San Francisco State University's CSC615 - Embedded Linux: Autonomous Bot Final Project

Report Prepared by Fried Pi

Submitted by Dec 13th, 2022
To Prof. Robert Bierman

As part of the requirement for CSC615 - Unix Programming/Embedded Linux

Fried Pi's Github Repository

https://github.com/CSC615-2022-Fall/csc615-term-project-DavidYeLuo

Team Members

Rafael Sunico (Github: Gravemind1142)
Marc Castro (Github: mcastro16)
Christian Francisco (Github: chrisf725)
David Ye Luo (Github: DavidYeLuo)

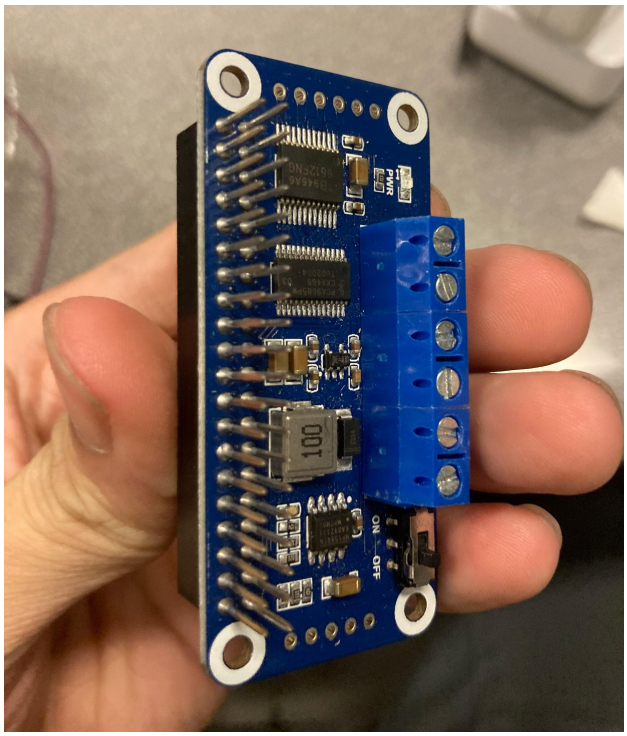
Task Description

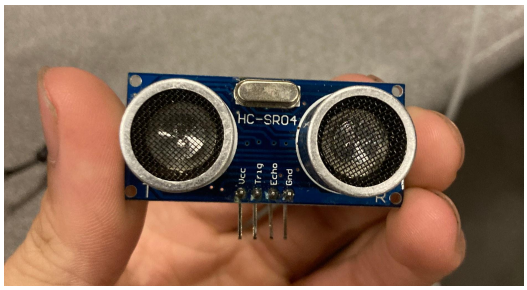
List each team member, and Github Username of primary github with your full source and Makefile

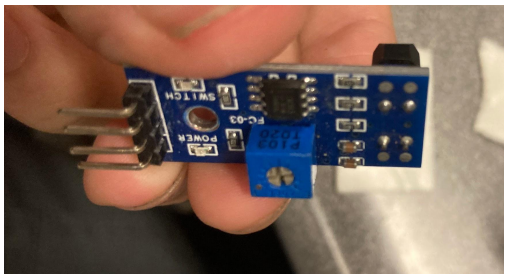
- Building the Robot (include photos)
- Parts / Sensors Used (include photo, and part numbers where applicable, such as HC-SR04 for the sonic echo sensor)
- How was bot built (photos good to include)
- What libraries/software did you use in your code (include full reference)
- Flowchart of your code
- Pin Assignments you used
- Hardware Diagram
- What worked well
- What were issues

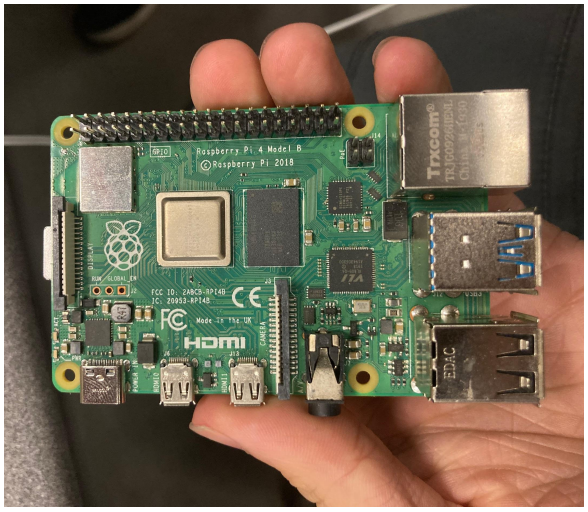
Parts / Sensors Used

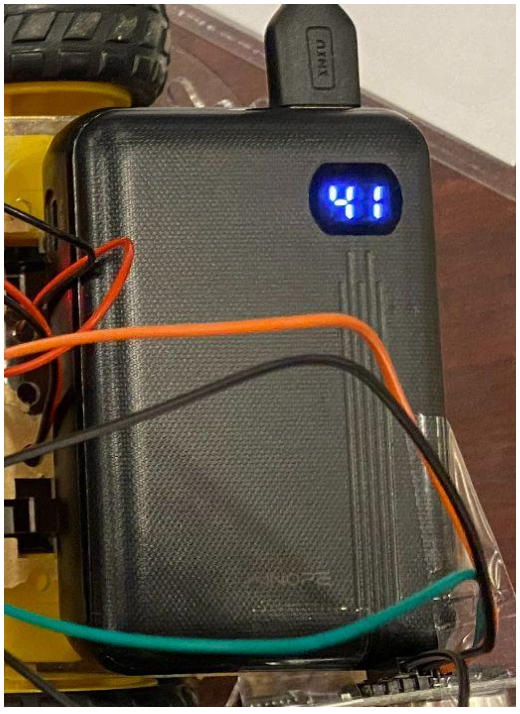
- 2 Line Sensors (TCRT5000 Reflective Optical Sensor)
- 3 Sonic Echo Sensors (HC-SR04)
- 4 Motors
- 1 Portable Battery Pack
- 1 Raspberry Pi HAT

