Lab 3 : Line Following Robot

Tommy Weir and Anusha Datar

October 12, 2018

1 Introduction

In this lab, we designed and programmed a line-following robot using a DC-motor based car, IR sensors, an Arduino UNO, and a 12V Motor Shield. We were given a basic chassis and had to design parts that would hold the Arduino and all of the other components we used for our project such that they were fixed to the chassis but could easily be removed. We also had to choose the correct electronic components and circuit configuration to use the IR sensors. Then, we had to write and tune Arduino code that would allow us to use the IR sensor data to determine if the robot needed to turn either of its wheels in order to stay on the line. See appendices for circuit diagram, circuit images, and source code.

2 Sensor Calibration

We took two discrete steps in our sensor calibration process : resistor determination and threshold tuning. Because our IR sensor circuit contains a divider with a phototransistor and a calibration resistor (such that we read the voltage from in between these two components), we needed to choose a resistor value to tune the voltage output to an informative range. To generate this value, we tried a variety of resistors with values of orders of magnitude ranging from 100Ω to $100k\Omega$. For each individual resistor, we tested the change in value between the tile floor and the tape and also recorded the overall sensitivity of the sensor : we wanted the sensor to be able to pick up on the difference between the tape and the tile reliably without overcorrecting or missing points. Through this process, we selected the $10k\Omega$ as the value for our R_{sense}. Following resistor selection, we had to choose a value for the black tape to trigger our controller. For this portion, we held the sensor over the tape and over the tile and calculated the average analog output value over several seconds using a small test script that reported the average to the serial output. After multiple trials on both the tile and the tape, we were able to determine reasonable thresholds that we then more finely tuned during controller operation and testing.

3 Mechanical System

The main mechanical parts are the chassis, Arduino holder, and the sensor holder. The chassis was a provided DIY kit, with an acrylic base and two motors that attached to plastic wheels. There was also a battery holder, but we were using a 12V adapter, so we chose not to attach it. In order to design the Arduino holder, we used a diagram with the placement of the holes for an Arduino Uno. Then, we designed a rectangular plate that had pegs to hold the Arduino onto the plate. Next, we made extended sections on the plate that aligned with the holes on the chassis so that we could screw on the holder. This allowed our components to be secured and also independent and easily removable.

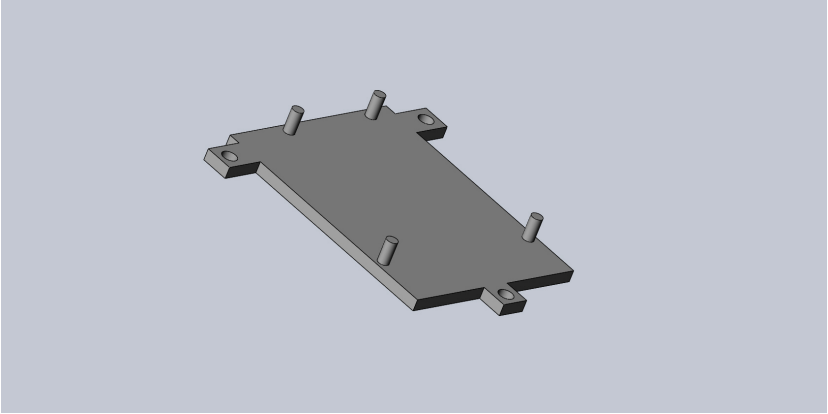

Figure 1: The Arduino holder

After that, we designed separate sensor holders for underneath the chassis. We decided to have one hole in the holder for a screw, since we can tighten the screw and nut enough to prevent movement for the holder and adjust the positioning of the sensors in real time during testing. We had originally planned on securing the holder directly to the chassis, but in order to make the sensor readings more accurate we had to add spacers between the holder and the chassis. This way the sensors were closer to the tape, and could determine more accurately if the sensor was over tape or over tile.

See a diagram of the completed mechanical system in Figure 3.

