Lab 3 - Building a Line Following Robot

Annabel Consilvio, Charlie Mouton, and Nur Shlapobersky

October 13, 2015

Abstract

In this lab, we created a motorized robot which follows a line of electrical tape around an abnormal circular track. Using a standard chassis, we created a mount plate that incorporated an Arduino, motor shield, and small breadboard. Using two photo-transistors, along with a Proportional and Derivative feedback control system, we were able to traverse the track speedily and reliably.

1 Sensor Calibration & Electronics

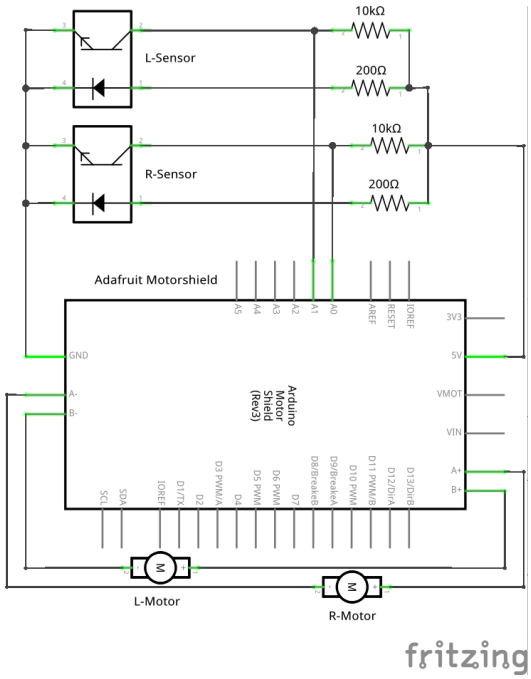

Figure 1: The schematic of our circuit with a generic motor shield, which was mounted onto our Arduino

We began our electronic work by choosing resistor values to use with our reflectivity sensors. For the diodes we used values of 200Ω, which was given to us. For the phototransistors we had to choose a resistance value which would be on an order of magnitude of the peak forward current of the transistor. We chose values of 10kΩ, because that would place our current at .5mA, comparable to the values of between 1-5mA that the phototransistor outputs. Our resistor could have been smaller, but since we had success with them, we found no reason to experiment further.

For our initial bang-bang control implementation, we had to choose certain sensor readings for which we should begin to turn, which we discuss more in the Control System section. All we had to ensure physically was that our sensors were farther apart than the the width of the tape line, and could thus react independently.

For the PD control system, calibration of specific values was done with respect to the proportionality constants as opposed to specific sensor values. However, to optimize our line-following capabilities, the

