**ELEC 291 Section 20C Section L2C**

Project 1 Report

Benches 2A and 2B (Team 2)

March 9, 2016

|  |  |  |
| --- | --- | --- |
| **Student Name** | **Student Number** | **Contribution Percentage** |
| Timothy Leung | 14189147 | 1/6 |
| Derek Tam | 24630148 | 1/6 |
| Kevin Qiu | 14188149 | 1/6 |
| Andy Ruan | 36863141 | 1/6 |
| Kevin Wong | 32105132 | 1/6 |
| Clarence Su | 36387132 | 1/6 |

**Contribution Summary**

Andy Ruan managed and organized the workflow, skeletal code, and documentation. He was in charge of the master-slave communication between the two Arduino Unos and wrote the code for the main loop, mode selection, and LCD display.

Kevin Wong implemented logic for Additional Functionality. He wrote code to decode IR signals and to drive the robot. He also helped implement the serial communication from Arduino to Processing and the an early prototype of the Processing diagram.

Clarence Su implemented logic for Additional Functionality. In Arduino, he wrote the probability distribution algorithm that determines the amount of obstruction the sonar reads. In Processing, he wrote code to implement the user interface to change the color of matrices depending on probability density.

Timothy Leung implemented the logic for Principle Function 2. He modeled and printed the 3-part wheel magnet mount and contributed to working on the PID drive and turn functions. He also worked on the circuitry for the optical line sensors and hall effect sensors.

Kevin Qiu implemented the logic for Principle Function 1. He also designed several mounting pieces that were 3D printed for the sensors and servo. The drive train, servo and ultrasonic system, and LCD module circuitry were completed by him.

Derek Tam worked on the code for both principle functionalities. He performed the majority of the testing and debugging for each system of the robot. The optical line sensor circuit and the hardware of the robot were completed by him.

Each team member contributed equally on Project 1.

**Table of Contents**

A. Cover Page - 1

Table of Contents - 2

B. Introduction - 3

C. Project Description - 4

D. References - 13

Appendix A - Robot Pictures - 17

Appendix B - Code - 21

Appendix C - Fritzing - 46

Appendix D - Other - 47

**B. Introduction**

This report outlines the completion of Project 1 by Group 2 (Teams 2A and 2B). The objective of this project was to design and construct a two wheeled robot that is capable of running several autonomous functions: scanning its surroundings and moving according to the scans, detecting and following a line, reacting to an infrared signal, and obstacle detection. The robot utilizes several sensors to calculate the next actions when running each functionality. First, we constructed the robot from the provided kit and added the necessary sensors to the chassis. Then, we wrote the algorithms for each functionality and tested them with the robot. Finally, we combined all the functions together and placed them within a overhead loop with button controls and menu selection. The motivation for this project was to interface various sensors to a physical moving object and be able to control its movements through algorithms.

**C. Project Description**

**Robot Construction**

We first began with the DFRobot 2WD Mobile Platform. The build consisted of several metal plates as the chassis, two motors with accompanying wheels, a caster wheel, a battery mount, a barrel plug, a flip switch, and mounting hardware. The construction process was straightforward as the instruction manual was thorough and descriptive enough. The included wires were soldered onto each terminal of both DC geared motors. Since each motor is rated for 6V, we needed to hook up each motor into a motor shield which will be powered by our 7.5V power supply. The positive battery wire was soldered onto one of the terminals of the switch for running toggleable power to the rest of the robot.

**Sensor Usage**

We have used several sensors to accommodate for the required functionalities of each principle and additional functions. For Principle 1, the driving and scanning function, we needed to use the ultrasonic sensor (HC-SR04) for detecting objects, the servo (SG92R) to rotate the ultrasonic sensor, and one hall effect sensor (SS443A1) over each wheel to encode the rotation of the wheel with magnets. For Principle 2, the line following function, we utilized three reflective optical sensors (TCRT5000) to detect a line on the ground. Our IR remote control function uses an IR receiver (TSOP38238) and a standard TV remote (Samsung BN59-01178W).



**Sensor Mounting Hardware**

In order to mount some of these sensors on the robot, we needed to create custom parts to hold the sensors. These parts would be 3D printed and attached on the robot using the extra hardware left

over. We designed a servo holder to keep the servo level on the front mounting plate and an ultrasonic sensor holder that will be attached on to the servo. Small L-shaped brackets were added on either side of the robot in the rails to allow for the hall effect sensor to hover over the magnets in each wheel.

We were initially given only 2 magnets. As we discussed basic code, we realized that we would have an issue with only using 1 magnet per wheel. The hall sensor would only trigger once per rotation of the wheel. This meant that between the first trigger and second trigger of the hall sensor, we would not know where the wheel is. It could have rotated 1 degree, or 359 degrees. If we wanted to use the hall sensor to determine the difference in position of each wheel, we could have up to 359 degrees of movement before we realize the wheels are out of sync. We decided to design and implement a magnet mount for 12 magnets on each wheel.

Refer to Appendix D for 3D model and design of the wheel mount.

**Arduino Shield Setup**

In order to attach the motors and all the sensors to our Arduino Uno, we needed to utilize two Arduino shields. First, the Arduino Uno was mounted to standoffs and placed in the holes in the upper plate.

Then, we needed to attach the DFRobot Motor Shield. This shield will allow the motors to receive their rated 6V at 2A. We connected our main power source, the 5 AA batteries (7.5V), into the motor shield to allow the motors to draw current not from the Arduino. After the motor shield was attached, we connected the DFRobot Protoboard to provide us ease of wiring for each sensor on the robot. Next to our master Arduino Uno, we placed an Arduino Uno to act as a slave for master-slave communication.

**Wiring and Port Configuration**

When wiring, we had to take into consideration the limited number of I/O ports on the Arduino. The three reflective optical sensors needed an analog port each, thus we assigned them to A0 to A2. We had to manage our analog ports to ensure there were enough ports for a master-slave communication between two Arduinos. This meant that we had to leave A4 and A5 free of any sensors and will only be used to connect the two Arduino Unos together. The motor shield used digital ports 4-7 which meant when wiring the circuit for the sensors. Digital ports 0 and 1 were reserved for Serial communication when testing. Digital ports 2 and 3 were reserved for both the hall effect sensors as we needed the Arduino’s interrupt service routine to update a global count every time a change in signal was detected. Digital ports 8 and 9 were connected to the echo and trigger pins of the ultrasonic sensor respectively. We connected the servo motor into digital port 10 as the servo needs to utilize PWM from the Arduino. Port 11 is used by the infrared sensor. A toggle button was attached to port 12 to allow for mode selection which will be discussed later in the report.

The slave Arduino Uno had a separate breadboard that contained the LCD display and menu scrolling buttons. For our circuit, we used 12 of the 16 pins on the LCD. Pins 1 and 2 were ground and 5V respectively for the LCD unit. The potentiometer was wired to pin 3 of the LCD and was used as a voltage dial for the screen backlight. The register select pin was connected to analog port A3. Sending a high signal to pin 4 would set the LCD to read and write data where as a low signal would set the LCD to take instructions. Pin 5, the read/write mode pin, was grounded as we only sent instructions to the LCD and had no need to read from it. Pin 6, the enable pin, was connected to analog port A2. Because we used the 4 bit data transfer method, we skipped pins 7 to 10 (DB0 to DB3). Pins 11 to 14 were used for sending each half of the 8 bits from the Arduino to the LCD and connected to digital ports 11, 10, 9, and 8 respectively. Finally, pin 15, connected to 5V through a resistor, and pin 16, connected to ground, were used to power the LCD’s backlight. Two momentary switches were connected with pulldown resistors to digital ports 2 and 3 to allow us to scroll through the LCD menu.

**Sensor Testing**

To ensure each motor and sensor was working as intended, we developed a series of test suites for each sensor. For the hall effect sensors, we first displayed the signal output of the sensor on the serial monitor. This allowed us to determine the correct orientation of the magnets in the wheel mount. Then, we configured the Arduino to enable the interrupt service routine on digital ports 2 and 3 for a change in signal (ie. high to low, or low to high). We then displayed the number of changes, or ticks, each hall effect sensor has read to see if the ISR was triggering. We used the servo library functions to change the rotation of the servo horn. The servo and horn mounting was adjusted such that 0, 90 and 180 degrees correspond to the left, centre, and right positions needed for the first principle functionality. We encountered an issue with our printed servo mount, ultrasonic mount, and servo horn where an alignment issue between these three components caused the angles to be not exactly square. This issue was solved by adjusting the servo horn to be slanted along the ultrasonic mount. For the ultrasonic sensor, we used similar code from Lab 5. This code sends a pulse to the trigger pin and reads a pulse from the echo pin that will be used to determine the distance of an object in front of the ultrasonic sensor.

**Basic Movement Functions**

Each motor for the drivetrain required two pins, one to set the rotation direction and one to determine the speed. We created a simple drive and turn function that ran both motors in the respective direction based on the speed and time parameters of each function. Although we set both motor powers to the same power value, the robot did not drive straight and started to veer toward the left.

For our Principle Function 1: Drive and Scan, we needed the robot to move straight as well as turn in perfect 90 degree turns. To aid in making perfect turns, we wanted to use a gyroscope sensor to measure the angle of the robot with respect to the earth, but this sensor was not included in our lab kit. We decided to use our hall sensor to increase the accuracy of our turn and drive functions.

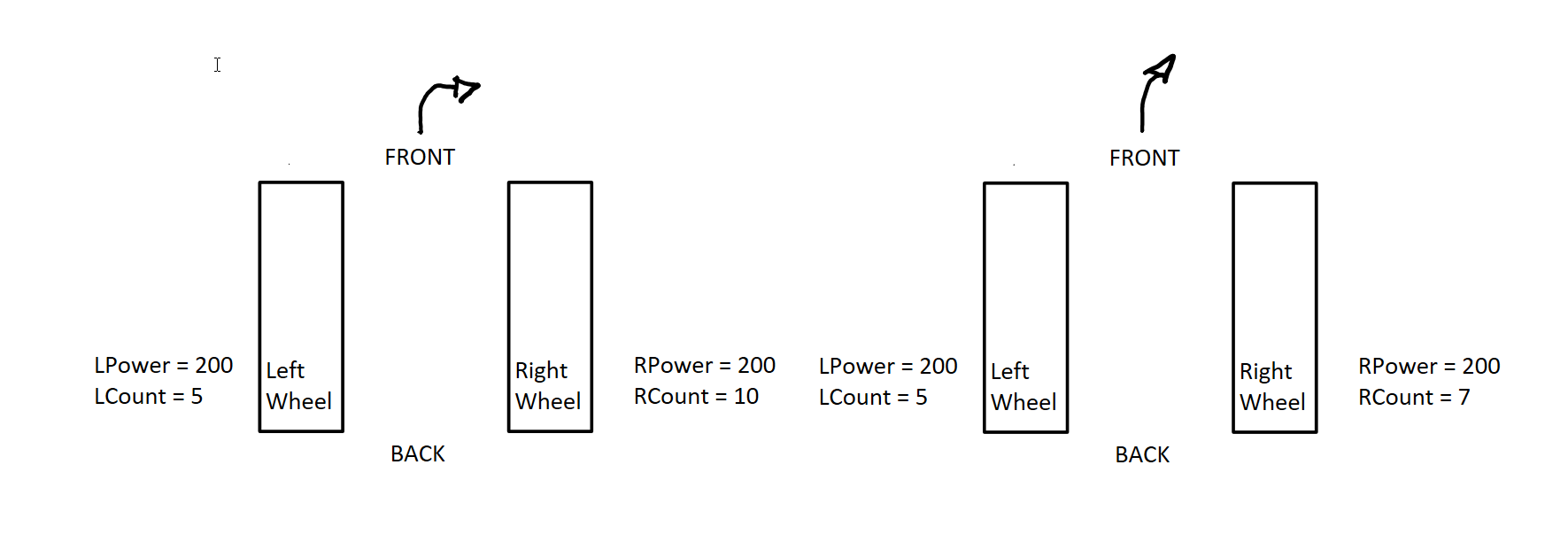
Our program has two global integer counters for each hall sensor, hallLCount and hallRCount, that increase by 24 ticks for every rotation of the wheel. By using interrupts to update these counters, we could use them to provide feedback data to a loop that controls the speed of each side of the wheel.

For our turn\_sensored function, we utilized the hall sensor values as a condition for a while loop to control the angle of the turns. This would cause the robot to turn the same amount, regardless of battery voltage, wheel traction issues as well as variances in arduino’s delay() function. We did not run into any issues; the turn was consistent as long as we kept the speed at 200 power.

For driving straight, a basic approach to our problem would be to have three unique states. The program would detect whether or not the robot is currently driving straight or veering one direction. The code would then change to motors to correct the robot’s heading.

|  |
| --- |
| if(hallLCount == hallRCount) // Driving straight  // Set motors to the same power  else if(hallLCount > hallRCount) // Veering to the right, left is faster  // Slow down the left side, speed up the right side  else if(hallRCount > hallLCount) // Veering to the left, right is faster  // Slow down the right side, speed up the left side |

A problem with this basic feedback loop is that it does not differentiate based on how far out of sync the wheels are (how far the robot has veered off). This means that it may overcorrect or undercorrect the speed. To solve this issue, we changed the feedback loop to adjust the motor speed based on the difference between the two encoder values. If the two are far off, we would adjust the speed by a large amount. If the two are relatively close to each other, we would adjust by a small amount. See Figure 1.

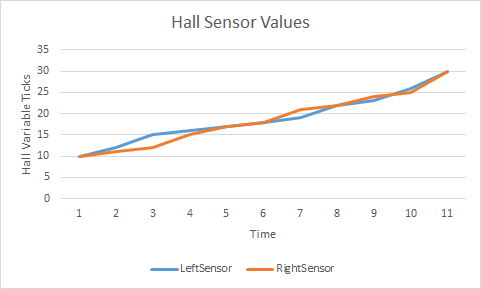


**Figure 1: The previous code cannot tell the difference between the two cases above. Using a feedback controlled loop, we can adjust the amount to swerve in either direction based on the difference between RCount and LCount.**

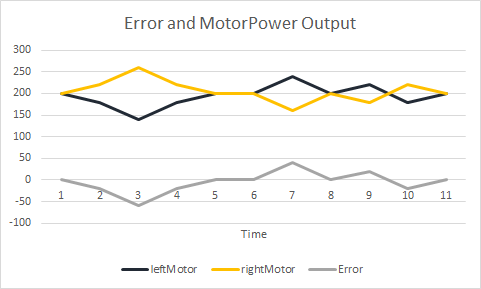
This was done by calculating the error between the two sides and multiplying it by a constant. A positive error means that the right side has moved more. A negative error means that the left side has moved more.

