ECE 556 Term Project Report
Team 7

Introduction

The need for developing a control system is ubiquitous with its application varying from domestic appliances to fully autonomous cars and robots.The Path tracking ability of an autonomous vehicle is one of the most challenging tasks for developers due to its constraints on mobility, speed and internal mechanisms like battery. The ideal control system is enabled with cheap sensors and capacity to handle large data in real time.

The project demonstrates the use of a well tuned closed loop controller to develop a Lego EV3 “Bot” Car to perform certain tasks. Sensors are the heart of any intelligent system, in this project we are using two types of sensors. i. Ultrasonic Sensor (HC SR04), ii. Light Sensors (EE-SF5B).

These sensors capture data and are most of the time compared to a reference value to compute the error. Depending on the error the closed loop controller tries to reduce it and hence, does a particular action to do so. The Ultrasonic sensor has a distance range from 2cm to 400 cm and is used to measure the distance between the bot and an object. Whereas, the photodiode gives us a varying reading for different colors.

The Lego EV3 is a great tool to teach the fundamentals of control systems with practical examples. There are two large motors with a built-in rotation sensor with 1-degree resolution for precise control.The control is developed in the Lego Mindstorm app, and the data is analysed using MATLAB.

The project compares the use between four different controllers P, PI, PD, PID and develops an algorithm that inculcates the most ideal controller for the given sensor placement. The controller tuning is the most essential part of this project and we used the Zeigler Nichols method to tune our controller.

Project Setup

The model makes use of the following setup:
i. Lego EV3 Brick (Microcontroller) with 2050 mAh DC battery.
ii. Lego EV3 large motors:

Max RPM	160-170
Running torque	20 N.cm
Stall torque	40 N. cm

iii. Ultrasonic Sensor: HC SR04

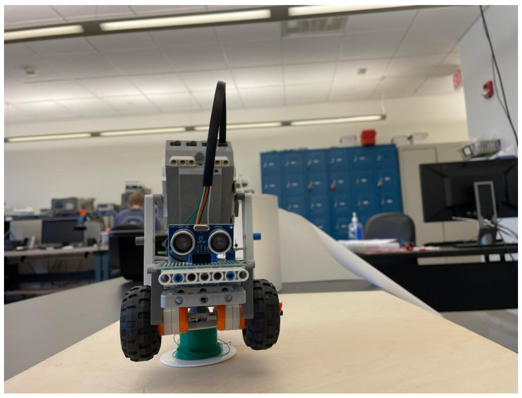
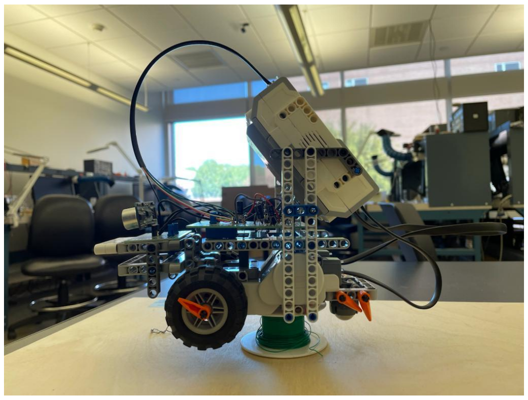
Dimension	45-20-15 mm
Working voltage (DC)	5 V
Range (cm)	2 - 400
Measuring angle	15°

iv. Light Sensor: EE-SF5B

Operating Voltage (DC)	5 V
Tolerance (3mm - 6mm)	± 0.375

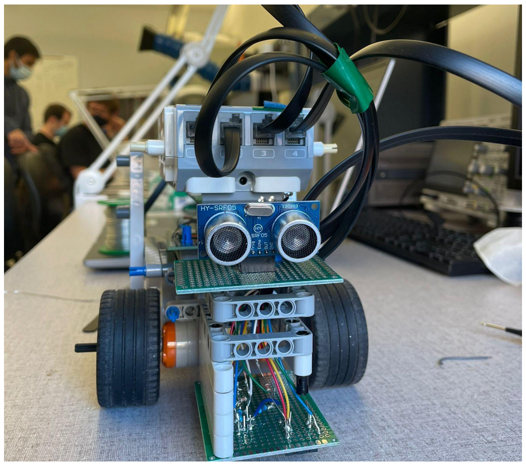
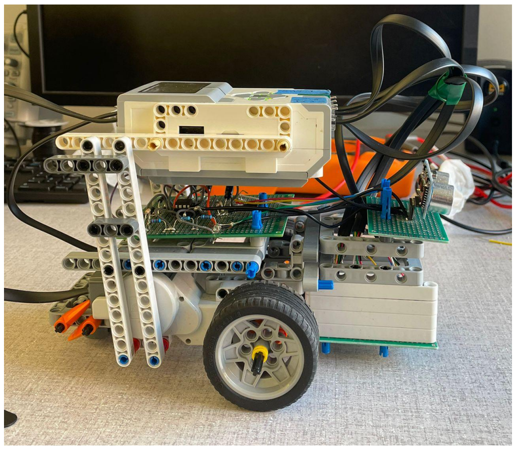
v. Bot Setup:

- Initial Setup (Milestone 1): It consisted of an ultrasonic sensor fixed at the center of the bot. This setup worked well for milestone 1 as the bot was not moving very fast.

