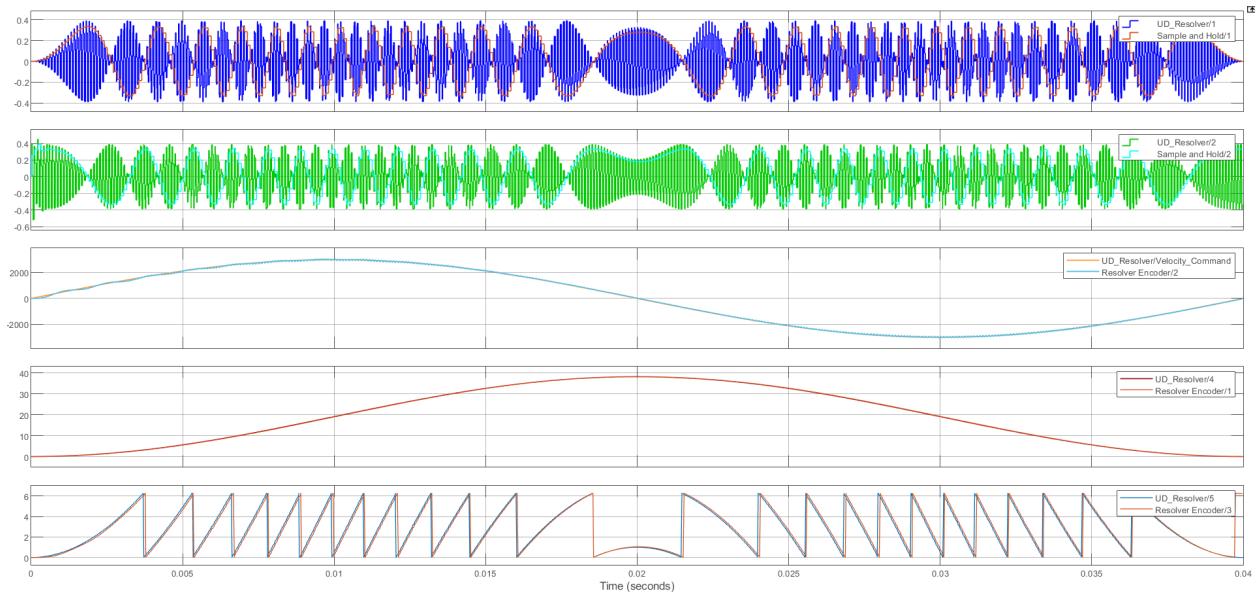


Figure 19: Screenshot of the performance of the Spring's PLL with high gains. The top color of the legend is the expected signal, and the bottom is the result.