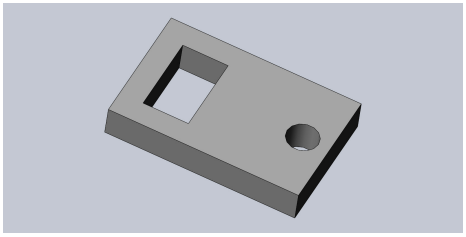

Figure 2: The IR sensor holder

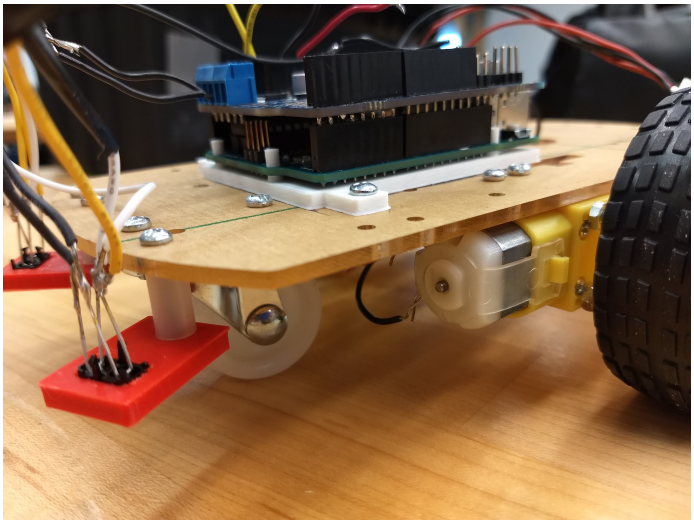

Figure 3: A closeup of the finished car chassis

4 Control System

After considering the merits of implementing both a bang-bang and proportional controller, we chose to use a bang-bang controller for our final implementation. The reason we selected bang-bang over PID was because we determined that the signal associated with the tape did not have an ample gradient for careful proportional control and that while considering the difference between each sensor could create a more complicated controller, the amount of oscillation associated with straight-line paths would likely outweigh the increased quality of turns. As a result, we have a very simple and straightforward controller. The controller had three main parameters: the threshold, the speed delta, and the delay. Essentially, for every timestep associated with the delay, the robot samples points from the right and left sensor. If the right sensor is above the empirically determined threshold for black tape, the robot moves its right wheel backwards and its left wheel forwards at the value of the speed delta to compensate. Similarly, if the left sensor reports a value above the threshold,

the robot turns right to make up for the difference. If both sensors are below the threshold, the robot moves forward, and if the both sensors are above the threshold, the robot oscillates back and forth on each time step. By tuning these parameters through a serial interfaced based update mechanism, we built a fairly robust and capable controller.

To correct for the mechanical limitations of the vehicle, we had to add a constant to the right wheel's commands to ensure that the robot would behave as we modeled. To find this constant, we set the wheel speeds of the left and right motor to the same value and then incremented the speed of the right motor until the the robot was driving forwards in a straight line.

Because our bang-bang controller is fairly straightforward, the figure (showing sensor data and motor speeds from a trial run) is quite simple. Note that the sensor and motor data is time-synchronized but shown on two separate plots to maximize clarity and account for offsets. Note that our threshold for the sensor value during this trial run was 950 units.

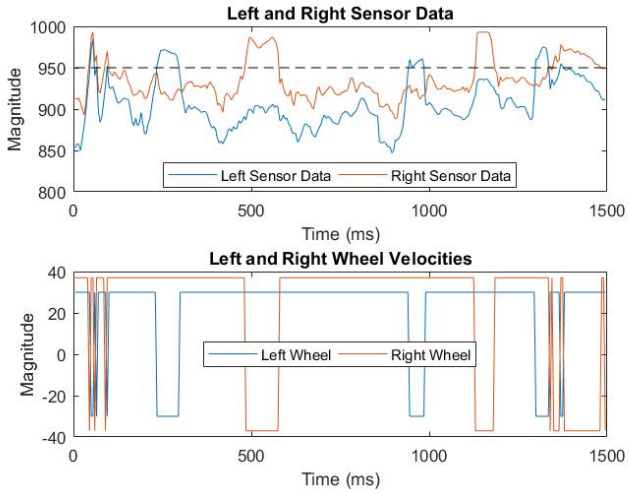

Figure 4: A closeup of the finished car chassis

5 Results

We were able to tune our robot to move around the track. We have posted a video of our working robot here.

6 Reflection

In this project, we learned a lot about how to work with systems that are not reliably repeatable. Depending on some physical disturbance, our code could

appear to have a bug when it did not. For example, when powering both wheels the same amount, the robot did not drive in a straight line and instead turned slightly to one side. In order to counter this, we made a right motor constant that would be added to the right motor speed. While this strategy compensated for the error, it also illustrates the large variety of errors we have minimal control over. This diverges from the previous labs: while there were certainly components that we did not have control over each failure mode of, we could quickly find and eliminate uncontrolled variables or replace problematic parts. Here, when faced with motors in an existing chassis or elements such as friction and anomalous sensor readings, we had to face them through hardware and software fixes instead of general troubleshooting. We also learned about the difficulties of tuning and the importance of writing software and interfaces that facilitate quick adjustments: we saved a great deal of time by automatically updating various controller parameters over Serial and changing other values through variables instead of having to parse through a large amount of code to make minor adjustments to software.