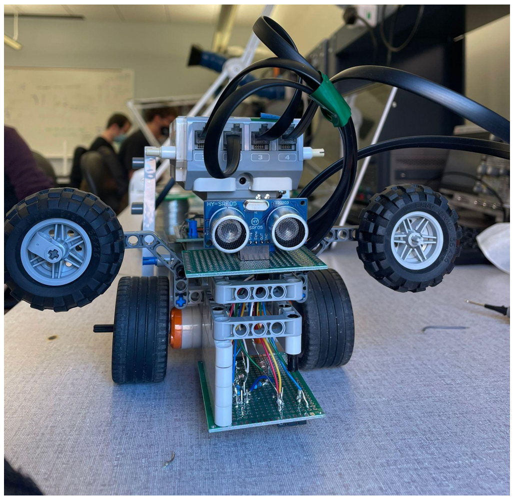
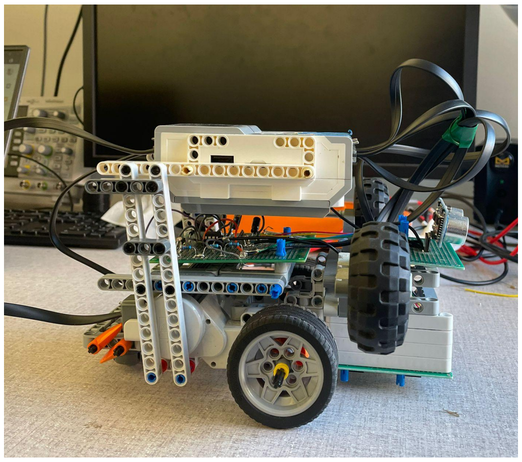



- Improved Setup (Milestone 2):

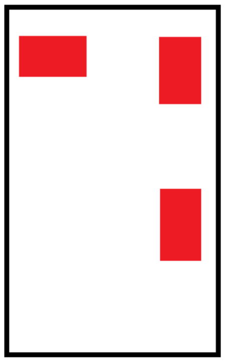
Problems with initial setup: During trails for milestone 2 we noticed that the wheels had been slipping so we changed it to the flat wheels. And we also noticed that all the weight due to the brick was on the back of the bot. So we changed the position of the brick. The light sensor is placed 7mm from the ground.

- Final Setup (Milestone 3):
  During trials for milestone three when the distance and speed of the bot was even larger we noticed that the bot was slipping due to lack of weight on the wheels so we added extra weight.

vi. Light Sensor placement:



The three light sensors are placed in the following way. The front-right and back-right sensors are 2 cm apart and the front-right and the front middle sensors are placed around 1 cm apart.

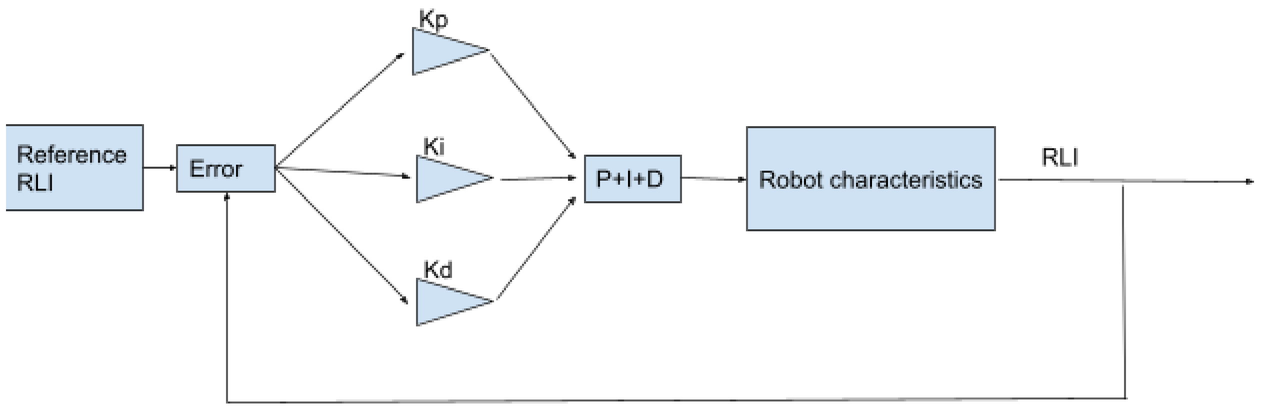
Algorithm design

i. Milestone 1

- ***Goal***: The goal of milestone 1 was to get the bot to a stationary object and park the bot as close as possible to the stationary object.
- ***Sensors***: For this task the only sensor we used was the SRF04 (ultrasonic sensor).
- ***Algorithm***: When the task is started, the robot is set to start moving straight at a speed of 10. The bot continues to do this until the ultrasonic sensor reads a raw value of 44, which translates to about 2 cm. Once this value is set, the bot starts moving at a lower speed of 3. The bot continues to do this until the ultrasonic fails (starts reading a raw value more than 55). When the ultrasonic sensor fails the program ends
- ***What is making the sensor fail***:: The SRF04 ultrasonic sensor, is able to reliably detect distances from 2 to 400 cm. When in this ideal range, the raw values read by the sensor range from 44 to 500. If the distance from the ultrasonic sensor drops below 2cm the sensor “fails" (the sensor starts to read raw values larger than 100). The way the sensor works is that it sends out an 8 cycle sonic burst. It then waits to hear this reflected signal and calculates the distance based on the time taken for the reflected signal to be detected.. One important point to note about this is that when the ultrasonic sensor is sending this 8 cycle sonic burst, it disables its receiver. This is due to the fact that if the receiver is left enabled, it will detect the 8 cycle sonic burst as the reflected signal and give incorrect data. This disabling of the receiver is what causes the sensor to not be able to detect less than 2 cm values and fail. If an object is placed less than 2 cm from the sensor the time taken for the reflected signal to arrive at the receiver will be less than the time taken to transmit an 8 cycle sonic burst. Since the receiver is disabled during this time duration, the sensor will “fail” to detect 2cm and read incorrect values.

