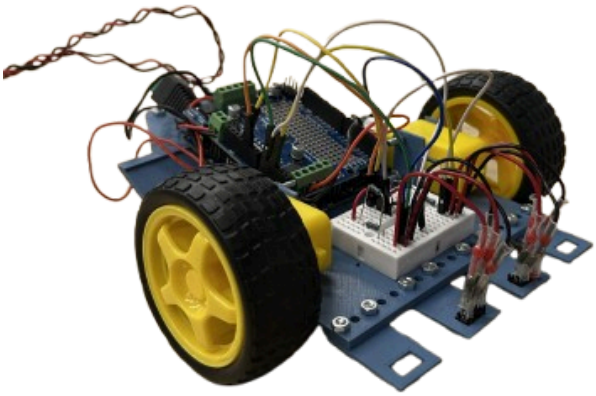
Mikey Ku
Heesung Han

Mini Project 3- DC Motor Control



Section 1: Overview and Introduction

Our goal for this mini project is to create a two-wheeled chassis that follows a track made of tape laid out on the floor. Firstly, we created a new mechanical chassis that matches our needs. We then developed a closed-loop controller that runs on an Arduino, integrating several IR reflectance sensors to determine whether the chassis is following the line correctly. After that, we coded a functionality to allow the user to test the robot and make changes in real time rather to reset and recompile the code.

This write up will dive deeper into the content we had talked about. We will discuss the system diagrams, the process of calibrating and integrating the sensors, the design of the chassis, the circuit diagram, the functionality of the controller, and the final results.

Section 2: System Diagram

To fully understand the components and interactions within our project, we created a data and energy flow diagram.

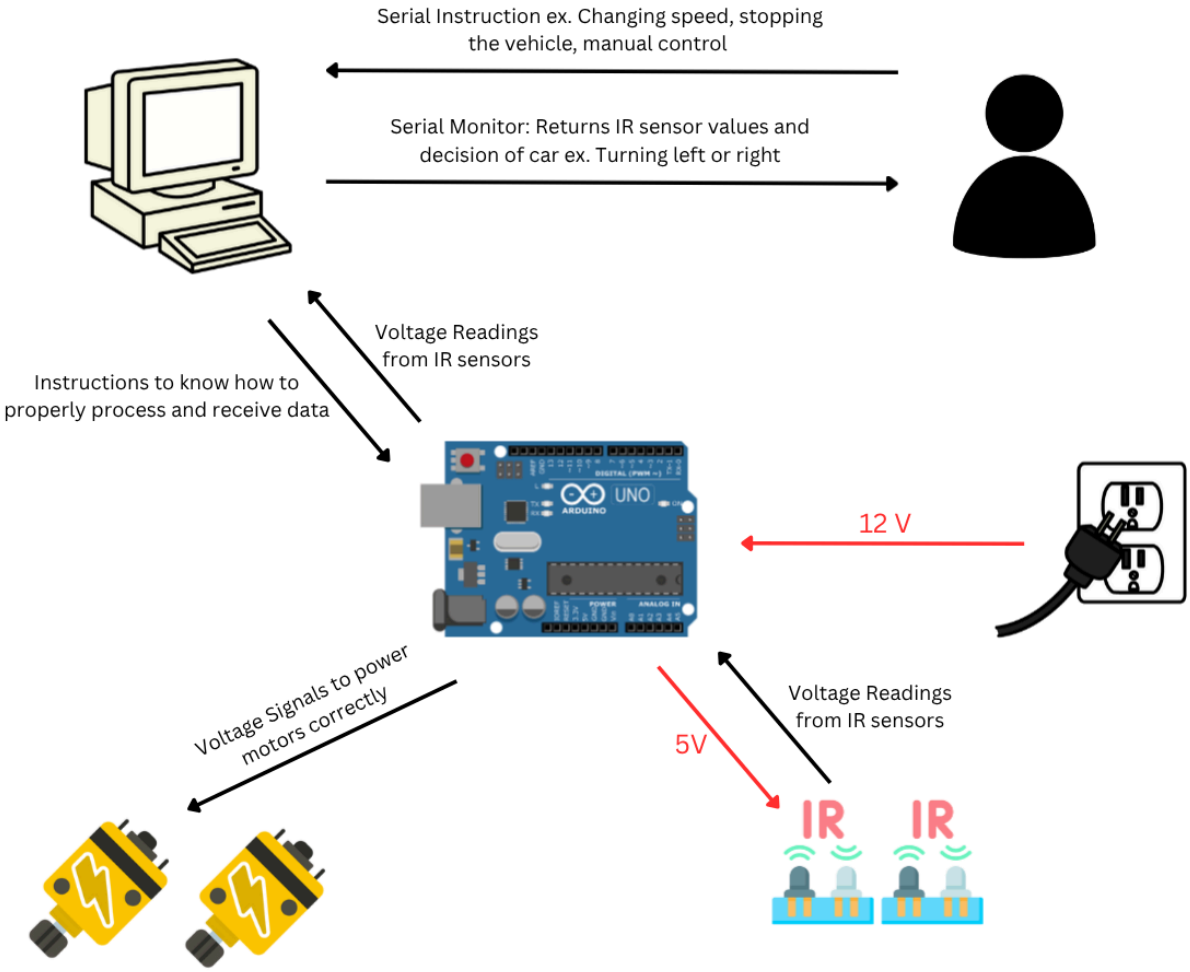


Figure 1: Data and Energy Flow Diagram

In this diagram, you can see how all the components in our system work together. It starts with the Arduino, which serves as the heart of the system. It connects nearly all the components, from the IR sensors to the DC motors, so that all data is processed through the Arduino. Next, we move to the computer, which provides the Arduino with instructions on how to process the inputs received from each component. The computer also receives live data from the sensors, which it uses to interpret the car's decisions and display them in the serial monitor for the user. The user, in turn, can type values or commands into the serial monitor to control the car. These commands are then passed back through the system to the motors, causing the car to move. Altogether, the system's components work in harmony to create an efficient flow of information and control.