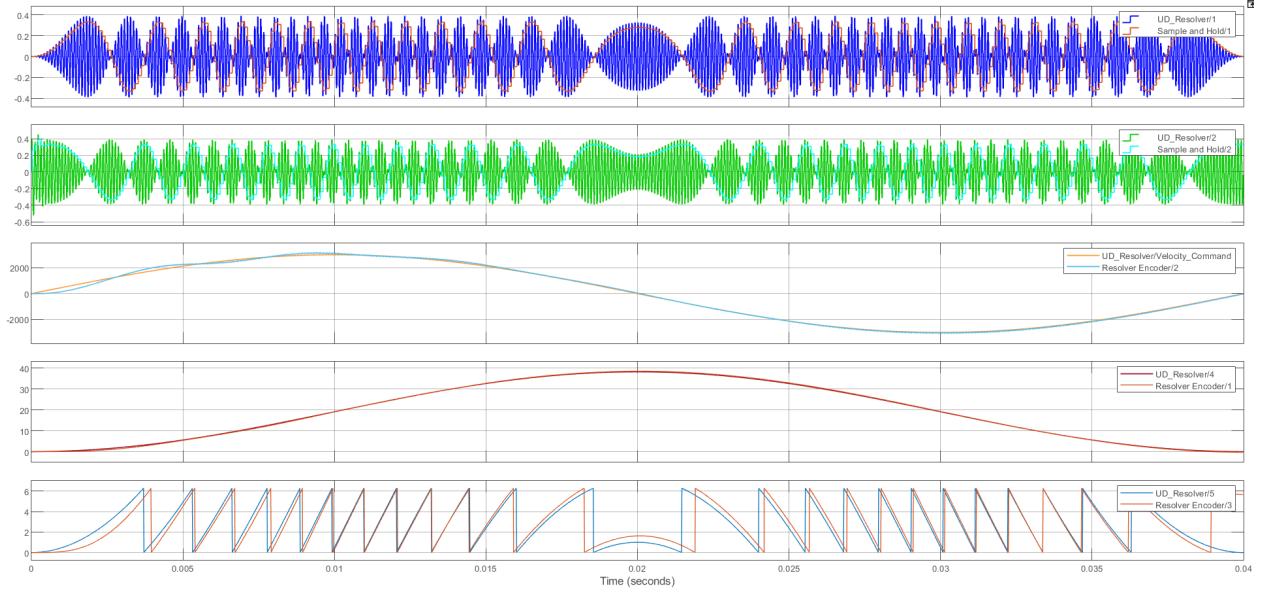


Figure 20: Screenshot of the performance of the Spring's PLL with moderate gains. The top color of the legend is the expected signal, and the bottom is the result.

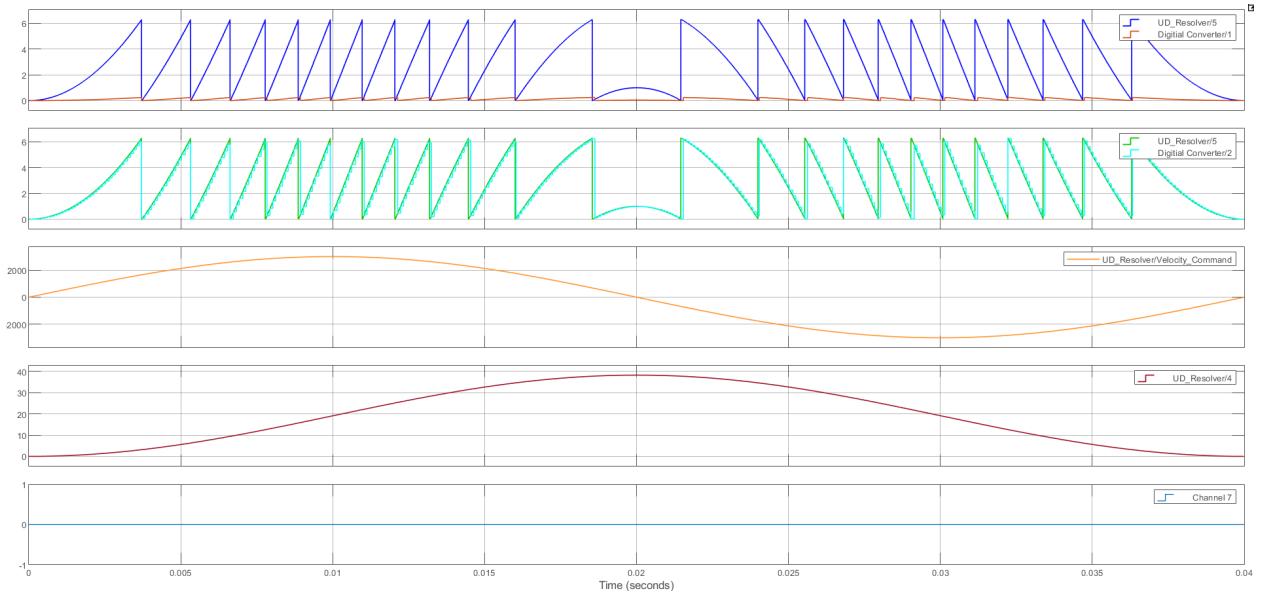


Figure 21: Screenshot of the Spring's third-order rational polynomial performance. The top color of the legend is the expected signal, and the bottom is the result.