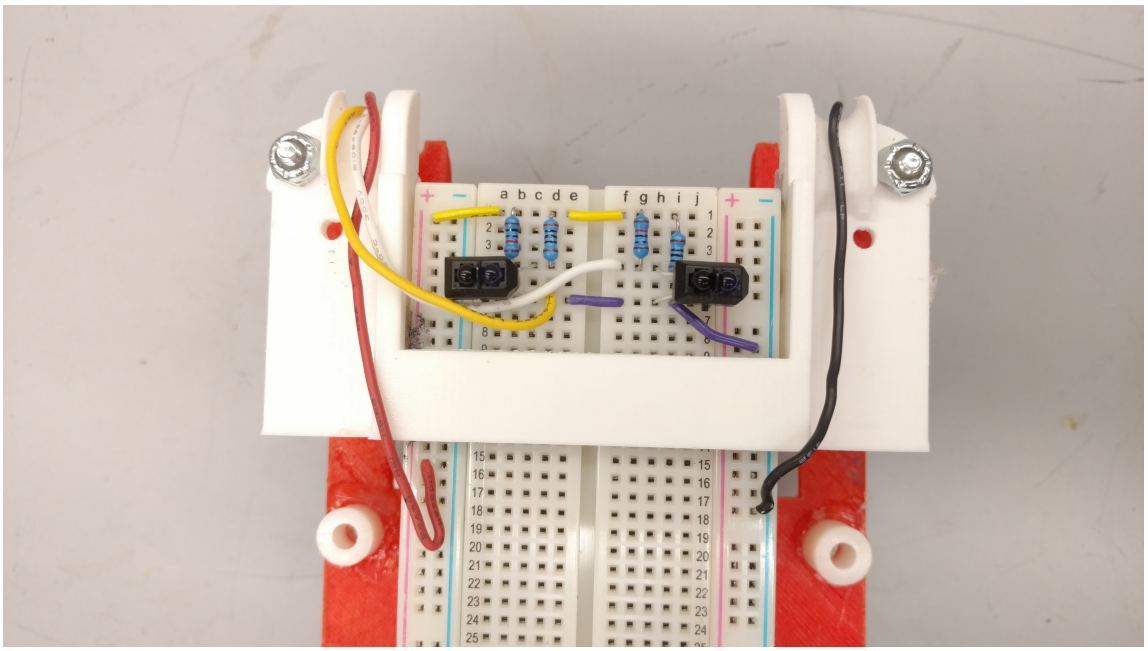

Figure 2: An in-depth view of our circuit located on our breadboard.

physical placement of the sensors had to be adjusted. We separated the sensors and placed them farther from the ground. This ensured that they had a spectrum of values between on or off the tape, as opposed to a binary one. This created a much smoother error for our PD control code to interact with.

2 Mechanical System

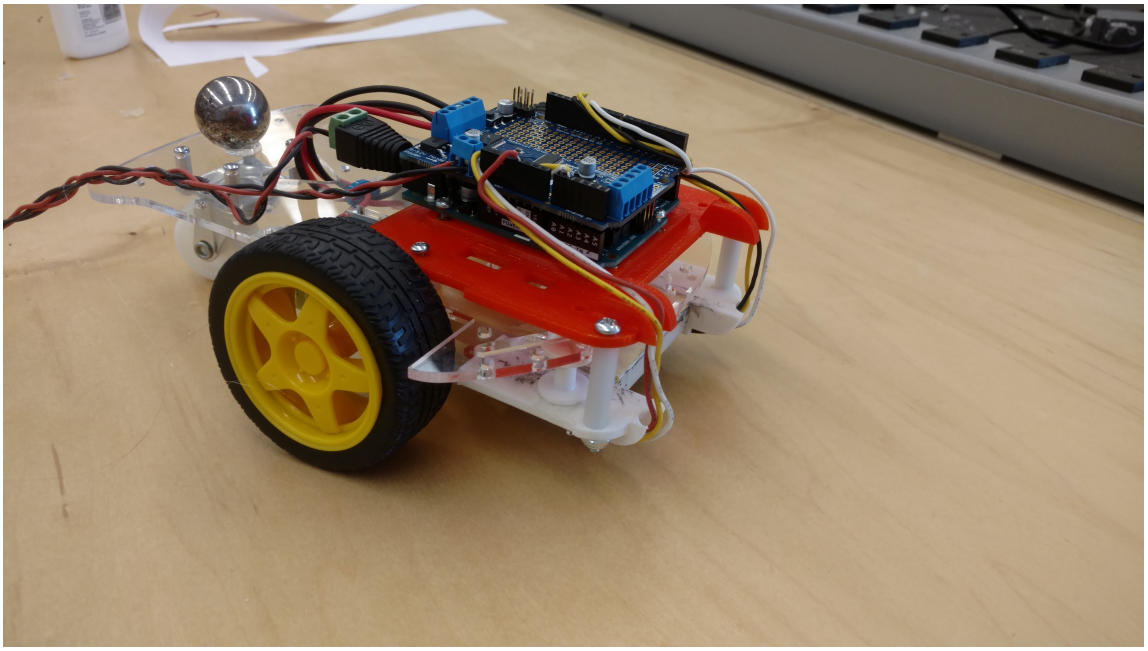

Figure 3: Our Chassis consisted of 3D printed parts, bolts, and several spacers made available off of the POE supply cart.