Section 3: Sensor Calibration

To calibrate our sensors, we began by collecting readings from both white and dark surfaces, as these are the two key contrasts the sensors detect. We needed to understand these values so the robot could determine when the sensors were positioned over the line and identify any discrepancies between the left and right sensors.

To test this, we placed our robot on the surface after wiring all the sensors and recorded the sensor readings for both the line and the surrounding ground. The values we obtained showed that the white surface produced readings around 60, while the dark surface produced readings around 750. To allow for small variations in lighting and surface texture, we set the cutoff threshold to around 80, providing some leeway in case conditions were not consistent across the tape, which could have darker spots from being worn down.

Section 4: Mechanical Chassis

Although we were given a chassis to begin with, we found some key flaws with it. The first key flaw was the instability of the laser-cut motor mounts. We found that it was difficult to get a precise location when the wheels themselves were not stable. Another flaw was how high it was above the ground. Ideally, we would want to be lower to the ground so that it will be more stable. With these changes in mind, we designed and made a new chassis with the 3D printer.

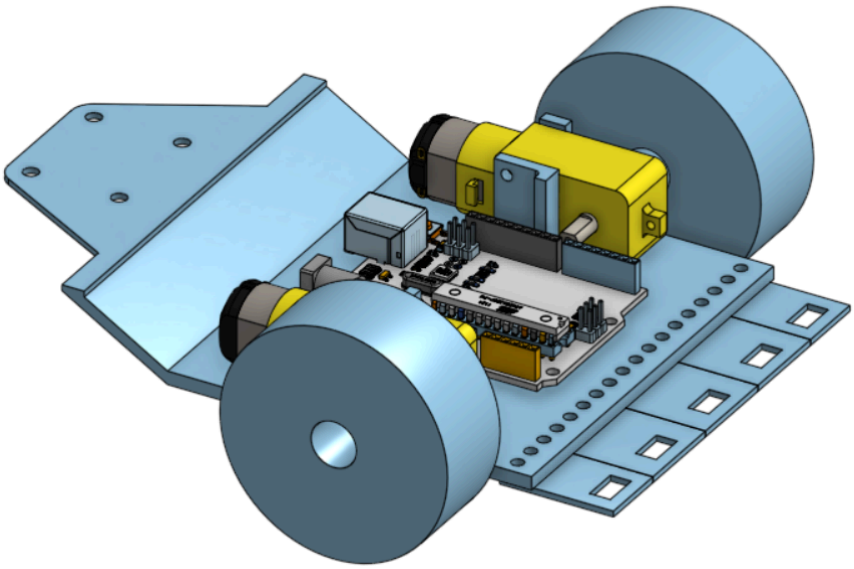


Figure 2: Full CAD of the chassis

Some noticeable features that we want to emphasize in this design are getting the robot closer to the ground, ensuring the robot is parallel to the ground, and the adjustable sensor mounts, which allow us to test different heights and widths. This design is composed of three distinct parts: the caster wheel mount, the gearmotor mounts, and the IR reflectance sensor mounts. We will go more in-depth, starting with the caster wheel mount.