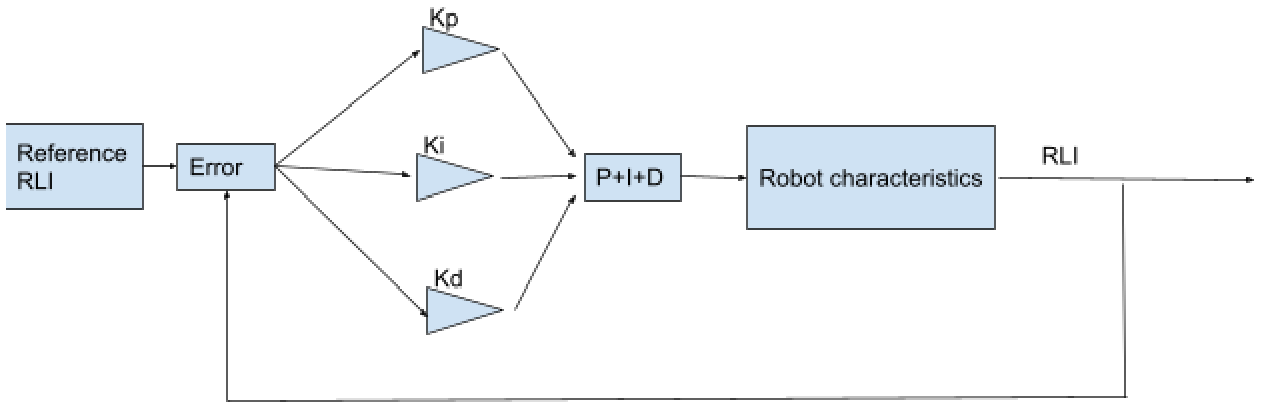
ii. Milestone 2

- ***Goal***: The goal of milestone 2 was to get the bot to follow a moving TA bot, and maintain a specific distance to the TA bot. The better the bot was able to maintain the specified distance to the TA bot (platoon), the better its performance would be. Once the bot reaches a red line, it is supposed to stop.
- ***Sensors***: For this task we used three sensors. 1 SRF04 (ultrasonic sensor) and two light sensors. The SRF04 is used to maintain the specified distance from the TA bot. One of the light sensors was used to detect the red line, and the other sensor was used to follow the black line, so that the bot would stay straight.
- ***Algorithm***: When the task is started, the robot is set to sample its current distance from the TA bot. This is the distance that will be used as the reference distance from the TA bot. Once this sample is taken, our PID controller starts to run. Using this PID controller we are able to maintain the reference distance from the TA bot as it moves. While the bot is following the TA bot, we also have a P controller gathering data from one of the light sensors to make sure it is accurately following the straight black line. Once the second light sensor detects a redline the program stops.
- ***How to detect the red line***: The readings from the light sensors we were reading were raw values. These values range from about 80 for black all the way to 500 for white. Raw values in between this range are used to detect different colors. To find out the raw value for the red line, we wrote a sample program which would display the raw values from the light sensor onto the EV3 display. Using this we were able to get values for the red line. The raw value we determined that signaled a red line was 310.
- ***PID code in Mindstorm***: To write the PID code we used the mindstorms software. In a PID there are 3 main parts. Proportional, Integral, and Derivative. All three of these values are dependent on the error. The error is the difference between the offset value (the average between raw value for white and raw value for black) and the current RLI reading from the light sensor. For proportionality the error is simply multiplied by the proportional gain. For Integral we have a variable called error accum. This variable stores the integral accumulation. This accumulated value adds to itself the current error value and continues to accumulate. This value is multiplied by the integral gain. The derivative part requires the difference in the current error and the previous error. This difference is multiplied by the derivative gain. At the end, all three parts, P, I, D are added up to get the controller output.

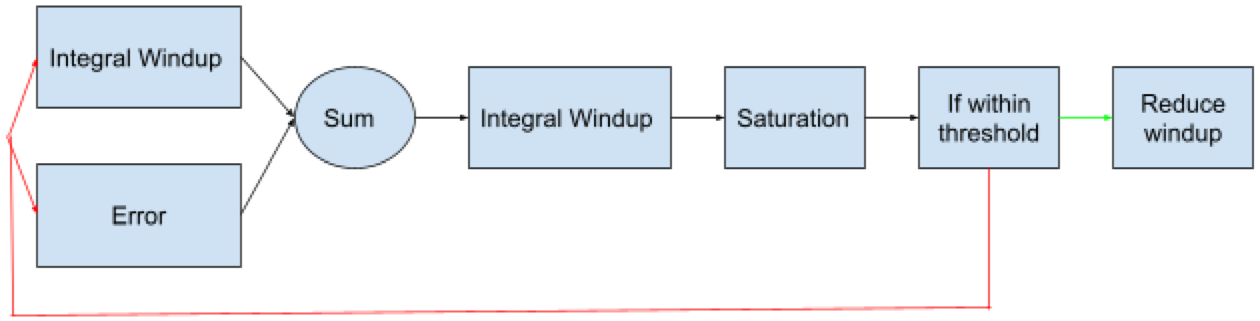


iii. Milestone 3

- ***Goal***: The goal of milestone 3 was to get the bot to follow a black line, and stop really close to the wall.The PID was tuned so that the bot is moving at its potentially max speed and it is "smooth". Once the bot reaches the wall it has to stop as close to the wall as possible.
- ***Sensors:*** For this task we used the front-right sensor to compute the error with the middle of the black and white line. For example when the sensor is in the middle of the black and white line the reading is 280, so that is our offset and we compute our error based on that.
- ***Algorithm***: The raw values from the light sensor is calculated and subtracted from the offset value to compute the error. This is fed to the PID controller, to keep the bot from moving along the black line. We introduced a logic to eliminate integral windup because our bot wasn't able to take sharp turns.
- ***PID code in Mindstorm***: The implementation of the PID controller is very similar to the implementation done in milestone 2. There were 2 main differences with the implementation for milestone 3. The first change is that for the integral portion, we implemented integral windup, and also implemented integral windup saturation. The second change we did was that we simulated having 2 sensors. This was done by multiplying the error by 2.
- ***Why saturation***: Saturation was added because our bot was taking sharp curves but then overshooting the line. This was due to the fact that we are dividing our saturation value to bring it close to zero. If the value is too large, the division is not enough to bring the value back close to 0.
- ***Why multiplication by 2***: Multiplying the error by 2 simulates 2 sensors. When two sensors are used, one on each edge of the black line, when one sensor moves more on to the white, the other sensor moves more onto the black. Because of this the error increases twofold. We simulated this by multiplying the error by 2.