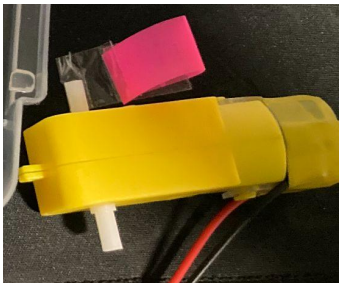






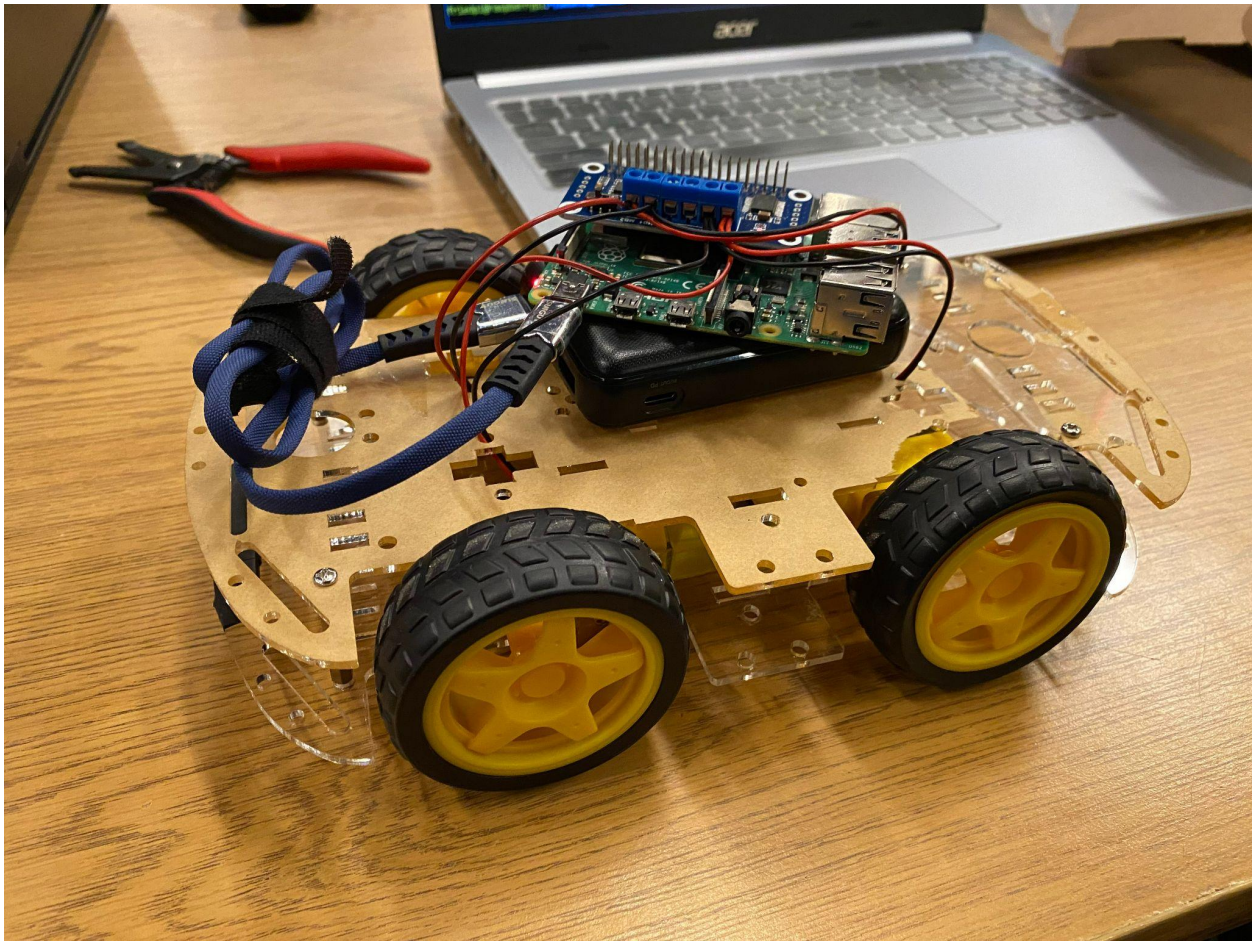




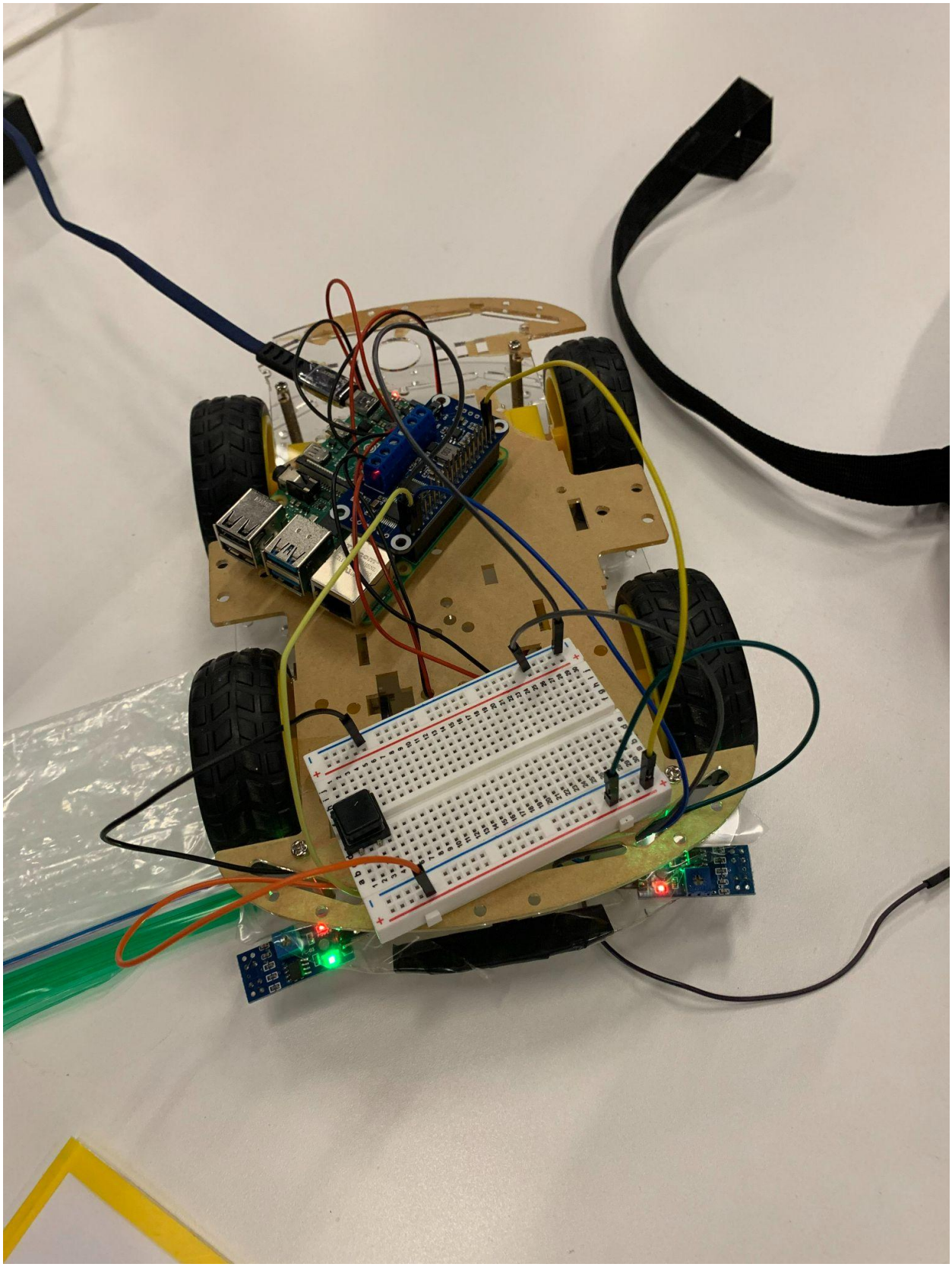


How was bot built

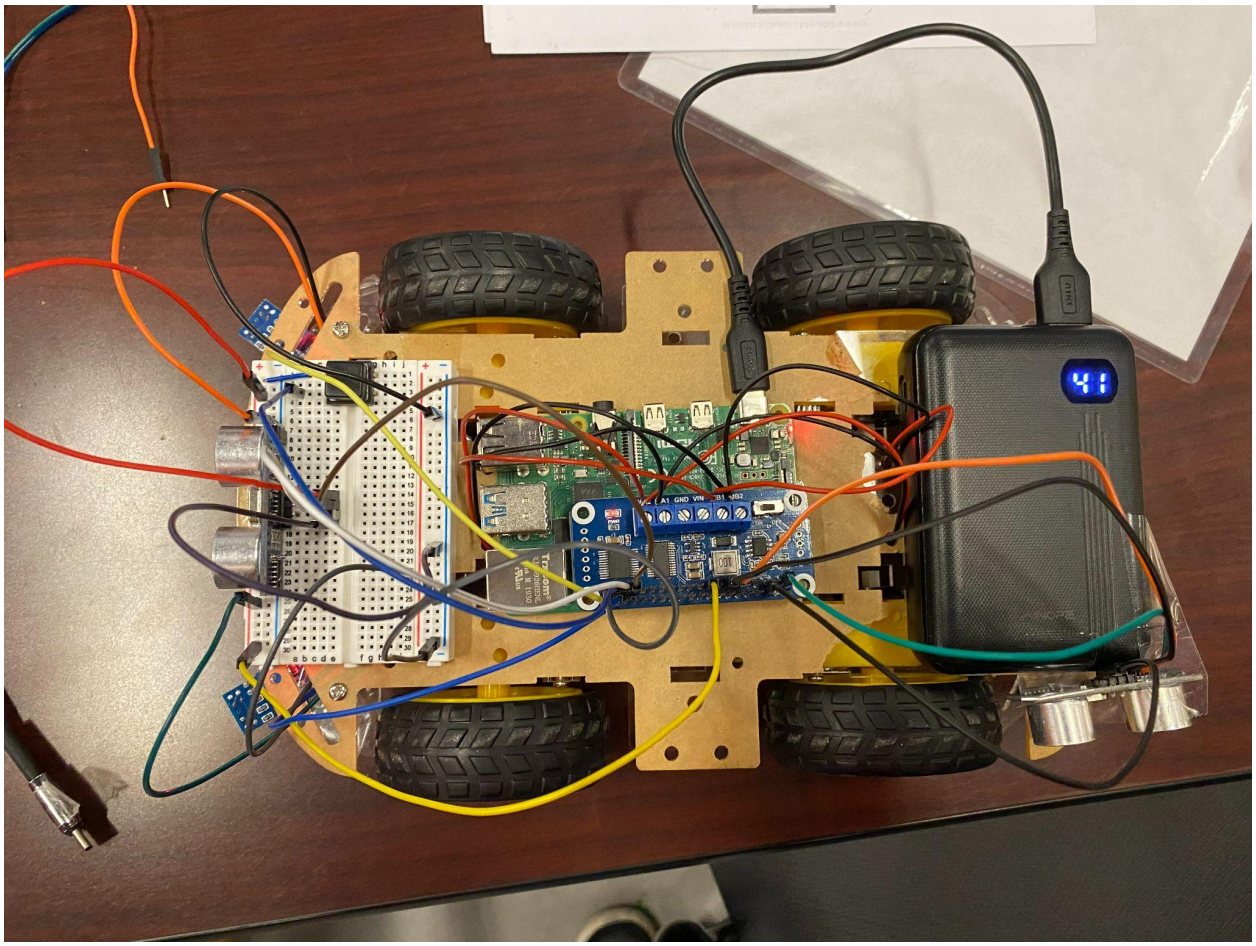
When thinking about the core functionality of the bot, we first thought that it should be able to move first. So, we built the bot up to the point where it begins moving. We used four motors and a portable battery pack in order to get the bot running. We then confirmed the bot properly moves by importing our code from assignment 3 since that was the assignment that allowed our motors to move.



