Line-Sensing Robot

Section 2, Mini-Project 3, Hong Zhang and Mateo Otero-Diaz

Introduction

In this mini-project, we designed and built an Arduino-based line-following robot that autonomously navigates a taped track using closed-loop feedback control. Our goal was to integrate infrared reflectance sensors with DC motor control to create a robot capable of following a non-circular course as quickly and accurately as possible.

This project provided hands-on experience with actuator control, sensor integration, and feedback theory. We implemented a closed-loop controller that uses data from four infrared (IR) reflectance sensors to independently drive two DC motors, keeping the robot centered on the track. A key feature of our system is a serial interface that allows for real-time tuning of the controller's behavior from a laptop without needing to recompile or upload new code.

To construct the robot, we used the provided acrylic chassis, an Arduino with an Adafruit v2 Motor Shield, two DC gearmotors, and two IR reflectance sensors. We also designed and built a custom, non-permanent mechanical mount to affix our sensors and electronics to the chassis, fulfilling the project's mechanical design requirement.

System Diagram (Data & Energy Flow)

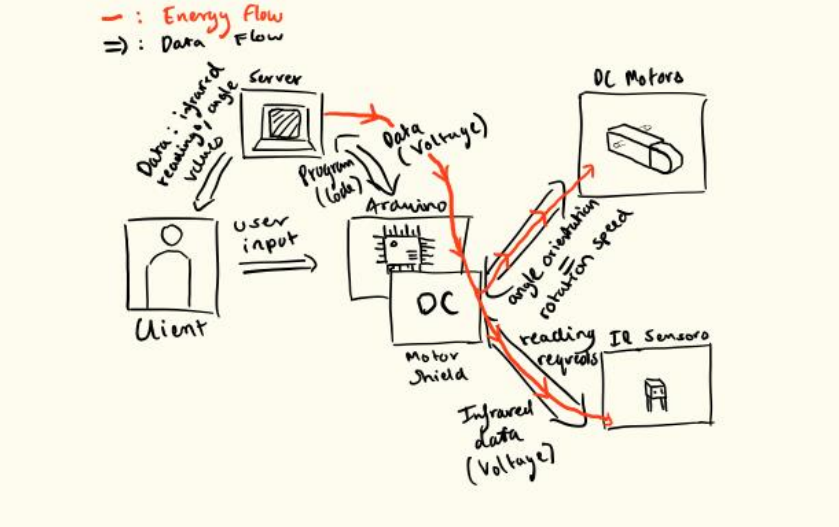


Figure 1: Energy and Data Flow Diagram for our System

IR Sensor Testing and Calibration:

To ensure our robot could reliably distinguish between the track tape and the floor, we performed a multi-step calibration process for our four infrared (IR) reflectance sensors. The primary goal was to establish a unique numerical threshold for each sensor that would define the boundary between seeing the line and seeing the floor.

First, we conducted an initial functionality test by creating a simple Arduino sketch. This program read the analog values from each of the four sensors in a continuous loop and printed them to the Serial Monitor. We confirmed that each sensor was working by waving a hand over them and observing the corresponding change in their output values.

When the sensors were confirmed to be working, we used the same program to collect performance data. We placed the robot on the floor and recorded the range of analog values for each sensor. Next, we positioned the robot so the sensors were directly over the white tape and again recorded the range of values. This process was repeated for all four sensors, yielding the following data:

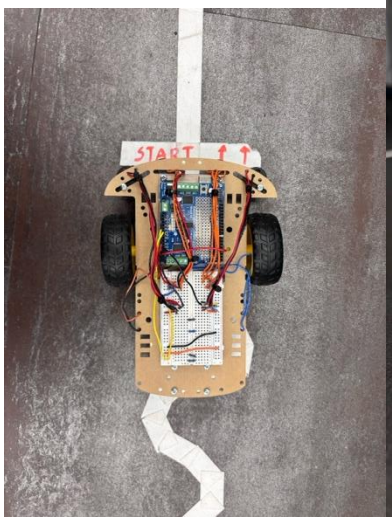
- Most Left Sensor (s2): Floor readings were ~943-955, while tape readings were ~640-674.
- Middle Left Sensor (s1): Floor readings were ~950-958, while tape readings were ~647-666.
- Middle Right Sensor (s4): Floor readings were ~810-820, while tape readings were ~589-620.
- Most Right Sensor (s3): Floor readings were ~815-826, while tape readings were ~580-624.