We aimed to create a physical attachment that could very easily be attached and detached from the chassis. To do this, we created a "U" shaped mounting piece that holds the Arduino and motor shield above the chassis and then supports a small breadboard below the chassis, allowing for the sensors to be close enough to get accurate readings. Wires are routed through two channels located at the front of our mount.

We started by first flipping the side of the chassis that the standoffs used with the caster were placed. Because we were creating our own mounting location for the Arduino, we decided that it was more valuable to have the entire chassis rest more level.

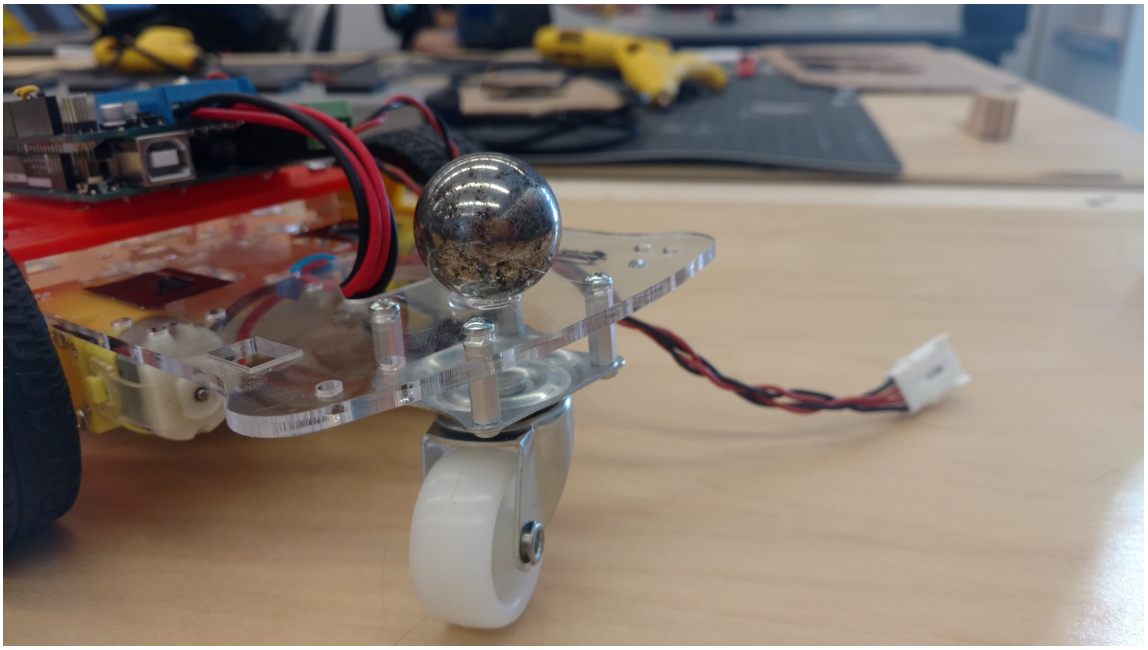

Figure 4: You can see where we switched the standoffs to below the plastic chassis plate and added a large ball to counterweight our system mounted to the front of the car.

We then 3D printed a plate that rested on top of the chassis with spacers to create the correct distance between the chassis and the sensors below. Using threaded inserts and bolts, we mounted the Arduino with the power supply cords sticking out of the back, reducing the interference caused by the tether.

Using bolts as pins, we use holes in the chassis as alignment holes that hold our mount in place. Finally, a steel ball is glued to the back of the chassis to act as a counterweight to our mount being located on the front.