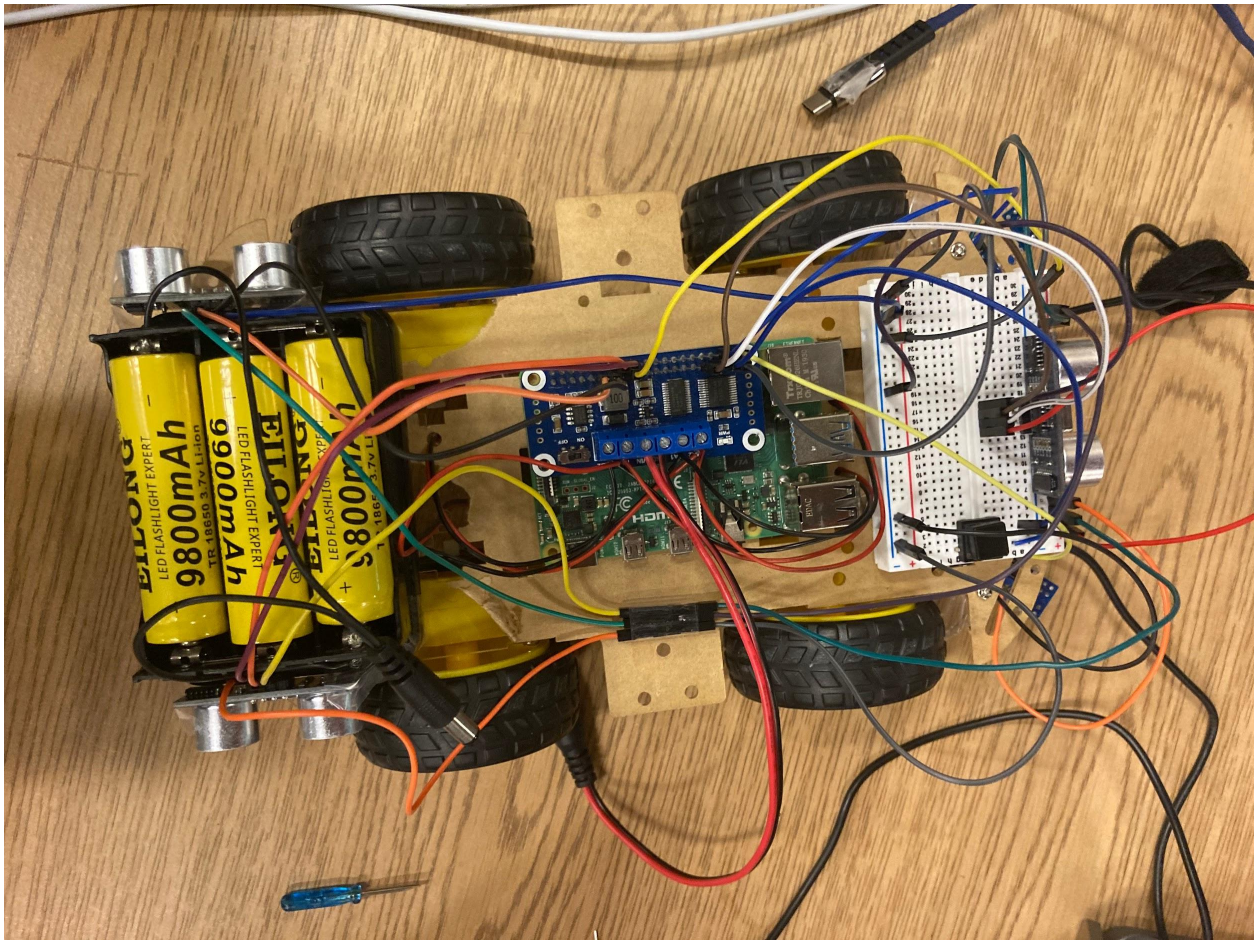
Once we got the bot to move, we then needed a way to read a black line so that the bot would follow it. We decided on using two line sensors, one on the left and one on the right, in front of the bot. By doing this, if the left sensor detects the line, the bot moves left, and if the right sensor detects it, the bot moves right. Adding these two line sensors required us to use the breadboard which we put at the front. Putting the breadboard at the front edge of the car allows the echo sensor to get a perfect read on the obstacle that is in front.



After getting the line sensors working, the next step was for the bot to move around an obstacle, and our group decided on using two echo sensors, one on the front and one on the back left side of the bot. The main logic for the bot to move around an obstacle revolves around these two sensors since the bot will stop after detecting an obstacle in front of it and then rotate right until the left echo sensor detects an object. The bot will move forwards repeatedly until the left echo sensor stops detecting an object. It will then turn left 90 degrees and move forwards repeatedly until the left echo sensor sees an object. The bot will turn left and move forward again until the right line sensor sees a black line. It will then move right until the left line sensor sees black and the right line sensor sees white.



After working on the implementation of the two sensors and verifying the functionality of such, we realized upon talking with the professor and examining the course that the professor could place the obstacle anywhere on the course to emulate real life scenarios. Therefore, we decided to implement another sensor for the right-back side of the robot to account for the possibility of the vehicle needing to go around the left side of the object rather than the right side. Giving us our final product for the robot.