Finally, we calculated a specific threshold for each sensor by finding the approximate midpoint between its tape and floor reading ranges. This method ensures a clear decision boundary. The final thresholds were determined to be 800 for the two left sensors (s1, s2) and 710 for the two right sensors (s3, s4). In our control logic, any reading below this threshold is interpreted as "tape," and any reading above it is interpreted as "floor."

Mechanical Integration

Our system has minimal, but useful mechanical implementation in terms of mounting points for sensors and relevant actuators, as well as cable management. As seen in figures 2 and 3, our cabling for each of the four sensors is designed to restrict incomprehensibility and allow for a visual understanding of our circuit. The “packets”, comprising of the anode, cathode, emitter,

and collector, are zip-tied together in order to restrict their motion, allowing for their function to remain without the “messy” bunching that can occur when these systems are assembled.



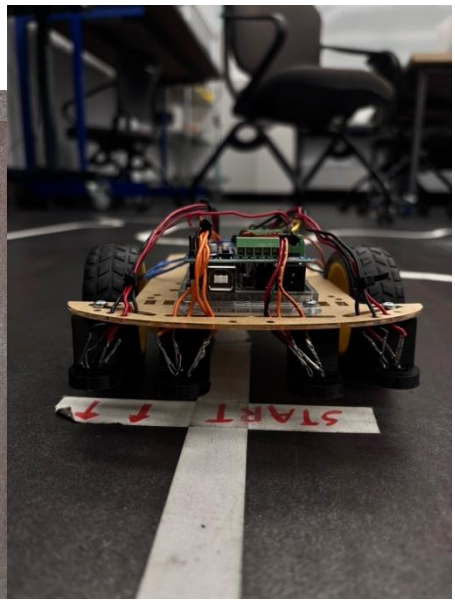


Figure 2: Bird’s eye Perspective Figure 3: Front-view Perspective

Figure 4 shows the 3D model designed for our sensor mounts. This design positions the sensors with the necessary clearance from the ground while also guiding the cables toward holes in the chassis to allow for easy routing. For improved aesthetics, the model incorporates clean fillets and a sloped transition between its mounting surface and the plane where the sensor sits. Using four independent mounts, rather than a single unified piece, provides modularity and makes it simple to adjust the sensor spacing based on insights gained during testing.

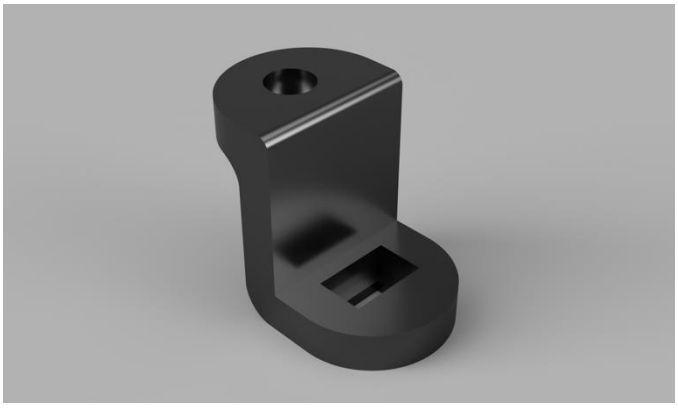


Figure 4: Sensor Mount 3-D Model

This modular approach avoids a wasteful design cycle. A single, unified mount would require guessing at the optimal sensor spacing without any test data, meaning the entire part would have to be discarded if that initial guess proved to be ineffective. Our system allows for easy two-axis mobility of our sensor location. It is possible to move the sensors closer or farther apart as well as rotate their orientation along the z-axis of their center, based on our test insights. This makes for facile modulation and the reuse of the same mounts, independent of failures.