* If the right side is moving faster, we want to speed up the the leftMotor and slow down the rightMotor. Adding a positive value to the base leftMotor speed and subtracting a positive value from the base rightMotor speed accomplishes this.
* If the left side is moving faster, we want to speed up the the rightMotor and slow down the leftMotor. Adding a negative value to the base leftMotor speed and subtracting a negative value from the base rightMotor speed accomplishes this.

|  |
| --- |
| int Kp = 20;  while (true){  int error = (HallRCount - HallLCount)\*kp; // Compute error  leftMotor = power + error + sumOfError\*Ki; // Add error to base power value  rightMotor = power - error - sumOfError\*Ki; // Add error to base power value  } |



**Graph 1: Hall Sensor Values showing motors out of sync**



**Graph 2: Error calculation and MotorPower change applied to correct wheel sync issues**

Utilizing a constant multiplied by the error allows us to tune the amount of response that the error calculation applies to the motor. Finding the correct value means that we will not overcorrect and not undercorrect. As soon as any side starts moving out of sync, a small error will apply a gradual correction to the motor powers. If any physical change in the environment occurs causing a large error, a strong correction will be applied to the motors.

After testing, we learned that the motor speed does not map linearly to the Analog values that we input as parameters. This caused a problem as small to medium errors did very little to correct variation in motorSpeed over time. As a solution to this, we utilized the ‘I’ portion of a PID loop.

|  |
| --- |
| int Kp = 20;  int Ki = 0.05;  int sumOfError = 0;  while (true){  int error = (HallRCount - HallLCount)\*kp; // Compute error  sumOfError += error; // Compute error over time  leftMotor = power + error + sumOfError\*Ki; // Add error to base power value  rightMotor = power - error - sumOfError\*Ki; // Add error to base power value  } |

By summing up the previous errors, we can add additional power to the respective sides to assist the small to medium errors that were not adjusted correctly. The sumOfError grows the longer the error takes to correct the path of the robot. As sumOfError grows, a larger change is applied to the motor. Additionally, a constant is used with sumOfError to allow us to fine tune the response time of sumOfError.

Our implementation works in theory to allow our robot to drive straight. After tuning the Kp and Ki constants, our robot drove relatively straight compared to before. However, we were still seeing a slight zig-zag pattern. We think this is because the motors do not change enough in RPM when adjusting the power from 200±50. The error continues to accumulate until a certain value at which the motors do actually respond. It then overcorrects and the process begins again. The performance of our PI loop isn’t too dissimilar from our original three case feedback loop. We think this is a hardware problem rather than a software problem, so we were satisfied with our Ki and Kp constants.

**Principle Function 1: Drive and Scan**

The first principle functionality we had to implement for this project involved autonomously moving the robot up to an obstacle, detecting the objects to its left and right, and turning towards the direction with a larger gap. The hall effect sensors, ultrasonic sensor, and servo are used for this functionality. First, the robot will ping the ultrasonic sensor for a distance until a valid distance is returned. Then, the required distance in terms of rotation ticks is calculated based on the distance of the object, a stopping distance threshold, and a conversion constant from ticks to centimeters (24 ticks per 20.5cm = ~1.17).

A loop will then run until the wheel has rotated the required amount of ticks. This loop will calculate the new motor speed using the Arduino’s map function [6]. The map function will take the current distance away from the object (x value) and return a motor speed (y value). We used the stopping distance to a constant as the bounds for the x range and the minimum movement speed (100) to the maximum speed (255) for the y range. Once reaching the distance defined by MAP\_CONST, the returned speed will begin decreasing down to the minimum movement speed. The updated motor value is then passed to the next function in the loop, the PID drive function (mentioned above). This will enable the robot to stay on a straight path while decelerating up to the predetermined stopping point.

After the robot reaches the end location, the motors will be stopped and the robot will begin its scanning algorithm. The robot will scan its surroundings using the ultrasonic sensor and the servo to turn the ultrasonic to scan the robot’s left and right sides [7]. The left and right distances are compared and turn direction is determined by whichever side has a greater gap to traverse. If either values are invalid distances, the robot will scan again until it can acquire proper values. After scanning, the robot will then turn 90 degrees in the distance with a greater gap and the drive and scan function will loop again.

When testing the algorithm, we used a simple drive command that did not correct the straightness of the robot’s movement. After the integration of the hall effect sensors and the drive-straight function, we needed to change how the loop ran. Before the changes, the robot would ping the distance each loop and use the ultrasonic sensor’s distance as our mapped value. This allowed for the testing arena to be dynamically changing as the robot can update the speed as objects in front of it moved. In the new implementation we realized updating the distance with the ultrasonic sensor was too resource-intensive which in turn did not allow enough calls to the drive correction function. This meant that the method of calculating the distance needed to change. Instead of using the ultrasonic sensor to find the distance the robot is away from an object, we opted to use a fixed distance which only pinged for the distance once. This fixed distance was calculated using the distance to wheel rotations conversion mentioned above. This removed the ability for the robot to stop if a new object is moved in front of it, but allowed for more precision in adjusting the straightness of the robot’s movement.

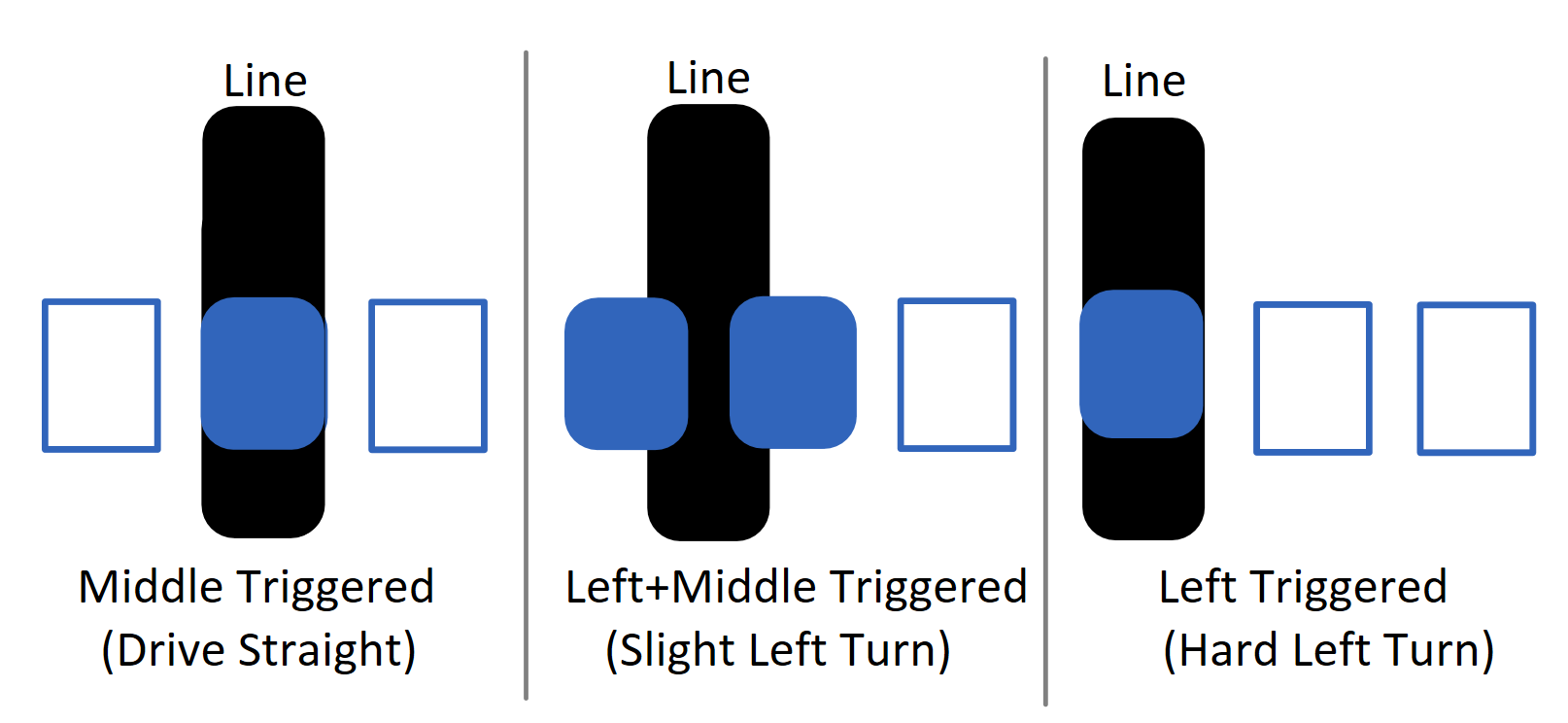
**Principle Function 2: Line Following**

For the second principle functionality of this project, we implemented a line following mechanism which allows the robot to guide itself along a designated path on the ground. The goal was to have the robot traverse a closed loop track of moderate, varying curvature as quickly as possible, without veering too far off from the path.

The three reflective optical sensors were used for this functionality. Each sensor detects the amount of visible light on the surface directly in front. Since the track path was a distinct line on the ground, the optical sensors allow the robot to distinguish the line from its background based on the amount of light. The three sensors were mounted facing the ground on a breadboard positioned at the front of the robot. They were spaced roughly 1 cm apart from each other, allowing the robot to recognize different sections of the line. We chose to mount the sensors far ahead at the front, rather than behind, so that our robot would have more time to turn in response to any changes in the line.

Our algorithm for this functionality was fairly simple in concept, although it required some tuning to get a smooth result. The main algorithm is to read from the left, middle, and right sensors continuously to determine the position of the line relative to the robot, then turn the robot towards the middle of the line accordingly to keep it on track. The black electrical tape used for the track returned double-digit values to the sensors, so we set a maximum threshold value of 100 to indicate when the line is detected. Our initial implementation used only three cases to adjust the robot’s heading. If the middle sensor was being read, the robot continued driving straight. Otherwise, if only the left or right sensors were reading the line, indicating the robot was about to head off track, the inner motor would slow down to turn the robot until the middle sensor detected the line again. This algorithm worked sufficiently well at low speeds and for paths of small curvature.

When testing our algorithm at higher speeds and on tighter curves however, the robot would have trouble turning on time and staying on track. In addition, with only three cases in our algorithm, the robot would wait too long before beginning to turn, thus having to constantly make hard adjustments and making the driving not very smooth. The solution was to add more cases in our algorithm and tune the turning mechanism. We added two middle cases for when the middle sensor and either the left or right sensor was being read, indicating that the robot is just beginning to head off track. This allowed our robot to make smaller adjustments earlier, and simulate the accuracy of having five separate sensors.



**Figure 2: Diagram showing three cases of adjusting the robot heading based on sensor readings.**

We also drastically reduced the speed of one motor when making a turn. Although this effectively stops the motor, it allows the robot to complete the turn faster and handle tighter curves. With these changes, the robot was able to drive smoothly at maximum speed without veering off track. We did not implement functionality to handle any intersections on the path as this was not required for the project.

**Master-Slave Communication**

The ability to switch between functionalities was an idea that our group had the desire to implement since the beginning of the project. This can be seen through how every functionality algorithm was designed to loop externally and does not depend on past inputs, reducing the number of global variables and memory usage. As we neared the completion of our principle functionalities, we realized that we were at a shortage of available pins to implement the buttons and LCD required for user interface.

To solve this issue, we decided to make use of a second Arduino in order to power and display text on the LCD and also read the two momentary switches. Using the Wire library the two Arduinos communicate over the I2C pins, A4 and A5, in a master-slave setup [10]. Grounds must also be shared in order to have a reference voltage between both devices. Arduinos may be powered individually, in series(5V into Vin), or in parallel which was our final configuration. This solution allows us the flexibility of another Arduino’s worth of pins and the ability to offload large calculations if we need to the slave Arduino. The slave Arduino, controlling the LCD, is always ready to receive inputs from the master; the master Arduino, is in charge of when transmission happens, both to and from the slave [11][12]. In our case, the master would send data when we wanted to display elements on the LCD and request data representing which momentary switches were pressed if any. The mode select toggle switch was left on the master Arduino as we wanted to avoid communications during the looping of algorithms which would introduce another potential area for bugs which was not tested in isolation.

During our isolated tests on a breadboard, in order to reduce latency during Arduino communications, we sent as few bytes as possible over the wire and left the logic to decode the byte on the slave; this is similar to how a CPU would decode an instruction before execution. As an example, suppose the slave receives one byte of data. This would tell the slave that this is a command to display LCD text during the selectMode() function as opposed to the speed and direction data of principle functionality one, which is sent as a two bytes at once. The value of the data would then tell the slave what to text display specifically. This is a much more efficient method as opposed to sending the entire String to be displayed. As the value of the momentary switches are the only pieces of information need to be sent from the slave, a single byte can represent this information whenever the master requests and the slave only needs debounce the buttons in its main loop.

Unexpected behaviour did occur as we incorporated the isolated setup onto the robot. This included problems initializing LCD, incorrect LCD display, and incorrect behaviour during functionalities. After double and triple checking our code, we realized that the issue had to due with instability in voltages due to the LCD backlight and how the power between the Arduinos was connected in series. We had two options: power both Arduinos individually and power them in parallel with the battery box. The second option required no additional mounting, and voltages were stable enough for our use. Another factor which we had to work around was updating displayed values too quickly on the LCD, especially real time speed. Although the slave Arduino could handle the volume of commands sent, we could not read the data displayed. Thus we decided to implement a piece of code using similar logic to the “blink without delay” code on Arduino’s website to limit the frequency of updates to the slave [13]. We settled on updating every 0.5 seconds as a good balance between accurate real time data and readability.

**Additional Functionality: Navigation Algorithm**

With the data from the sonar and the angle tracked by the servo, we can calculate the exact position of the obstacles detected by the sonar. However, this process neglects the fact that the sonar does not provide precise detection of an obstacle. Instead, the sonar will return data within the effective range of 30 degrees. Therefore, to be more precise about the position of the obstacles, we implemented a probability distribution model. Instead of mapping one position from one reading, the model will map out the distribution of the effect from one data to its effective area. Gaussian distribution will be more precise in this regard. Due to the constraint of time and the processing power from Arduino, we decided use a linear distribution model. We decided to collect 91 data points, one for every two degrees turned by the servo. Then, we calculate the effect of data on a 11 x 11 matrix.