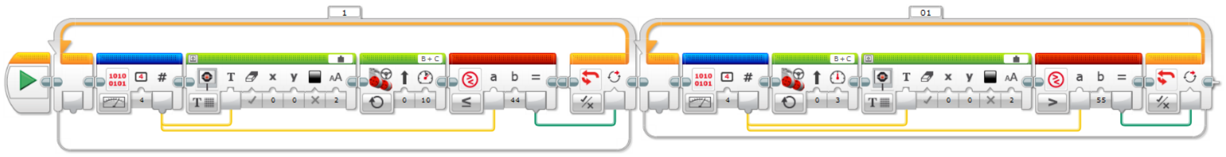
The controller diagram for Milestone 3 is the same as for milestone 2. The change is in the Robot Characteristics block



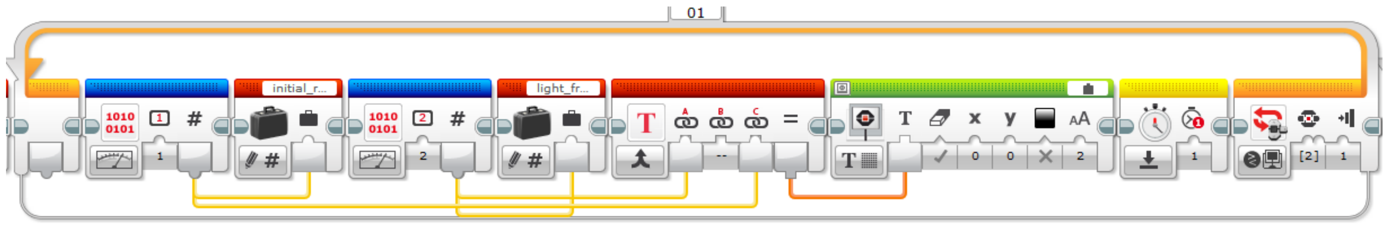
This diagram shows the addition of integral windup, and how it is manipulated in our control algorithm.

Algorithm development

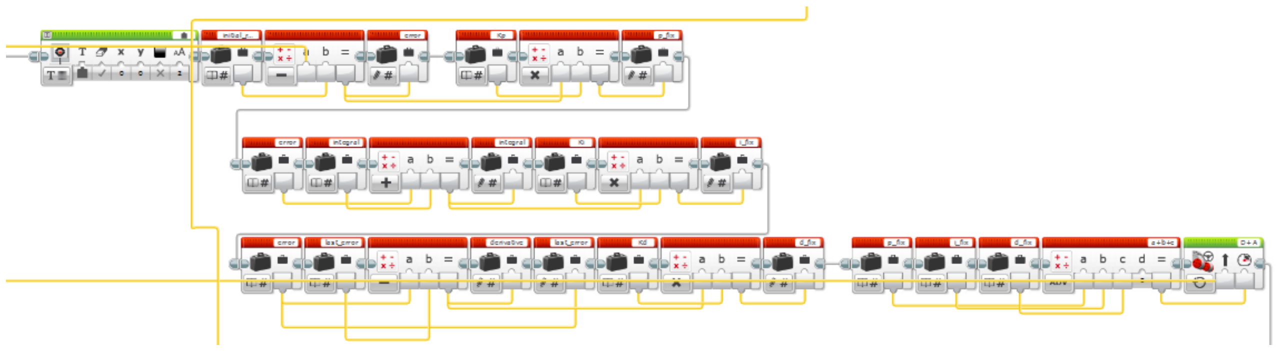
i. Milestone 1: In this code the first loop makes the bot travel straight till it reaches a certain threshold and then moves on to the next loop. There we decrease the speed of the bot even more and slowly move towards the threshold. The loop breaks and the bot stops once the value is greater than the threshold, i.e when the sensor “fails”.



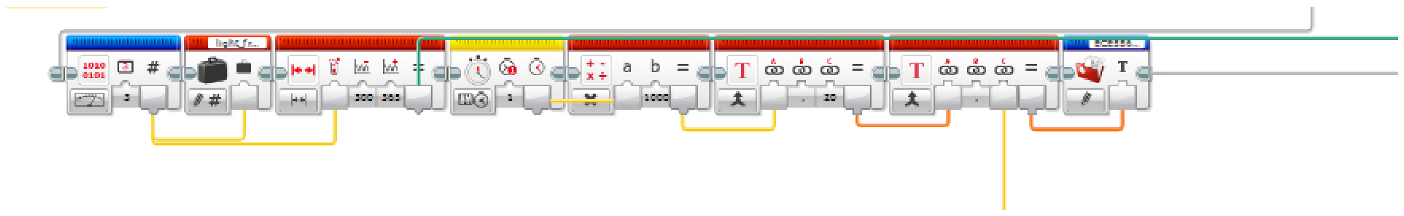
ii. Milestone 2:
In this loop we set the offset for the ultrasonic sensor. During the demo the TA bot is placed 20cms from our bot. So this value is not being hardcoded.



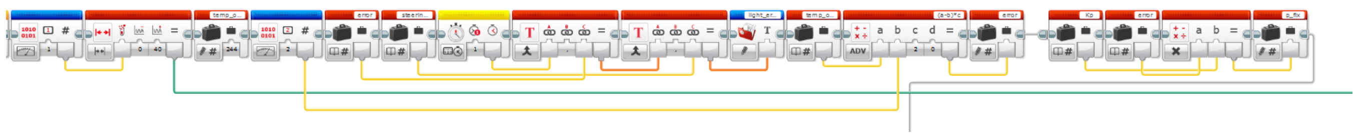
Later the raw value is subtracted from the offset value and error is computed. This error goes into the PID controller which controls the steering.



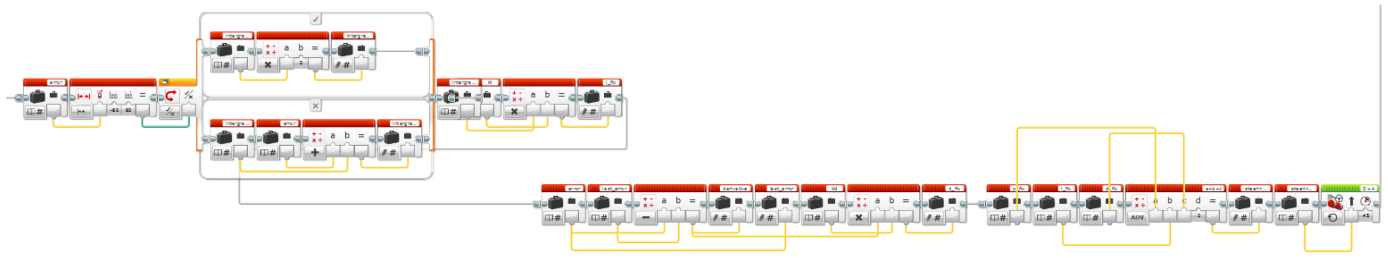
Later we have a block for data accusation.



