PROJECT 1: Inverted Pendulum

I wanted to build a robot that would truly apply the principles I learned in my controls classes at Purdue. As a result, I decided to build an inverted pendulum. This is simply a free-spinning pendulum attached to an encoder on a cart that is actuated by a motor. By referencing the encoder readings, we can establish an output to our system, and the input to the system is the acceleration of the motor. Since an input and output are established, a closed-loop controlled system can be created, and with utilization of a PID, steady-state error and settling time can be reduced to a minimum. Below is a picture of the pendulum itself balancing on its end.

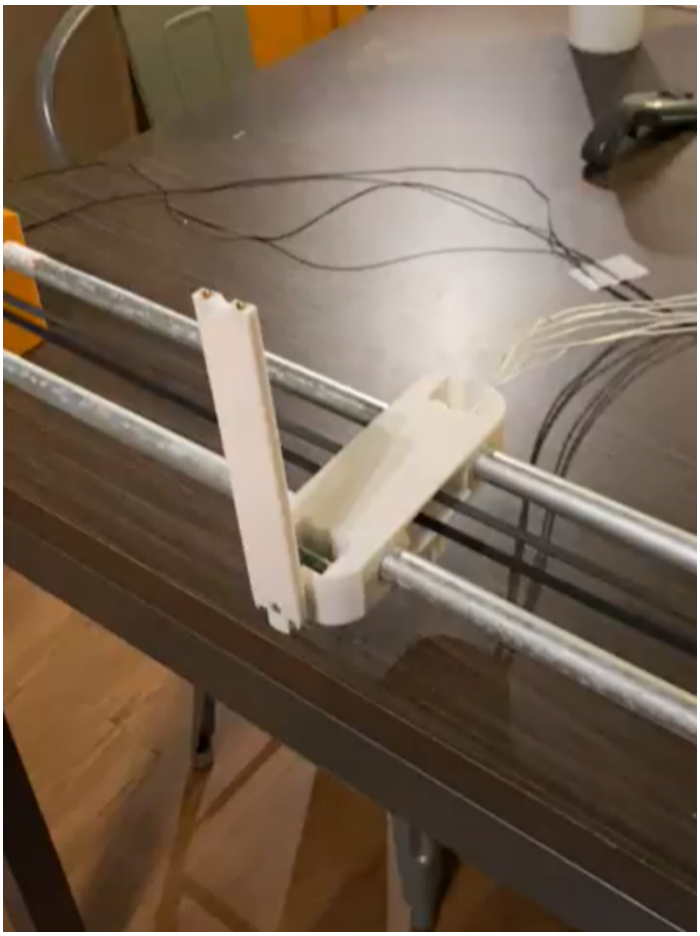


Figure 1 – Inverted Pendulum Cart Actively Balancing

Figure 2 – Full Inverted Pendulum Assembly (with Triscuit Cameo)

Before diving into PID implementation, I first had to build the robot and I wanted to establish a rudimentary control system beforehand just to test the concept. The pendulum carriage, pendulum, and support brackets were 3D printed due to the need for custom geometry and fabrication speed, but all other components were bought from Amazon (encoder, wires, metal guide bars, timing belt/pulley, and motor.

I am using a quadrature rotary encoder (Figure 3) to ensure that I can tell which direction the pendulum is falling at any given time. I am planning on changing this encoder in the future though, as it only has 96 counts per revolution, resulting in a resolution of 3.75 degrees. I need a higher angular resolution than that to achieve lower steady state error and settling time when the control system is implemented, however, this 96 count encoder is sufficient for now to create the rudimentary control system.

Figure 3 – Pendulum Attached to Quadrature Encoder

For electronics, an Arduino Uno will be running the software, and everything is connected to it via breadboard (Figure 4). A Nema 17 stepper motor (Figure 5) was initially used, along with a properly rated stepper motor driver to actuate it.

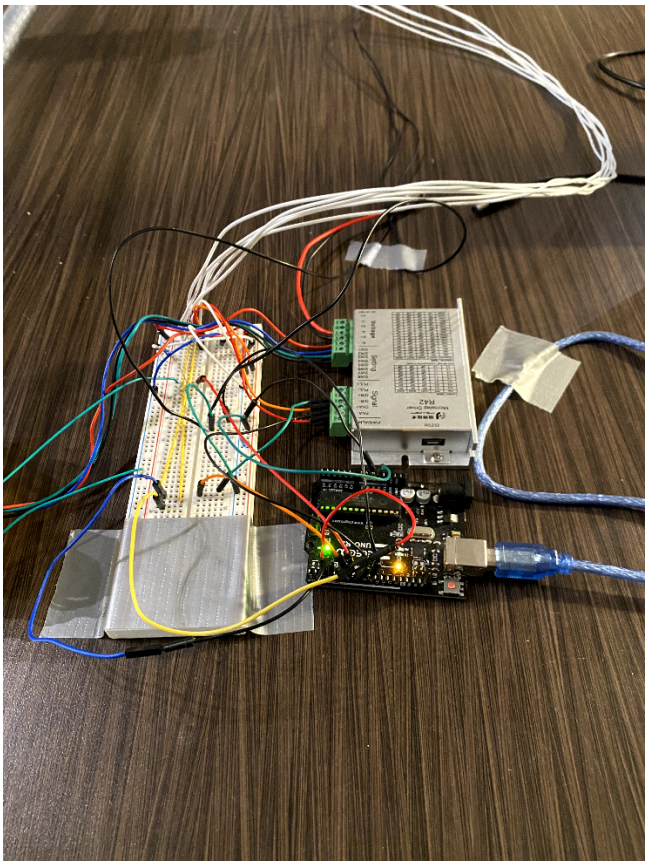


Figure 4 – Electronics (Arduino, Motor Driver, and Breadboard) Secured to Table

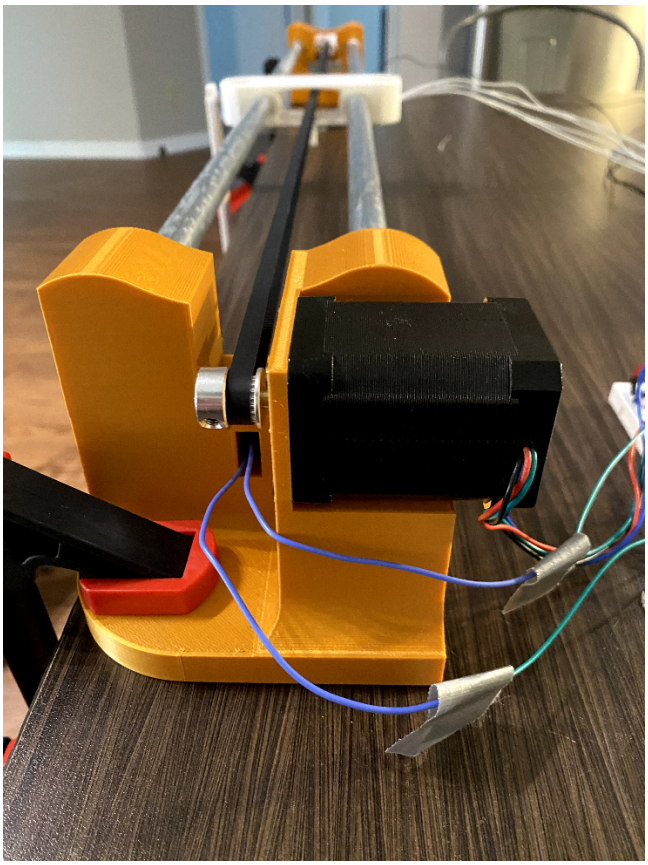


Figure 5 – Nema 17 Stepper Motor

Since a stepper motor is being used, and considering at the start of every run the carriage is not necessarily in the same position each time, a homing/calibration sequence was created that executes before active balancing occurs. This system consists of two limit switches: one attached to the motor holding bracket and one attached to the non-motor bracket. The carriage has prongs on its underside that depress these limit switches when in close proximity. By having some code run that forcibly traverses the carriage towards the motor, I can detect when the motor wall is hit, store the motor step count at this location, and instantly change the carriage's direction of travel. The carriage then drives all the way to the other side of the assembly until it hits the other bracket, and the step count is recorded and direction changed as well. The carriage then travels to the halfway point between the two brackets, and once that location is reached, transitions to active balancing. Now, I have the lower motor step count and the upper motor step count to be used as bounds for the carriage's travel during the balancing stage. If these limits are ever met or exceeded, the code immediately stops to prevent hardware damage.

The current control system in place is a simple one – encoder counts are converted to angle values, and when the angle that is read deviates from 90 degrees (the ideal position of the pendulum), the motor spins at a rate proportional to the difference between 90 and the read value. For example, if the encoder read angle is at 80 degrees, the motor will spin relatively

slowly in the direction to push the pendulum back to 90 degrees; yet, if the read angle is at 60 degrees, the motor will spin faster to account for the larger angular distance that needs to be traversed. This simple control system works well for very low-level testing, however, it is nowhere near sufficient to be able to actively balance the pendulum. Often times when running this control method, the pendulum will still fall down to resting position, or it will remain balanced, but the balancing will be incredibly jittery. I spent some time tuning this P controller to be fast enough to correct the pendulum as it falls down, and barely is just over-tuned to oscillate. I then added a D controller to dampen the oscillations as well as an I controller to reduce steady-state error. This was able to balance the pendulum well, but the carriage moves all over the place. I am currently working on implementing and tuning a carriage position PID controller as well that will be added to the angle controller to keep the pendulum balanced and the cart centered on the track.

PROJECT 2: 3D Printer with Embedded 3D Vision for Closed Loop Control

3D printing is a fantastic tool. It allows us to create custom, functional components for all stages of the design lifecycle. However, one of the biggest problems that prevents 3D printing (specifically FDM printing) from being used more ubiquitously is its reliability. Any 3D printer that currently exists works in essentially the same manner - a file containing a set of predetermined instructions is fed into the printer, and the part is printed by reading and interpreting those instructions step-by-step. The reason this process is unreliable is because if any physical anomaly happens during the print, the entire process needs to be restarted, wasting time and material - for example, if the printer is bumped by someone mid-print, the molten plastic coming out of the nozzle can drip, causing undesirable blemishes on the model's surface. The current way printers work does not account for environmental changes, which my research aims to fix. We are taking a commercial FDM 3D printer and modifying it by integrating it with a 3D vision system. By allowing the printer to see what is happening as it is printing, we can allow it to make real-time corrections should an unforeseen problem occur. This system can fix a lot of potential issues that commonly plague commercial users, including stringing, poor bed adhesion, over/under-extrusion, and even layer delamination. The creation of this system will massively change the way we think of 3D printing as a tool and will allow for larger scale adoption of the technology across all industries. By no longer having to worry about reliability, it will easily become the single best choice for fabrication in terms of time and cost efficiency.