The algorithm is as follows (more detailed version will be attached to appendix):

1. Given distance and angle, it will compute the centre of the affected area with the following computation:
2. After obtaining the centre position, algorithm will compute the terminal point of the effective range.
3. With the terminal point and effective range, the algorithm will map out the distribution effect, based on the geometry distribution of the segments from the effective range.

After obtaining the result from the algorithm, we map the result to a obstacles probability density map in processing. Each grid in the map represent a 15 cm by 15 cm grid in the real world, which is roughly the dimension of the robot. We chose the color code as follows:

* Orange: unexplored area (unreachable from sonar)
* White: low chance of obstacles (less than 30% chance to have a obstacle based on our algorithm)
* Red: high chance of obstacles (more than 30% chance to have a obstacle based on our algorithm)

The location of the robot will always be on (5,0) as that is our reference point for scanning. However, the result we got from this algorithm did not perform up to our original expectations. This is due to the nature of the ultrasonic sensor we are using. It creates random noise data during the scanning process and the accuracy of the data from this sensor is not satisfactory enough to create a precise mapping. The mechanism we implemented to avoid the noise is to have a larger delay for the sonar scan. Additionally, we have a threshold for the data to exclude data that is completely off.

**Additional Functionality: Infrared (IR) Communication**

In order to move the robot to our desired location, the team implemented remote control using IR. The IR receiver, TSOP38238, only accepts light at 38 kHz. For this reason, light from the surroundings would not affect the IR transmission. Since our IR transmitter was a Samsung BN59-01178W, our team needed to implement a way to detect the Samsung protocol. The Samsung protocol uses a series of 1s and 0s to send command and address to the receiver so the team needed to determine how long these 0s and 1s are on. From reference [9], 1 bits will be ON for 590 microseconds and OFF for 1690 microseconds while 0 bits will be ON for 590 microseconds and OFF for 590 microseconds. Using this information, our team could implement a loop that detects when a signal is pulsed from LOW to HIGH. Then when it is pulsed again from HIGH to LOW and the time for the signal is around 1690 microseconds, we know it is a 1-bit (from the Samsung Protocol) hence we can initialize our array value that represents bits to 1. We repeat this process in a loop until we get all 32 bits for the Samsung protocol. With that bit we can determine which button was pressed and can implement drive functions depending on which buttons were pressed on the remote.

For this robot, we used IR to configure seven keys from the remote to perform our commands. One button was used to drive the robot forward by changing the motor speed. A second button was used to turn the robot left 90 degrees by changing the directions of the motors and adding speed. A third button was used to turn the robot right 90 degrees by changing the directions of the motors and adding speed. A fourth button was used to drive the robot in reverse by changing the direction of the motors and changing motor speed. A fifth button was used to stop the robot from driving by turning off the motor. A sixth button was used to start the navigation algorithm by calling the function. The seventh button was used to Serial print the data so that it can be passed to Processing to draw.

A design consideration with the IR was to move the robot depending on how long the move button on the remote is pressed. This could not be done in this case because the “data” received from the remote will remain the same value until another button is pressed to update that value. Therefore, the team decided to implement a timed feature to drive and turn the robot instead.

**D. References and Bibliography**

Datasheets:

[1] SS443A Hall Effect Sensor - <http://sensing.honeywell.com/honeywell-sensing-ss400-series-product-sheet-009050-3-en.pdf>

[2] IR Receiver TSOP38238 -

<https://www.adafruit.com/datasheets/tsop382.pdf>

TCRT5000 Reflective Optical Sensor datasheet provided in lab handout.

DFRobot Motor Shield datasheet provided in lab handout.

DFRobot 2WD Mobile Platform assembly guide provided in lab handout.

Fritzing:

[3] DFRobot Motor Shield - <https://github.com/DFRobot/Fritzing-library>

[4] TCRT5000 Reflective Optical Sensor - <https://github.com/robertoostenveld/fritzing>

[5] HC-SR04 Ultrasonic Sensor - <http://fritzing.org/projects/hc-sr04-project>

Code:

[6] Arduino map function - <https://www.arduino.cc/en/Reference/Map>

[7] Servo sweep example - <https://www.arduino.cc/en/Tutorial/Sweep>

[8] Infrared - <https://learn.sparkfun.com/tutorials/ir-communication>

[9] Infrared - <http://www.techdesign.be/projects/011/011_waves.htm>

I2C Communications:

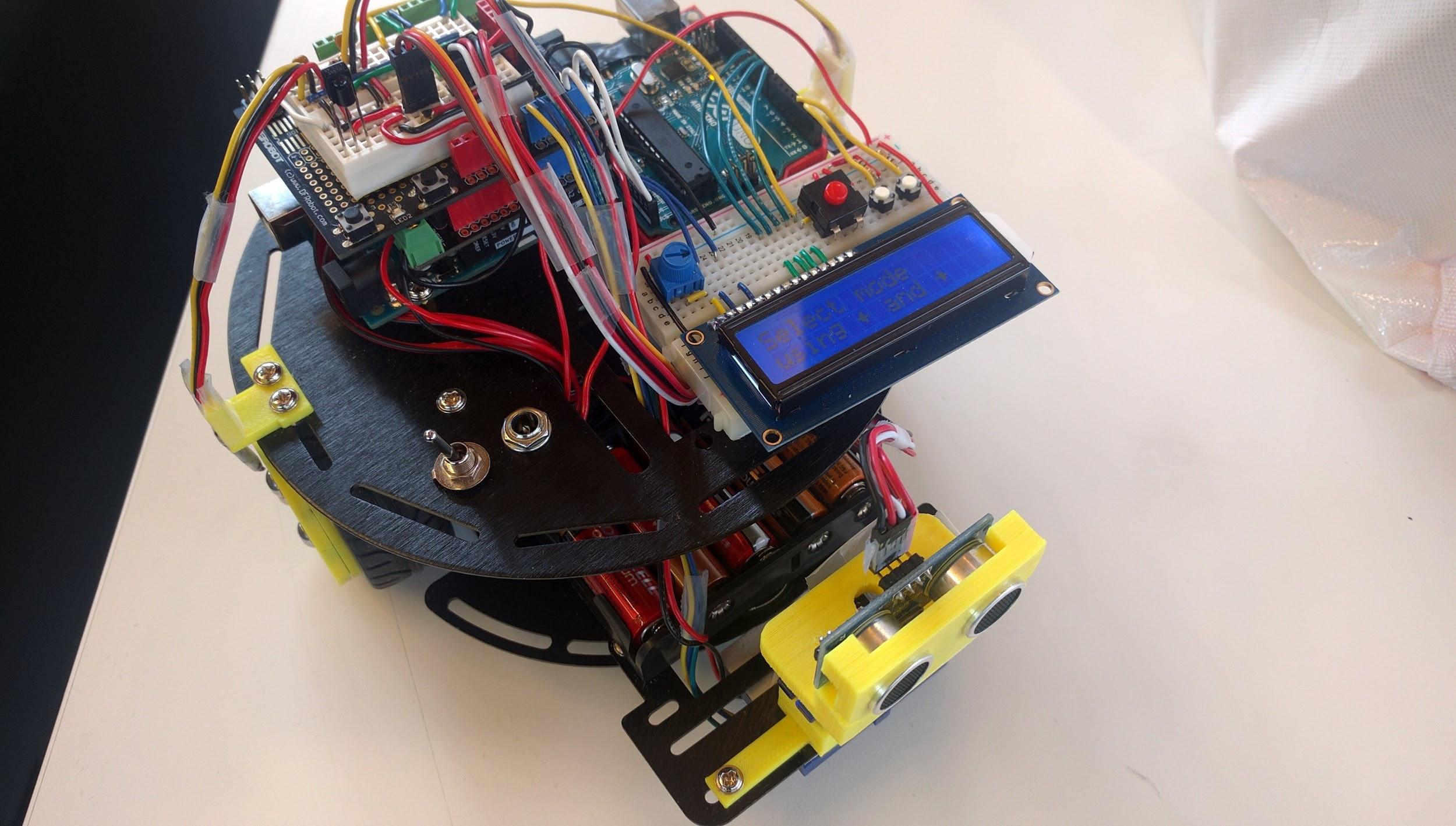
[10] Wire Library - <https://www.arduino.cc/en/Reference/Wire>

[11] Master writer example - <https://www.arduino.cc/en/Tutorial/MasterWriter>

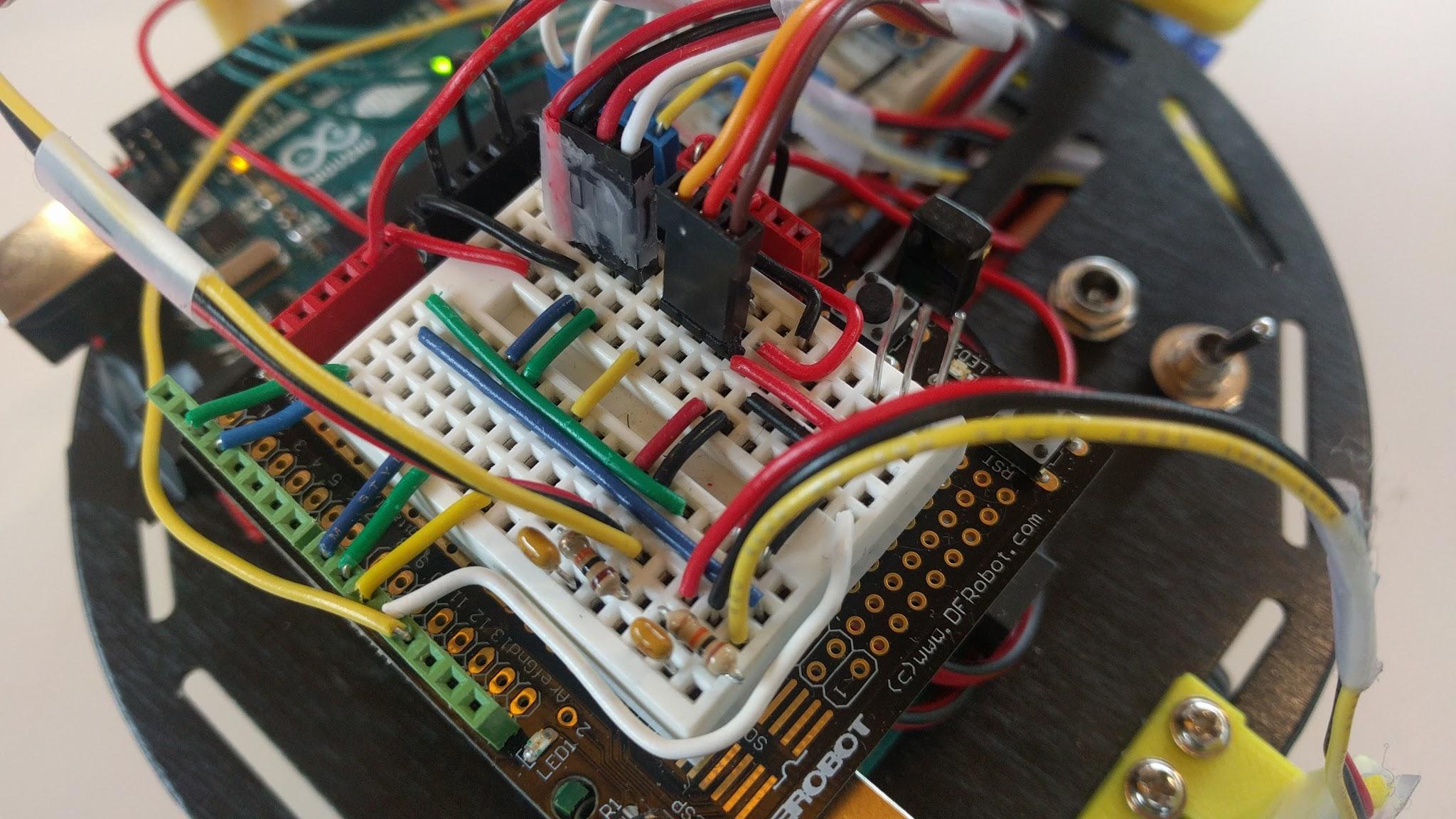
[12] Master reader example - <https://www.arduino.cc/en/Tutorial/MasterReader>

[13] Blink without delay - <https://www.arduino.cc/en/Tutorial/BlinkWithoutDelay>