A Appendix A : Electronics

A.1 Circuit Diagram

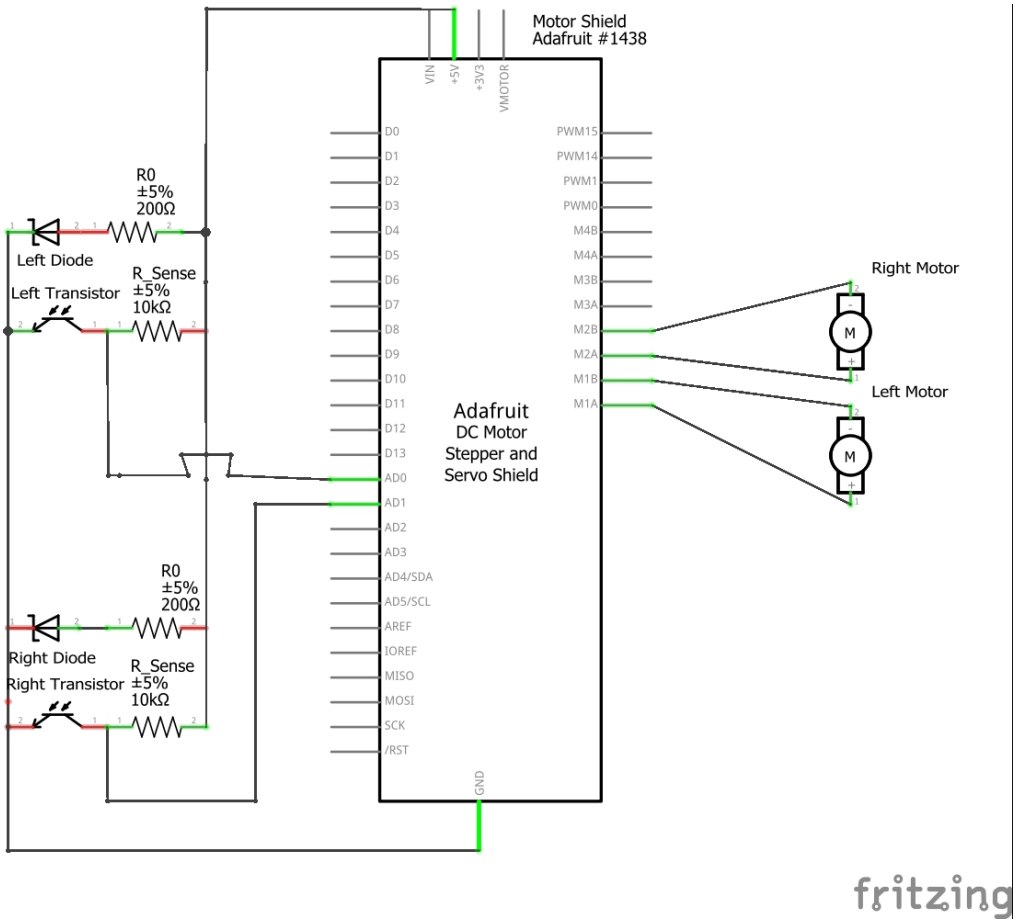

Figure 5: Circuit diagram for robot.

A.2 Photographs

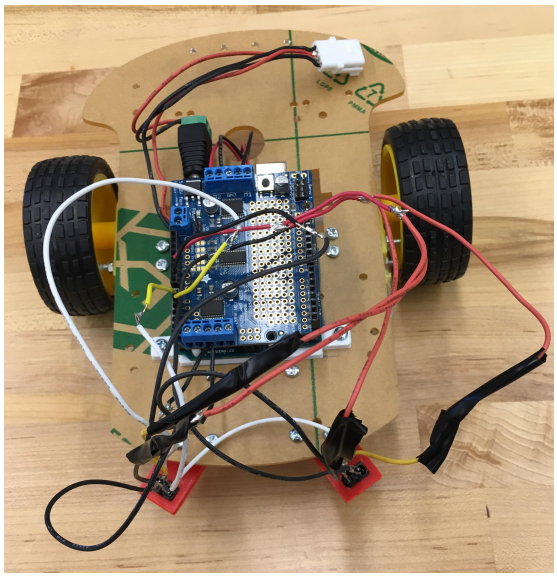

Figure 6: Top Circuit. Resistors are wrapped in electrical tape.