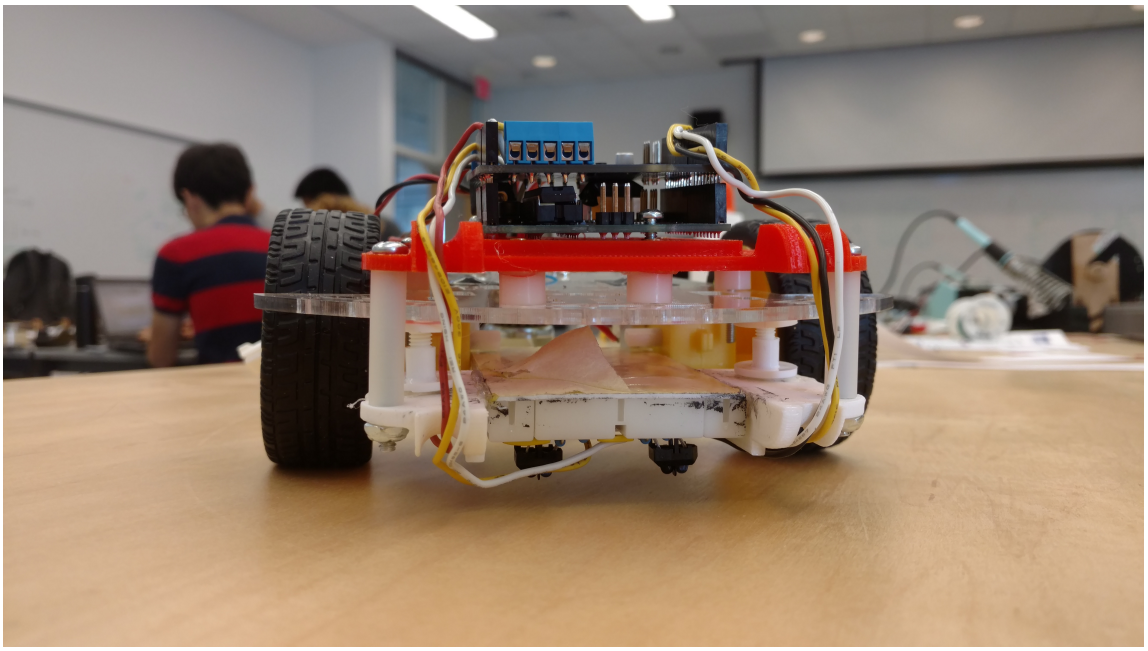

Figure 5: Using adjustable spacers on the bottom part of our mount, we were able to create a system that fit snugly against the chassis plate allowing our sensors to have a standard distance that they sit from the floor.

3 Control System

3.1 "Bang-Bang" Control

As an initial iteration, we used a bang-bang control system to ensure that our robot would follow the line. A bang-bang control is binary, in the sense that once a sensor reads a value indicating it is on the tape (¿ 700), it would make a sharp turn in the opposite direction. This creates a zig-zag motion, which, while being inneficient and slow, did successfully navigate the track. A sample of our control code can be seen below:

```
1       if (sensorValueR-sensorValueL >= 350)  // turn left
2       {
3         MotorR->setSpeed(turning_speed);
4         MotorL->setSpeed(low_speed);
5         Serial.print(turning_speed);
6         Serial.print(", ");
7         Serial.print(low_speed);
8         Serial.print(", ");
9         Serial.println(reading);
10      }
11      else if (sensorValueL-sensorValueR >= 250) //turn right -> 250 instead of
            350 because of the difference in natural sensor readings
12      {
13        MotorL->setSpeed(turning_speed);
14        MotorR->setSpeed(low_speed);
15        Serial.print(low_speed);
16        Serial.print(", ");
17        Serial.print(turning_speed);
18        Serial.print(", ");
19        Serial.println(reading);
20      }
21      else
22      {
23        MotorR->setSpeed(input);
24        MotorL->setSpeed(input);
```

```
25            Serial.print(input);
26            Serial.print(", ");
27            Serial.print(input);
28            Serial.print(", ");
29            Serial.println(input);
30        }
```

This code assumes a set turning speed, low speed, and reading. Reading, in the code above, is a user specified input, turning speed is slightly higher than the normal speed of the robot, and low speed, is 0. This essentially causes one wheel to stop moving entirely and the opposite wheel to speed up, creating a very sharp turn that over-corrects in order to find the line again. After we were able to implement the bang-bang control mechanism well enough to get the robot around the track, we moved on to the PID controlling method.