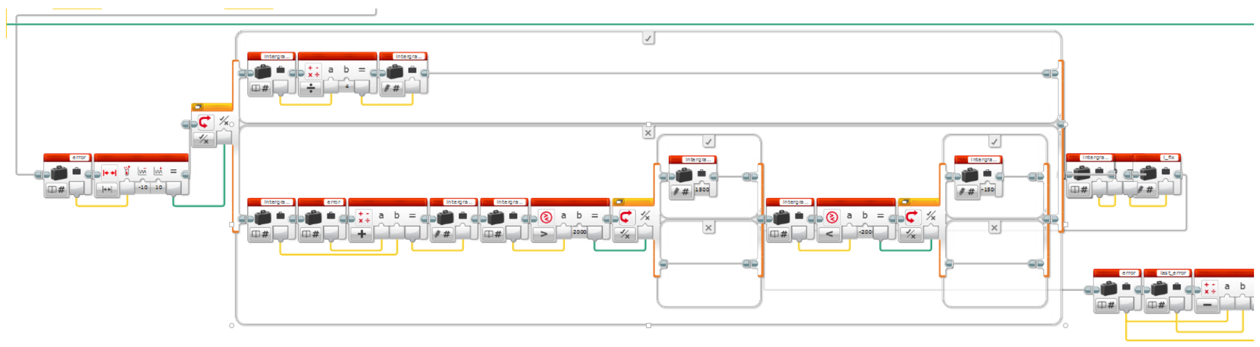
iii. Milestone 3:
Similar to milestone 2, we compute the error first and put it through the PID controller. We also have a code to stop the bot before it touches the obstacle in front of it



We have a block which negates the integral windup if it is between a certain range.

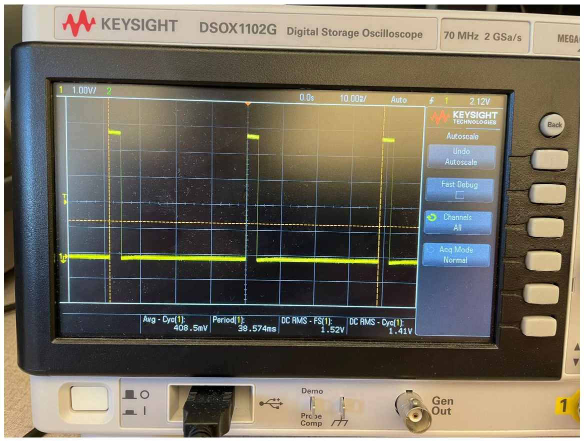


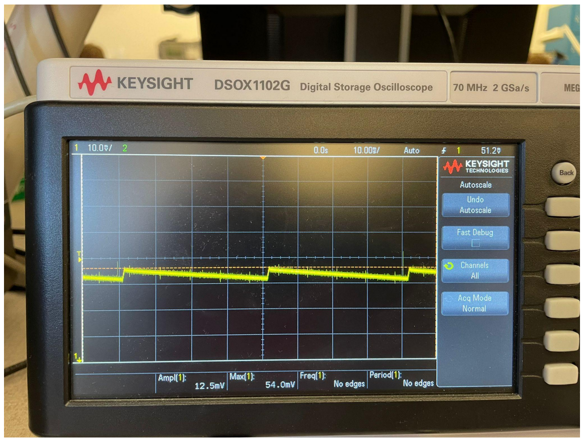
After further testing we concluded that we needed the integral accumulation to saturate. This is because if we let the integral windup then we weren't able to negate it quick enough and bring it down back to a value near zero. Because of this our robot was overshooting after taking sharp curves. The following pictures show the changes made to the integral part of the code