The system consists of the 3D printer itself (Item 1), as well as the 3D scanning components: the light stripe projector (Item 2), and the camera (Item 3). In order to take a 3D scan, the light stripe projector shines varying patterns of light lines (patterns that kind of look like barcodes) on objects being printed on the printer's build plate, and the camera captures how the light deforms when it interacts with the object. From this deformation, we can reconstruct the object as a 3D file, as a collection of individual points in 3D space, called a pointcloud. We then use Matlab on the computers (Item 4) to denoise these pointclouds, and compare them with the originally modeled 3D file to see if everything is printing as expected. If the pointcloud does not match up

with the 3D file well, then an error has occurred, and we use custom software to add extra printing instructions to fix the problem.

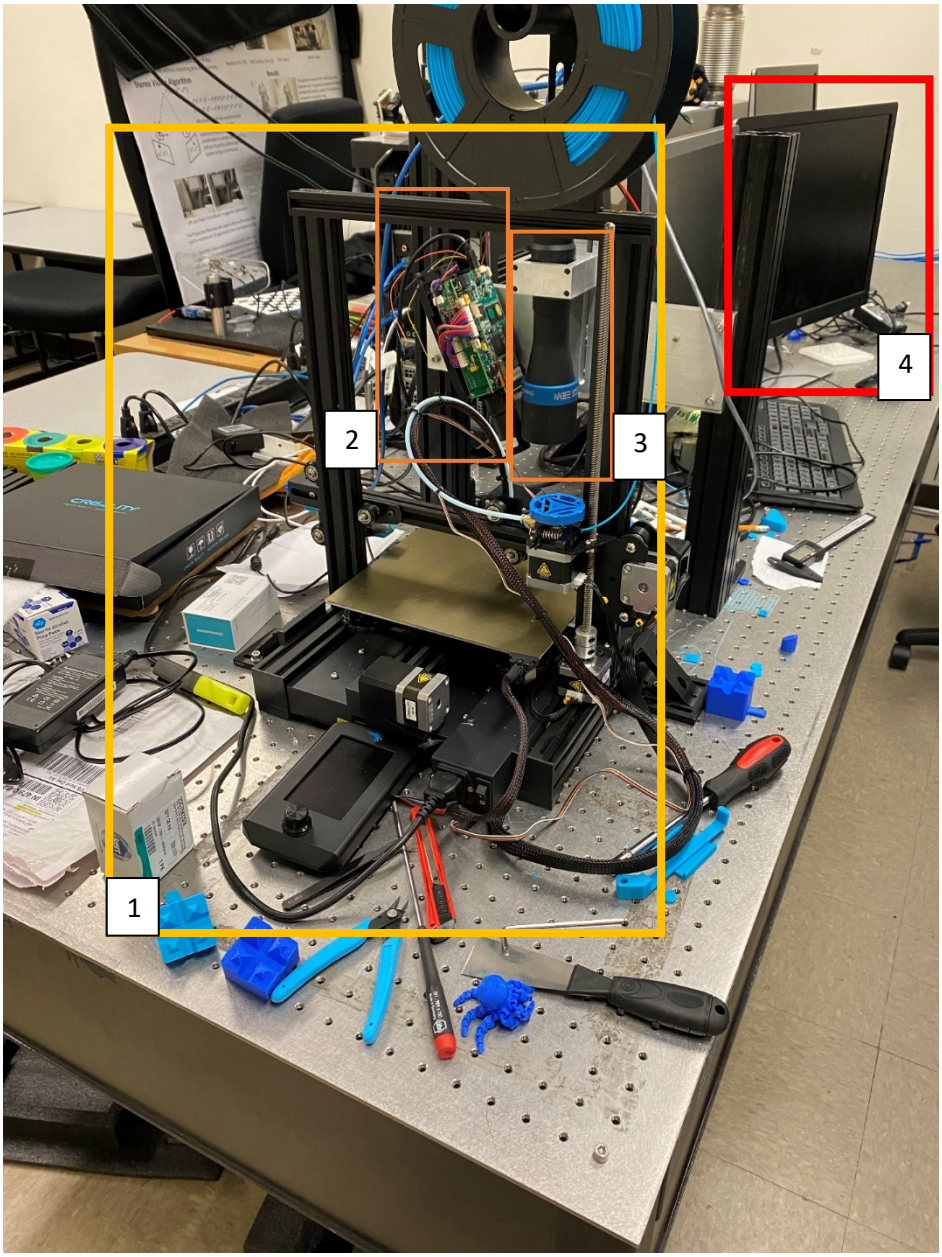


Figure 1 – Overview of the System

The main problem that my work focuses on is calibration. The 3D printer has its own 3D coordinate system that it operates in, and so does the 3D scanner. However, these two coordinate systems do not agree with one another. Due to bed warping and thermal effects between prints, the magnitude of this mismatch between coordinate systems can change drastically at any given time. Therefore, I needed to come up with a method to determine a transformation between points in the scanner coordinate system and points in the printer for every single print. This process is important because the pointcloud data and mismatch analysis happens in the scanner's coordinate system, and in order to make dimensionally accurate error

fixes in the form of injected GCode instructions, this transformation needs to be as accurate as possible. The method I came up with is this:

1. At the start of a print, wait until the bed heats up, and take a scan of the empty bed for reference later.
2. After the empty bed scan, inject custom GCode to draw a rectangle at the edge of the build plate (out of the way of the to-be-printed component) that fits just inside the camera's field of view. This rectangle has known GCode coordinates that do not change between prints – they always stay the same. While this would not typically work for large-scale printed parts (as the part's profile may take up the full build plate), our system is meant for small-sized, microscale error detection, and as such, can only print components a couple of centimeters wide (therefore the part will never stray away from the center of the build plate and will never collide with the calibration rectangle).

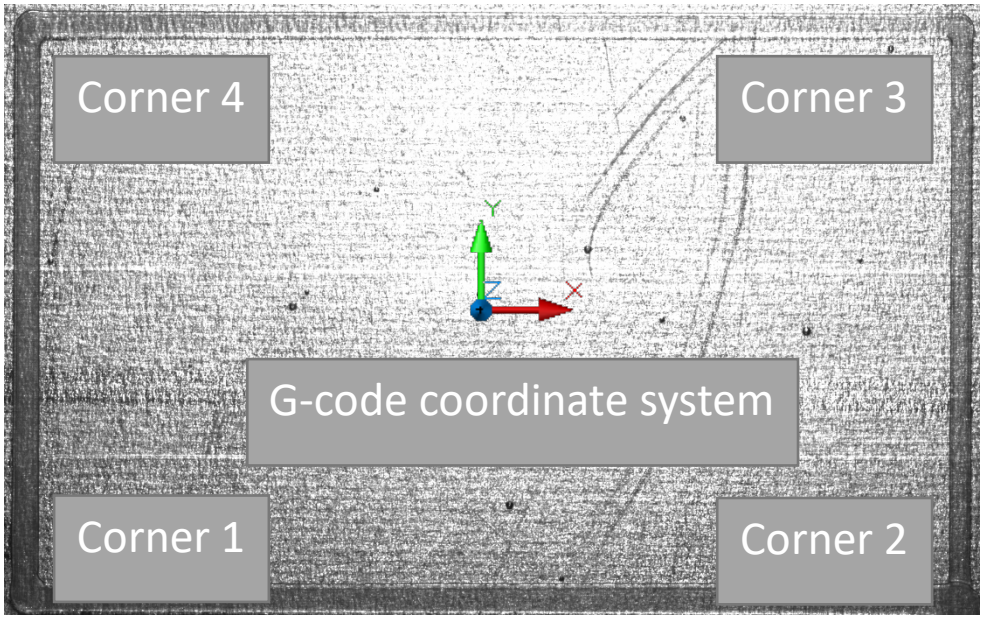


Figure 2 – Picture of 3D Printed Calibration Rectangle

3. Once the rectangle is printed, a scan is taken. We now have a pointcloud of the empty bed and a pointcloud of the rectangle. As seen in the picture, the rectangle consists of the printed plastic, but the scan also picks up the build plate in the middle region where plastic was not deposited. These build plate points need to be removed, which is done by comparing the rectangle scan with the empty bed and removing any similar points between the two. We now have an isolated pointcloud of just the deposited plastic at the edges of the rectangle.
4. 3D linear regression is performed on each of the edges of the rectangle, and the x, y, and z locations at which the lines intersect (or get close to intersecting) are recorded

as the locations of the 4 corners in the scanner coordinate system (Figure 3).

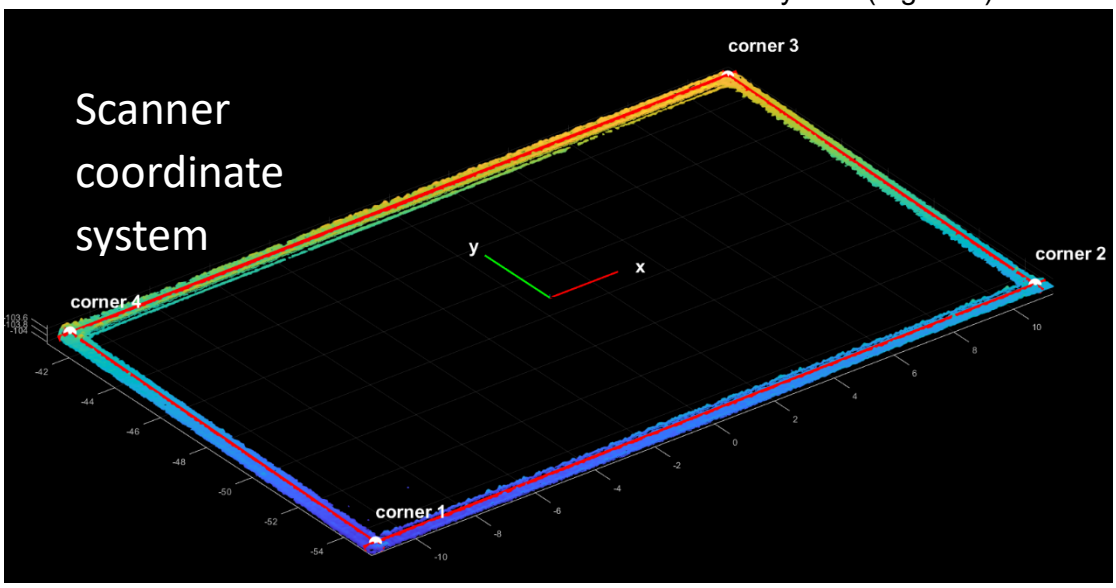


Figure 3 – Corner Detection Using 3D Linear Regression

5. We now have the x, y, and z locations for all 4 corners in both the printer's coordinate space and the scanner's coordinate space. Using linear algebra, a transformation matrix is determined, and is used throughout the duration of the print.

Once dimensional accuracy in scans was established, the next step in the project was to identify areas of underfill and overfill in the 3D print. This process was relatively simple, as it just involved a comparison between the ideal 3D geometry of the current layer from the sliced file and the 3D scan that was taken. Any areas of mismatch could then be classified as underfill depending on if too little material was present in the mismatch. A graphic of this can be seen in Figure 4, where areas marked blue are where the two 3D profiles match, while red areas are locations of underfill.