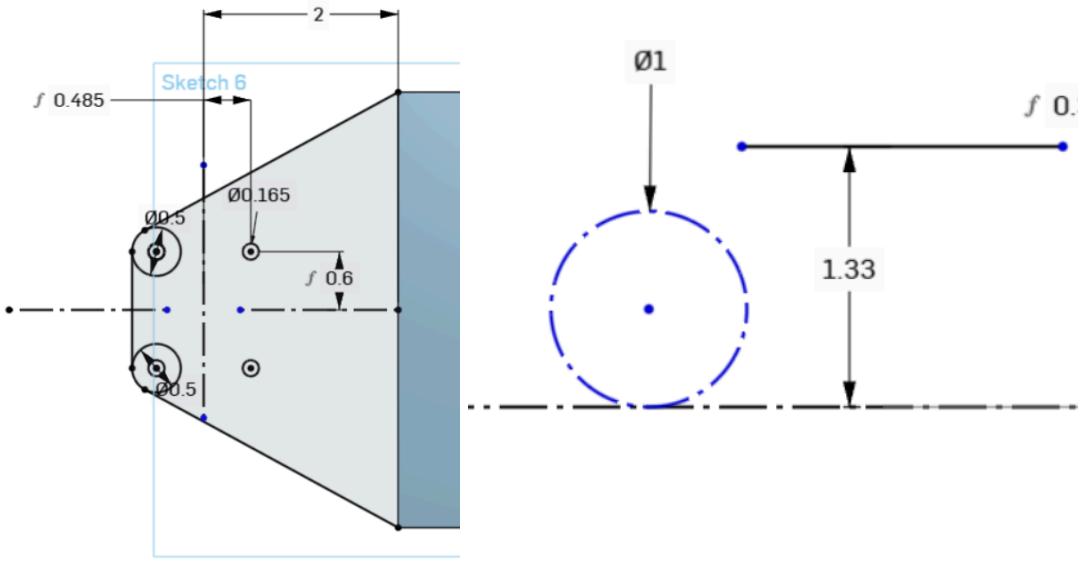


Figure 3: Initial Sketches of the Caster wheel

These are the initial sketches of the caster wheel mount. First, we found the mounting patterns of the caster wheels. We then found the height the caster wheel needs to be parallel to the ground by finding the exact model of the caster wheel. With this information, we knew the height and where to place the wheels. We decided on a slope feature down to the rest of the platform rather than going straight up to provide more support and a lower chance of snapping. We can now explore the mechanical decisions behind the gearmotor mount.

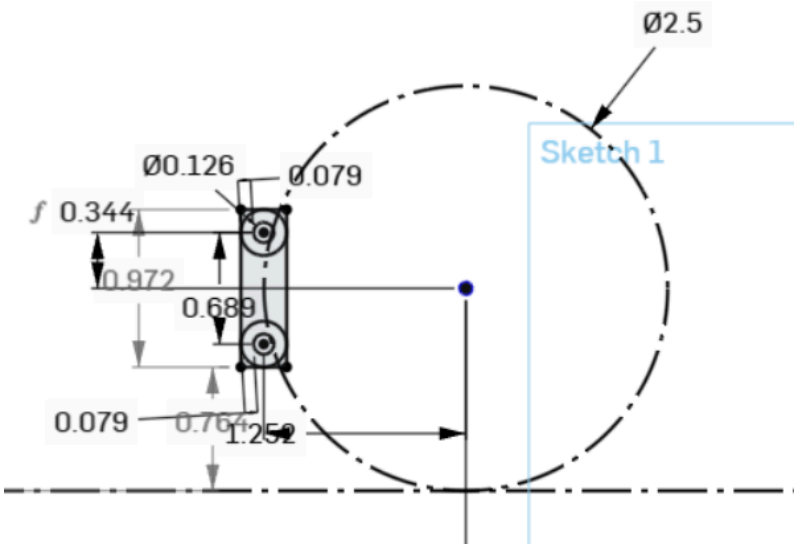


Figure 4: Initial Sketches of the gearmotor mount

This may seem confusing, but the general concept is to find the best location for the motor mount. We were given the gearmotor wheels, which were 2.5 inches. We knew the mounting

holes of the gearmotor mounting points, which were 17.5 mm apart from each other. From this, we were able to find the perfect position to put the mounting holes.

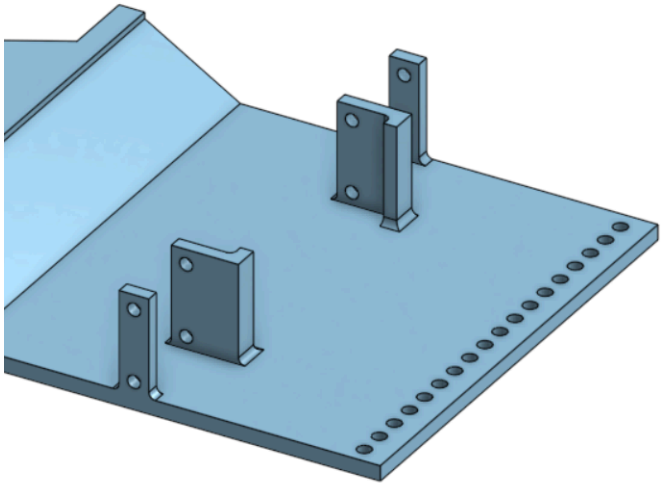


Figure 5: Detailed View of the gearmotor mount

With the initial sketches, we now knew where to put the mounts. Since one of the major issues was the instability of the laser-cut mounts, we wanted to sandwich the motors to be certain that there was no play or slack on the wheels. On the inner mounting holes, we gave it an L shape so that it wouldn't snap from external forces. We also had 2.625 inches between the two motors because we wanted to fit an Arduino in the middle.

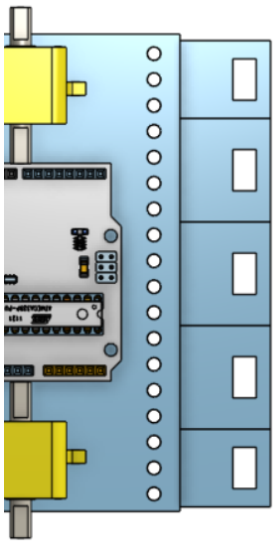


Figure 6: Detailed View of the IR reflectance sensor mount

Finally, we will expand on the mechanical reasoning for the IR reflectance sensor mount. We knew that we didn't want to just stick with one design, as there are unforeseen circumstances that

might make us need more than two sensors, or if they need different heights. Using screws and nuts, we can move the different sensor mounts around. This helps us by saving time and filament from 3D printing numerous times.