3.2 Exploring PID

For our initial iteration of PID (proportional-integral-derivative) control, we focused first on optimizing the proportional element of the function. This code can be seen in Appendix A. To begin with, we calculated the difference between the two sensor values and then compared that to our desired difference (in this case 0). This value was then multiplied by the proportional constant of our control code, which was then either added or subtracted from the speed of the motor based on the motor's location.

After a successful proportional implementation, which still moved relatively slowly around the track, we began adding the derivative or dampening control. We calculated our derivative control by tracking the change in our error at any given moment. This value was then multiplied by our derivative control constant, and added to our error control. Although this value is quite low in our final code, it allowed us to complete the track with an average speed of 45 rather than 40, which was the maximum speed at which we could complete the track solely with Proportional correction.

For this particular iteration, we did not use the integral part of the PID algorithm. Because there was not significant error in our readings and because we were able to reach an adequate speed without introducing another variable, we left this out. In the future, we could continue to tune the variables to reach a faster speed because the integral based code is already implemented (The kI variable is set equal to 0, which causes it not to be functional in the actual sensor processing).

3.3 Control Demo

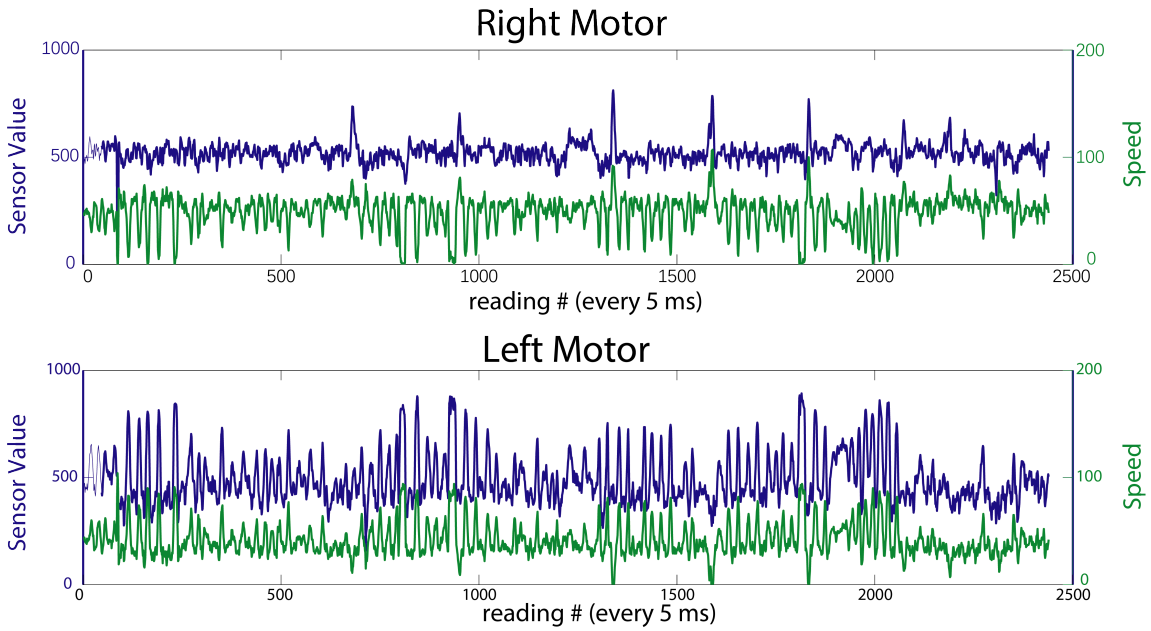

Figure 6

In order to test that our control code was functioning properly, we used Matlab to plot the sensor values and the motor speeds against each other. A long time scale plot from a full path run is shown below.