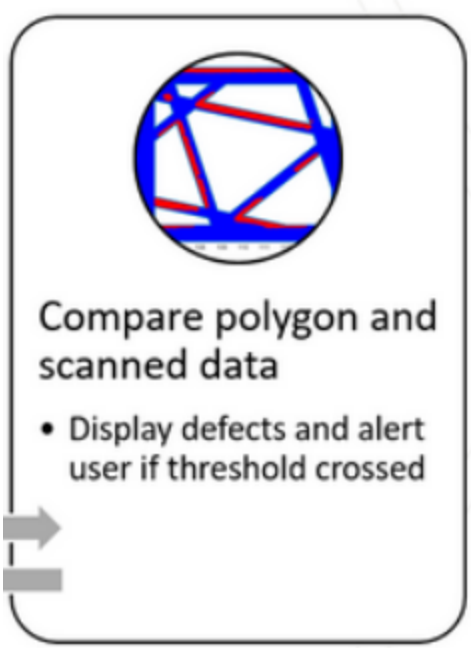


Figure 4 – Error Detection Method

Once these areas of underfill are determined, we can now use the Slic3r MATLAB library to generate specified G-code commands to fix the underfill (Figure 5). For now, all the system detects is underfill, as this is the easiest problem to remedy. However, overfill capabilities are currently being developed, and will be implemented in the coming semester.

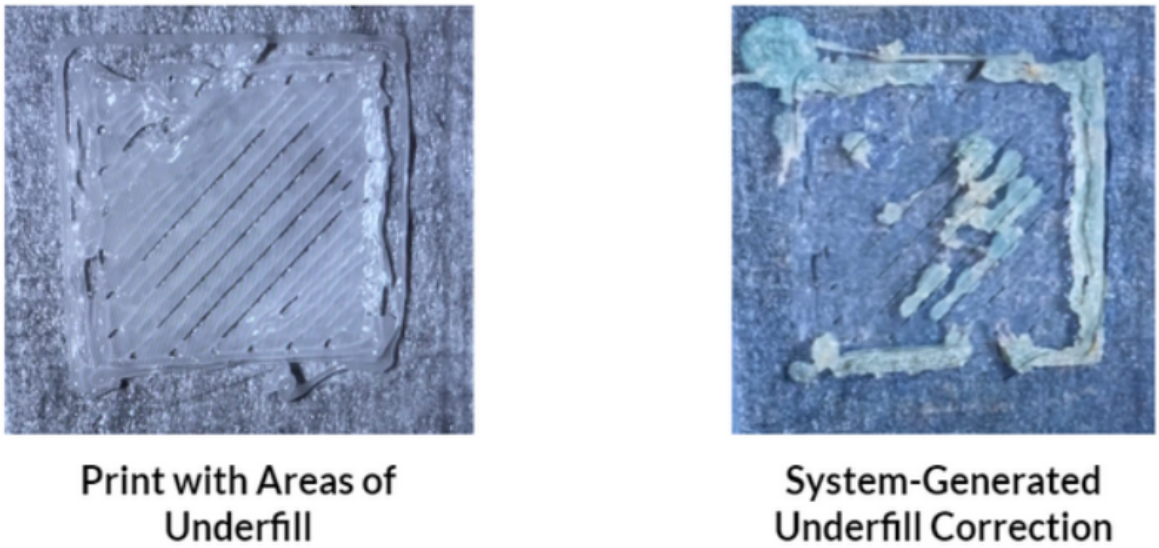


Figure 5 – Algorithm-Generated Underfill Correction G-code

Project 3: PID Control for Training Visuomotor Robot Manipulation Policies

I. Introduction

The goal of this project was to create an imitation learning policy that can learn to control a UR5 robot arm in a straight line from one point to another. This learning was to happen with only images that were taken at each timestep along the robot's journey and compared with the expert trajectory of the robot following PID control. The UR5 robot arm was loaded into a pybullet environment with a ground plane, and learning was performed using pytorch.

This project was split up into three tasks - planning, control, and learning. The planning and control stages contributed to data collection for the imitation learning model to train off of.

There were a couple of assumptions that were created to ensure the completion of this project with the given time constraints. While a robot can be controlled to achieve a desired position and orientation, for this project, only positional control was considered. This way, only 3 degree of freedom kinematics need to be considered rather than 6. Additionally, no obstacles were included within the robot's environment - while this also helped development, this assumption was mainly chosen to ensure that the entire robot arm could be seen when an image was taken. If the arm was hidden behind an obstacle, there would be unnecessary strain on the neural network as parts of the observation space became obfuscated. Finally, in order to perform accurate inverse kinematics, a third-party library called kinpy was used. Kinpy performs the IK directly from the robot's URDF file, which allows it to choose configurations that disallow self-collision, which was really useful for development.

II. Planning

The path planning for this task was relatively simple, as it only required generation of a straight-line path between two randomly generated points. However, these start and goal points needed to lie within the reachable workspace of the UR5 - this was achieved using a strategic sampling technique. From the URDF file, each link measurement could be obtained and summed together. This resulted in the maximum reach distance of the robotic arm, which will be denoted as R. The x-position of the point could then be determined by sampling between -R and R. At this x-coordinate, due to the reachable workspace being a sphere, there exists a maximum reachable z-coordinate that the robot can achieve. This max z can be calculated with the following equation: $z_{max} = sqrt(R^2 - x^2.)$ Therefore, the z-coordinate can be sampled between 0 and z_{max}. Finally, the corresponding y bounds can be determined through the equation of a sphere: $y_{bound} = sqrt(R^2 - x^2 - z^2)$. The y-coordinate can then be sampled between $-y_{bound}$ and y_{bound}. This process ensures a randomly generated point whose coordinates all lie within the reachable working hemisphere of the UR5.

After generating a start and goal node with the above strategy, a trajectory then needed to be created between the two. A global step size was created, which in this case, was 0.01. The Euclidean distance between the two points was found, and the corresponding dx, dy, and dz were found as well. By dividing the distance by the step size, the number of steps in the trajectory could be determined. By dividing dx, dy, and dz by the number of steps, an increment in each axis was found, which could then be iteratively added to the start point's coordinates to create a straight-line trajectory to the goal node. The cartesian position of each point was stored in an array called trajectory. Inverse kinematics were done at each point along the trajectory and the joint configurations at each step were stored in an array called confTrajectory.

III. Control

A PID controller was implemented to follow the generated trajectory. Since each joint in the robotic arm corresponds to an independent drive motor, each joint must be individually controlled by PID to achieve

accurate results. By comparing the robot's current configuration at each joint with the stored ideal position within the confTrajectory array, the error can be determined. By storing the previous error and logging time, de/dt can also be determined, and as such the following PD controller could be implemented on each joint: Torque = $k_p * e + k_d * de/dt$. Torque was applied to the UR5 through pybullet's applyExternalTorque() function applied to the link frame. The k_p and k_d values for each joint were tuned by plotting the error over time for each joint and adjusting the gains according to the plot behavior. If the response time was slow, increasing k_p would force the response to be quicker, yet increasing k_d would help fix oscillations/overshoot.

The control portion of this project was the trickiest, as 6DOF robotic arms are subject to singularities along their travel - if a singularity is hit, kinpy's inverse kinematics func- tion teleports the joints of the robot arm to their new config- uration. This caused discontinuities within the confTrajectory array, and as such, in these situations, the controller would not have the best performance. Therefore, while this classical manual PID approach was a great learning opportunity, this implementation was not used in the final code.

Pybullet has a lot of useful in-built functions, one of them being setMotorControl2, which allows the user to specify the type of motor control each joint in the robot can have, with the options of position control, velocity control, and torque control. The above implementation had each joint in velocity control mode. However, by setting the control mode to position, setMotorControl2 can also take in gain parameters for both position and velocity, as well as a target position and target velocity, effectively allowing pybullet to implement the PD controller on its own. While a lot of time was still spent tuning the gains for each joint, the resultant behavior was a lot smoother and handled singularities a lot better (although still crossing one caused much undesired oscillation). However, since there were 5 joints that determined the position of the end effector, this required tuning of 10 different parameters, which was very time consuming and effort intensive.

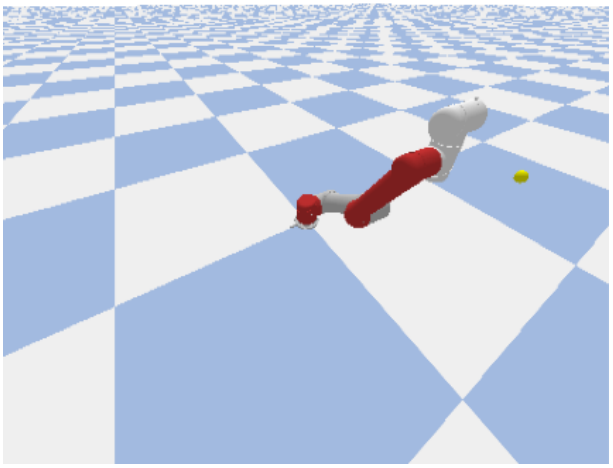


Fig. 1. Example Image of Environment 1/100

IV. Learning

After the PID controller was made, data needed to be collected for training of the imitation learning policy. A variable called numEnvs was created and set to 100. An environment describes an instance of a pair of start and goal points that the robot travels between using PID control. By randomly generating 100 different environments, 100 sets of data could be collected. At each timestep along the PID travel, pybullet.getCameraImage() was called to collect a 320x460 image for that specific time in the environment. Each of these images was stored in chronological order under its environment's

corresponding directory within the data folder. Therefore, by the end of runtime, the training data folder contained 100 separate directories, each corresponding to the 100 randomly generated environments. Each of these directories contained an image of the robot using PID control to travel along the trajectory at every timestep.

A behavioral cloning imitation learning policy was used due to its simplicity as well as the fact that it is not a supervised learning modality. The policy took in the entire 320x460 image as an input and mapped that to a 1x5 action space representing the torque to apply to each joint. The optimizer used was Adam, and upon tuning the hyperparameters, a learning rate of 3e-4 was chosen. The training loop was structured as follows:

1. Choose random environment from dataset, load in start and goal nodes, set robot joints to start configuration
2. Take image, query policy to predict actions for each joint
3. Call pybullet.applyExternalTorque() on each joint with the predicted action, step the simulation, and take another picture
4. Use expert image at current time t as label, perform backpropagation between collected image and expert image
5. Repeat for 5 total environments, chosen stochastically from the original 100

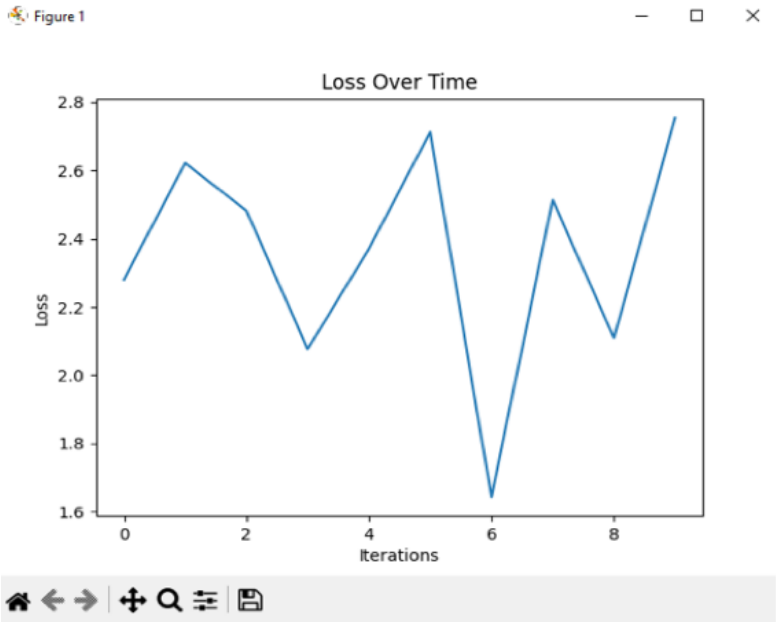


Figure 2. Initial Results of Training, 10 Iterations, 5 Envs per Iteration

The reason there are multiple environments per iteration is to ensure the model does not overfit to any one given environment - rather, in an iteration, it will calculate loss based on multiple different environments to be as general-purpose as possible. This loop was repeated for 10 iterations to collect some initial results, as seen in Figure 2. Due to the size of the input as compared to the output of the network, there was much loss, especially considering the initial implementation only had 3 layers to the network.