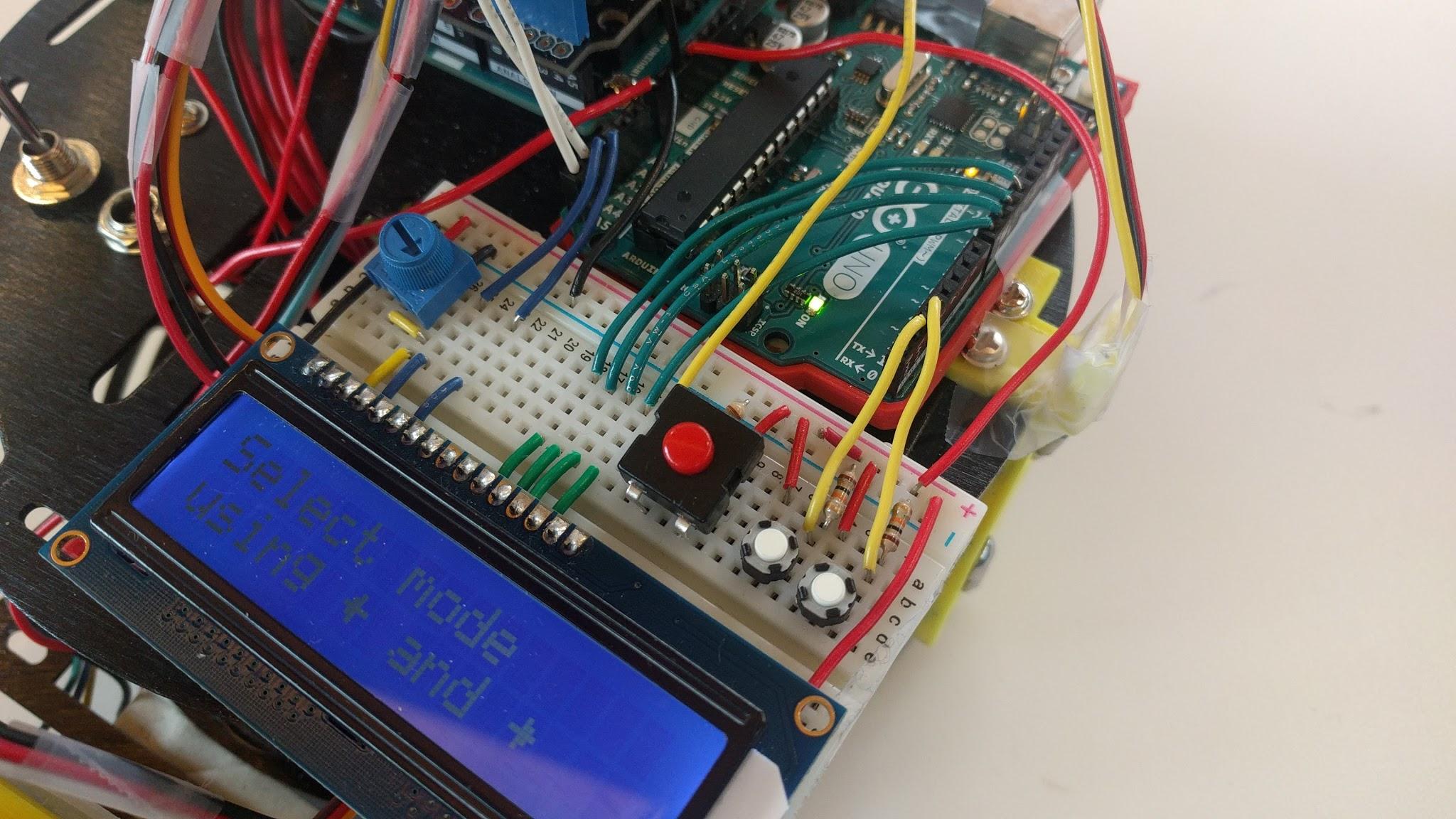
**Appendix A - Robot Pictures**



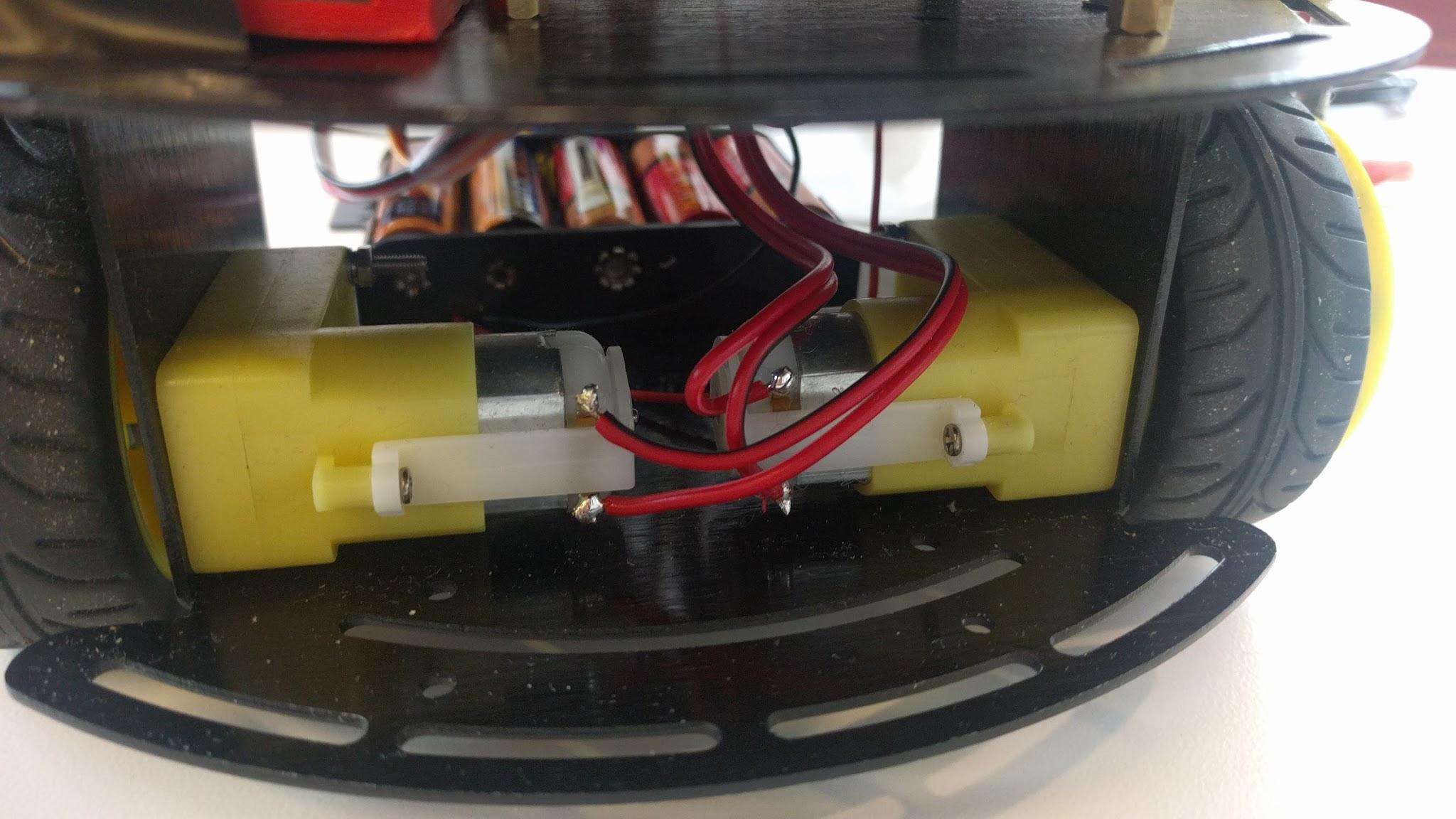
**Figure 3. Overview of the robot**



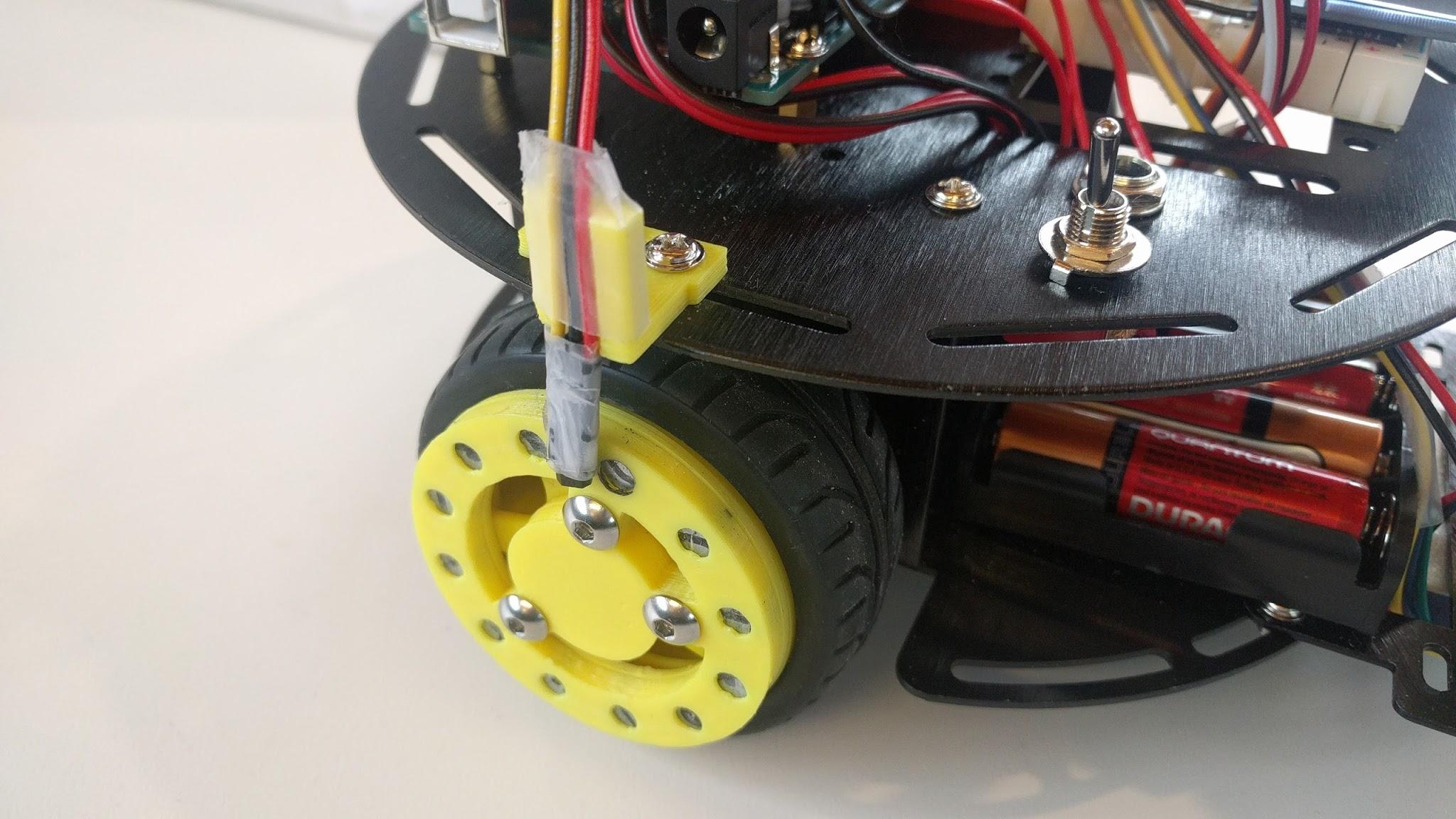
**Figure 4. Master Arduino Uno circuitry and shields**



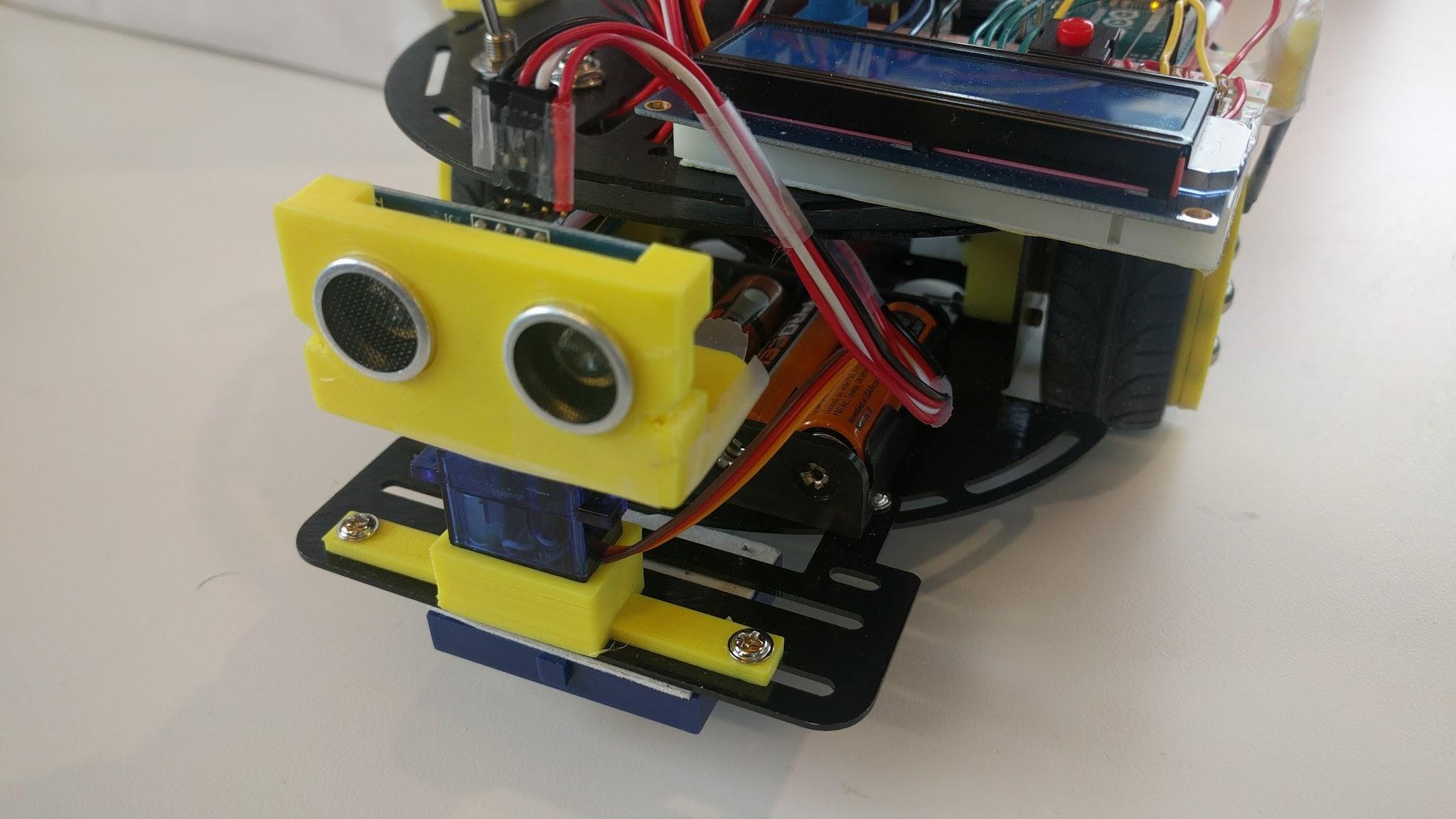
**Figure 5. Slave Arduino Uno, LCD module, and menu selection button setup**