3.4 User Input

We also implemented a user input functionality through our Arduino code so that you did not have to re-upload to the Arduino in order to change the default speed. While currently commented, this code can be seen implemented in Appendix A. Through the function Serial.read in Arduino, users can reset the robot's default speed in the Serial Monitor. This speed will remain until the robot reaches an end point and stops or until the user resets the speed.

4 Conclusion and Reflections

A video of a successful run can be found here. In this lab, we were able to successfully implement two different control systems. Bang-bang was our initial attempt, and we improved on this by implementing a PD control system. Additionally, we also made mechanical and electrical improvements along the way to make the robot more efficient.

Overall, our team grew tremendously. Annabel learned introductory control theory and got much more comfortable with solo Arduino programming. Charlie learned about integrating electrical and computing systems into the mechanical and allowing for adjustment to tweak our design as we aim to improve our robot. Nur enjoyed coordinating between what mechanical corrections would best complement the software, and how to best position and use our sensors to implement a feedback control loop.

A Arduino Code

```
//Adding Libraries for motor shield
#include <Wire.h>  //include Motor Liraries
#include <Adafruit_MotorShield.h>
#include "utility/Adafruit_PWMServoDriver.h"

Adafruit_MotorShield AFMS1 = Adafruit_MotorShield();  //create motor shield object
Adafruit_MotorShield AFMS2 = Adafruit_MotorShield();  //create motor shield object
Adafruit_DCMotor *MotorR = AFMS1.getMotor(1); //set up motor in port 1
Adafruit_DCMotor *MotorL = AFMS2.getMotor(2); //set up motor 2

//Define Variables
int setpoint_value = 0; //find true value
int sensorPinR = A0; //the input pin for right IR sensor
int sensorPinL = A1;  //input pin for left IR sesnor
long sensorValueR = 0; //initializing sensor value
long sensorValueL =0; //initializing sensor value
int speedR;  //initalizing speed variables
int speedL;
int defaultSpeed = 45;  //speed that the robot will converge to
double kP = .127; // proportinal value for PID control
double kI = 0;   // integral value for PID control, not actually used in our
    functioning model
double kD = 0.001; //dampening value for PID control
int ref = 0;  //temp variable for calculating error
int sensorDiff = 0; //difference between the two sensor values

//correctional values for PID
int corrDiff = 0;
int control = 0;
int corrVal = 0;
int corrStore = 0;
int corri = 0;
int corrf = 0;

boolean moving = true;  //only keeps robot running if this is true

byte input = 40;  // serial user input variables
```

```
byte reading = 40;


void setup() {
  Serial.begin(250000);  //initalize serial monitor
  delay(5000);  //so matlab can get its shit together
  AFMS1.begin();   // begin motors
  AFMS2.begin();
  MotorR->setSpeed(defaultSpeed);  //set both motors too default speeds
  MotorL->setSpeed(defaultSpeed);
  MotorR->run(FORWARD);  //start running motors
  MotorL->run(FORWARD);
  moving = true;
}


void loop() {
  //reading = Serial.parseInt();  //make sure user input is one number (50) rather
       than 5 and then 0

  if (moving){  //if robot is moving
   // defaultSpeed = input;  //set default speed to whatever the user input is

    //initialize the variables we're linked to
    sensorValueR = analogRead(sensorPinR);  //set sensorValue to current value of
        analog pin
    sensorValueL = analogRead(sensorPinL);  //set sensorValue to current value of
        analog pin
    sensorDiff = sensorValueR - sensorValueL;  //get difference between 2 sensors
        for PID
    control = ref - sensorDiff;  //for integral - not actually used in final model
    corri = control;
    corrDiff = corrf - corri;
    corrStore += (control);

    //make correctional value determined by proportinal difference in sensor value
         (kP)
    //then dampen this (kD) so that the robot doesn't go cray
    corrVal = (control)*kP + corrStore*kI + corrDiff*kD;

    speedR = defaultSpeed - corrVal;  //change speeds of motors based on control
        PID algorithm
    speedL = defaultSpeed + corrVal;

    if (speedR < 0)  //making sure motors don't loop and go from 0 to 244 when it
        gets a negative
    {
      speedR =0;
    }

    if (speedL < 0)
    {
      speedL =0;
    }

    MotorR->setSpeed(speedR);  //reset motor speeds
    MotorL->setSpeed(speedL);
    corrf = corri;
    delay(5);  //wait 1 second before printing next value


    //print a bunch of values to send to matlab for plotting
    Serial.print(sensorValueR);
    Serial.print(", ");
    Serial.print(sensorValueL);
    Serial.print(", ");
    Serial.print(speedR);
    Serial.print(", ");
```

```
    Serial.print(speedL);
    Serial.print(", ");
    Serial.println(reading);

    if (sensorValueR >=600 && sensorValueL >= 650)  //if both sensors are on the tape, stop running
    {
      moving = false;   //once its not running, send a bunch of 0s to matlab so that it knows to stop
      Serial.print(0);
      Serial.print(", ");
      Serial.print(0);
      Serial.print(", ");
      Serial.print(0);
      Serial.print(", ");
      Serial.print(0);
      Serial.print(", ");
      Serial.print(0);
      Serial.println(0);
      MotorR->setSpeed(0);
      MotorL->setSpeed(0);
    }

    //user input things
        if (reading > 0)  // if there is a user input (reading) that is greater than 0
    {
      input = reading;  // change the input to the value of the reading
    }

  }

    delay(5);  //wait 1 second before printing next value

}
```