Researching further into image-based neural networks revealed something called resnet18, which is an 18-layer pre-trained neural network designed specifically for image classification. Therefore, by piping

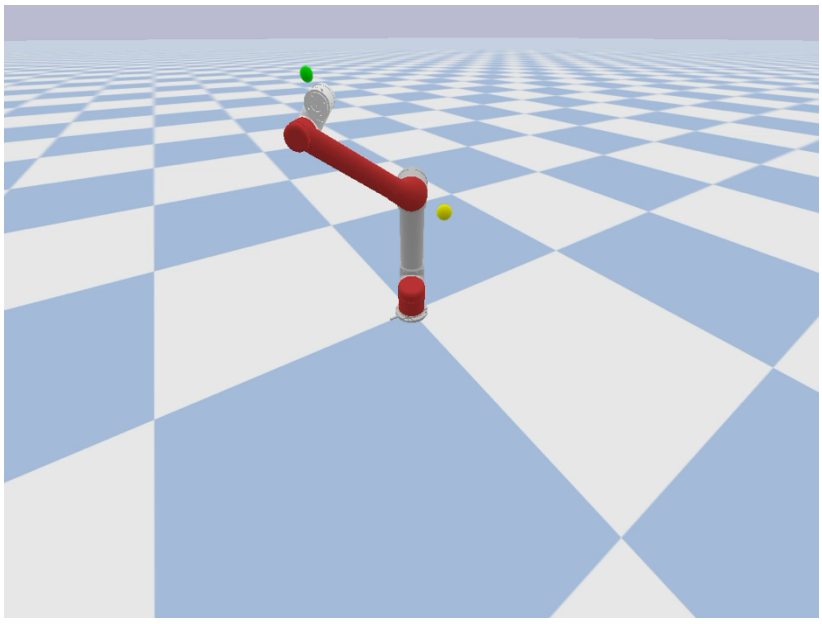
resnet18 into an output 1x5 layer, the loss from processing the large input image was reduced. The resultant policy can be seen in Figure 3.

```
class Policy(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(Policy, self).__init__()
        self.net = resnet18(pretrained=True)
        self.net.fc = nn.Linear(self.net.fc.in_features, output_size)
        self.net.conv1 = nn.Conv2d(4, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)

    def forward(self, x):
        return(self.net(x))

def choose_action(state, il_policy):
    state = torch.tensor(state, dtype=torch.float32, requires_grad = True).unsqueeze(0).to(device)
    action = il_policy(state)
    return action
```

Figure 3: Policy for Imitation Learning

Training was then repeated using this new policy structure for 200 iterations at 5 environments per iteration. The batch size was the length of the trajectory for the environment being trained on, which on average, was around 300.

Despite a tumultuous loss graph, it can be seen that valid training has occurred by running testing.py as seen in Video 1. While the end effector does not get to the goal, it is moving in the correct direction, albeit slowly towards the goal node. This further influences the idea that in order to get better performance, the model simply needs more data and iterations to train off of.



Video 1: Learned Control of Robot Arm

V. Conclusion

While this loss is less than ideal, it is most likely due to the number of environments per iteration being only 5, rather than a larger number. Additionally, the dataset of 100 total environments is small and should realistically be higher by an order of magnitude as well. However, the hardware used to perform the training of the model is really old and cannot handle much more than what was presented in this report. Certain techniques such as using the action space rather than the entire image of the expert trajectory were attempted, but could not be completed within the given time constraints. Perhaps another future point of improvement after the completion of this course would be to implement a replay buffer in order to ensure the model does not forget what it has already learned.

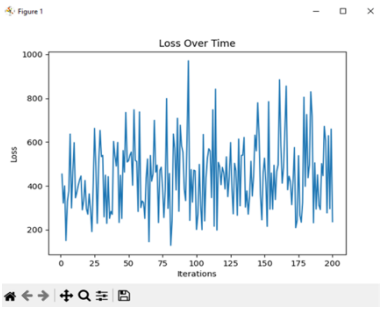
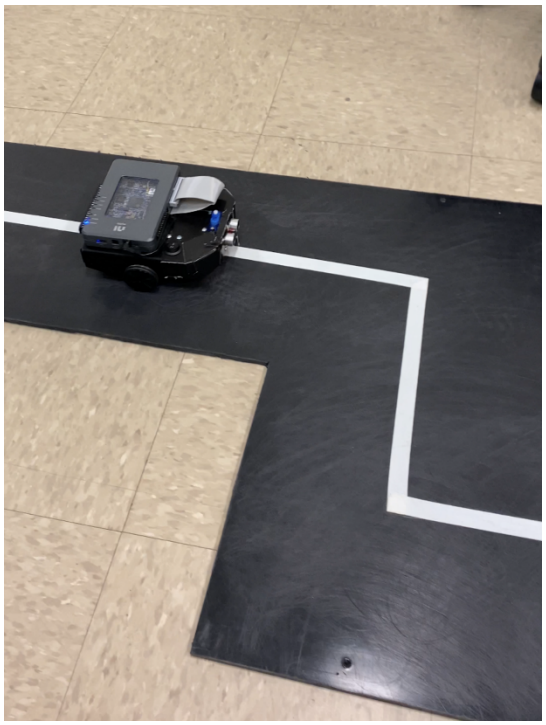


Figure 4. Results of Updated Policy Training, 200 Iterations, 5 Envs per Iteration

PROJECT 4: Skitter Robot

1. Introduction

The task of this project was to create a robot that could achieve two goals - that of line-following and that of herding. Line following used a 5-light line-following sensor, and herding used an ultrasonic sensor. The goal of line-following was to follow a line of white tape laid on a black track in order to guide the robot from start to finish. When arriving at the finish, the robot turns into herding mode, where if an object is placed in front of it, it must move to be as close to 9 inches away as possible, with ability to make corrections both forwards and backwards. The approach taken was a simple one: for line-following, a state machine was used in order to coordinate the movements between the two motors. Specifics of this state machine will be addressed in the Project section. For herding, a PID controller was implemented, where the desired location was 9 inches, and the response was the reading from the ultrasonic sensor. Gains of this PID controller were varied to produce low steady state error and low settling time. This process will also be further explored in the project section. Below is a picture of the robot:



2. Project

State Machine Design:

When placed on a black board with a white line, any time one of the bulbs on the line-following sensor passes over the line, the sensor reads true, and black reads false. With this simple logic in mind, a state machine could be built. As part of the robot hardware, there was a breakout board that connected the line-follower sensor to Port B on the myRIO. However, this breakout board only allowed 4 analog inputs, so one of the sensors on the line follower could not be used. Of the five total sensors (U1 - far left, U2 - left, U3 - middle, U4 - right, and U5 - far right), we decided to use all but U4. The U1 and U5 sensors were chosen as they were far away from middle, and therefore would not be falsely triggered when driving straight. Once the sensors were chosen, simple logic was set up to run the track. The track was a long straight line followed by a right turn, with a solid horizontal line at the end meant to activate all of the sensors to go into herding mode.

The first part of the state machine built was simple line following - that is the series of moves that moves the robot along the straight line portions of the track. If just U3 was activated, the robot would move both wheels at the same velocity (7.5 in/s), moving straight. If U2 was activated while U1 and U5 were not activated, this meant that the robot slightly deviated to the right. We check U1 and U5 to ensure that we haven't gotten to the finish line. To fix this rightward deviation, we keep the left wheel moving at the same velocity, but move the right wheel faster by 2.5 in/s. This turns the robot slightly to become centered again on the line. Since we did not have access to U4, a similar case could not be implemented for left deviation. In order to check for this deviation, we decided to check if all sensors read black (as this essentially would be the same positional condition for U4 to activate if it were connected). This

worked really well, and the deviation fix was similar to the rightward case, where the left wheel was accelerated to 2.5 in/s faster than the right to coerce the robot back to center.

The next part of the state machine was a bit more complex, as it addressed the turning. By using the inbuilt encoders on the wheels, we could detect distance traveled. By sensing when U5 detected white (as the line for the turn is only present when this happens), we started a counter calculating distance traveled. Once this distance value became greater than the distance from the sensor to the wheels, the robot was prompted to turn by turning the left motor at 10 in/s and the right motor at -10 in/s for a smooth motion without activating any other states. The reason this distance was checked was because when the motors are moving in opposite directions, the entire robot rotates around an axis orthogonal to the motors' axis. Since the sensor starts the detection of needing to rotate, and the sensor is offset from the motors' axis, the additional distance needed to be traveled.

Controller Design:

For herding, this state activated when all sensors read true. For the herding process itself, a PID controller was utilized. From the activities in Lab 5, we determined that the best option for this was a PI controller, as it had low steady-state error, although it had some noise/overshoot during system identification. To find Kp, a P controller was implemented and run on both motors. A +/- 8V pulse signal was sent to the motors and the step response was captured, with static gain and time-constant being recorded for the motors. The average of these parameters was taken, and using the process from prelab 5, a Kp was determined. The model was then updated to a PI controller, where the Kp and Ki were also determined using the process from prelab 5. These controllers were compared, and although the PI had higher overshoot, it had a lower e_{ss}, so that is what we used.

Performance:

Following are some performance plots of the implemented PI controller.