Section 5: Circuit Diagram

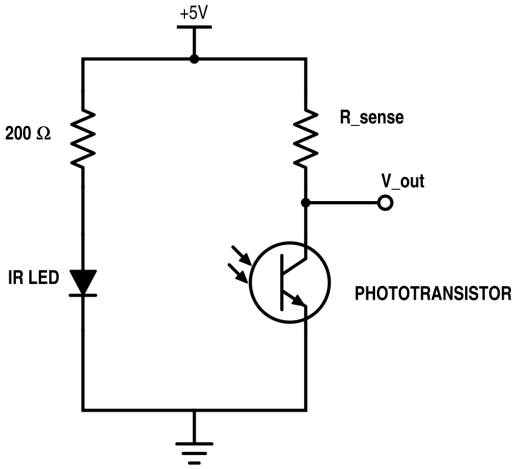


Figure 7: Circuit Diagram

When developing the circuit, there was one key component we needed to determine: the value of **R_Sense**, which can be seen in the circuit diagram provided on the left (Figure 7). This diagram represents the circuit for the IR sensor. The IR sensor consists of four pins: the anode, cathode, emitter, and collector. In this diagram, the anode and cathode correspond to the positive and negative terminals of the IR LED, while the emitter and collector represent the terminals of the phototransistor. With this understanding, we could determine how to wire the entire circuit.

Next, we needed to calculate the R_Sense value. Based on the TRCT5000 datasheet, the emitter current of the phototransistor was tested at approximately 1 mA. Using this information, we applied Ohm's Law to find the resistor value. Since the supply voltage for the circuit is 5 V and the current is 1 mA, the resistor value can be calculated as follows:

$$R = \frac{V}{I} = \frac{5v}{0.001\,A} = 5000\ \Omega$$

Figure 8: Calculations using Ohm's Law

From this, we determined that the R_Sense value should be 5 kΩ. With this information, we finalized our circuit and wired everything together in KiCad. The circuit diagram above was used to guide the connections for the IR sensors. Additionally, we used a custom KiCad Arduino motor shield design to show how the motors are wired and how the 12 V power source is integrated into the system.

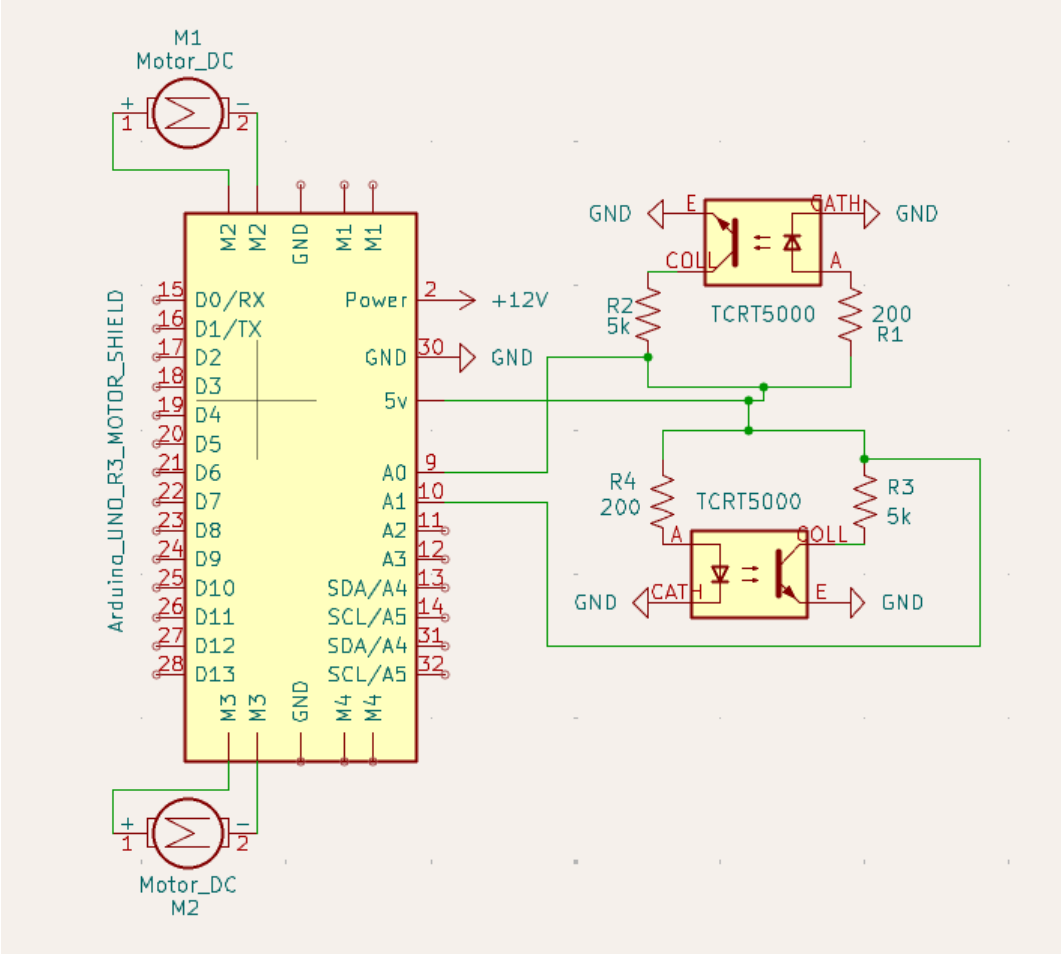


Figure 9: KiCad wiring diagram

Section 6: Controller

When developing our line-following logic, we used the strict line-following approach, starting with two IR sensors. By “strict line following,” we mean that the sensors are spaced just wide enough to fully cover the line.