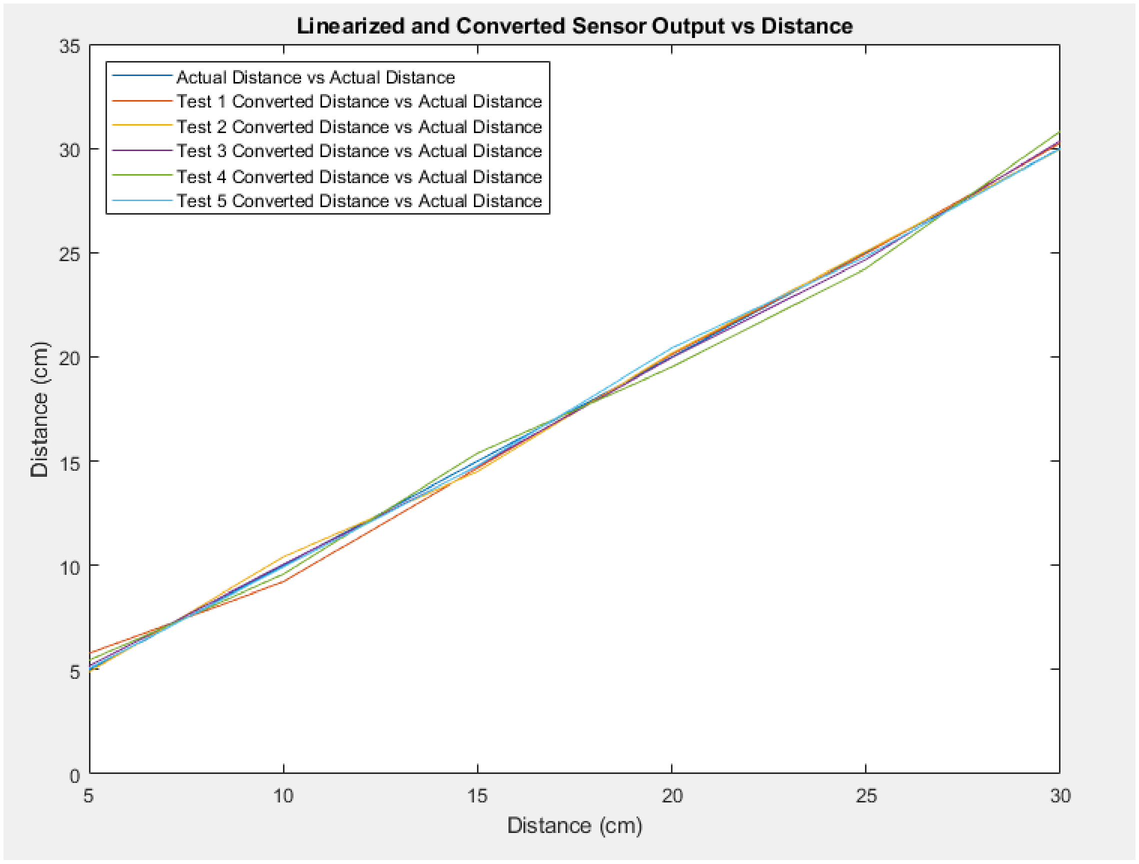
Results

1. Ultrasonic sensor output:





2. Task 2:
   In task two we experimented with linearized vs non linearized ultrasonic sensor values. The plot below shows the result of those experiments. Using MATLAB we collected ultrasonic outputs from multiple distances ranging from five to thirty centimeters. Once we had the raw values, we found the linear function that best fit the points and converted the raw values into centimeters. The blue actual distance vs actual distance shows the plot that would result from a perfect sensor. The plots labeled as converted distance tests are the actual plots we received from the ultrasonic sensor at specific distances. In the final vehicle design we chose to emit this conversion from the code to improve the speed and eliminate the error that is in the linearization functions. Instead these results simply show the accuracy of the ultrasonic sensor so that it can be properly utilized.



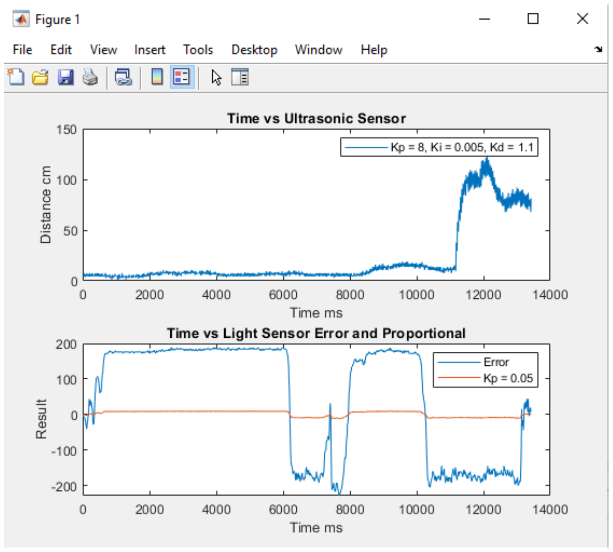
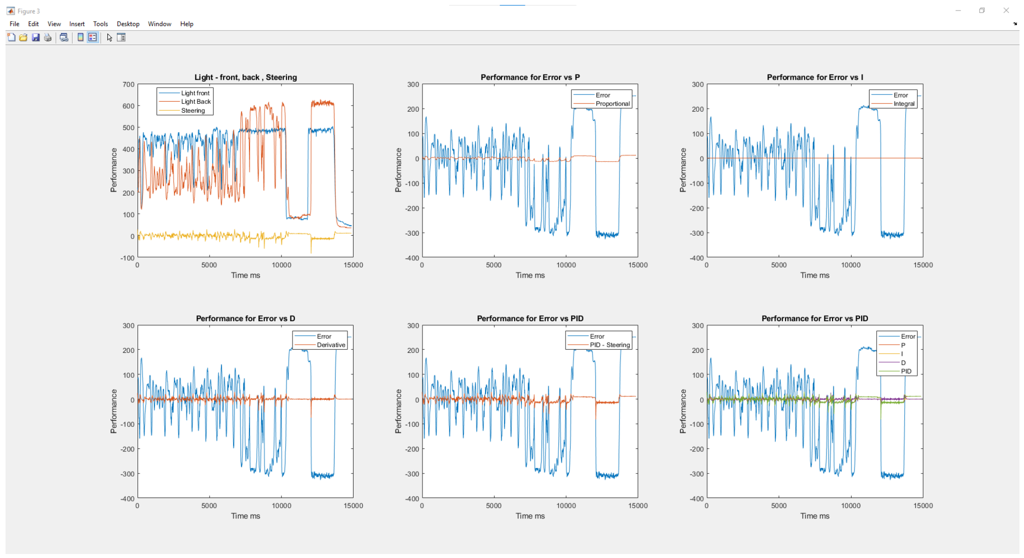


Figure 1
File Edit View Insert Tools Desktop Window Help
Time vs Ultrasonic Sensor
Kp = 8, Ki = 0.005, Kd = 1.1
Distance cm
Time ms
Time vs Light Sensor Error and Proportional
Error
Kp = 0.05
Result
Time ms

For the platooning task we used a P controller and the most optimal steering was found for when Kp = 0.05. There were more oscillations so we reduced the Kp further to 0.045. Which was ideal for most of our project ahead.

Task 3:
For task 3 we started with working with a PD controller. The outputs were promising but we noticed that for consecutive turns the bot was slow to react. But if we increased the Kd the bot is jerky during straight lines.
[Graph for when Kp = 0.045, Ki = 0, Kd = 0.3]



Later we used Zeigler Nichols to tune the PID. We used a PI controller- The results showed that the bot could take consecutive turns well. But it was not consistent.