As the figures above show, we have significantly improved performance. The S-Transform requires too much time to process the signal, thus a lack of accurate results for that simulation. Northrop Grumman had us explore ways to reduce oscillations in the positional signal seen toward the beginning of the simulation run. The only way we have discovered, after exploring several options, to reduce these oscillations was to increase the frequency of them and thereby reduce the amplitude by increasing the integral gain of the encoder. The difference can be seen in Figures 19 and 20. Lastly, in Figure 21, we show our best results which came from the third-order

rational polynomial. The third-order rational polynomial outputs arctangent of the position, which, if summed, would result in the position, but a sum of such caliber cannot be easily realized in Simulink, nor did it need to be. The third-order polynomial follows the arctangent the best and does not result from massive gains.

Recurrent Neural Network (RNN)

As we approached the final stage of the semester, we aimed to further expand our project by implementing a Recurrent Neural Network (RNN). The motivation behind this step was the expectation that the RNN would outperform any Simulink MATLAB models regarding the accuracy and predictive capabilities.

To facilitate this implementation, we turned to Google Colab, a robust data analysis and machine learning tool. Google Colab provides an environment where executable Python code can be combined with rich text, charts, images, HTML, and LaTex, all stored conveniently in Google Drive. This collaborative platform offered us the advantage of developing as a team with easy access to shared resources.

One notable advantage of Google Colab is its access to GPUs (Graphics Processing Units), eliminating the need to transfer our project to a separate system for GPU utilization. This streamlined our workflow and allowed us to take full advantage of the computational power offered by GPUs, which is particularly beneficial for training complex neural networks.

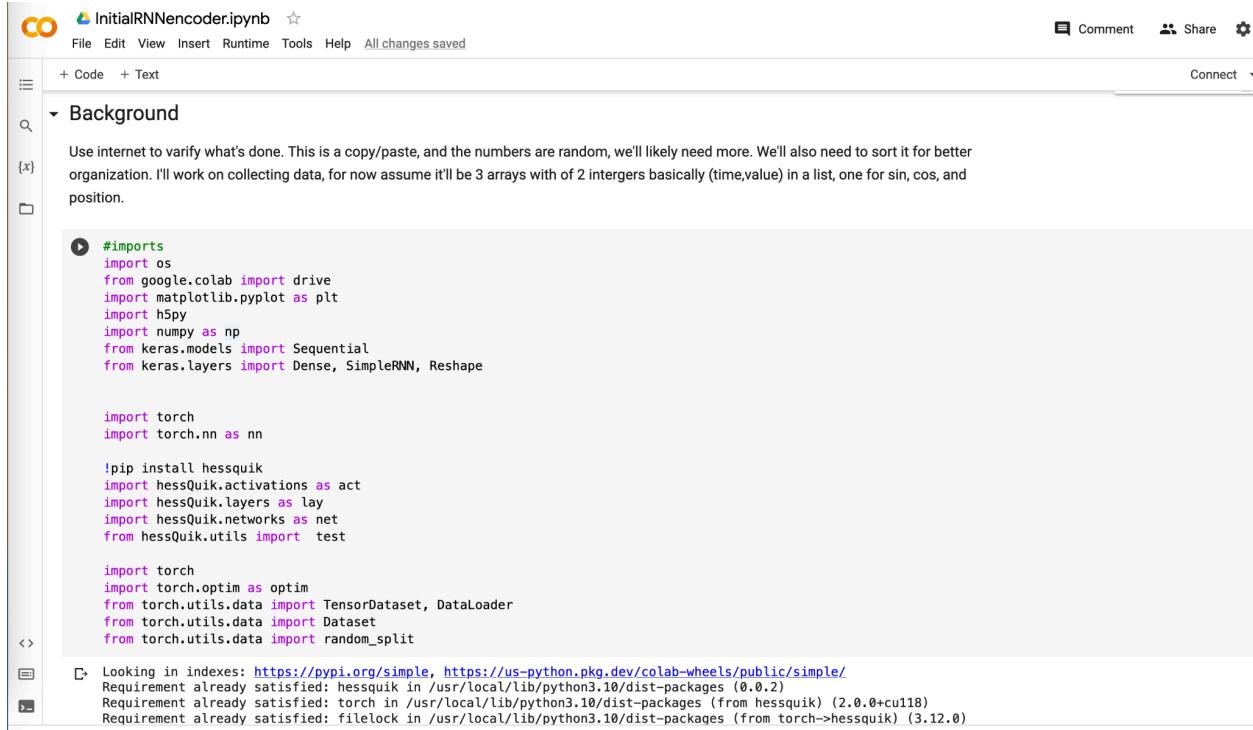
As we began utilizing Google Colab, we recognized the need to enhance our understanding of Neural Networks. Neural networks are machine learning algorithms that enable us to provide input data and their corresponding desired outputs, allowing the network to learn from these examples. This learning process empowers the RNN to generalize its knowledge and make accurate predictions when presented with new data.