9800mAh
LED FLASHLIGHT EXPERT

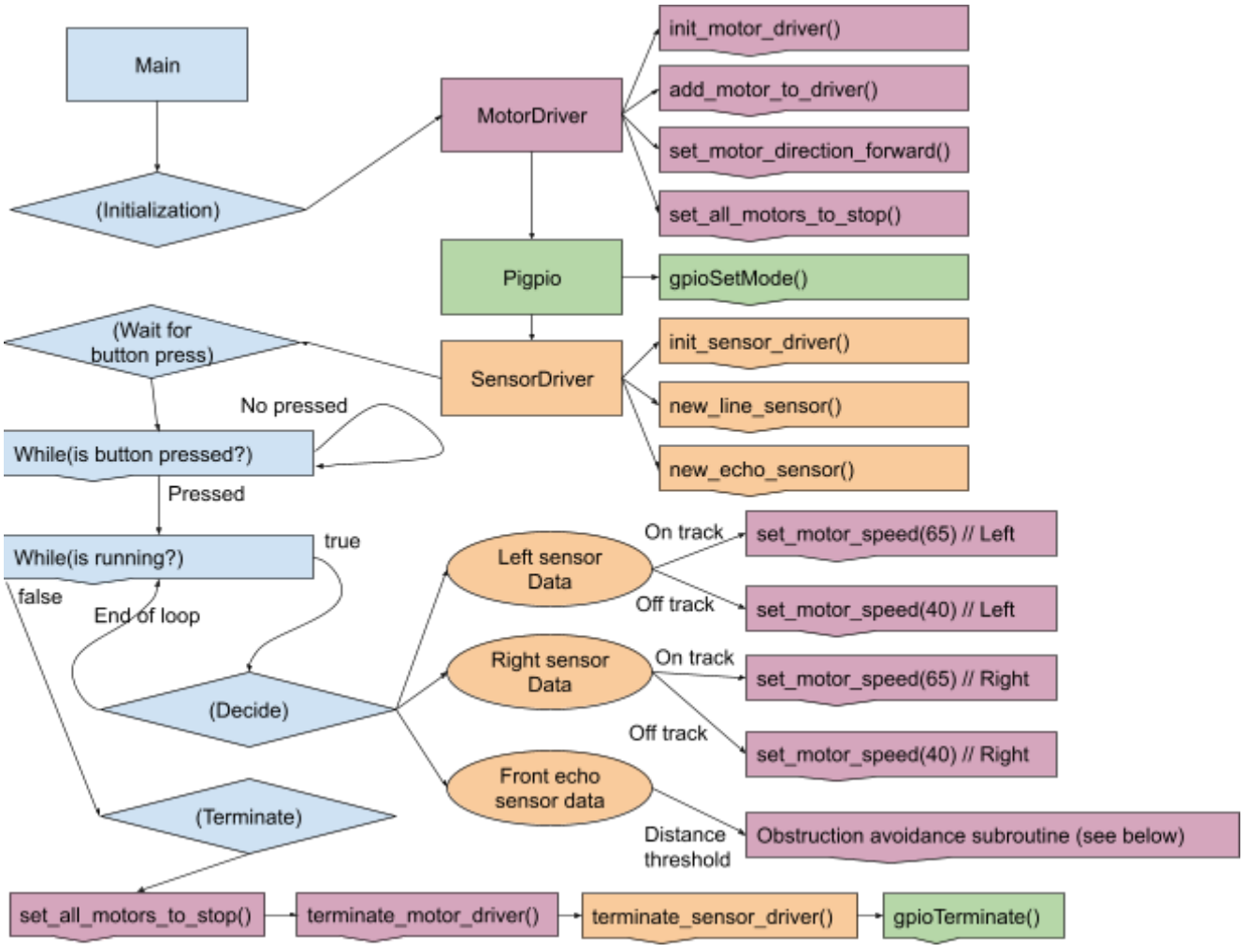
What libraries/software did you use in your code

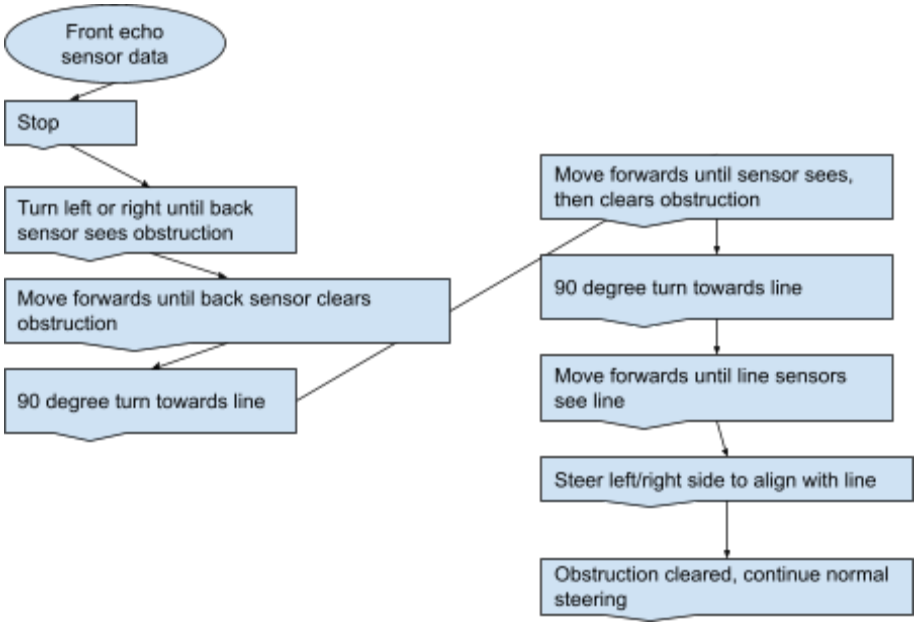
We need our Raspberry Pi to communicate with our sensors. Our approach for such communication is using GPIO pins, also known as General Purpose Input Output Pins, which the Raspberry Pi has many. These pins are identified by a number in the Raspberry Pi documentation. However, not all pins in the Raspberry Pi is a GPIO pins so we need to pick pins carefully. The hardware side is simple where we pick the pins suitable to the sensors' needs and the GPIO pin to recieve data. Software side, we need to enable the pin and set the pins to input or output mode. We could've implemented it on our own using Direct Memory Access or communicating through the Linux virtual file system. However, we decided to use the external library called Pigpio to save time on the other parts of our project.