Our control logic is simple:

- If both IR sensors detect white, the robot moves straight.
- If the left sensor detects a darker surface, the robot turns right.
- If the right sensor detects a darker surface, the robot turns left.

The robot continuously adjusts itself until both sensors once again detect the same surface, ensuring it stays centered on the line. We also implemented a feature where, if neither sensor detects the line, the robot will keep pivoting until it finds it again. The following section shows the code that implements this logic.

To start, we defined several base values that are used throughout the program. These include the different speed settings, such as base, turn, and pivot, as well as the threshold value that distinguishes the white tape from the surrounding surface.

```
// ---------------- Thresholds & Speeds ----------------
const int threshold = 80;     // Tape vs Not tape
int baseSpeed = 40;           // Normal forward speed
const int turnSpeed = 35;     // Turn Speed
const int pivotSpeed = 30;    // Pivot speed when line is lost
```

Figure 10: Base Values

From here, we can start breaking down how everything works. Let's look at the different types of movements the car can perform:

Forward: The concept of forward movement is very simple. To make the car move forward, both motors must run at the same time and at the same speed. This behavior is shown in the code on the right.

```
void forward(int spd) {
  leftMotor->setSpeed(spd);
  rightMotor->setSpeed(spd);
  leftMotor->run(FORWARD);
  rightMotor->run(FORWARD);
}
```

Figure 11: Forward Movement

Left/Right: The concept of turning is slightly different. In this case, one motor runs faster than the other, causing the car to turn in the direction of the slower motor. For example, if the right motor speed is lower than the left, the car will turn to the right because the left motor is doing more work. In our setup, the base speed is higher than the turn speed, so whichever motor is set to the turn speed determines the direction of the turn. For instance, if the left motor is running at the turn speed, the car will turn to the left.

```
void turnLeft() {
  leftMotor->setSpeed(turnSpeed);
  rightMotor->setSpeed(baseSpeed);
  leftMotor->run(FORWARD);
  rightMotor->run(FORWARD);
}
```

Figure 12: Turn Left /Right

Pivot Left/Right: The concept of pivoting is similar to turning. The main difference lies in the speeds used and how it is applied later in the program. In the basic code, the same principle applies: to pivot in one direction, one motor must run slower than the other, causing the car to rotate in that direction. The pivot speed is slower than the base

```
void pivotRight() {
  leftMotor->setSpeed(pivotSpeed);
  rightMotor->setSpeed(pivotSpeed);
  leftMotor->run(FORWARD);
  rightMotor->run(BACKWARD);
}
```

Figure 13: Pivot Left /Right

speed, so whichever motor is set to the pivot speed determines the direction of the pivot. The car will continue to pivot slowly until it realigns with the line.

Now that we have established how the car moves, we can discuss the logic that controls its operation and looping behavior. Below is the full set of conditionals that determine how the car responds to different sensor readings.

```
// ----- Line following logic -----
// Left IR on line
if (leftOnWhite && !rightOnWhite) {
  Serial.println("Turning LEFT");
  turnLeft();
  lastTurn = -1;
}
// Right IR on line
else if (!leftOnWhite && rightOnWhite) {
  Serial.println("Turning RIGHT");
  turnRight();
  lastTurn = 1;
}
// Both on line
else if (leftOnWhite && rightOnWhite) {
  Serial.println("Going STRAIGHT");
  forward(userSpeed);
  lastTurn = 0;
}
else {
  // Both sensors off the line (pivot)
  Serial.print("Lost line – Pivoting ");
  if (lastTurn <= 0) {
    Serial.println("LEFT");
    pivotLeft();
  } else {
    Serial.println("RIGHT");
    pivotRight();
  }
}
```

Figure 14: Line Following Logic

There are four main scenarios:

1. **Left sensor on the line, right sensor off:** This means the line is closer to the left side, so the car needs to turn left until both sensors detect the same surface again.

2. **Right sensor on the line, left sensor off:** This is the opposite situation, where the line is closer to the right side. In this case, the car turns right until it realigns with the line.

3. **Both sensors on the line:** This is the simplest case. When both sensors detect the line, the car continues moving straight at a set speed.

4. **Both sensors off the line**: In this situation, the car has lost the line completely. It will pivot until the sensors detect the line again.

In addition to the main logic and controller, we added several features to allow user interaction with the car. These features include the ability to stop the car, set custom speeds, and control basic movements such as moving forward, moving backward, and resuming normal line-following behavior.

To achieve this, we first take input from the Serial Monitor in the Arduino IDE, which the Arduino reads and processes in real time.

```
void checkSerialCommand() {
  if (Serial.available() > 0) {
    String command = Serial.readStringUntil('\n');
    command.trim();
```

Figure 15: Serial Monitor Input

From here, we process the user input and define a few specific commands for the Arduino to recognize. In our case, there are five main commands that the system looks for:

1. “stop”
2. “resume”
3. “speed <value 0-255>”
4. “backward”
5. “auto”

Here is the code that represents what happens when each thing is typed:

```
if (command.equalsIgnoreCase("stop")) {
  isStopped = true;
  forward(0);
  Serial.println("Car stopped.");
}
```