By implementing an RNN within Google Colab, we aimed to leverage the capabilities of this robust neural network architecture. The RNN's ability to retain information from previous inputs makes it suitable for sequential data, time series analysis, and tasks involving temporal dependencies. By training the RNN with appropriate datasets, we could achieve improved performance and accurate predictions for our project.

Our project's final stage involved incorporating a Recurrent Neural Network (RNN) using Google Colab. This collaborative tool provided a unified data analysis and machine learning platform, enabling team development and easy access to shared resources. The RNN, a robust machine learning algorithm, allowed us to input data and desired outputs, and with training, it would provide accurate predictions for new data. By harnessing the capabilities of Google Colab and the RNN, we aimed to enhance the accuracy and performance of our project.

Implementation of RNN

The task at hand seemed straightforward initially: implementing an RNN based on the documentation we discovered from Keras. However, as is often the case with seemingly simple tasks, the execution proved to be more challenging than expected. To begin with any coding assignment, the first step is to import the necessary libraries. The image provided below displays the final list of imports we incorporated. As depicted, we utilized PyTorch, Keras, and Numpy to facilitate our implementation.



The screenshot shows a Google Colab notebook titled "InitialRNNencoder.ipynb". The code editor contains the following import statements:

```
#imports
import os
from google.colab import drive
import matplotlib.pyplot as plt
import h5py
import numpy as np
from keras.models import Sequential
from keras.layers import Dense, SimpleRNN, Reshape

import torch
import torch.nn as nn

!pip install hessquik
import hessQuik.activations as act
import hessQuik.layers as lay
import hessQuik.networks as net
from hessQuik.utils import test

import torch
import torch.optim as optim
from torch.utils.data import TensorDataset, DataLoader
from torch.utils.data import Dataset
from torch.utils.data import random_split
```

A status bar at the bottom indicates:

- Looking in indexes: <https://pypi.org/simple>, <https://us-python.pkg.dev/colab-wheels/public/simple/>
- Requirement already satisfied: hessquik in /usr/local/lib/python3.10/dist-packages (0.0.2)
- Requirement already satisfied: torch in /usr/local/lib/python3.10/dist-packages (from hessquik) (2.0.0+cu118)
- Requirement already satisfied: filelock in /usr/local/lib/python3.10/dist-packages (from torch->hessquik) (3.12.0)

Figure 22: Google Colab Import Section

The import section in Google Colab is essential for setting up the required libraries and dependencies, ensuring effective code execution. It includes essential libraries like PyTorch, Keras, and Numpy, which are crucial for various aspects of our project. PyTorch provides functionalities for building and training neural networks, while Keras simplifies the development of neural network models with its high-level API. Numpy, on the other hand, enables efficient numerical operations on arrays and matrices. By incorporating these libraries, we ensure our code has the necessary resources to implement and train the Recurrent Neural Network (RNN). These libraries offer data manipulation, model creation, training, and evaluation functions. Overall, the import section lays the foundation for our project, enabling us to leverage the capabilities of these powerful libraries within the Google Colab environment.