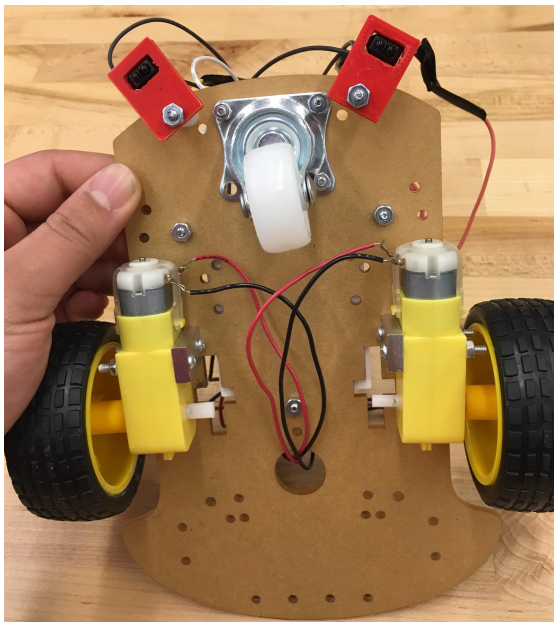

Figure 7: Connections for motors and sensors underneath the robot.

B Appendix B : Source Code

```
// Tommy Weir and Anusha Datar
// PoE Lab 3: Line-Following Robot

/*
 * This file contains all of the arduino code associated with Principles of Engineering lab
 * 3 as a consolidated unit. All files used to test electronics and mechanical assemblies can
 * be found in the tests/ directory in this repository (https://github.com/anushadatar/follow-that-line)
 */

// Include headers for motor shield and connections.
#include <Wire.h>
#include <Adafruit_MotorShield.h>

// Pin definitions and associated constants.
int leftMotorPin = 1;
int rightMotorPin = 2;
int leftSensorPin = A0;
int rightSensorPin = A1;
// Baudrate of serial. Important to match this on all programs.
int baudrate = 9600;

// Initial constants for motion controller.
// The initial speed of each motor.
int initialSpeed = 30;
// The initial speed by which to turn the motors if the tape is recognized.
// NOTE: This slows down the corresponding motor and speeds up the opposite motor
// at the same time, so the net delta is 2 x initial_speed_delta/
int initialSpeedDelta = 30;
// The initial threshold for the IR sensor - ABOVE this value is black tape, BELOW
// this value is the tile floor.
int initialThreshold = 950;
// The samples per measurement to start with for the IR sensor.
int initialSamplesPerMeasurement = 1;
// The amount of time to wait in between adjustments.
int initialWait = 5;
// Our right motor is a little slower than our left motor. We can adjust for that in software :)
int rightMotorConstant = 7;

// Create the motor shield object with the default I2C address.
// Configure motor as specified in program.
Adafruit_MotorShield AFMS = Adafruit_MotorShield();
Adafruit_DCMotor *leftMotor = AFMS.getMotor(leftMotorPin);
Adafruit_DCMotor *rightMotor = AFMS.getMotor(rightMotorPin);

//// Helper Methods.
int verify_IR_sensor(int sensorPin, int threshold, int samplesPerMeasurement) {
/* Utility method to measure IR sensor data on some pin for some number of
 * samples to check if the value is above some user-specified threshold.
 *
 * sensorPin : INT The pin of the sensor to read data on - should be
 * an analog input pin ideally.
 * threshold : INT The value BELOW which the value is black tape and ABOVE which
 * the value is tile floor.
 * samplesPerMeasurement : INT the number of samples to read per measurement.
 * Increasing the number of samples will decrease the
```

```
 * amount of jerkiness but also increase the amount of error.
 *
 * Returns : FALSE if the sensor value is HIGHER than the threshold and TRUE
 * if the value is LOWER than the threshold.
 *
 * In the case of the line following robot, FALSE means that the value is ABOVE the
 * threshold and the robot is ON the black tape, while TRUE means that the value
 * is BELOW the threshold and the robot is OFF the black tape.
 *
 * Kept all values as ints to maximize for efficiency here because of the high
 * volume of calls made on this function.
 */
  int sum = 0;
  for (int numberOfReads = 0; numberOfReads < samplesPerMeasurement; numberOfReads++) {
    sum += analogRead(sensorPin);
    delay(10);
  }
  // Grab average based on samples per measurement.
  int average = sum/samplesPerMeasurement;
  if (average > threshold) {
    return false;
  }
  else {
    return true;
  }
}
void set_speed_and_direction(Adafruit_DCMotor *motor, int inputSpeed) {
  /* Set wheel speed and direction for a given motor.
   *
   * motor = Pointer to Adafruit_DCMotor to change speed of.
   * inputSpeed = INT speed to set the motor to run at.
   *
   * Runs sanity and direction check and then assigns value to motor.
   */
  int defaultDirection;

  // If the direction is to stop.
  if (inputSpeed == 0) {
    motor->run(RELEASE);
    return;
  }

  // If the direction is backwards.