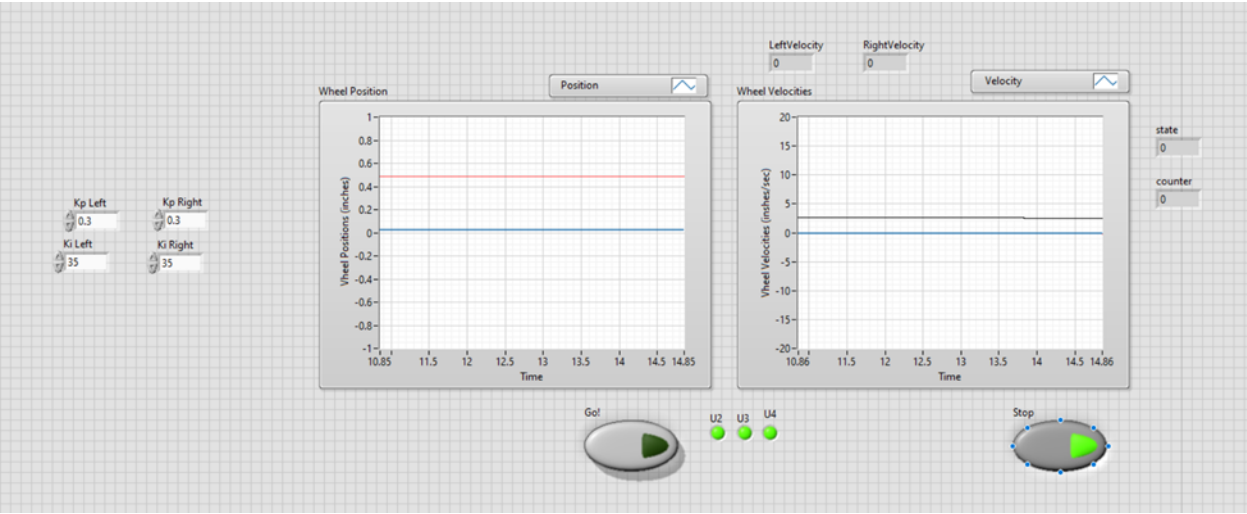


Figure 1 - Idling (State 0) Step Response

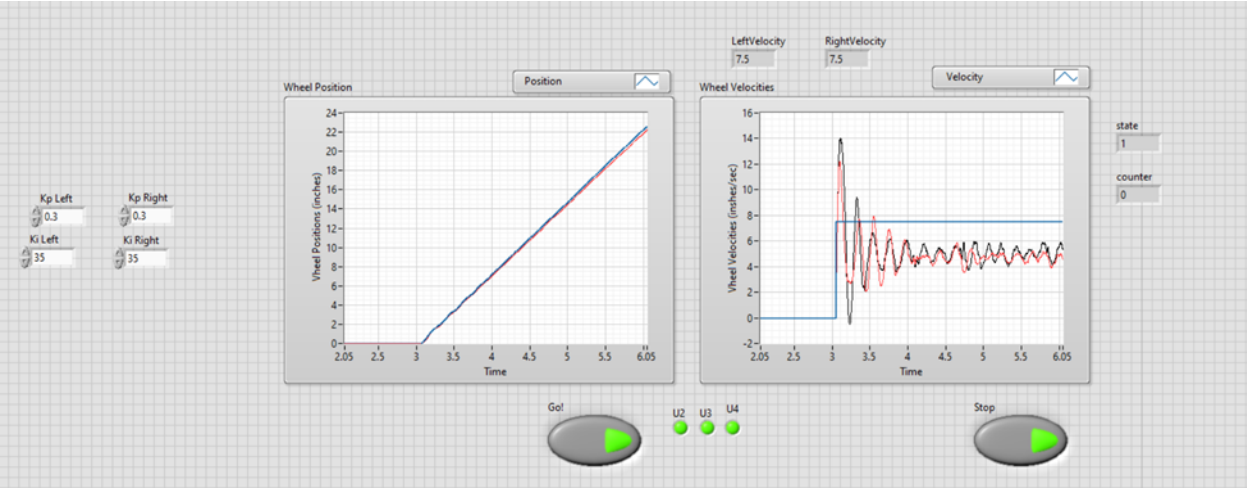


Figure 2 - Line Following, Forward (State 1) Step Response

The forward state of line following provided some interesting results. The positional steady state error and settling time are extremely low, with the desired and measured lines almost being collinear. However, with the velocity response, the overshoot is high, there is a lot of high frequency noise, and the steady-state error is high as well, around 2.5 in/s. This is possibly due to us not implementing friction compensation in the controller design.

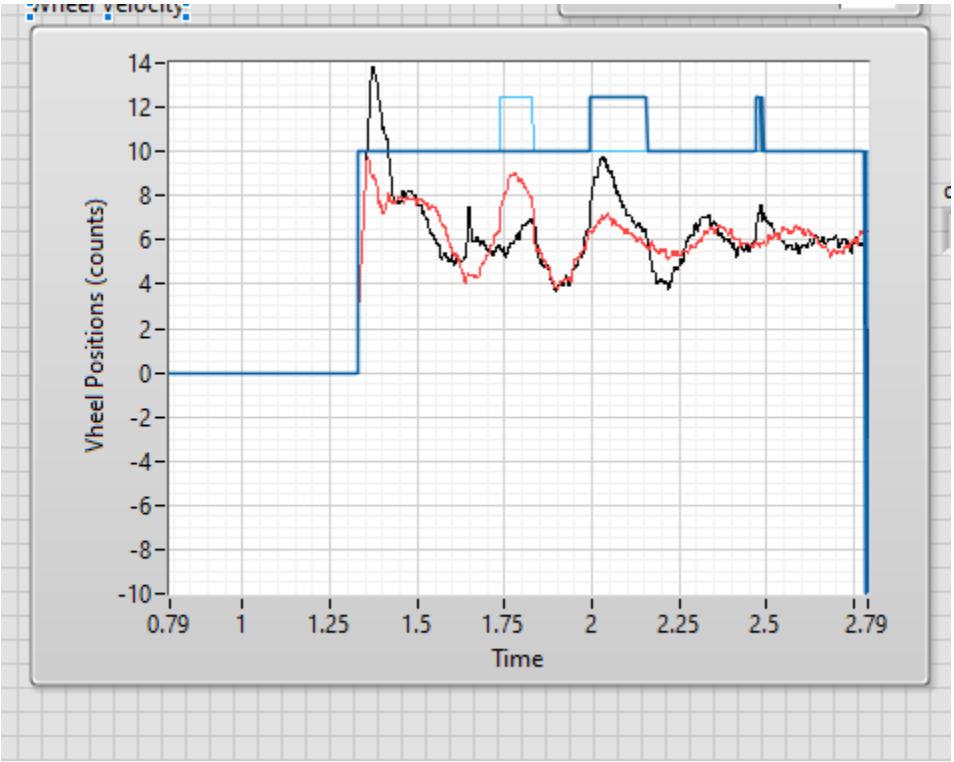


Figure 3 - Line Following, General, Step Response

The general line following and right turn states (Figures 3 and 4) both exhibit the same behavior as the forward state, where noise and steady state error is high, again possibly due to lack of friction compensation. This error seems to remain constant, even in the state changes. In Figure 3, at 1.75s, the state changes to left adjustment (state 3), and the response line moves accordingly. At 2s, the state changes to right adjustment (state 2), and the right wheel response line follows as well, with the same steady state error offset throughout the entire operation. An additional right adjustment happens just before 2.5s.

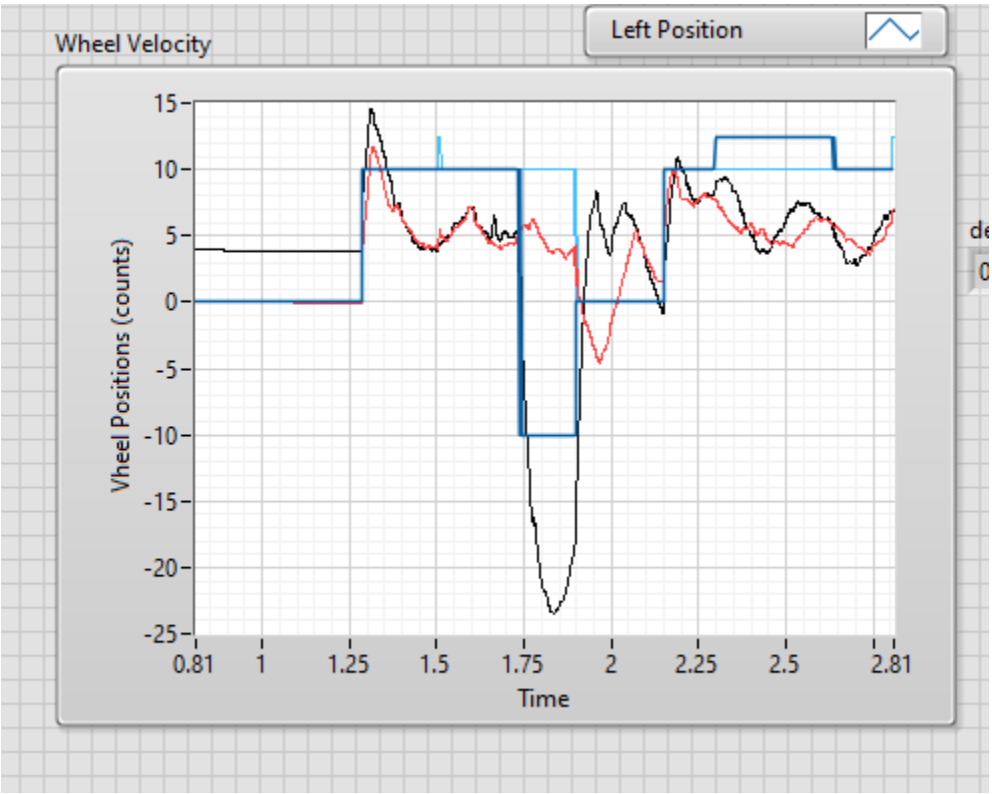


Figure 4 - Right Turn Step Response

Code Description:

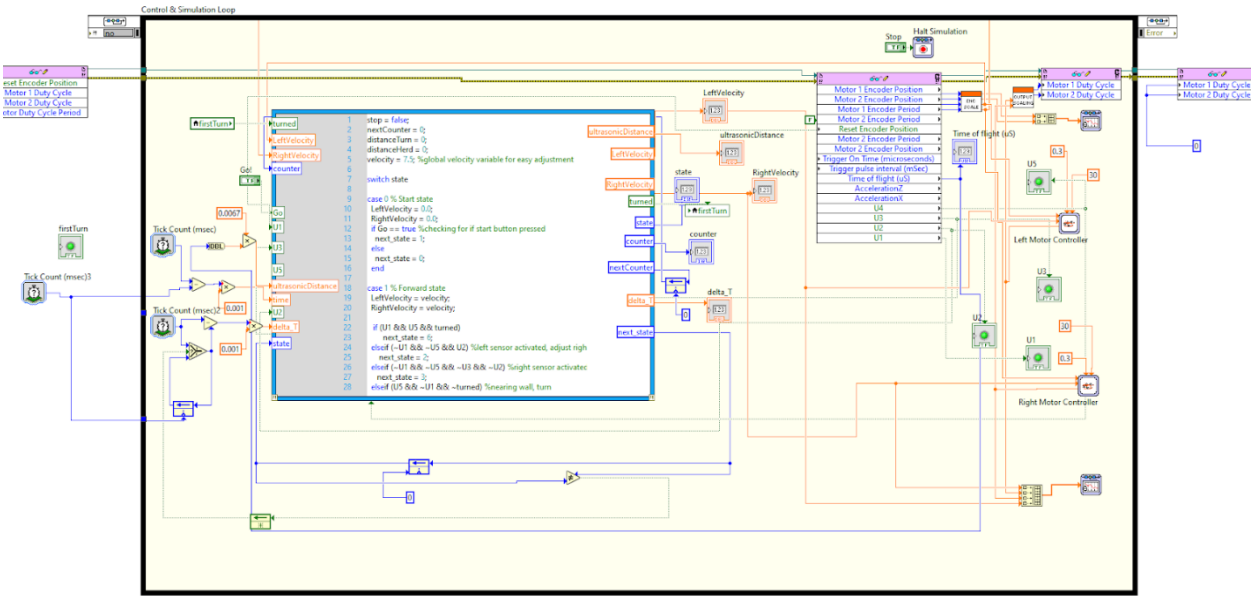


Figure 5 - Code Overview

Figure 5 shows the layout of the LABVIEW code that was used to run the robot. The pink block outside of the control loop loads the FPGA file, which is a pre-downloaded bitfile from brightspace. While we did look into compiling our own bitfile in order to get access to all 5 sensors, this approach did not end up working, so we just ended up using the default sheepdog bitfile. Inside of the control loop, to the left of the mathscript node, there are a couple of blocks that calculate time elapsed, however, this portion of the code was not written by me. There are various signals read from the sensors, mainly the U1, U2, U3, and U5 digital signals, which are fed in as inputs into the mathscript node for use within the state machine. For analog inputs, we feed in time, ultrasonicDistance, RightVelocity, LeftVelocity, and delta_T, which is used to measure time elapsed between state changes. For integer inputs, we feed in state, which is updated using a feedback loop within the mathscript.