Figure 16: Code “stop” is typed

If the Serial Monitor receives the word “stop,” the program sets a variable called isStopped to true and sets the motor speed to zero, resulting in no movement. The car will remain in place until a new command is entered, since the main loop continues to skip execution while isStopped is set to true.

```
else if (command.equalsIgnoreCase("resume")) {
  isStopped = false;
  Serial.println("Resuming line following...");
```

Figure 17: Code “resume” is typed

If the word “resume” is entered, the program sets isStopped back to false, allowing the original line-following algorithm to run normally again.

```
else if (command.startsWith("speed ")) {
  int val = command.substring(6).toInt();
  if (val >= 0 && val <= 255) {
    userSpeed = val;
    baseSpeed = val;
    Serial.print("Speed set to ");
    Serial.println(userSpeed);
  } else {
    Serial.println("Invalid speed. Enter 0–255.");
  }
```

Figure 18: Code "speed" is typed

Once the car is stopped, there are a few options for what the user can do next. One option is to type "speed" into the Serial Monitor to change the car's running speed. To enter a specific value (from 0 to 255), type the word "speed," followed by a space and then the value. That number becomes the new speed, and the code above includes logic to filter out invalid inputs.

```
else if (command.equalsIgnoreCase("forward")) {
  manualMode = true;
  forward(userSpeed);
  delay(1000);
  forward(0);
  Serial.println("Moving forward manually.");
}
```

Figure 19: Code "forward" is typed

The code above shows what happens when the word "forward" is entered. In this case, the car moves straight for about one second and then stops. This function is useful for nudging the car back onto the track if it has drifted off.

```
else if (command.equalsIgnoreCase("backward")) {
  manualMode = true;
  leftMotor->setSpeed(userSpeed);
  rightMotor->setSpeed(userSpeed);
  leftMotor->run(BACKWARD);
  rightMotor->run(BACKWARD);
  delay(1000);
  forward(0);
  Serial.println("Moving backward manually.");
}
```

Figure 20: Code "backward" is typed

The code above performs the opposite action. It makes the car move backward for about one second, allowing a small backward nudge to help the car get back on the track if needed.

```
else if (command.equalsIgnoreCase("auto")) {
  manualMode = false;
  Serial.println("Switched back to auto line-follow mode.");
}
else {
  Serial.println("Unknown command. Try: stop, resume, speed <0-255>, forward, backward, auto");
}
```

Figure 21: Code "auto" is typed

The code above shows the final two conditionals. The "auto" command returns the car to automatic line-following mode, preparing it to resume autonomous driving. The final conditional handles invalid inputs and ensures that the user enters correct commands for the Arduino to process. Altogether, this is how the Arduino interprets input values and controls the car to successfully follow the track.

Section 7: Plots

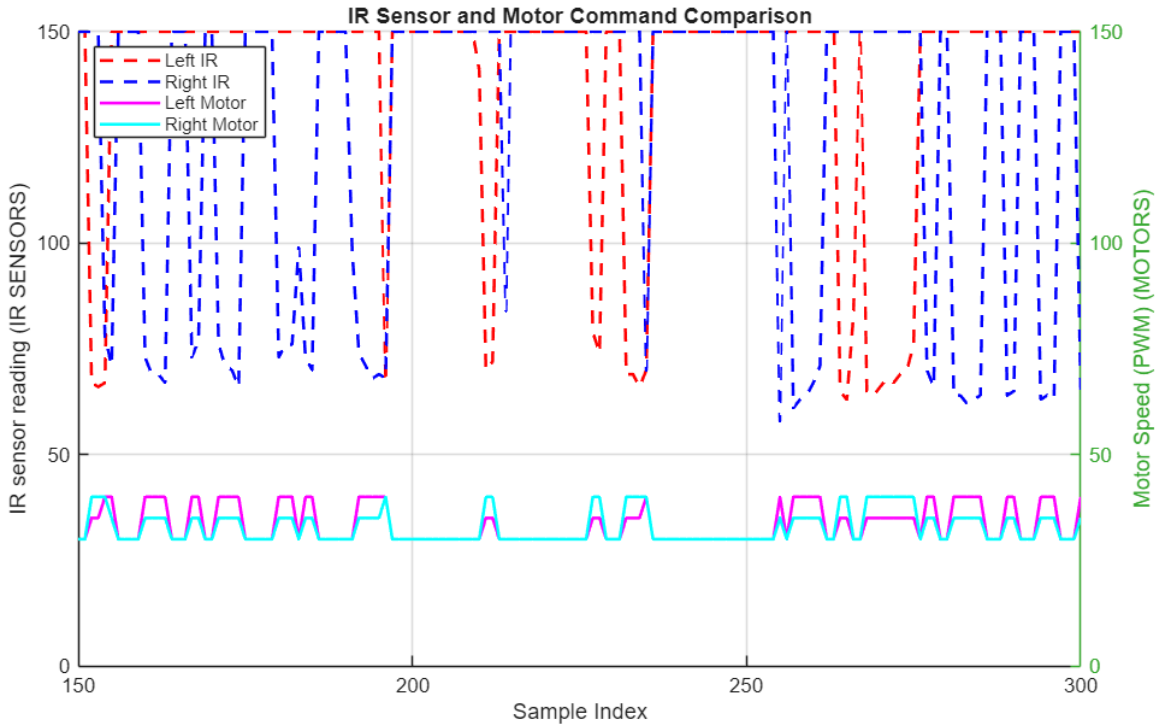


Figure 22: Plot of IR Sensor and Motor Command Comparison

To better visualize how the car behaved during a run, we created a plot in MATLAB showing the IR sensor readings and motor outputs over time. We applied a cutoff value to the IR sensors because the large difference between the minimum and maximum readings made the data difficult to interpret. In the plot, each drop in the IR sensor values corresponds to a reaction from the motors, illustrating how the car responded to different sections of the track. This visualization clearly shows how the car perceived and reacted to its environment throughout the run. It also provides an approximate visualization of the car's movement along the track. The points where the motor voltages flatline represent straight sections of the course where no turns were needed, while the regions with significant fluctuations in IR values and motor speeds indicate turns or corrective adjustments. Overall, this plot provides valuable insight into the car's behavior and control performance during the run.