  if (inputSpeed < 0) {
    inputSpeed = -inputSpeed;
    defaultDirection = BACKWARD;
  }

  // If the direction is forwards.
  else {
    defaultDirection = FORWARD;
  }

  // If the speed is too high, cap it at 255.
  if (inputSpeed > 255) {
    inputSpeed = 255;
  }
```

```
113
114   // Run the motors as directed.
115   motor->setSpeed(inputSpeed);
116   motor->run(defaultDirection);
117 }
118
119 void setup() {
120   /*
121   Sets up appropriate pins, motor shields, and dynamic constants.
122   */
123   pinMode(rightSensorPin, INPUT);
124   pinMode(leftSensorPin, INPUT);
125   Serial.begin(baudrate);
126   AFMS.begin();
127   // Set the initial constants. It can be tuned over serial as well, but we should have a starting point before se
128   int speedDelta = initialSpeedDelta;
129   int threshold = initialThreshold;
130   int samples = initialSamplesPerMeasurement;
131   int wait = initialWait;
132 }
133
134 void loop() {
135   /*
136     Check the serial interface for commands, move along the track, check each WAIT and
137     turn to adjust as needed.
138   */
139
140   // Check the serial interface in case any of these constants need to be changed.
141   // Command formatting should be as such (note place value limitations):
142   // - To set speed delta : "D : ###"
143   // - To set threshold : "T : ####"
144   // - To set sample rate : "S : ###"
145   // - To set delay : "W : ##"
146   if(Serial.available()){
147         String command;
148         command = Serial.readStringUntil("\n");
149         Serial.println(command);
150         if (command[0] == ('D')) {
151           speedDelta = (command.substring(4,7)).toInt();
152           Serial.print("Delta␣updated␣to␣:␣");
153           Serial.print(speedDelta);
154           Serial.print('\n');
155         }
156         if (command[0] == 'T') {
157           threshold = (command.substring(4, 8)).toInt();
158           Serial.print("Threshold␣updated␣to␣:␣");
159           Serial.print(threshold);
160           Serial.print('\n');
161         }
162         if (command[0] == 'S') {
163           samples = (command.substring(4, 7)).toInt();
164           Serial.print("Sample␣rate␣updated␣to␣:␣");
165           Serial.print(samples);
166           Serial.print('\n');
167         }
168         if (command[0] == 'W') {
169           wait = (command.substring(4, 6)).toInt();
```

```
          Serial.print("Wait␣time␣update␣to␣:␣");
          Serial.print(wait);
          Serial.print('\n');
        }
    }
  bool leftSensorValue = verify_IR_sensor(leftSensorPin, threshold, initialSamplesPerMeasurement);
  bool rightSensorValue = verify_IR_sensor(rightSensorPin, threshold, initialSamplesPerMeasurement);
  // If the left sensor is on the tape, we want to speed up the left motor to get back on track.
  if (!leftSensorValue) {
    // Stop for a second to tune turning.
    set_speed_and_direction(leftMotor, 0);
    set_speed_and_direction(rightMotor, 0);
    delay(2);
    // Move in opposite direction
    set_speed_and_direction(leftMotor, -speedDelta);
    set_speed_and_direction(rightMotor, speedDelta + rightMotorConstant);
    Serial.print("Left␣above␣threshold.␣Speed␣delta:␣");
    Serial.print(speedDelta);
    Serial.print("\n");
    delay(100);
  }
  if (!rightSensorValue) {
    set_speed_and_direction(leftMotor, 0);
    set_speed_and_direction(rightMotor, 0);
    delay(2);
    set_speed_and_direction(rightMotor, -speedDelta - rightMotorConstant);
    set_speed_and_direction(leftMotor, speedDelta);
    Serial.print("Right␣above␣threshold.␣Speed␣delta:␣");
    Serial.print(speedDelta);
    Serial.print("\n");
    delay(100);
  }
  if (leftSensorValue && rightSensorValue) {
    set_speed_and_direction(rightMotor, initialSpeed + rightMotorConstant);
    set_speed_and_direction(leftMotor, initialSpeed);
    Serial.print("Going␣forwards.␣\n");
  }
  delay(wait);
}
```