The mathscript node is the state machine. It governs the behavior the robot takes under certain conditions. Figure 6 shows the initialization of variables and starting state - a state used to detect when the Go button is pressed to actually start running the robot.

```
stop = false;
nextCounter = 0;
distanceTurn = 0;
distanceHerd = 0;
velocity = 7.5; %global velocity variable for easy adjustment

switch state

case 0 % Start state
 LeftVelocity = 0.0;
 RightVelocity = 0.0;
 if Go == true %checking for if start button pressed
  next_state = 1;
 else
  next_state = 0;
 end
```

Figure 6 - Variable Initialization and Starting, State 0

Once Go is pressed, the robot transitions to state 1 (Figure 7), which is the forward line following state. It enters this condition only when just U3 is activated. Within state 1 (and the subsequent other states used in pure line following), there are conditional statements to check for if the state needs to change based on sensor readings, which were mentioned in the state machine design section of this report.

```
case 1 % Forward state
 LeftVelocity = velocity;
 RightVelocity = velocity;

  if (U1 && U5 && turned) %reached finish, enter herding
      next_state = 6;
 elseif (~U1 && ~U5 && U2) %left sensor activated, adjust right
    next_state = 2;
 elseif (~U1 && ~U5 && ~U3 && ~U2) %right sensor activated, adjust left
   next_state = 3;
 elseif (U5 && ~U1 && ~turned) %turn 90 degrees
    next_state = 4;
 elseif (~U1 && ~U2 && U3 && ~U5) %center sensor on only, keep straight
    next_state = 1;
 else %used to avoid entering state 0 again
   next_state = 1;
 end;
```

Figure 7 - Forward, State 1

States 2 and 3 (Figures 8 and 9) are the right- and left-deviation correction states that slightly speed up their respective motors to maintain the robot following the line. The same conditional tree is included to ensure these states are entered only when needed.

```
case 2 %U2 activated, adjust right
  LeftVelocity = velocity;
  RightVelocity = velocity + 2.5;

  if (U1 && U5 && turned)
      next_state = 6;
 elseif (~U1 && ~U5 && U2) %left sensor activated, adjust right
    next_state = 2;
 elseif (~U1 && ~U5 && ~U3 && ~U2) %right sensor activated, adjust left
   next_state = 3;
 elseif (U5 && ~U1 && ~turned) %nearing wall, turn
    next_state = 4;
 elseif (U1 && U2 && ~U5)
    next_state = 6;
 elseif (~U1 && ~U2 && U3 && ~U5)
    next_state = 1;
 else
   next_state = 1;
 end;
```

Figure 8 - Right Adjustment, State 2

```
case 3 %U4 activated, adjust left
  LeftVelocity = velocity + 2.5;
  RightVelocity = velocity;

  if (U1 && U5 && turned)
      next_state = 6;
 elseif (~U1 && ~U5 && U2) %left sensor activated, adjust right
    next_state = 2;
 elseif (~U1 && ~U5 && ~U3 && ~U2) %right sensor activated, adjust left
   next_state = 3;
 elseif (U5 && ~U1 && ~turned) %nearing wall, turn
    next_state = 4;
 elseif (U1 && U2 && ~U5)
    next_state = 6;
 elseif (~U1 && ~U2 && U3 && ~U5)
    next_state = 1;
 else
   next_state = 1;
 end;
```

Figure 9 - Left Adjustment, State 3

States 4 and 5 (Figure 10) are entered when the line turns horizontal, and a left or right 90 degree turn must be conducted. These states solely measure distance to ensure the motor axis lies above the line turn rather than the sensors. Once this distance is achieved, the robot transitions to state 7 (left turn) or state 8 (right turn) (Figure 11). While the project did not call for a left turn, the competition did, and we wrote this code as a baseline for our competition robot - therefore case 5 was not used for the scope of the project.

```
case 4 %U5 activated, measure distance
 RightVelocity = velocity;
 LeftVelocity = velocity;
 distanceTurn = distanceTurn + min(RightVelocity, LeftVelocity) * delta_T;
 turned = true;

 if (distanceTurn < 3) %turned too far, leave loop
    next_state = 4;
 else
     distanceTurn = 0;
     nextCounter = 1;
     next_state = 8;
 end

case 5 %U1 activated, measure distance
 RightVelocity = velocity;
 LeftVelocity = velocity;
 distanceTurn = distanceTurn + min(RightVelocity, LeftVelocity) * delta_T;

 if (distanceTurn < 3) %turned too far, leave loop
    next_state = 6;
 else
     distanceTurn = 0;
     next_state = 7;
  end
```

Figure 10 - Start Measuring Distance from Line Turn, States 4 and 5

```
case 7 %turn left
  RightVelocity = 10;
  LeftVelocity = -10;

  if (~U3)
    next_state = 7;
  else
    next_state = 9;
  end

case 8 %turn right
  RightVelocity = -10;
  LeftVelocity = 10;

  if (~U3)
    next_state = 8;
  else
    next_state = 9;
  end
```

Figure 11 - Conduct the Correct Turn, States 7 and 8

After each turn, state 9 is entered, which is a brief pause state to allow both motors to achieve the same speed before starting line following again. This is important, as if we were to transition directly back to state 1, one wheel would need to decelerate 2.5 in/s to get to the target velocity, while the other would need to accelerate 17.5 in/s. This acceleration disparity was large enough

where it was causing some very fast and large deviations away from the line, so the pause state pushes both motor velocities to 0 in/s for 0.25s before proceeding (Figure 12).

```
case 9 %stop after turn
  RightVelocity = 0;
  LeftVelocity = 0;
  if(delta_T < 0.25)
    next_state = 9;
  else
    next_state = 1;
  end
end;
```

Figure 11 - Pause, State 9

Finally, state 6 (Figure 12) shows the herding conditional tree. This state does not allow exit, as for the project, it occurs at the end of the track. If the object is detected to be less than 8.5 in away from the robot, this means that we need to move backwards, so we either move -5 in/s or the difference between the sensor and the object in in/s, whichever is closer to 0 (this is necessary to limit the velocity of the robot in case very large blips occur due to sensor inaccuracy). A similar case condition is set up for if the object is farther than 9.5 in away from the robot. The 0.5in buffer in either direction from 9in was arbitrarily chosen, however it was necessary to ensure relative stability while stationary.

```
case 6 %herding
  next_state = 6;
  if (ultrasonicDistance <= 8.5)
    LeftVelocity =  max((ultrasonicDistance - 9) , -5);
    RightVelocity = LeftVelocity;
  elseif (ultrasonicDistance >=  9.5)
   LeftVelocity =  min((ultrasonicDistance - 9) , 5);
    RightVelocity = LeftVelocity;
  else
    LeftVelocity = 0.0;
    RightVelocity = 0.0;
  end;
```

Figure 12 - Herding, State 6

This is the extent of the state machine. As far as actually controlling the motors, that is when the PI controller from before is used, as seen in Figure 13.

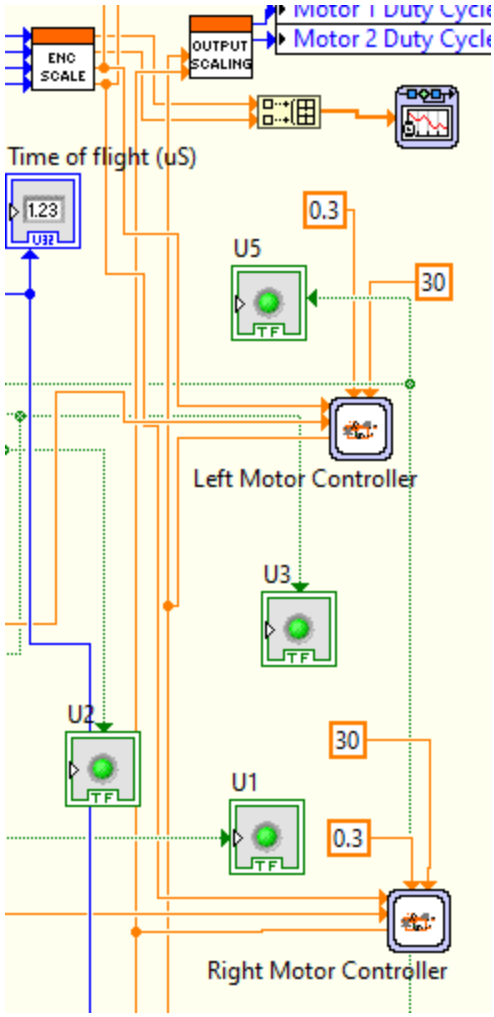


Figure 13 - Controller Implementation

3. Discussion

Many issues were encountered and solved in clever ways, some of which were referenced above. The main problem was the line-follower sensor was not very accurate and gave a lot of false positives. This was avoided by adding extra conditions as redundancy within our state machine. Additionally, the special turning method of turning motors opposite directions was created as a result of noisy sensor readings during traditional turning methods tested earlier in the project.

Discussion of robot performance of herding was addressed earlier in the report, yet certain improvements can be made. Friction compensation can be implemented, as well as derivative control implemented to reduce noise and overshoot. Further system ID could also be implemented to select more optimal gain values.