After setting up the necessary libraries, our next step involved loading the data required for our project. To accomplish this, we used Google Drive, which provided a convenient

way to store and access our recorded Sin, Cos, and Velocity signals. Using Google Drive, we ensured our data was easily accessible and well-organized.

In the figure below, we demonstrate how we loaded these three signals into our Google Colab environment, enabling us to perform further analysis and processing:

The screenshot shows a Google Colab notebook titled "InitialRNNencoder.ipynb". The code cell contains Python code for loading data from Google Drive. The code uses the `drive` API to mount the drive, change the working directory to a specific folder containing datasets, and then loads signals for Sine, Cosine, and Velocity. The code includes comments explaining the purpose of each step, such as keeping data loading the same for different signal types and adding noise if it's a sine or cosine wave.

```
drive.mount('/content/drive')
# check current working directory, make sure the data is stored in the directory
os.getcwd()
# change working directory to the data folder
os.chdir('/content/drive/Shared drives/M.A.R.S/Neural Network/datasets')

# function to keep data loading the same and adds the noise if it's a sine or cosine wave, simply make enough if it's a velocity
def load_signals(folder):
    numNoise = 1 #number of noisy versions of the signals that should be created
    noiseSDPercentage = 3 #percent of largest absolute value that the standard deviation of noise be'
    signals = []
    if folder == "Sine" or folder == "Cosine":
        for filename in os.listdir(folder):
            filepath = os.path.join(os.getcwd(), folder, filename)
            with h5py.File(filepath, "r") as f:
                data = np.array(f["data"])[();1]
                noiseSD = 0.01*noiseSDPercentage*np.max(np.abs(data))
                for i in range(0,numNoise):
                    signals.append(data) #+ np.random.normal(0, noiseSD, data.shape)
    elif folder == "Velocity":
        for filename in os.listdir(folder):
            filepath = os.path.join(os.getcwd(), folder, filename)
            with h5py.File(filepath, "r") as f:
                data = np.array(f["data"])[();1]
                for i in range(0,numNoise):
                    signals.append(data)
    return signals
```

Figure 23: Loading Data from Google Drive

Loading signals from Google Drive provided seamless access to the Sin, Cos, and Velocity data, which is crucial for training and testing our RNN. Google Drive's centralized storage facilitated collaborative development and eliminated the need for manual data transfer or individual storage. Overall, Google Drive streamlined data loading, enabling easy access to our signals in Google Colab for analysis, processing, and RNN training.

After successfully loading the data, the next crucial step was to process and shape the data to suit the requirements of the RNN model. The figure below visually depicts the data shaping process, often considered the most laborious part of model development.

The process of shaping the data for the RNN model required meticulous attention to detail and careful consideration of the specific model's input requirements. It involved transforming the raw data into a suitable format that effectively captured the temporal nature of the problem.

▼ Step 2: Pre-process Data

```
❶ sin = ((np.array(sines))) #I kind of feel like if you looked back at the load data function and changed a few things half of theses would be unnecessary
cos = ((np.array(cosines)))
sin = sin[None,:]
cos = cos[None,:]
# print(sin.shape)
x = torch.tensor(np.concatenate([sin,cos]))
# print(x.shape)
reshaped_x = x.view(120, 40001)
# print(reshaped_x.shape)
x = reshaped_x.reshape((60, 2, 40001)).mean(dim=1)
print(x.shape)
np.reshape(velocities, [-1,1])
y = torch.tensor(velocities)
print(y.shape)

[] torch.Size([60, 40001])
torch.Size([60, 40001])
```