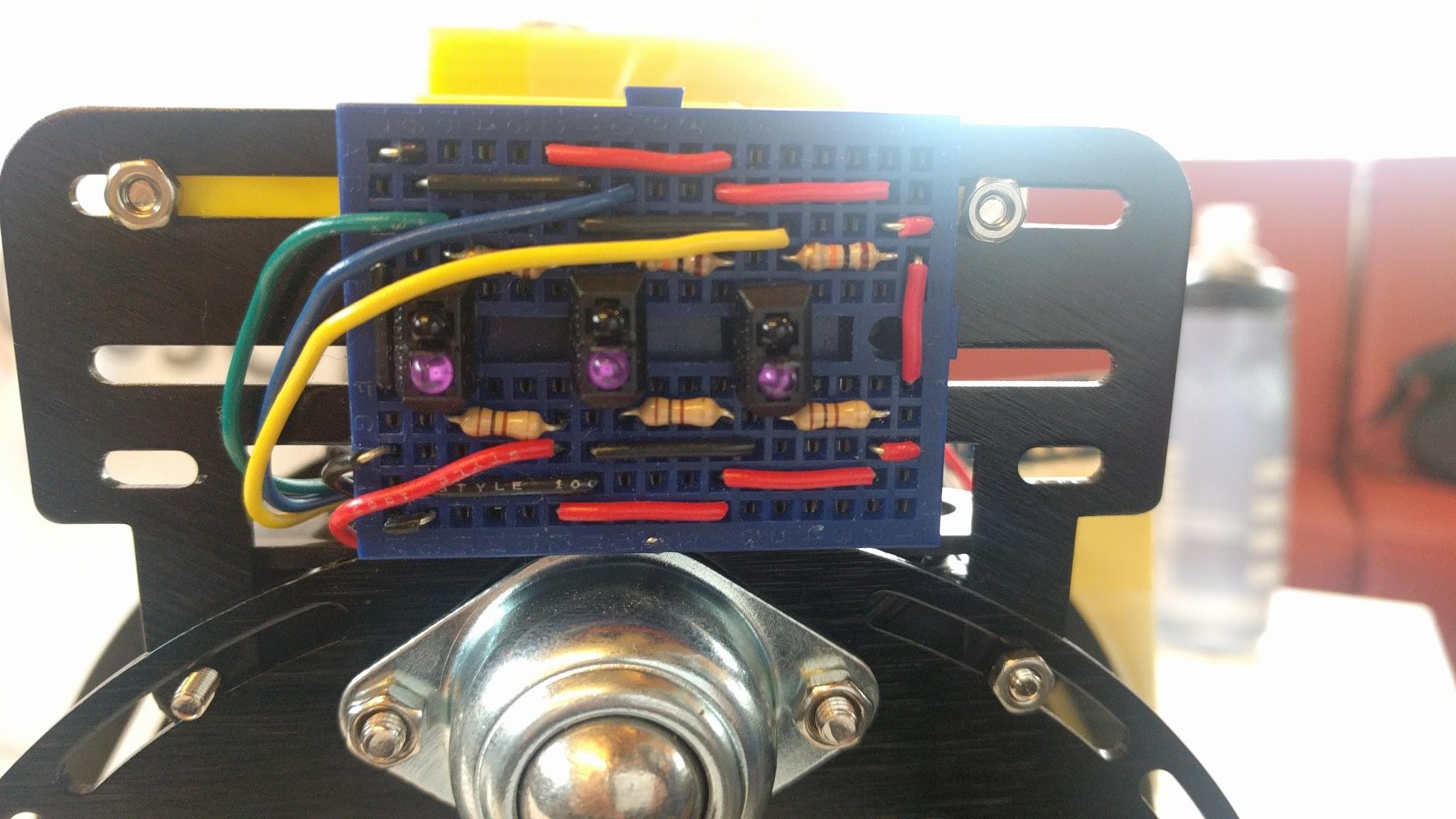
**Figure 6. Motor wiring**



**Figure 7. Hall effect sensor mount and wheel mount for magnets**



**Figure 8. Ultrasonic sensor and servo mount**



**Figure 9. Reflective optical sensor array underneath the robot**

**Appendix B - Code**

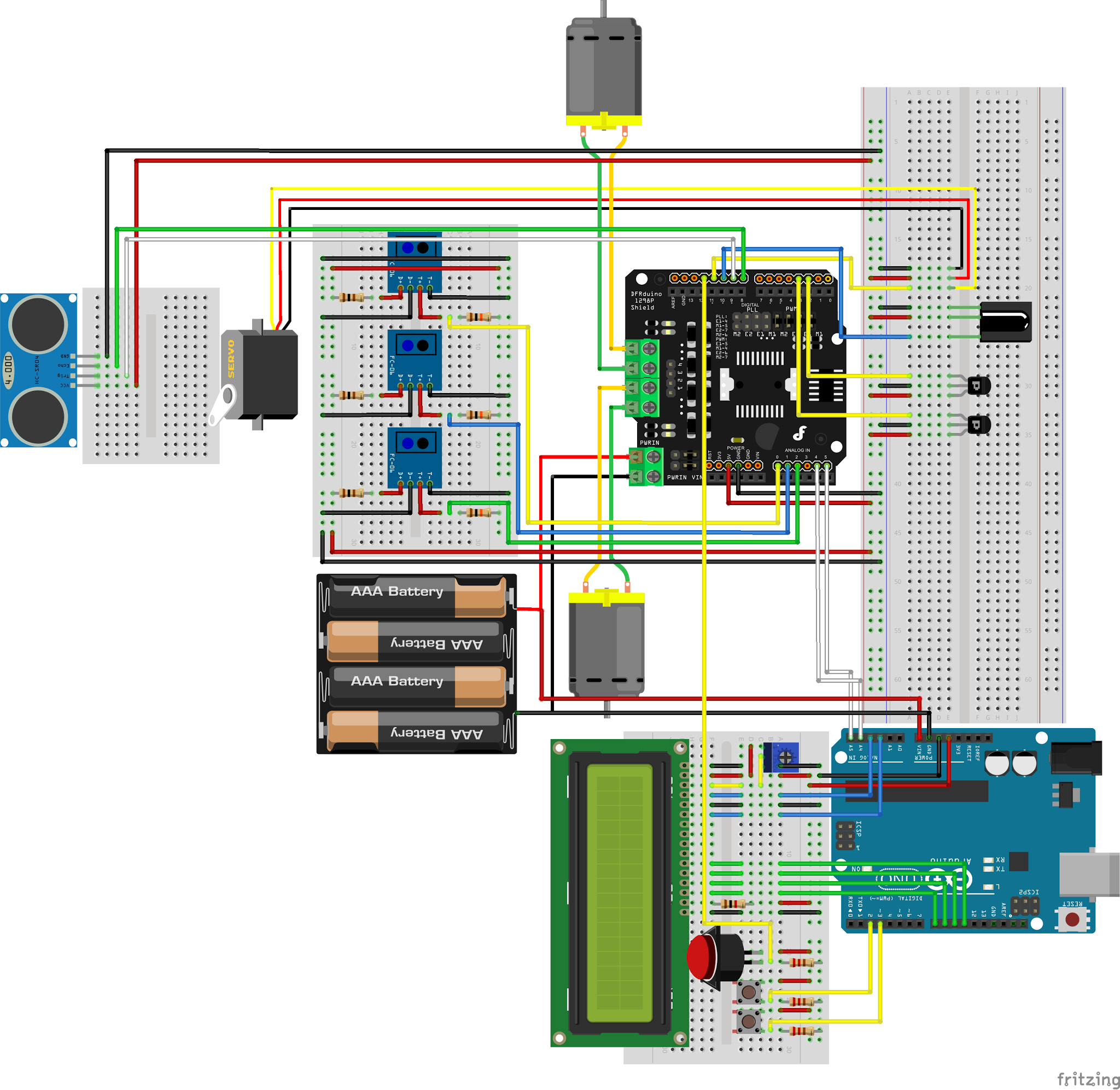
**Master Code**

|  |
| --- |
| #include <Wire.h> #include <Servo.h>  ////////////////////////////////////////////////////// // PIN LAYOUT // ////////////////////////////////////////////////////// #define LEFT\_DRIVE\_SPEED 5 #define LEFT\_DRIVE 4 #define RIGHT\_DRIVE\_SPEED 6 #define RIGHT\_DRIVE 7  #define LINE\_LEFT A0 #define LINE\_MIDDLE A1 #define LINE\_RIGHT A2  #define SERVO\_PIN 10 // Servo requires PWM pin Servo servo;  #define TRIG\_PIN 9 #define ECHO\_PIN 8 #define TEMP\_PIN A3 #define IRPIN 11  #define HALL\_L\_PIN 2 #define HALL\_R\_PIN 3  #define MODEBTN\_PIN 12 ////////////////////////////////////////////////////// // CONSTANTS // ////////////////////////////////////////////////////// #define HALL\_LEFT 0 #define HALL\_RIGHT 1  #define FORWARD 0 #define BACKWARD 1 #define LEFT 2 #define RIGHT 3  #define scanLeft 180 #define scanRight 0 #define centerServo 90  /\* Conversion constant from analogRead() output to temperature in C\*  Derived from LM35 temperature sensor datasheet  Voltage vs. Temperature 0-1500mV <--> 0-150C \*/ #define CONVERT\_TO\_TEMP 1/1024.0\*500  // Define motor speed control and distance constants #define DIST\_THRESH 5.0 #define MAP\_CONST 40 #define MIN\_MOVEMENT\_POWER 100 #define MAX\_SPEED 255  #define ticksPerCM 1.1765  #define LINE\_THRESHOLD 100  // Bonus boolean data\_ready = false; const int rowNum = 11; const int colNum = 11; const int gridLength = 15; String serialString = ""; float global\_maps[rowNum][colNum]; //a 3 dimensional matrices that represent the surrounding //0: clear //1: obstacle //(0,1) the chance that there is obstacle in that box //-1: unexplore //represent a point on the map typedef struct {  float x;  float y; } point;  #define pi 3.14159265358979323846 #define maxRange 200 //unit cm point robot\_location;  //buffer for data const int data\_number = 91;  ////////////////////////////////////////////////////// // VARIABLES // //////////////////////////////////////////////////////  // determines which functionality to run int mode = 0;  // slave variables byte prevDirToSlave = -1;  // debouncing variables int modeBtnState; int prevModeBtnState = LOW; unsigned long lastMBDebounceTime = 0; long debounceDelay = 50;  // drive variables volatile int hallLCount = 0; volatile int hallRCount = 0; double drive\_integral = 0;  void setup() {  // enable motor shield for motors  pinMode(LEFT\_DRIVE, OUTPUT);  pinMode(RIGHT\_DRIVE, OUTPUT);  pinMode(LEFT\_DRIVE\_SPEED, OUTPUT);  pinMode(RIGHT\_DRIVE\_SPEED, OUTPUT);   // enable ultrasonic sensor  pinMode(TRIG\_PIN, OUTPUT);  pinMode(ECHO\_PIN, INPUT);   // enable servos  servo.attach(SERVO\_PIN);   // enable interrupt service routine for hall effect sensors  attachInterrupt(digitalPinToInterrupt(HALL\_L\_PIN), updateHallL, CHANGE);  attachInterrupt(digitalPinToInterrupt(HALL\_R\_PIN), updateHallR, CHANGE);   // begin coms with slave Arduino  Wire.begin();    Serial.begin(9600);   delay(100); }  void loop() {  // choose mode  selectMode();  switch ( mode ) {  // drive and scan mode  case 1:  // enable interrupts for hall effect sensors  hallLCount = hallRCount = 0;  attachInterrupt(digitalPinToInterrupt(HALL\_L\_PIN), updateHallL, CHANGE);  attachInterrupt(digitalPinToInterrupt(HALL\_R\_PIN), updateHallR, CHANGE);  while ( debounceModeButton() == HIGH ) {  driveAndScan();  }  stopMotors();  break;   // line following mode  case 2:  // disable interrupts when line following  detachInterrupt(digitalPinToInterrupt(HALL\_L\_PIN));  detachInterrupt(digitalPinToInterrupt(HALL\_R\_PIN));  while ( debounceModeButton() == HIGH ) {  lineFollow(255);  }  stopMotors();  break;  case 3:  hallLCount = hallRCount = 0;  attachInterrupt(digitalPinToInterrupt(HALL\_L\_PIN), updateHallL, CHANGE);  attachInterrupt(digitalPinToInterrupt(HALL\_R\_PIN), updateHallR, CHANGE);  while ( debounceModeButton() == HIGH ) {  IR();  }  stopMotors();  break;  default:  stopMotors();  selectMode();  break;  } }  ////////////////////////////////////////////////////// // MODE SELECT FUNCTIONS // ////////////////////////////////////////////////////// /\*  Prompts the user to select the functionality to run on the robot \*/ void selectMode() {  sendToSlave(0); // prompt user to select mode on LCD on slave Arduino  byte dispCode = 0;  int button = -1;  while ( debounceModeButton() == LOW ) { // mode is confirmed when user toggles mode button  Wire.requestFrom(8, 1); // reads button presses on slave Arduino's LCD shield  while (Wire.available())  button = (int)Wire.read();  // increment or decrement the display code(looping between 1-2)depending on button presses  if ( button == 1 ) {  if ( dispCode >= 3 )  dispCode = 1;  else  dispCode++;  }  else if ( button == 0 ) {  if ( dispCode <= 1 )  dispCode = 3;  else  dispCode--;  }  sendToSlave(dispCode); // shows user what mode will start if mode button is toggled  delay(250);  //Serial.print(dispCode);  }  mode = (dispCode == 0) ? mode : dispCode; // resumes previous functionality if user toggles mode button without selecting a mode  sendToSlave(mode); }  /\*  Writes a single bytle to the slave Arduino over I2C \*/ void sendToSlave(byte x) {  Wire.beginTransmission(8);  Wire.write(x);  Wire.endTransmission(); }  /\*  Tells the slave Arduino to display the given speed and direction on the LCD \*/ void sendSpeedToSlave(byte speed, byte direction) {  Wire.beginTransmission(8);  Wire.write(speed);  Wire.write(direction);  Wire.endTransmission(); }  void sendLineFollowToSlave(byte direction) {  Wire.beginTransmission(8);  Wire.write(direction);  Wire.write(0);  Wire.write(0);  Wire.endTransmission(); }  /\*  Debounces the mode button (toggle switch) \*/ int debounceModeButton() {  int reading = digitalRead(MODEBTN\_PIN);  if ( reading != prevModeBtnState ) {  lastMBDebounceTime = millis();  }  if ( (millis() - lastMBDebounceTime) >= debounceDelay ) {  if ( reading != modeBtnState )  modeBtnState = reading;  }  prevModeBtnState = reading;  return modeBtnState; }  ////////////////////////////////////////////////////// // DRIVE FUNCTIONS // //////////////////////////////////////////////////////  // Uses P to lower/increase motor speed to match the rotation speed to drive straight // Needs to be called in a while loop, does not stop the motors void drive\_P\_Straight(int direction, int speed) {  digitalWrite(LEFT\_DRIVE, (direction == 2) ? LOW : HIGH);  digitalWrite(RIGHT\_DRIVE, (direction == 3) ? HIGH : LOW);   // PI constants and error calculation  double Kp = 70;  double Ki = 0.1;  double dt = 0.01;  double error = (hallRCount - hallLCount) \* Kp;  drive\_integral = drive\_integral + error \* dt;   double leftSpeed = 0;  double rightSpeed = 0;   // Calculate left motor speed based on error  if (speed + error + Ki \* drive\_integral > 255)  leftSpeed = 255;  else if (speed + error + Ki \* drive\_integral < MIN\_MOVEMENT\_POWER)  leftSpeed = MIN\_MOVEMENT\_POWER;  else  leftSpeed = speed + (error) - Ki \* drive\_integral;   // Calculate right motor speed based on error  if (speed - error - Ki \* drive\_integral > 255)  rightSpeed = 255;  else if (speed - error - Ki \* drive\_integral < MIN\_MOVEMENT\_POWER)  rightSpeed = MIN\_MOVEMENT\_POWER;  else  rightSpeed = speed - (error) - Ki \* drive\_integral;   /\*Set motor speed  Serial.print(leftSpeed);  Serial.print(" ");  Serial.print(rightSpeed);  Serial.print(" ");  Serial.print(hallLCount);  Serial.print(" ");  Serial.print(hallRCount);  Serial.print(" ");  Serial.println(error\*Ki\*drive\_integral);  // \*/   // Set the motor speed  analogWrite(LEFT\_DRIVE\_SPEED, leftSpeed);  analogWrite(RIGHT\_DRIVE\_SPEED, rightSpeed); }  void turn\_Sensored(int direction, int speed, int distance) {  // Reset hall effect sensor counts  hallLCount = hallRCount = 0;   // Set direction  digitalWrite(LEFT\_DRIVE, (direction == LEFT) ? LOW : HIGH);  digitalWrite(RIGHT\_DRIVE, (direction == RIGHT) ? HIGH : LOW);   // Set motor speed  analogWrite(LEFT\_DRIVE\_SPEED, speed);  analogWrite(RIGHT\_DRIVE\_SPEED, speed);  sendSpeedToSlave(speed, direction);   // Run motors until hall effect count reaches distance  while (hallLCount < distance) {}   // Stop motors  analogWrite(LEFT\_DRIVE\_SPEED, 0);  analogWrite(RIGHT\_DRIVE\_SPEED, 0); }  // Drive without acceleration or decleration, Speed 0-255, timed // Sets the motors, waits and then stops the motors void drive\_Timed(int direction, int speed, int time) {  // Set direction  digitalWrite(LEFT\_DRIVE, (direction == 2) ? LOW : HIGH);  digitalWrite(RIGHT\_DRIVE, (direction == 3) ? HIGH : LOW);   // Set motor speed  analogWrite(LEFT\_DRIVE\_SPEED, speed);  analogWrite(RIGHT\_DRIVE\_SPEED, speed);   delay(time);   // Stop motors  analogWrite(LEFT\_DRIVE\_SPEED, 0);  analogWrite(RIGHT\_DRIVE\_SPEED, 0); }  // Stop motors void stopMotors() {  analogWrite(LEFT\_DRIVE\_SPEED, 0);  analogWrite(RIGHT\_DRIVE\_SPEED, 0); }  /\*  Update hall effect sensor count every time  the Arduino receives an interrupt from either  hall effect sensor. \*/ void updateHallL() {  hallLCount++; } void updateHallR() {  hallRCount++; }   ////////////////////////////////////////////////////// // PRINCIPLE FUNCTION 1 // ////////////////////////////////////////////////////// /\*  Drive the robot up to an object up to a predetermined threshold.  