PROJECT 5: Automatic Bicycle Brake Light

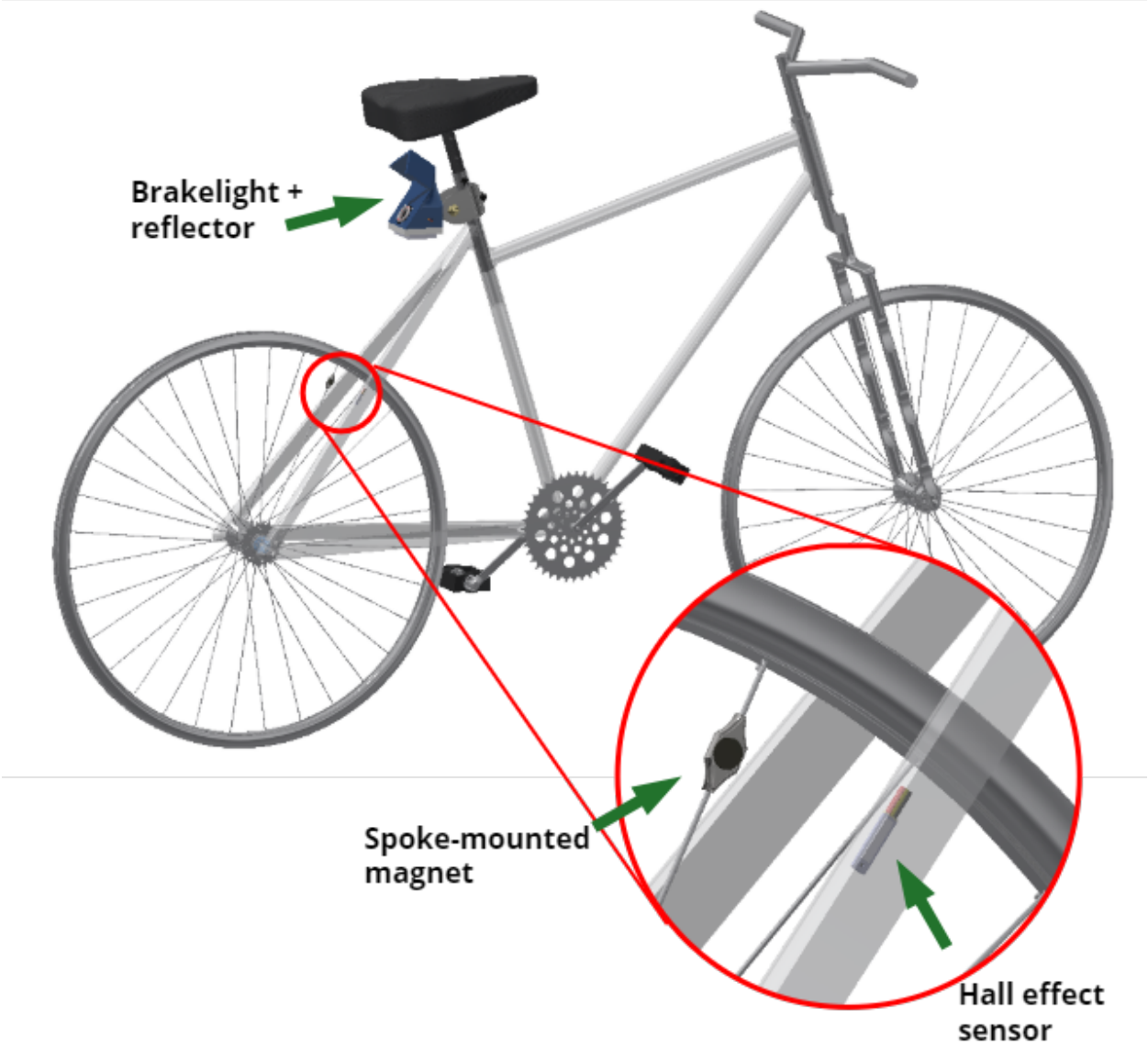


Figure 1 – Automatic Brake Light Mounted on Bicycle

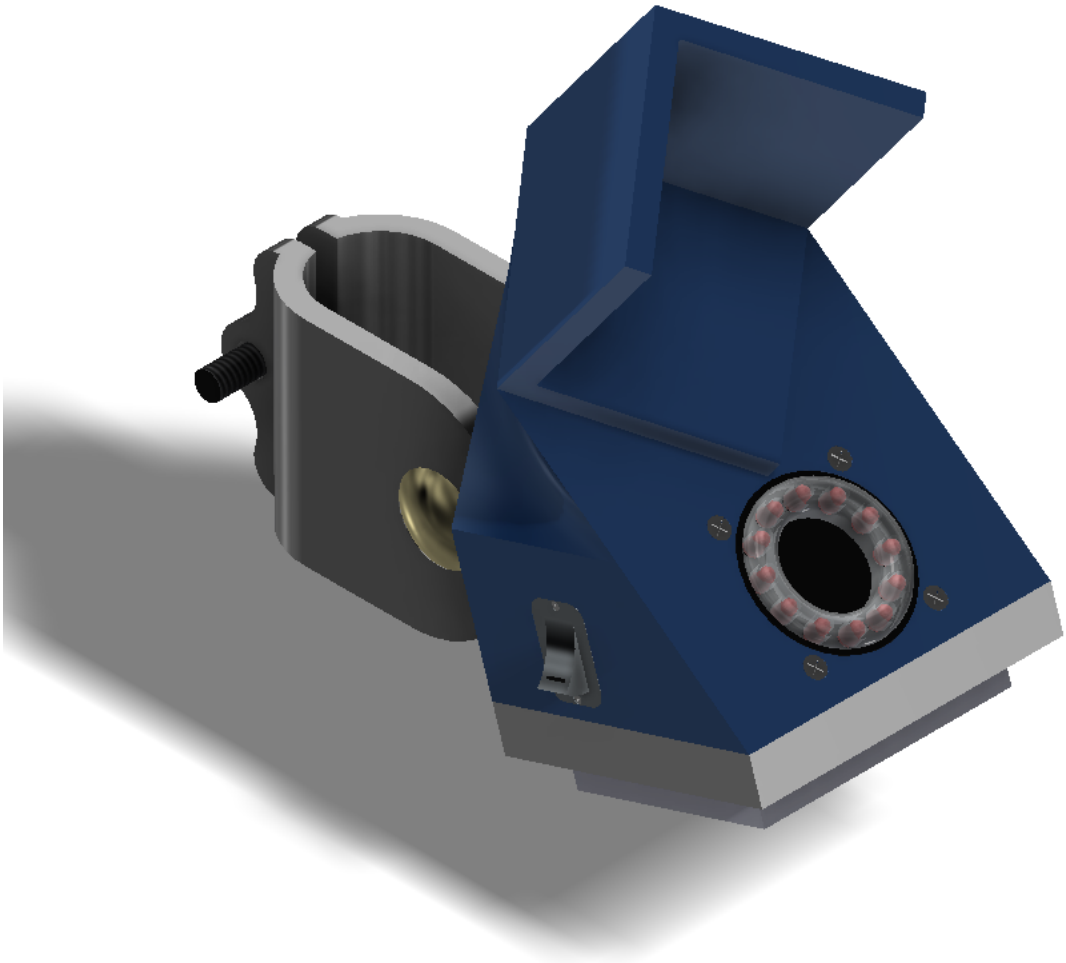


Figure 2 – Brake Light Isolated

This is an automatically activated brake light for a bike that I made for ME 263, the Purdue sophomore design course for mechanical engineering majors. The basic idea is that a magnet is mounted onto the spokes of the bike wheel, and a hall effect sensor on the frame detects when the magnets pass by, indicating a full rotation. This allows the internal microcontroller to calculate RPM, and when the RPM value becomes low enough, the brake lights activate. Additionally, the unique geometry is a corner reflector, used to make the bike more visible to cars with radar capabilities. Here is a video demonstrating the lights activating (made with a 3D printed model): https://www.youtube.com/watch?v=SNwyz9qo01U.
I made the bike CAD model completely from scratch as well and challenged myself to make it as detailed as possible. Every component down to the bearings in the wheels were made from scratch.

Figure 3 – Full Bike Assembly

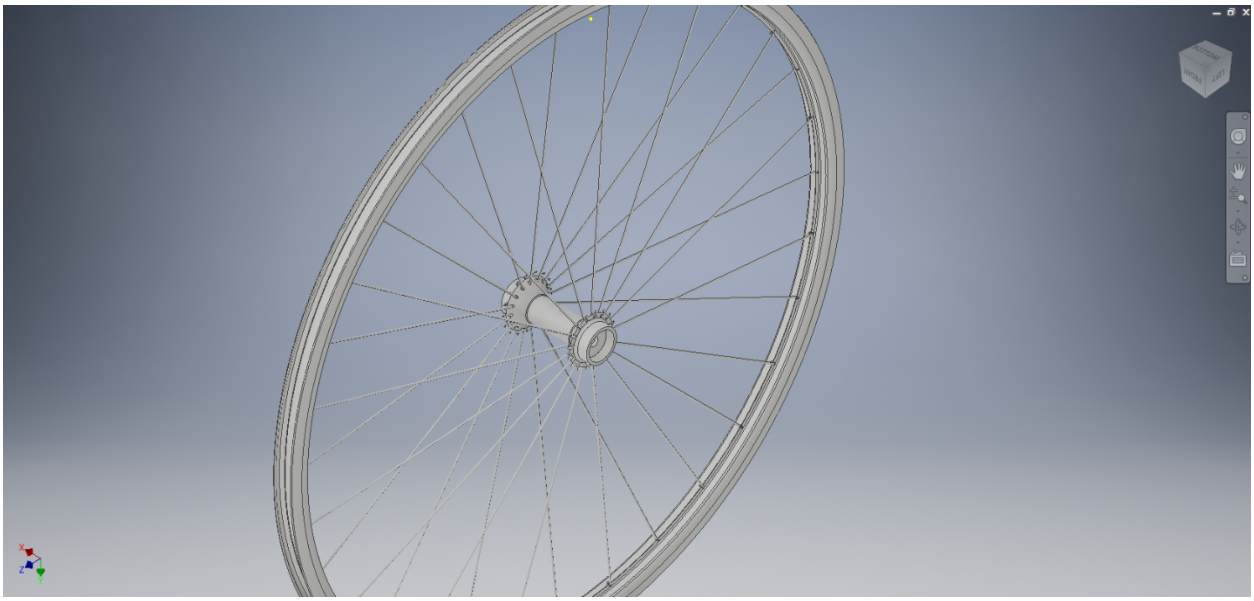


Figure 4 – Bike Wheel

The bike wheel was interesting due to the spokes being offset from each other, so they were not symmetrical along the center plane.