Section 8: Video

The following is a video of the system with line following.
https://youtube.com/shorts/_yGmEujxP80?si=z3EPTMJkCsCh1JCX

The following is a video of the serial input with behavior change of the system.
https://youtube.com/shorts/wv4oc4SpL1k?si=m6YTCcEHzHFikN6-

Section 9: Reflection

Overall, the project went very smoothly. We were able to finish at a good pace without feeling too much pressure from deadlines. Each of us focused on different aspects of the project, and we worked efficiently as a team, communicating well whenever we needed input or feedback from each other.

There are a few mechanical improvements we could make in future iterations. While designing the chassis, the gear motor mount was too thin and eventually snapped on one side. Next time, we can reinforce areas that experience higher stress to improve durability. We also realized that we did not fully consider how the chassis would be 3D printed. Some parts required excessive support material, which wasted filament and printing time. In the future, we will take print orientation and support usage into account during the design process.

There are also improvements to be made on the electrical and programming side. Our current robot is somewhat limited by its speed and accuracy. At times, it struggles to detect the line or react quickly enough to changes in the path. A major improvement would be to add more IR sensors to increase the detection range and resolution. This would allow the robot to detect the line earlier and respond with greater precision. Placing the sensors in an arc configuration is a common and effective method to achieve this, as it provides a wider field of view for tracking the line.

Finally, we can improve our teamwork by collaborating more simultaneously. There were times outside of class when only one person was working while the other was unavailable, which sometimes affected our communication. In future projects, working together in real time more often would help maintain consistency and strengthen coordination.