We were using the WaveShare Motor Hat to control the motors. It supports PWM duty cycle control which is how the motor controls the speed of the motor. Normally, the motor doesn't have control of the speed because it can only spin at full speed or stop. How we control the speed is turning the voltage high and low rapidly. This is what the PWM duty cycle is doing which our car needs. We are using the components of the WaveShare libraries that handle the underlying implementation of PWM as well as other features to save time. The one component that we didn't use was the MotorDriver because we didn't like their implementation.

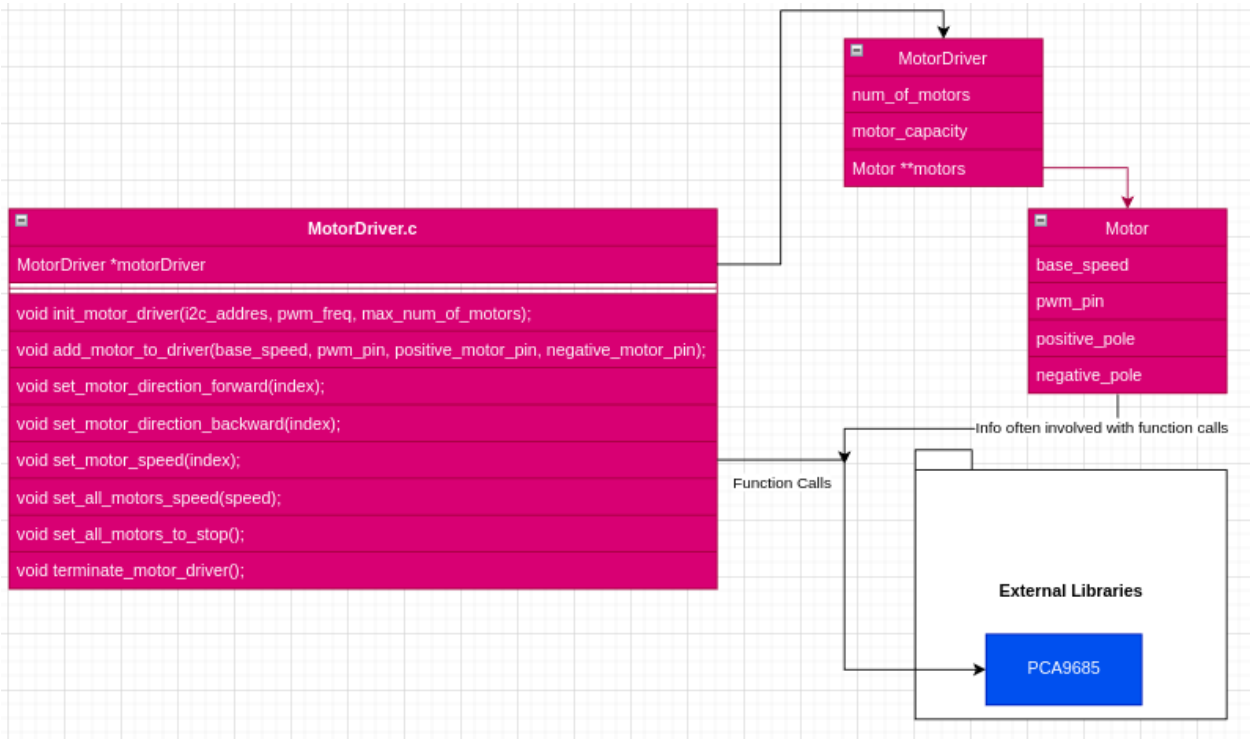
Flowchart of your code

The main.c file has many responsibilities. Its primary job is making decisions on how the car should operate. Two of the components that the main.c use are the SensorDriver and MotorDriver. Based on different data from the sensor, it will decide to interact with the motors in a specific way. This decision logic lives in this file, for example, normally the sensor has a direct influence on the wheel. If the left wheel is off course then it will decide to rotate the car to the right until it is on course. Another thing that main.c is in charge of is waiting for a button press before starting. The program waits for a GPIO pin to send a signal so that the program can proceed. Lastly, the main.c file is also in charge of initializing these components to use and terminating when done.