Once the robot reaches the distance threshold,  the robot will scan its left and right side  and select a direction to turn to. \*/ void driveAndScan() {  // Get ultrasonic distance  double initDistance = 0.0;   // Set baseline speed  double motorSpeed = 150;  sendSpeedToSlave(motorSpeed, FORWARD);   // Get the distance  do {  initDistance = getUltrasonic();  drive\_P\_Straight(FORWARD, motorSpeed);  } while (initDistance >= 500 || initDistance == 0.0);  //Serial.println(distance);   // Calculate number for ticks to reach the required distance  double requiredTicks = (initDistance - DIST\_THRESH) \* ticksPerCM;   drive\_integral = 0;  hallLCount = hallRCount = 0;  unsigned long prevMillis = 0;  // Run loop until robot has reached the required distance  while (hallLCount < requiredTicks)  {  // Call getMotorSpeed to calculate speed based on distance traveled  motorSpeed = getMotorSpeed(initDistance - (hallLCount / ticksPerCM));   //Serial.print(initDistance\*ticksPerCM - ((hallLCount + hallRCount) / 2.0));  //Serial.print(" ");  //Serial.println(motorSpeed);  //Serial.print(initDistance - ((hallLCount + hallRCount) / 2.0 / ticksPerCM));  //Serial.print(" ");  //Serial.println(motorSpeed);   drive\_P\_Straight(FORWARD, motorSpeed);   // update the speed displayed on LCD every 500ms  unsigned long currMillis = millis();  if ( currMillis - prevMillis >= 500 ) {  prevMillis = currMillis;  sendSpeedToSlave(motorSpeed, FORWARD);  }  }   // Stop motors  stopMotors();  sendSpeedToSlave(0, 4);   // Find a direction to turn to, if scanSurroundings returns -1, scan again  int turnDir;  do  {  turnDir = scanSurroundings();  } while (turnDir == -1);   delay(500);   // Turn left or right based on clearance of each side  turn\_Sensored(turnDir, 125, 11);  sendSpeedToSlave(0, FORWARD);  delay(500); }  /\*  Calculate the motor speed based on  the measured ultrasonic sensor value and  the distance an object is away from the robot.  Param: distance - measured distance from ultrasonic  Return: motorVal - updated motor speed \*/ int getMotorSpeed(double distance) {  // Set motor value to MAX\_SPEED  int motorVal = MAX\_SPEED;   // Check if robot has not approached object  if (distance > DIST\_THRESH)  {  // Calculate new motor value based on distance of object using map() function  motorVal = map(distance, DIST\_THRESH, MAP\_CONST, MIN\_MOVEMENT\_POWER, 210);  }  else  {  motorVal = 0;  }  return motorVal; }  /\*  Check the left and right side of the robot  using the ultrasonic sensor on a servo mount.  Determine which direction to turn based on  the further of the two distances.  Returns: 0 - to turn the robot left  1 - to turn the robot right  2 - scan again \*/ int scanSurroundings() {  // Set the servo to scan left side  servo.write(scanLeft);  delay(1500);   // Get distance of object on the left side  double distanceLeft = getUltrasonic();  delay(250);   // Set the servo to scan right side  servo.write(scanRight);  delay(1500);   // Get distance of object on the right side  double distanceRight = getUltrasonic();  delay(250);   // Reset servo to face forward  servo.write(centerServo);   // If left and right distances are garbage, tell main loop to scan again  if (distanceLeft > 500 && distanceRight > 500)  return -1;  // Determine which direction to turn in based on surroundings  if (distanceLeft > distanceRight)  return LEFT;  else  return RIGHT; }  // Return distance in centimetres double getUltrasonic() {  //double temp = analogRead(tempPin) \* CONVERT\_TO\_TEMP;  double temp = 21; // Standard temp in celsius  double speedOfSound = (331.5 + (0.6 \* temp)) \* 0.0001; // cm/microsecond   // Ping the ultrasonic   digitalWrite(TRIG\_PIN, HIGH);  delayMicroseconds(10);  digitalWrite(TRIG\_PIN, LOW);   // Return pulse from echo  return (pulseIn(ECHO\_PIN, HIGH) / 2.0) \* speedOfSound;  // return (ultraSonReading/2.0)\*speedOfSound; }  void servoTest() {  servo.write(scanLeft);  delay(2000);  servo.write(centerServo);  delay(2000);  servo.write(scanRight);  delay(2000); }  ////////////////////////////////////////////////////// // PRINCIPLE FUNCTION 2 // ////////////////////////////////////////////////////// /\*  Send the robot to trace a line. Each optical sensor's value  is read and the speed of each wheel is determined based on  which optical sensors detect the line. \*/ void lineFollow(int motorValue) {  int speed = motorValue;  int L\_LINE = analogRead(LINE\_LEFT);  int M\_LINE = analogRead(LINE\_MIDDLE);  int R\_LINE = analogRead(LINE\_RIGHT);  byte dirToSlave;    if (R\_LINE > LINE\_THRESHOLD)  {  // Swerve right  digitalWrite(LEFT\_DRIVE, HIGH);  digitalWrite(RIGHT\_DRIVE, LOW);   // Set motor speed (Right)  analogWrite(LEFT\_DRIVE\_SPEED, speed);  analogWrite(RIGHT\_DRIVE\_SPEED, 0);  dirToSlave = 0;  }  else if (L\_LINE > LINE\_THRESHOLD)  {  // Swerve left  digitalWrite(LEFT\_DRIVE, HIGH);  digitalWrite(RIGHT\_DRIVE, LOW);   // Set motor speed (Right)  analogWrite(LEFT\_DRIVE\_SPEED, 0);  analogWrite(RIGHT\_DRIVE\_SPEED, speed);  dirToSlave = 2;  }  else if (R\_LINE > LINE\_THRESHOLD && M\_LINE > LINE\_THRESHOLD)  {  // Slow turn right  digitalWrite(LEFT\_DRIVE, HIGH);  digitalWrite(RIGHT\_DRIVE, LOW);   // Set motor speed (Right)  analogWrite(LEFT\_DRIVE\_SPEED, speed - 75);  analogWrite(RIGHT\_DRIVE\_SPEED, 0);  dirToSlave = 3;  }  else if (L\_LINE > LINE\_THRESHOLD && M\_LINE > LINE\_THRESHOLD)  {  // Slow turn left  digitalWrite(LEFT\_DRIVE, HIGH);  digitalWrite(RIGHT\_DRIVE, LOW);   // Set motor speed (Right)  analogWrite(LEFT\_DRIVE\_SPEED, 0);  analogWrite(RIGHT\_DRIVE\_SPEED, speed - 75);  dirToSlave = 4;  }  else if (M\_LINE > LINE\_THRESHOLD)  {  // Drive foward  digitalWrite(LEFT\_DRIVE, HIGH);  digitalWrite(RIGHT\_DRIVE, LOW);   // Set motor speed  analogWrite(LEFT\_DRIVE\_SPEED, speed);  analogWrite(RIGHT\_DRIVE\_SPEED, speed);  dirToSlave = 1;  }  // Update the LCD data if it differs from previously  if( dirToSlave != prevDirToSlave ) {  sendLineFollowToSlave(dirToSlave);  prevDirToSlave = dirToSlave;   } }  void lineTest() {  int leftVal = analogRead(LINE\_LEFT);  int midVal = analogRead(LINE\_MIDDLE);   Serial.print(leftVal);  Serial.print(" ");  int rightVal = analogRead(LINE\_RIGHT);   // LEFT < LINE\_THRESHOLD? 1 : 0;  // MIDDLE = MIDDLE < LINE\_THRESHOLD? 1 : 0;  // RIGHT = RIGHT < LINE\_THRESHOLD? 1 : 0;  Serial.print(midVal);  Serial.print(" ");  Serial.println(rightVal); }  ////////////////////////////////////////////////////// // ADDITIONAL FUNCTIONALITY // ////////////////////////////////////////////////////// /\*  IR Remote code to decode signals from remote control \*/ void IR() {  //variables for IR  boolean binary[32]; //array for Samsung 32 bits  int data = 0; //data to determine the button pressed   //While pulseIn to go from LOW to HIGH  while (pulseIn(IRPIN, LOW) < 4000);  for (int i = 0; i < 32; i++)  {  //According to Samsung Protocol, the 1-bit will stay as HIGH for about 1000 microseconds  if (pulseIn(IRPIN, HIGH) > 1000)  //If arund 1000 microseconds, we know it is a 1-bit so add to binary array.  binary[i] = 1;  else  binary[i] = 0;  }  //Decode data from binary into data  for (int i = 0; i < 8; i++)  {  if (binary[i + 16] == 1)  data += 1 << i;  }   //if detect button to drive forward  // 96 is a key on the remote  if (data == 96)  {  //Drive forward  digitalWrite(LEFT\_DRIVE, HIGH);  digitalWrite(RIGHT\_DRIVE, LOW);  analogWrite(LEFT\_DRIVE\_SPEED, 180);  analogWrite(RIGHT\_DRIVE\_SPEED, 200);  }  //if detect button to turn left  // 101 is a key on the remote  if (data == 101)  {  //Turn left  turn\_Timed(LEFT, 255, 300);  }  //If detect button to turn right  // 98 is a key on the remote  if (data == 98)  {  //turn right  turn\_Timed(RIGHT, 255, 300);  }  //If detect button to drive reverse  // 97 is a key on the remote  if (data == 97)  {  //Drive reverse  digitalWrite(LEFT\_DRIVE, LOW);  digitalWrite(RIGHT\_DRIVE, HIGH);  analogWrite(LEFT\_DRIVE\_SPEED, 200);  analogWrite(RIGHT\_DRIVE\_SPEED, 200);  }  //If detect button to stop robot  //104 is a key on the remote  if (data == 104) {  //Stop the motors  analogWrite(LEFT\_DRIVE\_SPEED, 0);  analogWrite(RIGHT\_DRIVE\_SPEED, 0);  }  // Press button to start navigation algorithm  // 71 is a key on the remote  if (data == 71) {  completed\_mapping\_algorithm();  data\_ready = true;  sendToSlave(4);  }  // Prints data onto Serial Monitor  // 73 is a key on the remote  if (data == 73 && data\_ready) {  printData();  data\_ready = false;  }   delay(50);  }  //Function to turn robot left or right void turn\_Timed(int direction, int speed, int time) {  int turnDirection;  // Set direction  if (direction == LEFT) {  //direction is Left  turnDirection = 0;  }  if (direction == RIGHT) {  //direction is Right  turnDirection = 1;  }  //Set the wheels according to the direction  digitalWrite(LEFT\_DRIVE, (direction == LEFT) ? LOW : HIGH);  digitalWrite(RIGHT\_DRIVE, (direction == RIGHT) ? HIGH : LOW);   // Set motor speed to turn  analogWrite(LEFT\_DRIVE\_SPEED, speed);  analogWrite(RIGHT\_DRIVE\_SPEED, speed);   delay(time);   // Stop motors  analogWrite(LEFT\_DRIVE\_SPEED, 0);  analogWrite(RIGHT\_DRIVE\_SPEED, 0); }  ////////////////////////////////////////////////////// // NAVIGATION ALGORITHM // ////////////////////////////////////////////////////// //the comments for the following code is not detailed due to  //the complexity of the algorithm. For detail comments on the algorithm //refer to the algorithm appendix in the report(Asked Professor Farshid)   void completed\_mapping\_algorithm() {  init\_map();//initialize the array and the robot location  float sonarData[data\_number] = {0};  float angleData[data\_number] = {0};   // receive\_data(sonarData, angleData);  scan\_in\_front(sonarData, angleData);  