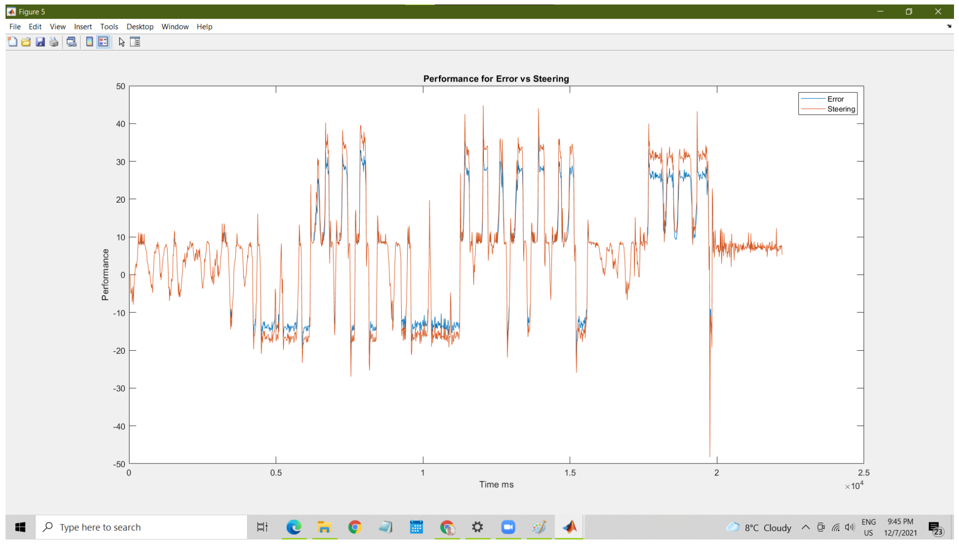


Fig. Kp = 0.03

We kept the Kp = 0.045 as the value that is most ideal.

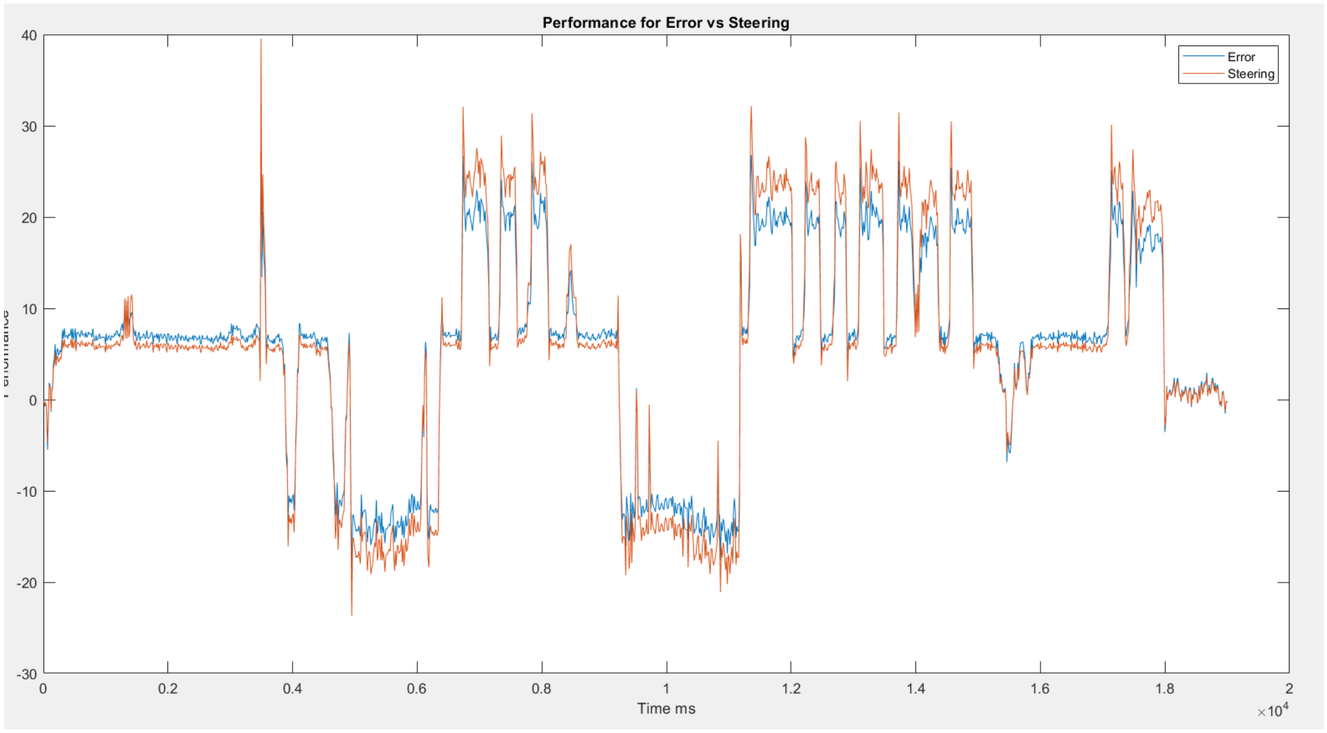
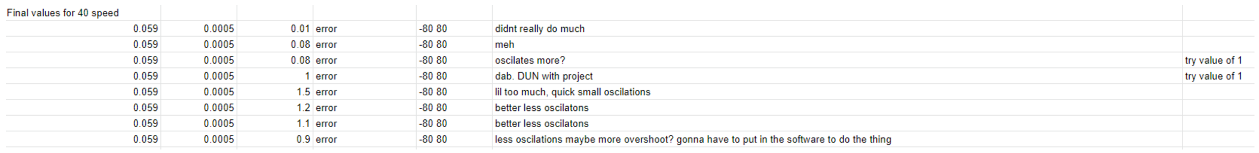


Fig. Kp = 0.045, we can see the steering is relevant to the error.
Ki = 0.3

Tuning the PID for milestone 3

To tune the PID for milestone 3 we tried to take multiple approaches. Initially we tried to change the position of our sensors, so that one sensor was on one edge of the line and the other sensor was on the other side of the line, however, we didn't solder the sensors far enough and this was not possible. We then decided to switch to using just one sensor. We first decided to use the sensor in the front. To do this we followed a simple method. Set, Ki and Kd to zero, then slowly increase Kp until the bot is able to follow the line fairly well with only minor oscillations. Once this was done Ki was added. If the robot was not able to take the sharp curves quick enough, the Ki value was increased. If there were too many oscillations on the straight line, then the Ki was reduced. After a suitable Ki value was obtained, then Kd was added. The Kd value was started off at 1 and reduced if there was too much jitter on straight lines. The following table shows the first attempt at tuning the PID.