B Matlab Plotting Code

```
function M = motorirread()
s = serial('COM5');  % set to match arduino port
set(s, 'BaudRate', 250000, 'Timeout', 0.005); %set rate to match arduino output
fopen(s);  %open serial port where arduino is printing
M = []; % initalize empty array
moving = true;  %initalize scanning variable, true at the beginning
                    %false when arduino stops printing

finishup = onCleanup(@() cleanup(s));  %cleans up functions at the end of run
    function cleanup(s)
        toc  %amount of time passed
        fclose(s);  %stop reading serial from arduino
        delete(s);
        clear s
        disp('Cleaned Up.')
    end

fscanf(s) % clear garbage data up through newline terminator
i = 0;
tic  %time starts
while(moving)  %while the arduino is still sending data
    if(get(s, 'BytesAvailable') >= 1)  %if data is coming in
        i = i + 1;
        data = sscanf(fscanf(s), '%u, %u, %u, %u, %u');  %reformat string into integer data
       if  sum(data) <= 150  %if arduino sends a bunch of zeros or very low number
            moving = false;    %stop scanning
        else
```

```
            M = vertcat(M,[data(1), data(2), data(3), data(4), data(5)])
             %concatonate arrays for plotting
        end

    end

end

%%plotting

hold on

time = [];

for i = 1:length(M(:,1))
    time = vertcat(time,i);
end

subplot(2,1,1)
[hAx,hLine1,hLine2] = plotyy(time, M(:,1), time,M(:,3))
title('Right Motor')
xlabel('reading # (every 5 ms)')
ylabel(hAx(1),'Sensor Value') % left y-axis
ylabel(hAx(2),'Speed') % right y-axis
hLine1.LineStyle = '--';
hLine2.LineStyle = ':';


subplot(2,1,2)
[hAx,hLine1,hLine2] = plotyy(time, M(:,2), time,M(:,4))
title('Left Motor')
xlabel('reading # (every 5 ms)')
ylabel(hAx(1),'Sensor Value') % left y-axis
ylabel(hAx(2),'Speed') % right y-axis
hLine1.LineStyle = '--';
hLine2.LineStyle = ':';



end
```