Figure 24: Shaping Data for RNN Model

By shaping the data in accordance with the figure depicted above, we ensured that the RNN model could process the data correctly and make accurate predictions. This critical step paved the way for successful training, evaluation, and application of the RNN model to our problem domain.

```
❶ def build_model(input_shape):
    # Define the model architecture
    model = Sequential()
    model.add(SimpleRNN(units=64, activation='tanh', input_shape=input_shape))
    model.add(Dense(units=input_shape[0], activation=None))
    #model.add(Reshape((1, input_shape[-2], input_shape[-1])))

    # Compile the model
    model.compile(optimizer='adam', loss='mse')

    return model

# N_samples=120
# Length_sample=40001
# x= np.random.rand(N_samples,Length_sample) # simulated dataset of 120 samples with 40001 time steps each
# model = build_model((Length_sample,1))
# model.summary()

[] model = build_model(x.shape)
model.compile(optimizer='adam', loss='mse', metrics=['mae'])

history = model.fit(x_train, y_train, batch_size=32, epochs=10, validation_data=(x_val, y_val))

# loss = model.evaluate(x_test, y_test)

model.summary()
```

Figure 25: RNN Model Build

Lastly, our focus shifted towards building the RNN model, encompassing several vital points. While constructing the model itself was relatively straightforward, the most challenging aspect lay in training the model effectively. During this process, we encountered a specific obstacle: incorporating different types of noise to enhance the model's ability to handle real-world disturbances, the exact nature of which remains unknown.

Initially, our approach involved introducing various forms of noise during the training phase. The intention was to expose the model to various disturbances, thereby simulating real-life scenarios more accurately. However, despite correctly organizing and preparing the data, we needed help achieving the desired output, even after multiple attempts at reshaping the data and refining the training process.

Our results did not align closely with the intended output. Due to our persistent challenges and the lack of progress in attaining the desired performance, we temporarily set aside this specific problem. Despite our diligent efforts, the outcome fell below our expectations. Nonetheless, we acknowledged the complexity and intricacy of the task, recognizing that further exploration and experimentation might be necessary to address this challenge effectively.

By acknowledging the limitations and complexities associated with training the RNN model and recognizing the need for additional refinement, we remain open to revisiting this problem. This experience has provided valuable insights into the difficulties of achieving accurate predictions in real-world scenarios, emphasizing the importance of continuously improving and fine-tuning the training process to enhance model performance.

Safety Issues

Safety issues involved in this project include data safety and electrical safety when working with voltage. Northrop Grumman required our team to sign a non-disclosure agreement to work on this project. This is because most of their projects are directly related to working with the United States government. Therefore, data safety is something our team needs to be mindful of when sharing details of our project with others and in presentations. The other safety issue involves practicing electrical safety when working with our physical apparatus. We were working with a high-wattage motor and PWM, which both need voltage in order to get them to function correctly.

Project Results

We have completed our Simulink models as required by Northrop Grumman. This includes the PLL (phase-locked loop) resolver encoder algorithm, the Third-Order Rational Polynomial resolver encoder algorithm, and the S-Transform resolver encoder algorithm. We have presented these models to Northrop Grumman and received positive feedback, and completed our part of the project for them. There were a lot of challenges and lessons that we learned from this project, including learning to work with MATLAB Simulink, using MATLAB to alter and optimize algorithms, getting to understand brushless motors on a more technical level, learning how to create a neural network, and probably the most important, learning to work with a third party. Some standard design modules that could potentially be used for future projects are the physical model we produced for the project and the Simulink models. Our group's physical model for this project was already reused material from a previous senior design project to be reused in the future. The Simulink models are very complex and potentially reusable, but it may depend on the NDA that our group signed with Northrop Grumman on it being able to be reused in the future. If we continue working with Northrop Grumman, we could work on additional Simulink models for resolver encoder algorithms. Doing this could lead to the discovery of a better algorithm for the purposes that Northrop Grumman needs to be fulfilled.

Another thing that we would like to continue working on is the development of the neural network. The neural network was not one of the goals given to us by Northrop Grumman, nor was it one of the goals set for our project by us, but it was something we wanted to test on the side. Because of this, there was not much time and effort put into it compared to what Northrop Grumman wanted from us, and it was primarily left unfinished. If we had more time for this project, the neural network could work better than a traditional Simulink algorithm, so it would be worth pursuing.