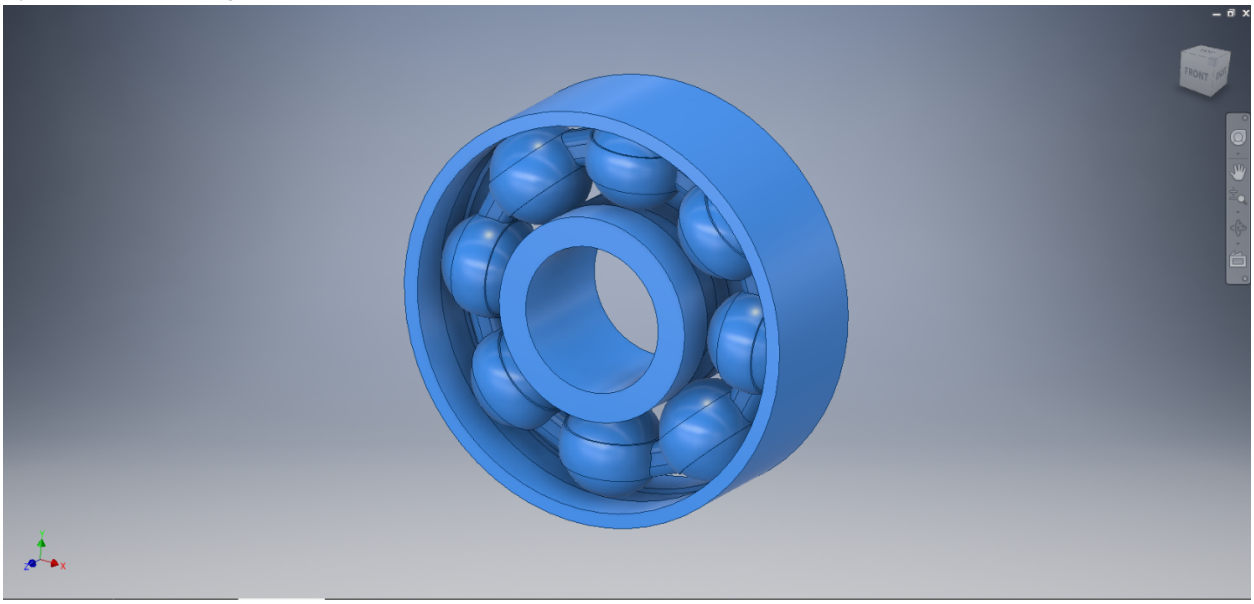


Figure 5 – Wheel Bearing

The bearing required some very precise measurements and some complex math for proper fitting of the components.

Figure 6 – Pedal Assembly

Here is the bike pedal I designed. I made a custom attachment for it to interface with the crankshaft without unscrewing itself during operation.

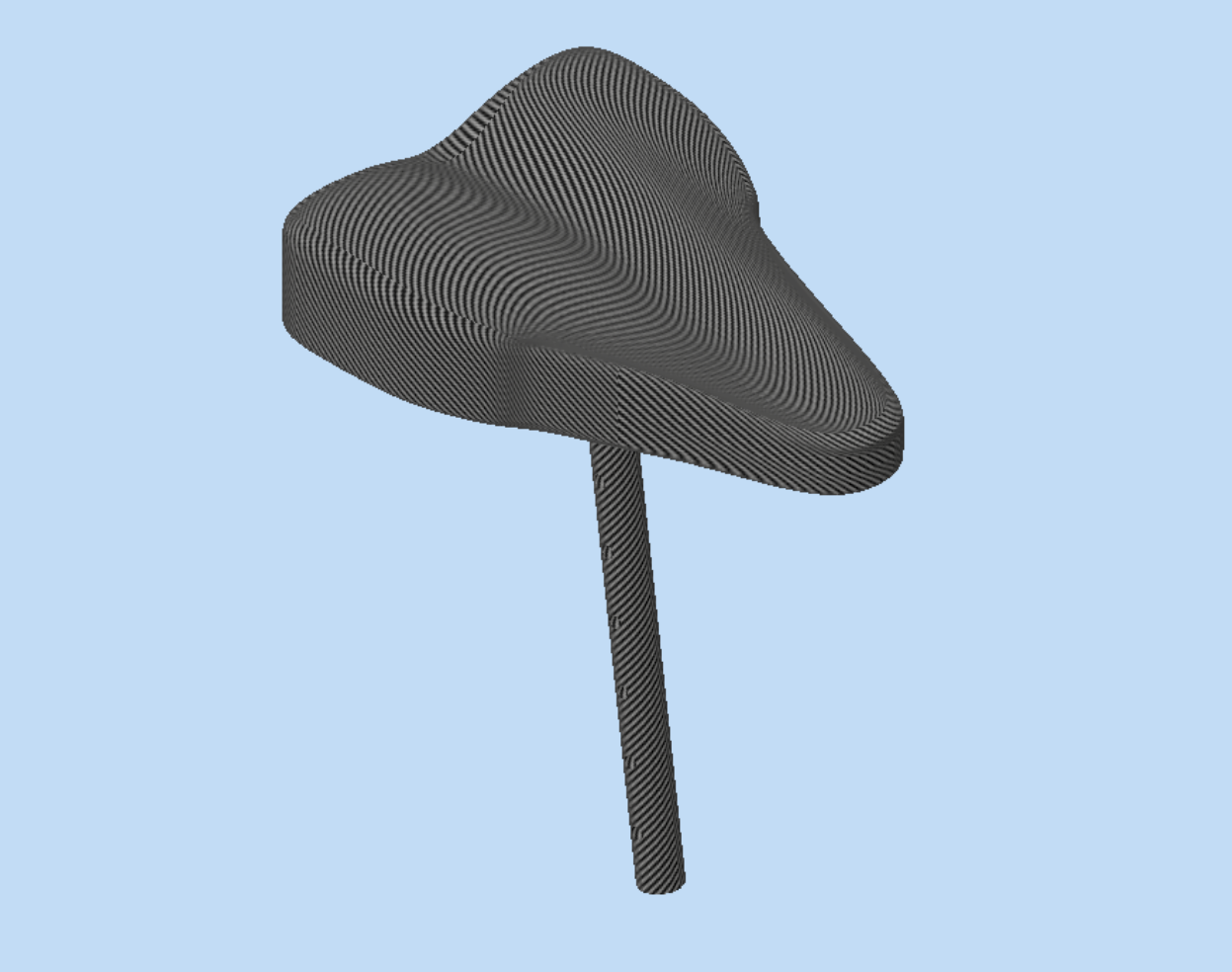


Figure 7 – Bike Seat

This is the seat for the bike. I spent a lot of time using the freeform tools to match the complex cushion shape to that of a real-life bike.

PROJECT 6: Automatic Winding Mechanism for Weight-Driven Mechanical Clock

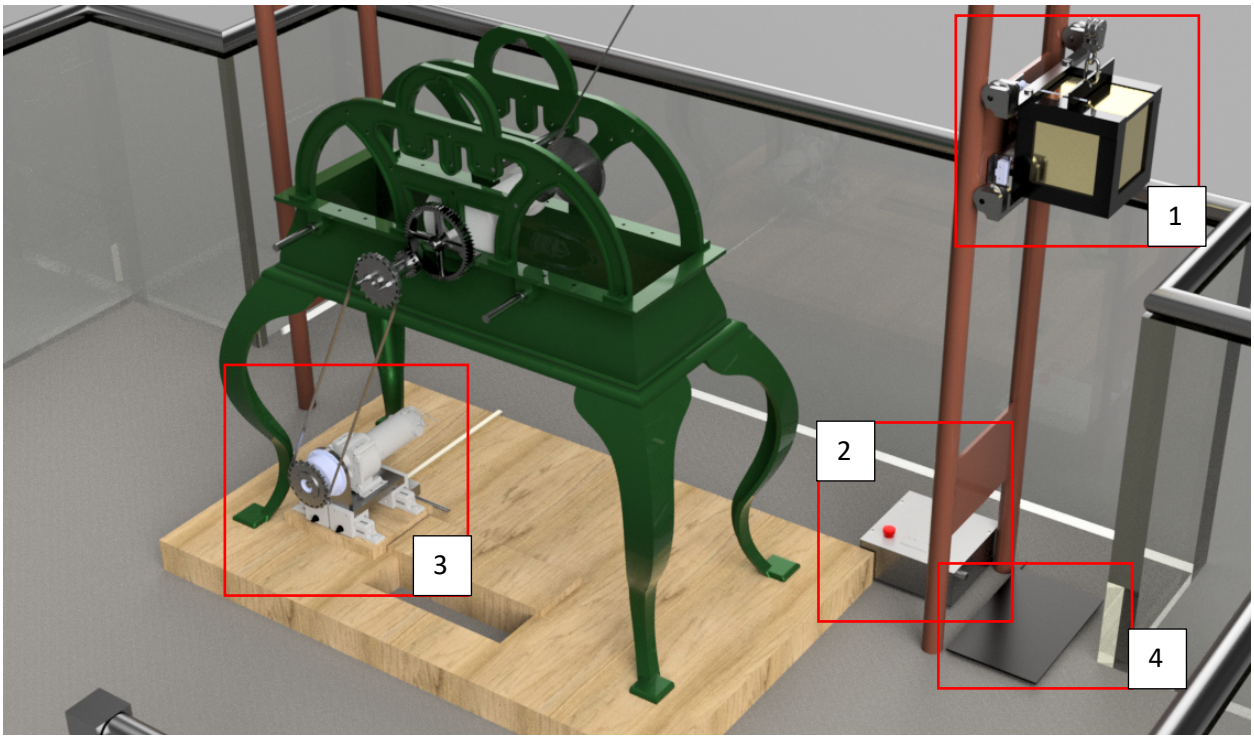


Figure 1 – Clock Mechanism Assembly in Fusion 360

This is an autonomous winding system I made for a weight-driven clock in the Purdue Mechanical Engineering Building. Currently, the clock has to be hand-cranked around every 3 days to lift the weighted box (Item 1) back up to the top. The autonomous mechanism works as follows: when the box is at the top, a set of limit switches is depressed. As the weight slowly travels down, the limit switches are disengaged, and an internal timer starts on the Arduino inside of the electronics box (Item 2). At the bottom of the box's travel, another set of limit switches is depressed, which engages the motor winding system (Item 3) to crank the box back up to the top, resetting the timer. If the limit switches fail, or the timer is not reset, a pressure plate is placed at the floor as a redundancy measure to ensure that the clock does not stop and lose its time. If a power outage were to occur, a backup battery was in place to also serve as redundancy in order for the clock to keep track of time. The CAD in Figure 1 was made entirely from hand measurements, as no documented drawings of the clock existed. This was a summer internship, and on top of the CAD model, I created 2 reports, 2 presentations, and a full animation of the clock movement. See https://www.dropbox.com/s/h75ob3x5iap2dts/Clock%20Animation.mp4?dl=0 for the full animation.

PROJECTS 7 & 8: Web Development (Terabit Design LLC and htmlhandbook.com)

I am the head of business operations and frontend developer at Terabit Design LLC. I started Terabit Design in August 2020 with my friend in order to provide small businesses with a more affordable option for fully customizable websites in order to expand their online presence. Current options such as Squarespace, Wordpress, and Wix do not offer the tools to fully express one's creativity, and demand quite a high monthly cost of their customers. So far, I have created the frontend for 3 different clients (https://flyclarity.com, https://simplypre.com/pages/preworkout-quiz, and https://point-and-figure.web.app/). The websites were all built using HTML, CSS, and JavaScript using the Vue.js framework. flyclarity.com was a particularly long project, as the client's needs changed drastically throughout the course of development. So far, Terabit Design LLC has generated over $10,000 in revenue from our work.

https://htmlhandbook.com is a website I co-developed to showcase different common elements found on webpages. The designs are made to be open-source and the HTML, CSS, and JavaScript used to create them are easily copy-and-pasteable for use in anyone's own personal website. I used the development of this website as a way to learn more about CSS while also having fun creating new ideas for web elements. This project heavily improved my web development skills, and prepared me for the types of problems I would later solve programming for Terabit Design.