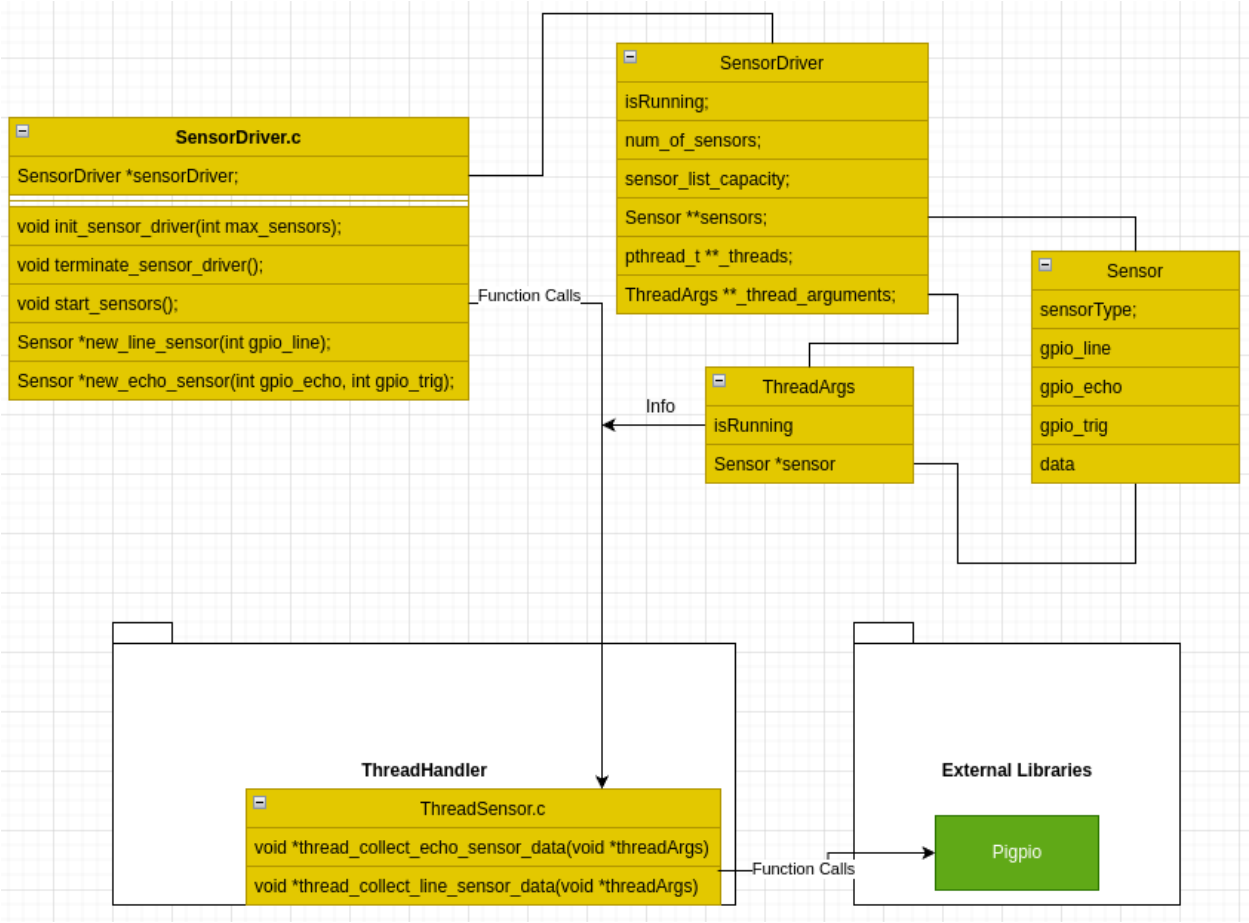




The MotorDriver.c file is not the same as the one from WaveShare. Although it uses PCA9685 from WaveShare, the driver is implemented differently from the counterpart. Our MotorDriver is designed as a wrapper to more easily interface with motor(s). First, it stores information in a struct called Motor from the initialization process. This information like PWM_pin is used to identify which motor to control. This structure allows us to create functions that minimize the number of arguments needed. In other words, they are wrapper functions to easily interact with the PCA9685 from the WaveShare library that does the interaction with the motor.



The SensorDriver.c file works similarly to the motor driver. c in which it also uses structs to store information for later use and manages an array of sensors. The information about the sensor is stored in the Sensor structure which contains the type of the sensor, the sensor GPIOs, and the output sensor as data. One of the benefits of creating a Sensor structure is that we have a common interface between the echo sensor and the line sensor. We are using multithreading so the information will be helpful to pass to the thread. However, we also pass in the state of the program so we can terminate the sensors. Because these two data are stored in different structures, one is stored in the SensorDriver and the other one is in Sensor, we created ThreadArgs structure to pass in more than one argument. These thread functions will be called from the Pigpio library to keep gathering data from their corresponding GPIO pins. The results will be placed in the data from the sensor structure.



Pin Assignments you used

Front Echo Sensor

Echo - GPIO 13
Trig - GPIO 19
VCC - 3V3 (Physical Pin 17) via breadboard
GND - Ground (Physical Pin 39) via breadboard

Back (Left) Echo Sensor

Echo - GPIO 27
Trig - GPIO 17
VCC - 3V3 (Physical Pin 17) via breadboard
GND - Ground (Physical Pin 39) via breadboard

Back (Right) Echo Sensor

Echo - GPIO 24
Trig - GPIO 23
VCC - 3V3 (Physical Pin 17) via breadboard
GND - Ground (Physical Pin 39) via breadboard

Left Line Sensor

DO - GPIO 20
VCC - 3V3 (Physical Pin 17) via breadboard
GND - Ground (Physical Pin 39) via breadboard

Right Line Sensor

DO - GPIO 21
VCC - 3V3 (Physical Pin 17) via breadboard
GND - Ground (Physical Pin 39) via breadboard