Engineering Standards

A few standards we defined are as follows:

IEEE Recommended Practice for Motor Protection in Industrial and Commercial Power Systems.

This protection describes the use of electric motors in industrial and commercial applications. This covers dc motor protection with factors based on types of protection, low-voltage and medium-voltage protection, fixed speed, and an adjustable speed drive application. [IEEE 3004.8-2016](#)

High-Integrity System Modeling Guidelines.

Provides model setting, block usage, and block parameter considerations for complete, unambiguous, robust, and verifiable models. Complying with the [DO-178C](#) / DO-331, [IEC 61508](#), [IEC 62304](#), [ISO 26262](#), or [EN 50128](#) industry standards.

MAB Guidelines.

A guideline for basic rules of modeling with Simulink. The purpose of this guideline is to allow for an easy and shared understanding of control systems. The objectives stated for this guideline are readability, simulation and verification, and code generation. [Model Advisor Checks for MAB and JMAAB Guidelines](#)

References

- <https://www.northropgrumman.com/>
- https://www.mathworks.com/help/simulink/mdl_gd/maab/model-advisor-checks-for-mab-and-jmaab-guidelines.html
- <https://standards.ieee.org/ieee/3004.8/5028/>
- <https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3231115/>

- <https://www.mathworks.com/help/simulink/gui/libraries.html>
- <https://www.hindawi.com/journals/amse/2016/1497360/>
- <https://ieeexplore.ieee.org/abstract/document/1185415>
- <https://www.tensorflow.org/guide/keras/rnn>
- <https://colab.research.google.com/>

Appendix

Equipment Needed

We have utilized various equipment and tools in our project to accomplish our objectives. We have used a brushless motor, a resolver, 3D printed pieces, a threaded rod, Arduino for PWM, a brushless ESC (Electronic Speed Controller), and two couplers. We also need an oscilloscope to take readings, a power supply, and waveform generator. These elements have played crucial roles in the functioning and operation of our physical system.

Additionally, implementing MATLAB has been integral to the success of our project. Within MATLAB, we extensively relied on Simulink, a dynamic systems modeling and simulation tool, to design and simulate our system's behavior and performance.

Furthermore, we received recommendations from Northrop Grumman during our project. They suggested employing several specific libraries and tools within MATLAB to enhance our work. One such tool is the "Fixed-Point Designer," which aids in developing and optimizing fixed-point systems. We also used the "Electronics and Mechatronics Sensors" library, which provides a range of sensor models for designing and testing electronic and mechatronic systems.

Finally, we leveraged the capabilities of the "Foundation Library Mechanical Sensors" to incorporate various mechanical sensors into our system. This library offers a comprehensive set of pre-built sensor models, enabling us to simulate and evaluate the performance of our mechanical components accurately.

Our project has relied on diverse equipment, including a brushless motor, resolver, 3D printed components, threaded rod, PWM, brushless ESC, and couplers. MATLAB and its Simulink platform have been essential for simulation and analysis. The recommendations from Northrop Grumman introduced us to valuable tools such as the "Fixed-Point Designer," "Electronics and Mechatronics Sensors," "Foundation Library Mechanical Sensors," and "HDL Coder," enabling us to enhance our project's performance and implementation.

Budget

Our group was given a budget of around \$200. We didn't need to spend that much extra money since we repurposed a previous senior design apparatus and motors. We were also provided with a brushless motor, resolver, brushless ESC, and Arduino. Therefore, the only cost we used on the physical apparatus was the two couplers to connect the resolver to the motor, the motor to the threaded rod, and a new resolver since the one we found from old senior designs didn't work. Both couplers cost us around \$25 combined. The resolver used cost us \$57. We also would need access to the MATLAB libraries stated in the "Equipment Needed" section, which would be hundreds of dollars over budget. In the end, these libraries were provided for free through the University, so no extra cost was generated there.