Final values for 40 speed						
0.059	0.0005	0.01	error	-80 80	didnt really do much	
0.059	0.0005	0.08	error	-80 80	meh	
0.059	0.0005	0.08	error	-80 80	oscilates more?	try value of 1
0.059	0.0005	1	error	-80 80	dab. DUN with project	try value of 1
0.059	0.0005	1.5	error	-80 80	lil too much, quick small oscilations	
0.059	0.0005	1.2	error	-80 80	better less oscilatons	
0.059	0.0005	1.1	error	-80 80	better less oscilatons	
0.059	0.0005	0.9	error	-80 80	less oscilations maybe more overshoot? gonna have to put in the software to do the thing	

After this we decided to experiment more. We tried to use the back sensor, as opposed to the front sensor. We used a similar process that we used to tune the PID for the front sensor. The following table shows the PID values that we got with our testing

Try back sensor at 40						
0.059	0.0005	0.9	error	-80 80	ok with 40. takes longer to respond	increase ki?
0.059	0.00055	0.9	error	-80 80	oscilates more	increase ki?
0.059	0.0005	0.9	error	-80 80	oscilates more	increase ki?

One mistake that we made was not including file writing in our tuning code. After adding the file, writing our code slowed down a lot and we had to re-tune our PID. The following table shows the PID tuning we did for the code with the file writing.

Speed 50					
0.055	0	0	not many oscilations	maybe reduce lil bit more	
0.05	0	0	barely any oscilations but longer to respond		
0.052	0	0	similar to before	add in ki	
0.052	0.002	0	reacting way to slowly	increasi ki	
0.052	0.001	0	less oscilations cant take turn well enough	add in kd	
0.052	0.001	0.3	beauty, lil bit of oscilations reduce ki?	reduce ki	
0.052	0.001	0.25	better, add more ki to react faster to curves	increase ki	
0.052	0.002	0.25	good bit of scilations maybe increase the ki 0 range		
0.052	0.003	0.25	increased to -100 to 100		

With these final PID values we were able to get our bot to follow the line without too much oscillation and able to take sharp curves without too much overshoot.

The final PID values that we are using for the task are Kp: 0.04, Ki: 0.001, Kd: 0.2

Discussions

Milestone 1: In the first milestone demonstration the unmanned vehicle stopped four millimeters from the object. This result was achieved by running the vehicle at variable speeds depending on the distance from the object. The distance output of the sensor and the speed are directly related. When the vehicle distance is low the speed is low. This procedure was chosen because of the variability and accuracy of the ultrasonic sensor as well as the momentum that the vehicle carries while running. The SRF05 sensor is not high enough quality to have a continuous and accurate output. While working on this milestone it was found that the SRF05 sensor output is in increments of around two to three centimeters and so sometimes the EV is closer to the object than measured and is therefore unable to stop in time once the stopping range is reached. In addition, the vehicle carries more momentum at high speed and because it does not have any braking mechanisms it will carry into the object if the speed is too high when parking.

Milestone 2: In the second milestone demonstration the vehicle followed the straight line and stopped on the red line, but was unable to match the speed of the TA vehicle while it was at its top speed. Even though this milestone was not timed, the maximum speed of the vehicle was the most important aspect of the vehicle. The difference in speed between the two vehicles is difficult to explain. The two vehicles had the same size wheels, neither used gears to scale the output of the motors, both had similar weights, and both traveled in a nearly straight path. One of the possibilities is that the TA vehicle happened to have motors with less wear and therefore higher top speeds. While working on this milestone we found that one of our motors was slower than the other. This factor could have had an overall negative effect on the test results for our group. This milestone was the only one that required both distance and speed control, however they were both simple to implement. The vehicle had to follow a straight line and so a simple proportional controller with a low Kp value was enough to keep the vehicle on track. Similarly, the distance control relied heavily on proportional control to achieve a quick response to the TA vehicles change in speed. One unexpected variable to consider in this milestone and milestone three was the track conditions. While the vehicle was running, if it collected enough dust on the wheels, the vehicle would often lose control or travel at a greatly reduced speed. To accommodate this frequent wheel and track cleaning was required while testing.

Milestone 3: This report is due before the demonstration for milestone three and therefore this discussion does not include information on that event. The direction control was the most important aspect of the vehicle while testing for this milestone. The main consideration while building the control was the set speed of the vehicle. The vehicle was unable to take the tightest track turns or travel in a straight line smoothly while at high speeds. The majority of the time allowed for preparing for this task was spent trying to balance the speed of the vehicle and accuracy of the track following. The vehicle design for this milestone will only run at approximately 50% of its maximum speed so that the line following is smooth. While preparing for this milestone there were many vehicle design problems that arose that could not be overcome because of lack of parts and foresight from the group. For example, the reflective light intensity sensor placement was a major component in the type of control that could be used for line following. However, because only three sensors were provided and they were permanently placed on the board to be used in milestone two, it was risky to attempt to experiment with different sensor orientations. The orientation could only be changed by deconstructing the circuits or cutting circuit components. In hindsight it would have been a brilliant idea to permanently attach each reflective light intensity sensor to its own perforated board so that they could be moved as needed.

Milestone Grading: The evaluation code for the milestones was thoroughly explained in the project description. The evaluation code also seemed like a great way to test the improvement of the vehicles between tests for an individual group. However, there was continuous discussion and questions among groups about how this code would work between different vehicles because of the difference in sensor readings and quality. For this reason it seems that the milestones may have been better graded by evaluating a couple of key aspects from the demonstrations. The first milestone could have been graded on how close the vehicle was able to park to the object. The second milestone could have been graded on accumulated error and whether or not the vehicle stopped at the appropriate spot. The final milestone could simply be graded on time to complete the course and a general visual evaluation of the line following quality. The grade statistics for the milestones were posted, but none of the individual group grades were released. The complexity of the grading seems to have left many groups in the dark on their true milestone performance.

What we Learned

Tuning a PID takes a LONG time with a lot of trial and error
Writing to a file is super time consuming and should be avoided if possible.

References

SRF04 Technical Documentation (robot-electronics.co.uk)
Micro Sensing Device Data Book (omron.com)
HOWTO create a Line Following Robot using Mindstorms - LEGO Reviews & Videos (thetechnicgear.com)
Lab4&5