Section 10: Source Code

Arduino Code:

```
#include <Wire.h>
#include <Adafruit_MotorShield.h>
#include "utility/Adafruit_MS_PWMServoDriver.h"

// Motor setup
Adafruit_MotorShield AFMS = Adafruit_MotorShield();
Adafruit_DCMotor *leftMotor  = AFMS.getMotor(3);
Adafruit_DCMotor *rightMotor = AFMS.getMotor(2);

// IR sensor pins
const int leftIR  = A2;
const int rightIR = A0;

// Thresholds & Speeds
```

```
const int threshold = 80;     // Tape vs Not tape
int baseSpeed = 40;           // Normal forward speed
const int turnSpeed = 35;     // Turn Speed
const int pivotSpeed = 30;    // Pivot speed when line is lost

// State tracking
int lastTurn = 0; // -1 = left, 1 = right, 0 = none

// Serial control states
bool isStopped = false;
bool manualMode = false;
int userSpeed = baseSpeed;

// Motor state tracking
int leftMotorSpeed = 0;
int rightMotorSpeed = 0;
String leftMotorDir = "STOP";
String rightMotorDir = "STOP";

// Setup
void setup() {
  Serial.begin(9600);
  if (!AFMS.begin()) {
    Serial.println("Could not find Motor Shield. Check wiring.");
    while (1);
  }

  Serial.println("2-Sensor Line Follower with Serial Commands Ready!");
  Serial.println("-------------------------------------------------");
  Serial.println("Commands:");
  Serial.println("stop, resume, speed <0-255>, forward, backward, auto");
  Serial.println("-------------------------------------------------");
  Serial.println("Left IR | Right IR | State | L-Speed | R-Speed | L-Dir |
R-Dir");
  Serial.println("-------------------------------------------------");
}

// Loop
void loop() {
  checkSerialCommand();
  if (isStopped) return;
  if (manualMode) return;

  // Read sensor values
  int leftVal  = analogRead(leftIR);
```

```
int rightVal = analogRead(rightIR);

bool leftOnWhite  = (leftVal  < threshold);
bool rightOnWhite = (rightVal < threshold);

// Print readings
Serial.print(leftVal);
Serial.print(" | ");
Serial.print(rightVal);
Serial.print(" | ");

// Line following logic
if (leftOnWhite && !rightOnWhite) {
  Serial.print("Turning LEFT");
  turnLeft();
  lastTurn = -1;
}
else if (!leftOnWhite && rightOnWhite) {
  Serial.print("Turning RIGHT");
  turnRight();
  lastTurn = 1;
}
else if (leftOnWhite && rightOnWhite) {
  Serial.print("Going STRAIGHT");
  forward(userSpeed);
  lastTurn = 0;
}
else {
  Serial.print("Lost line -- Pivoting ");
  if (lastTurn <= 0) {
    Serial.print("LEFT");
    pivotLeft();
  } else {
    Serial.print("RIGHT");
    pivotRight();
  }
}

// Print motor state
Serial.print(" | L:");
Serial.print(leftMotorSpeed);
Serial.print(" R:");
Serial.print(rightMotorSpeed);
Serial.print(" | DirL:");
Serial.print(leftMotorDir);
```

```
  Serial.print(" DirR:");
  Serial.println(rightMotorDir);

  delay(80);
}

// Serial Command Handler
void checkSerialCommand() {
  if (Serial.available() > 0) {
    String command = Serial.readStringUntil('\n');
    command.trim();

    if (command.equalsIgnoreCase("stop")) {
      isStopped = true;
      forward(0);
      Serial.println("Car stopped.");
    }
    else if (command.equalsIgnoreCase("resume")) {
      isStopped = false;
      Serial.println("Resuming line following...");
    }
    else if (command.startsWith("speed ")) {
      int val = command.substring(6).toInt();
      if (val >= 0 && val <= 255) {
        userSpeed = val;
        baseSpeed = val;
        Serial.print("Speed set to ");
        Serial.println(userSpeed);
      } else {
        Serial.println("Invalid speed. Enter 0-255.");
      }
    }
    else if (command.equalsIgnoreCase("forward")) {
      manualMode = true;
      forward(userSpeed);
      delay(1000);
      forward(0);
      Serial.println("Manual forward complete.");
    }
    else if (command.equalsIgnoreCase("backward")) {
      manualMode = true;
      backward(userSpeed);
      delay(1000);
      forward(0);
      Serial.println("Manual backward complete.");
```

```
    }
    else if (command.equalsIgnoreCase("auto")) {
      manualMode = false;
      Serial.println("Switched back to auto line-follow mode.");
    }
    else {
      Serial.println("Unknown command. Try: stop, resume, speed <0-255>,
forward, backward, auto");
    }
  }
}

// Motion Functions
void forward(int spd) {
  leftMotor->setSpeed(spd);
  rightMotor->setSpeed(spd);
  leftMotor->run(FORWARD);
  rightMotor->run(FORWARD);
  leftMotorSpeed = spd;
  rightMotorSpeed = spd;
  leftMotorDir = "FWD";
  rightMotorDir = "FWD";
}

void backward(int spd) {
  leftMotor->setSpeed(spd);
  rightMotor->setSpeed(spd);
  leftMotor->run(BACKWARD);
  rightMotor->run(BACKWARD);
  leftMotorSpeed = spd;
  rightMotorSpeed = spd;
  leftMotorDir = "REV";
  rightMotorDir = "REV";
}

void turnLeft() {
  leftMotor->setSpeed(turnSpeed);
  rightMotor->setSpeed(baseSpeed);
  leftMotor->run(FORWARD);
  rightMotor->run(FORWARD);
  leftMotorSpeed = turnSpeed;
  rightMotorSpeed = baseSpeed;
  leftMotorDir = "FWD";
  rightMotorDir = "FWD";
}
```

```
void turnRight() {
  leftMotor->setSpeed(baseSpeed);
  rightMotor->setSpeed(turnSpeed);
  leftMotor->run(FORWARD);
  rightMotor->run(FORWARD);
  leftMotorSpeed = baseSpeed;
  rightMotorSpeed = turnSpeed;
  leftMotorDir = "FWD";
  rightMotorDir = "FWD";
}

void pivotLeft() {
  leftMotor->setSpeed(pivotSpeed);
  rightMotor->setSpeed(pivotSpeed);
  leftMotor->run(BACKWARD);
  rightMotor->run(FORWARD);
  leftMotorSpeed = pivotSpeed;
  rightMotorSpeed = pivotSpeed;
  leftMotorDir = "REV";
  rightMotorDir = "FWD";
}

void pivotRight() {
  leftMotor->setSpeed(pivotSpeed);
  rightMotor->setSpeed(pivotSpeed);
  leftMotor->run(FORWARD);
  rightMotor->run(BACKWARD);
  leftMotorSpeed = pivotSpeed;
  rightMotorSpeed = pivotSpeed;
  leftMotorDir = "FWD";
  rightMotorDir = "REV";
}
```

MATLAB Code:

```
% Load the CSV data
data = readtable('ArduinoVals.csv');

% Extract columns
leftIR = data.("LeftIR");
rightIR = data.("RightIR");
```

```
leftMotor = data.("LeftMotor");
rightMotor = data.("RightMotor");

% Cap IR values at 150
leftIR(leftIR > 200) = 150;
rightIR(rightIR > 200) = 150;

t = (1:length(leftIR))';
startIdx = 150;
endIdx = 300;

t = t(startIdx:endIdx);
leftIR = leftIR(startIdx:endIdx);
rightIR = rightIR(startIdx:endIdx);
leftMotor = leftMotor(startIdx:endIdx);
rightMotor = rightMotor(startIdx:endIdx);

% Plot
figure;
hold on; grid on;

plot(t, leftIR, 'r--', 'LineWidth', 1.5);
plot(t, rightIR, 'b--', 'LineWidth', 1.5);
plot(t, leftMotor, 'm-', 'LineWidth', 1.5);
plot(t, rightMotor, 'c-', 'LineWidth', 1.5);

% Use unified Y-axis
ylim([0 150]);

% Labels, title, legend
xlabel('Sample Index');
ylabel('Value (IR and Motor)');
title('IR Sensor and Motor Command Comparison');
legend({'Left IR', 'Right IR', 'Left Motor', 'Right Motor'},
'Location', 'northwest');
```