probability\_distribution\_model(sonarData, angleData); }  //initialize the map void init\_map() {  int row, col;  robot\_location = {0.0, 5.0};  for (row = 0; row < rowNum; row++) {  for (col = 0; col < colNum; col++) {  global\_maps[row][col] = -1; //initialized the array to -1 to indice unexplored  }  } }  //for printing data to Serial monitor void printData() {  int row, col;  for (row = 0; row < rowNum; row++) {  for (col = 0; col < colNum; col++) {  float toPrint = global\_maps[row][col];   //Serial.println(String(toPrint));  if (col == 0) {  serialString = String(toPrint);  }  else {  serialString = serialString + "," + String(global\_maps[row][col]);  }  }  Serial.print(serialString + ",0");  Serial.print("\n");  delay(1000);  } }  //test void check\_radian\_degree() {  float result = sin(pi / 2);  if (abs(result - 1) < 0.1) {  Serial.print("the sin is in radia \n");  }  else {  Serial.print("the sin is in degree \n");  } } //given a distance and angle to compute its probability distribution effect to the map void probability\_distribution\_model(float sonarData[], float angleData[]) {  int i;  float effectRange;  point left\_terminal;  point right\_terminal;  point center;  for (i = 0; i < data\_number; i++) {  effectRange = compute\_effective\_range(sonarData[i]);  center = compute\_centeral\_point(sonarData[i], angleData[i]);  left\_terminal = compute\_terminal\_lpoint(center, angleData[i], effectRange);  right\_terminal = compute\_terminal\_rpoint(center, angleData[i], effectRange);  float effective\_x\_range = compute\_effective\_range\_x(effectRange, angleData[i]);  float effective\_x\_increment = effective\_x\_range / gridLength;  float effective\_y\_range = compute\_effective\_range\_y(effectRange, angleData[i]);  float effective\_y\_increment = effective\_y\_range / gridLength;  distribution\_helper\_fn(center, left\_terminal, right\_terminal, angleData[i], effective\_x\_increment, effective\_y\_increment);  } }  //compute effective range float compute\_effective\_range(float distance) {  return 2 \* distance \* tan(degree\_to\_radian(15)); }  //compute effective range x component float compute\_effective\_range\_x(float range, float angle\_a) {  if (angle\_a<90 && angle\_a>0) {  return range \* cos(degree\_to\_radian(90 - angle\_a));  }  else if (angle\_a > 90 && angle\_a < 180) {  return range \* cos(degree\_to\_radian(angle\_a - 90));  }  else if (angle\_a == 90) {  return range;  }  else if (angle\_a == 180) {  return 0;  }  else {  return 0;  } }   //compute effective range y component float compute\_effective\_range\_y(float range, float angle\_a) {  if (angle\_a<90 & angle\_a>0) {  return range \* sin(degree\_to\_radian(90 - angle\_a));  }  else if (angle\_a > 90 && angle\_a < 180) {  return range \* sin(degree\_to\_radian(angle\_a - 90));  }  else if (angle\_a == 90) {  return 0;  }  else if (angle\_a == 180) {  return range / 2.0;  }  else {  return range / 2.0;  } }  //act as a helper function for probability\_distribution\_model() void distribution\_helper\_fn(point center, point left\_t, point right\_t, float angle, float x\_incre, float y\_incre) {  float x = left\_t.x;  float y = left\_t.y;  matrix\_marked\_explored(center);  matrix\_marked\_explored(left\_t);  matrix\_marked\_explored(right\_t);  if (angle > 0 && angle < 90) {  for (; x < floor(right\_t.x);) {  global\_maps[(int)floor(y)][(int)floor(x)] += 1 / x\_incre;  x = cap(x + x\_incre);  y = cap(y + y\_incre);  //Serial.print("I am stucking at helper");  }  }  else if (angle >= 90 && angle < 180) {  for (; x < floor(right\_t.x);) {  global\_maps[(int)floor(y)][(int)floor(x)] += 1 / x\_incre;  x = cap(x + x\_incre);  y = cap(y - y\_incre);  }  }  else if (angle == 0) {  for (; y < floor(right\_t.y);) {  global\_maps[(int)floor(y)][(int)floor(x)] += 1 / y\_incre;  x = cap(x);  y = cap(y + y\_incre);  }  }  else {  for (; y > floor(right\_t.y);) {  global\_maps[(int)floor(y)][(int)floor(x)] += 1 / y\_incre;  x = cap(x);  y = cap(y - y\_incre);  }  } }  //initilize sub matrix void matrix\_marked\_explored(point p1) {  int lower\_bound\_x, lower\_bound\_y, upper\_bound\_x, upper\_bound\_y;  lower\_bound\_y = robot\_location.y;  upper\_bound\_y = p1.y;  //determine the lower and upper bound  if (p1.x >= robot\_location.x) {  lower\_bound\_x = robot\_location.x;  upper\_bound\_x = p1.x;  }  else {  lower\_bound\_x = p1.x;  upper\_bound\_x = robot\_location.x;  }   int x, y;  for (x = lower\_bound\_x; x <= upper\_bound\_x; x++) {  for (y = lower\_bound\_y; y <= upper\_bound\_y; y++) {  if (global\_maps[y][x] == -1) global\_maps[y][x] = 0; // marked as explored  }  } }   //receive data from another arduino void receive\_data(float sonarData[], float angleData[]) {  int i;  for (i = 0; i < data\_number; i++ ) {  sonarData[i] = random\_distance();  angleData[i] = random\_angle();  } }  //for testing: random float generator float random\_distance() {  float range = 200;  return ((float)rand() / (float)(RAND\_MAX)) \* range + 50; }  float random\_angle() {  float range = 180;  return ((float)rand() / (float)(RAND\_MAX)) \* range; }  //compute the left terminal point point compute\_terminal\_lpoint(point center, float angle\_a, float range) {  point terminal\_point = {0.0, 0.0};  if (angle\_a<90 & angle\_a>0) {  terminal\_point.x = cap(center.x - range \* cos(degree\_to\_radian(90 - angle\_a)) / gridLength);  terminal\_point.y = cap(center.y - range \* sin(degree\_to\_radian(90 - angle\_a)) / gridLength);  }  else if (angle\_a > 90 && angle\_a < 180) {  terminal\_point.x = cap(center.x - range \* cos(degree\_to\_radian(angle\_a - 90)) / gridLength);  terminal\_point.y = cap(center.y + range \* sin(degree\_to\_radian(angle\_a - 90)) / gridLength);  }  else if (angle\_a == 90) {  terminal\_point.x = cap(center.x - range / (2 \* gridLength));  terminal\_point.y = center.y;  }  else if (angle\_a == 0) {  terminal\_point.x = center.x;  terminal\_point.y = center.y;  }  else {  terminal\_point.x = center.x;  terminal\_point.y = cap(center.y + range / (2 \* gridLength));  }  return terminal\_point; }  //compute the right terminal point point compute\_terminal\_rpoint(point center, float angle\_a, float range) {  point terminal\_point = {0.0, 0.0};  if (angle\_a < 90 && angle\_a > 0) {  terminal\_point.x = cap(center.x + range \* cos(degree\_to\_radian(90 - angle\_a)) / gridLength);  terminal\_point.y = cap(center.y + range \* sin(degree\_to\_radian(90 - angle\_a)) / gridLength);  }  else if (angle\_a > 90 && angle\_a < 180) {  terminal\_point.x = cap(center.x + range \* cos(degree\_to\_radian(angle\_a - 90)) / gridLength);  terminal\_point.y = cap(center.y - range \* sin(degree\_to\_radian(angle\_a - 90)) / gridLength);  }  else if (angle\_a == 90) {  terminal\_point.x = cap(center.x + range / (2 \* gridLength));  terminal\_point.y = center.y;  }  else if (angle\_a == 180) {  terminal\_point.x = center.x;  terminal\_point.y = center.y;  }  else {  terminal\_point.x = center.x;  terminal\_point.y = cap(center.y + range / (2 \* gridLength));  }  return terminal\_point; }  //tested //compute the central point point compute\_centeral\_point(float angle, float distance) {  point center = {0.0, 0.0};  if (angle > 90 && angle < 180) {  center.x = cap(robot\_location.x - distance \* cos(degree\_to\_radian(180 - angle)) / gridLength);  center.y = cap(robot\_location.y + distance \* sin(degree\_to\_radian(180 - angle)) / gridLength);  }  else if (angle<90 & angle>0) {  center.x = cap(robot\_location.x + distance \* cos(degree\_to\_radian(angle)) / gridLength);  center.y = cap(robot\_location.y + distance \* sin(degree\_to\_radian(angle)) / gridLength);  }  else if (angle == 90) {  center.x = cap(robot\_location.x);  center.y = cap(robot\_location.y + distance / gridLength);  }  else if (angle == 180) {  center.x = cap(robot\_location.x + distance / gridLength);  center.y = cap(robot\_location.y);  }  else {  center.x = cap(robot\_location.x - distance / gridLength);  center.y = cap(robot\_location.y);  }  return center; }   //cap the point value inside the map float cap(float x) {  if (x < 0) return 0;  if (x > 11) return 11;  return x; }  //convert degree to radian float degree\_to\_radian(float degree) {  return degree / 180.0 \* pi; }  //scan the 180 degree in fron of the robot void scan\_in\_front(float sonarData[], float angleData[]) {  // start servo in left pos  servo.write(scanRight);  delay(200);  float previous\_data = getUltrasonic\_bonus();  delay(200);  while (previous\_data > maxRange) {  previous\_data = getUltrasonic\_bonus();  delay(200);  }  // sweep ultrasonic 180 degrees on servo  int i;  float currentReading;  for (i = 0; i < data\_number; i++)  {  // get distance  currentReading = getUltrasonic\_bonus();  delay(200);  if (currentReading < maxRange && currentReading != 0) {  sonarData[i] = currentReading;  // Serial.println(sonarData[i], 4);  // set servo to new angle  angleData[i] = i \* 2.0;  servo.write(i \* 2);  delay(200);  previous\_data = currentReading;  }  else {  sonarData[i] = previous\_data;  // Serial.println(sonarData[i], 4);  // set servo to new angle  angleData[i] = i \* 2.0;  delay(200);  servo.write(i \* 2);  }  }  servo.write(centerServo); }  // Return distance in centimetres float getUltrasonic\_bonus() {  //double temp = analogRead(tempPin) \* CONVERT\_TO\_TEMP;  float temp = 21; // Standard temp in celsius  float speedOfSound = (331.5 + (0.6 \* temp)) \* 0.0001; // cm/microsecond   // Ping the ultrasonic   digitalWrite(TRIG\_PIN, HIGH);  delayMicroseconds(10);  digitalWrite(TRIG\_PIN, LOW);   // Return pulse from echo  return (pulseIn(ECHO\_PIN, HIGH) / 2.0) \* speedOfSound; } |