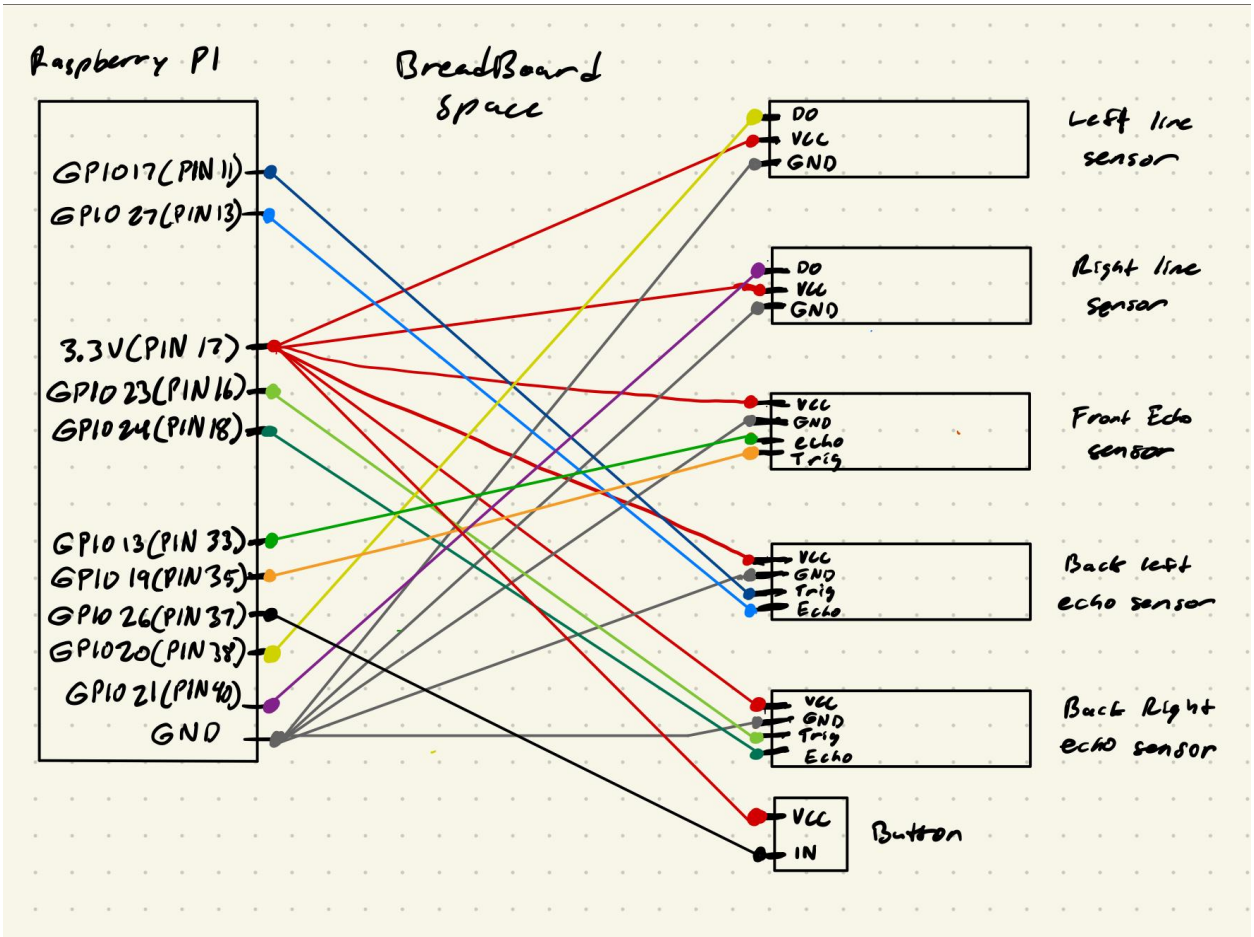
Button

Output - GPIO 26 (Physical Pin 37) via breadboard
VCC - 3V3 (Physical Pin 17) via breadboard

Breadboard

Physical Pin 17
Physical Pin 39

Hardware Diagram



What worked well

- Efficiency of line following
    - Our robot used only two line sensors at the front for following the line, and used 4 powered wheels to respond effectively to the inputs of the line sensors, resulting in quick navigation around curved paths and 90 degree turns most of the time
    - We also connected the two motors on each side to the same terminals in the motor hat, so that we wouldn't have to deal with controlling multiple i2c devices and individual motors.

- Speed and turning calibration
    - To help with calibrating our robot, we used command line arguments in our program to quickly change the variables for speed and turn duration without having to rely on github commits, and recompilation for edits

- Test runs working
    - Within our test runs, we were able to make steady progress especially when it came to making the robot move around the object. This was achieved by iterating through steps within the code, working slowly to make sure the robot would turn the correct amount of degrees, pass the object, turn, and get back on the line.

- Reading documentation on individual parts helped
    - Reading the documentation that was previously given to us in our assignments helped in our initial building and implementation of the parts because we learned some new things about the outputs of the parts. This allowed us to better optimize our code for multiple test runs and quick swaps of values and GPIO inputs when necessary.

- Implementation of past assignment work for functionality purposes
    - Upon reading the previous documentation of parts, we also revisited projects that displayed working instances of the individual parts. With these working instances, we were able to use them in our project, as well as have them work hand-in-hand to output functionalities to our robot. An example of this would be having the robot turn when it runs into a line turn/curve.

- Discussing the project with other groups
    - We were fortunate to discuss other projects with other groups and see how they went about approaching the project. This gave us ideas on how we felt we should handle the project as well as some conceptual ideas on how to solve some problems we had regarding getting around the obstacle.

What were issues

- Understanding how to implement everything in our code
    - Our initial problem was understanding how to implement everything in our code because previously all our implementations were independent of each other. The solution was far simpler than we thought it was, and was a matter of just understanding what each one did and having the others react via if-else statements, as well as running while-loops when certain conditions were met.

- Thread scheduling to read from both echo sensors
    - Because the pi has 4 cores, there were not enough resources to have them all run concurrently since we have more than 4 threads running (main, 2 line sensors, 2 echo sensors). The extra threads weren't being run properly, and there weren't enough resources allocated to run each thread. This problem was first found when our echo sensors were not updating with the correct distances. This was fixed by delaying the threads so that one thread does not hog all of the resources.

- Getting the robot to perform actions to move around the obstacle
    - There were some instances where the bot couldn't make a full 90 degree turn when an obstacle was in front of it causing its path to go off course, and as a result, the left/right echo sensor not detecting the object properly. This was fixed by turning the bot until the left/right echo sensor detected an object and moved the bot forward accordingly.

- Frying the motor hat
    - At the time during the testing phase, we believed that we were overloading the hat which caused it to fry.

- Getting the motors to run in the same direction
    - After frying the motor hat, we had to replace the old one with a new one which meant that we had to rewire the motors to the hat. When doing this, we accidentally put the wires into the wrong pins which caused the wheels to run in a way that is not intended. This problem was fixed by switching the wires of MA1 and MA2.

- Cable management
    - The bot had horrendous cable management to the point where the wires were getting hard to track, and some of them were on the verge of disconnecting from their pins. The only thing that was done about this was taping the wires together from the back sensors.

- Getting the robot to stay on the line
    - There were some instances where the line sensor was not detecting the line correctly which caused it to move off the line. The main reason for this was that we used the wrong GPIO pin for the line sensor, and we fixed this by changing the GPIO pin to the intended one.

- Problem with the makefile
    - We had some issues with the makefile when it came to compiling all the libraries and necessary files to run the components.

- The bot was getting stuck at 90 degree corners
    - This was an issue with the placement of the line sensors. Initially, we had the sensors placed at the round edge of the chassis which didn't work well when turning corners. This was fixed by placing them perpendicular to the wheels.

- When rotating, the bot stutters causing inconsistencies in the degree of rotation
    - Initially, we thought the issue is caused by low power going through the hat which is correct, but when replacing the batteries, the problem still persisted. Eventually, we pinpointed the problem which was the loose wheels. The loose wheels were causing the rotation stutters, and we fixed this by tightening the wheels.

Successful Run

https://www.youtube.com/watch?v=AheX5phQXsY