**Slave Code**

|  |
| --- |
| #include <LiquidCrystal.h> #include <Wire.h>  #define MODEBTNUP\_PIN 2 #define MODEBTNDOWN\_PIN 3  int prevLCDCode = -1; int prevLCDSpeed = 0;  // debouncing variables int modeBtnUpState; int prevModeBtnUpState = LOW; unsigned long lastModeBtnUpDebounceTime = 0; int modeBtnDownState; int prevModeBtnDownState = LOW; unsigned long lastModeBtnDownDebounceTime = 0; long debounceDelay = 50;  byte upArrow[8] = {  0b00100,  0b01110,  0b11111,  0b00100,  0b00100,  0b00100,  0b00100,  0b00000 };  byte downArrow[8] = {  0b00100,  0b00100,  0b00100,  0b00100,  0b11111,  0b01110,  0b00100,  0b00000 };  byte leftArrow[8] = {  0b00000,  0b00100,  0b01100,  0b11111,  0b01100,  0b00100,  0b00000,  0b00000 };  byte rightArrow[8] = {  0b00000,  0b00100,  0b00110,  0b11111,  0b00110,  0b00100,  0b00000,  0b00000 };  byte fullBox[8] = {  0b11111,  0b11111,  0b11111,  0b11111,  0b11111,  0b11111,  0b11111,  0b00000 };  byte emptyBox[8] = {  0b11111,  0b10001,  0b10001,  0b10001,  0b10001,  0b10001,  0b11111,  0b00000 };  byte eye[8] = {  0b00000,  0b00000,  0b01110,  0b10101,  0b01110,  0b00000,  0b00000,  0b00000 };  LiquidCrystal lcd(A3, A2, 11, 10, 9, 8);  void setup() {  lcd.begin(16, 2);   lcd.clear();  lcd.print("Setting up");   Wire.begin(8); // begin coms with master Arduino, this Arduino is on address 8   Serial.begin(9600);  lcd.createChar(0, upArrow);  lcd.createChar(1, downArrow);  lcd.createChar(2, leftArrow);  lcd.createChar(3, rightArrow);  lcd.createChar(4, fullBox);  lcd.createChar(5, emptyBox);  lcd.createChar(6, eye);    lcd.clear();  lcd.print("Setup complete");    Wire.onReceive(receiveEvent);  Wire.onRequest(requestEvent); }  void loop() {   debounceAllButtons();  }  void receiveEvent(int howMany) {  Serial.println("recieve triggered");  if ( Wire.available() == 1 ) { // one byte means that user is currently selecting a mode  int x = Wire.read();  if ( x != prevLCDCode ) { // reduces flickering text on LCD if user does not press a button  prevLCDCode = x;  lcd.clear();  switch (x) {  case 0:  lcd.print("Select mode");  lcd.setCursor(0, 1);  lcd.print("using ");  lcd.write(byte(2));  lcd.print(" and ");  lcd.write(byte(3));  break;  case 1:  lcd.print("Principle 1:");  lcd.setCursor(0, 1);  lcd.print("Drive and Scan");  break;  case 2:  lcd.print("Principle 2:");  lcd.setCursor(0, 1);  lcd.print("Line Following");  break;  case 3:  lcd.print("BONUS: ");  lcd.setCursor(0, 1);  lcd.print("IR Navigation");  break;  case 4: // special case for bonus matrix processing  lcd.print("Matrix Ready");  lcd.setCursor(2, 1);  lcd.write(byte(6));  lcd.print("Processing");  lcd.write(byte(6));  break;  default:  lcd.print("Select mode");  lcd.setCursor(0, 1);  lcd.print("using ");  lcd.write(byte(2));  lcd.print(" and ");  lcd.write(byte(3));  break;  }  }  }  else if ( Wire.available() == 2 ) { // two bytes means speed and direction of robot are to be displayed on LCD  int speed = Wire.read(); // first byte is speed  byte directionCode = Wire.read(); // second byte to be decoded into a direction   byte increasing = (speed > prevLCDSpeed) ? 0 : 1; // display an arrow for increasing or decreasing speed  prevLCDSpeed = speed;   String dir; // decode the directionCode byte into a String representing the robot's current direction  byte arrow;  switch ( directionCode ) {  case 0:  dir = "Forward";  arrow = 1;  break;  case 1:  dir = "Backward";  arrow = 0;  break;  case 2:  dir = "Left";  arrow = 3;  break;  case 3:  dir = "Right";  arrow = 2;  break;  case 4:  dir = "Scanning...";  arrow = 6;  }  // display data on LCD  lcd.clear();  lcd.print("Speed: ");  lcd.print(speed);  lcd.setCursor(15, 0);  lcd.write(increasing);  lcd.setCursor(0, 1);  lcd.print(dir);  lcd.setCursor(15, 1);  lcd.write(arrow);  }  else if ( Wire.available() == 3 ) { // 3 bytes means data is for line following  byte boxCode = Wire.read();  Wire.read(); // garbage  Wire.read(); // garbage   lcd.clear();  lcd.setCursor(0, 1);   byte empty = 5; // display an empty box for no detection on that sensor  byte full = 4; // full box for sensors that detect the line  switch(boxCode) {  case 0:  lcd.write(full);  lcd.setCursor(7, 1);  lcd.write(empty);  lcd.setCursor(14, 1);  lcd.write(empty);  break;  case 1:  lcd.write(empty);  lcd.setCursor(7, 1);  lcd.write(full);  lcd.setCursor(14, 1);  lcd.write(empty);  break;  case 2:  lcd.write(empty);  lcd.setCursor(7, 1);  lcd.write(empty);  lcd.setCursor(14, 1);  lcd.write(full);  break;  case 3:  lcd.write(full);  lcd.setCursor(7, 1);  lcd.write(full);  lcd.setCursor(14, 1);  lcd.write(empty);  break;  case 4:  lcd.write(empty);  lcd.setCursor(7, 1);  lcd.write(full);  lcd.setCursor(14, 1);  lcd.write(full);  break;  default:  lcd.write(empty);  lcd.setCursor(7, 1);  lcd.write(empty);  lcd.setCursor(14, 1);  lcd.write(empty);  break;  }  }   } /\*  Used for reading the momentary switches \*/ void requestEvent() {  Serial.println("request triggered");  if ( debounceModeUpButton() )  Wire.write(0);  else if ( debounceModeDownButton() )  Wire.write(1);  else  Wire.write(-1); }  void debounceAllButtons() {  debounceModeUpButton();  debounceModeDownButton(); }  int debounceModeUpButton() {  int reading = digitalRead(MODEBTNUP\_PIN);  if ( reading != prevModeBtnUpState ) {  lastModeBtnUpDebounceTime = millis();  }  if ( (millis() - lastModeBtnUpDebounceTime) >= debounceDelay ) {  if ( reading != modeBtnUpState )  modeBtnUpState = reading;  }  prevModeBtnUpState = reading;  return modeBtnUpState; }  int debounceModeDownButton() {  int reading = digitalRead(MODEBTNDOWN\_PIN);  if ( reading != prevModeBtnDownState ) {  lastModeBtnDownDebounceTime = millis();  }  if ( (millis() - lastModeBtnDownDebounceTime) >= debounceDelay ) {  if ( reading != modeBtnDownState )  modeBtnDownState = reading;  }  prevModeBtnDownState = reading;  return modeBtnDownState; } |

**Appendix C - Fritzing**

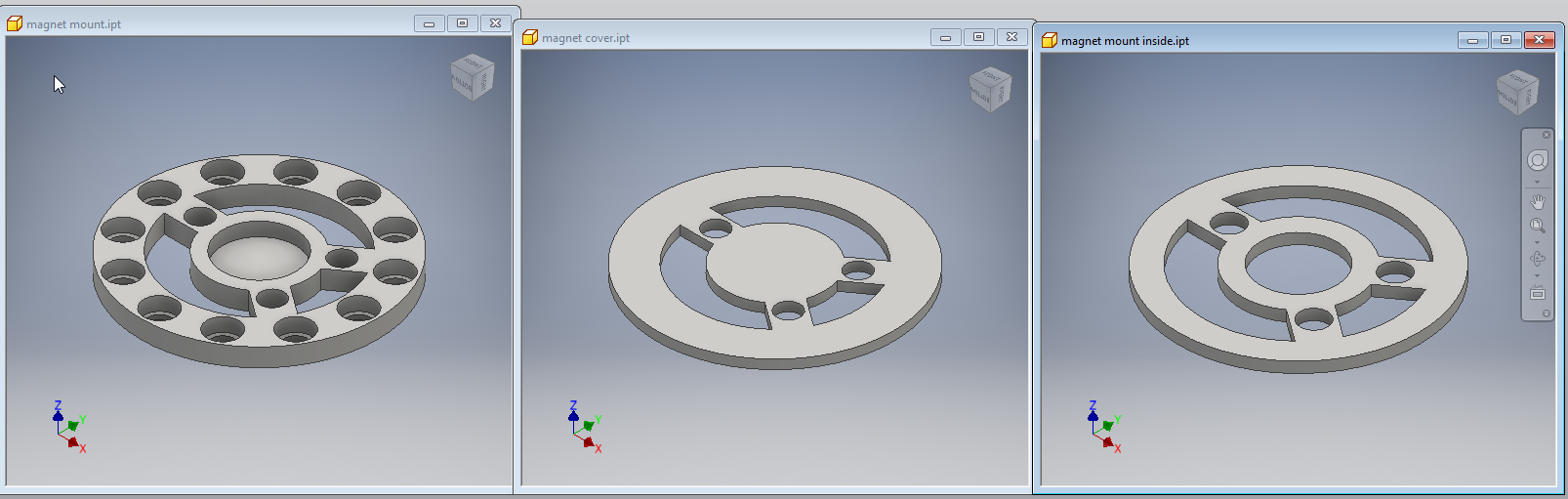


**Figure 10. Fritzing schematic for our robot’s circuitry**

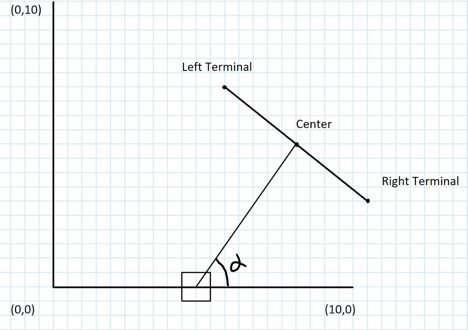
**Appendix D - Other**

**3D Printed Wheel Mount**

We designed 3D printed circular disks that attached on to each wheel using three screws. These disks were embedded with twelve magnets each in order to trigger each hall effect sensor. Additionally, by reading both the rising edge and falling edge of the hall sensor, we could have an effective resolution of 24 ticks per rotation of the wheel. This allowed us to update the a variable in the code every 15 degrees rather than every 360 degrees. The increase in resolution allows us to detect a difference in the position of the wheels sooner so that we can adjust the motor powers accordingly.



**Figure 11. CAD designed magnet mount parts for the DFRobot wheels**

**Additional Functionality Algorithm Explanation**

**Figure 12. Illustration of the algorithm**

The algorithm performs as follows:

The robot location is fixed on the point (5,0), so the map generated from the algorithm is going to be coordinated in that point with a grid dimension 15cm.

First, the algorithm take the effective angle 30 degree and the distance from the robot to the centre to calculate the distance from left terminal to right terminal,

|  |
| --- |
| distance(left terminal to right terminal) = tan(15) \*distance \*2 |

Second, the algorithm takes the angle alpha and distance to compute the location of center. We also cap the centre point inside the map so that we won’t get index out of bound error. The formula to compute centre

* 90 < angle < 180

|  |
| --- |
| center.x = cap(robot\_location.x - distance \* cos(degree\_to\_radian(180 - angle)) / gridLength);  center.y = cap(robot\_location.y + distance \* sin(degree\_to\_radian(180 - angle)) / gridLength); |

* 0 < angle < 90

|  |
| --- |
| center.x = cap(robot\_location.x + distance \* cos(degree\_to\_radian(angle)) / gridLength);  center.y = cap(robot\_location.y + distance \* sin(degree\_to\_radian(angle)) / gridLength); |

* angle = 90

|  |
| --- |
| center.x = cap(robot\_location.x);  center.y = cap(robot\_location.y + distance / gridLength); |

* angle = 180

|  |
| --- |
| center.x = cap(robot\_location.x + distance / gridLength);  center.y = cap(robot\_location.y); |

* angle = 0

|  |
| --- |
| center.x = cap(robot\_location.x - distance / gridLength);  center.y = cap(robot\_location.y); |

Third, after obtaining the centre and the distance from left terminal to right terminal. We can move forward to calculating the location of left terminal and right terminal.

Left Terminal Point:

* 0< angle <90

|  |
| --- |
| terminal\_point.x = cap(center.x - range \* cos(degree\_to\_radian(90 - alpha)) / gridLength);  terminal\_point.y = cap(center.y - range \* sin(degree\_to\_radian(90 - alpha)) / gridLength); |

* 90 < angle <180

|  |
| --- |
| terminal\_point.x = cap(center.x - range \* cos(degree\_to\_radian(alpha - 90)) / gridLength);  terminal\_point.y = cap(center.y + range \* sin(degree\_to\_radian(alpha - 90)) / gridLength); |

* angle = 90

|  |
| --- |
| terminal\_point.x = cap(center.x - range / (2 \* gridLength));  terminal\_point.y = center.y; |

* angle = 0

|  |
| --- |
| terminal\_point.x = center.x;  terminal\_point.y = center.y;\ |

* angle =180

|  |
| --- |
| terminal\_point.x = center.x;  terminal\_point.y = cap(center.y + range / (2 \* gridLength)); |

Right Terminal Point:

* 0< angle <90

|  |
| --- |
| terminal\_point.x = cap(center.x + range \* cos(degree\_to\_radian(90 - angle\_a)) / gridLength);  terminal\_point.y = cap(center.y + range \* sin(degree\_to\_radian(90 - angle\_a)) / gridLength); |

* 90 < angle <180

|  |
| --- |
| terminal\_point.x = cap(center.x + range \* cos(degree\_to\_radian(angle\_a - 90)) / gridLength);  terminal\_point.y = cap(center.y - range \* sin(degree\_to\_radian(angle\_a - 90)) / gridLength); |

* angle = 90

|  |
| --- |
| terminal\_point.x = cap(center.x + range / (2 \* gridLength));  terminal\_point.y = center.y; |

* angle =180

|  |
| --- |
| terminal\_point.x = center.x;  terminal\_point.y = center.y; |

* angle = 0

|  |
| --- |
| terminal\_point.x = center.x;  terminal\_point.y = cap(center.y + range / (2 \* gridLength)); |

Fourth, after calculating the centre, left and right terminal point. We move forward to calculating the probability distribution effect from the data. For this function, we implemented a linear distribution model. The probability increase within the effective area is linearly proportional to the effective range segment in that grid.

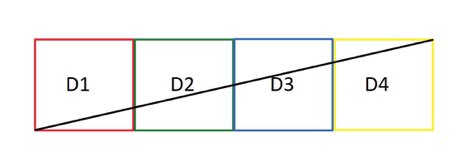
chance increased =

1/(distance from left terminal to right terminal projected on x axis / gridLength)

For example:

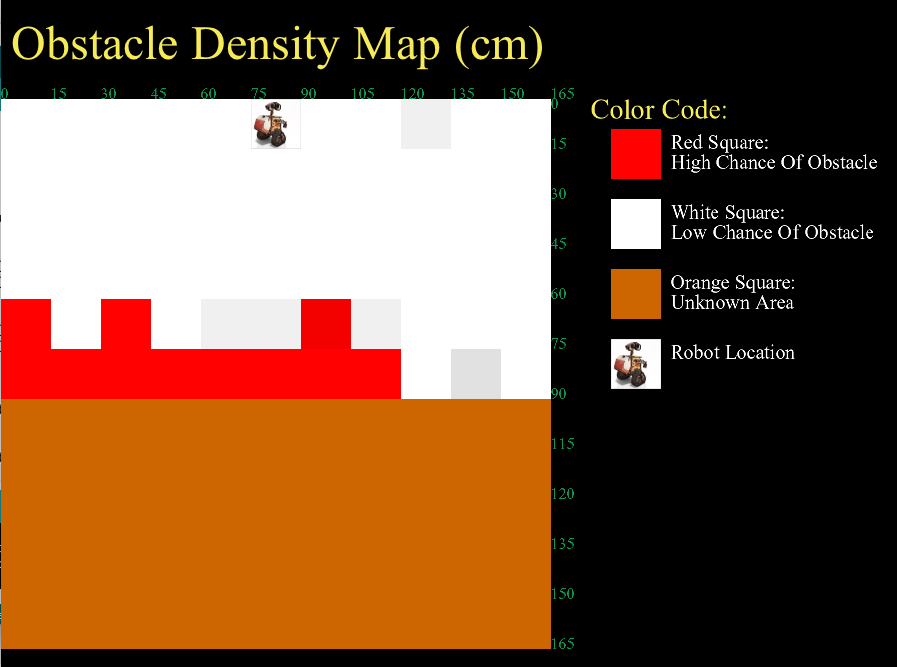
the chance increased in the red box is will be

1/(the projection of D1 to the axis / grid length)



**Figure 13. Illustration of algorithm, example**

Lastly, the algorithm will initialize all other area enclosed by the terminal points and the robot location to 0 to indicated the area has been explored. As result of the all the steps above, we will obtain the distribution effect from one data point and translate that effect into a matrix representation. Combining all 91 data points, we will get the complete representation of the surroundings in front of robot from 0 to 180 degrees.



**Figure 14. Processing Image from the navigation scan algorithm, with an obstacle in front**