Finally, figure 5 represents an original mechanical choice to create an aesthetic “cover” for the system internals. The purpose of the housing component was purely aesthetic and served no real functional purpose but would have allowed the finished product to achieve a certain quality of presentation.

Figure 5: Housing Design with Base and Sliding Lid

The housing was designed, prototyped over 5 iterations, and mounted during testing, but it was ultimately removed due to its hampering on cable management as well as deterrence of consistent performance. Simply said, the spacing for the housing walls did not allow “breathing space” for the components, which simply means that cable routing, modulation, and iteration on system organization was limited and made exceedingly difficult, even with the design of a lid. While it is quite disappointing that the housing was “scrapped” at the tail end of our test cycle, its purpose allowed for its removal to be quite non-consequential.

Circuit Diagram

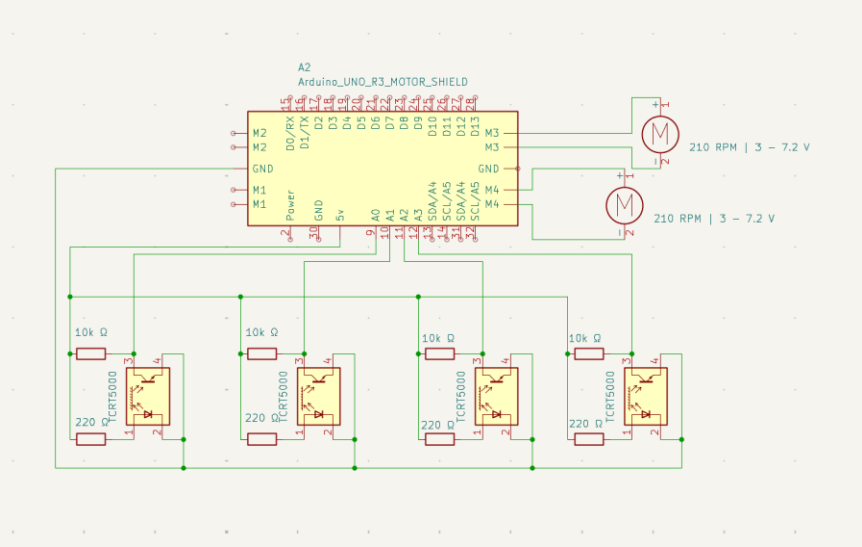


Figure 6: A complete circuit diagram showing the integration of the power system, Arduino controller, motor shield, four TCRT5000 IR reflectance sensors, and two DC motors.

Our circuit is designed around an Arduino Uno with an Adafruit v2 Motor Shield, which serves as the central hub for control and power distribution. The primary sensing component consists of an array of four TCRT5000 IR reflectance sensors. Each TCRT5000 package contains an infrared LED and a phototransistor. For each sensor, the IR LED's anode is connected to the Arduino's 5V rail via a 220Ω resistor, and its cathode is tied to ground. The phototransistor's emitter is connected directly to ground, while its collector is pulled up to the 5V rail through a 10kΩ resistor. This

configuration creates a voltage divider, where the output signal (V_out) is taken from the junction between the collector and the 10kΩ resistor. This analog voltage, which varies based on the amount of reflected infrared light, is fed into one of the Arduino's analog input pins (A0-A3). The two DC motors are connected directly to the motor shield's M3 and M4 screw terminals. The entire system is powered by a 12V external supply, which is split to power both the motor shield directly and the Arduino through its DC barrel jack, allowing the Arduino's onboard regulator to provide a stable 5V to the sensor array.

Controller and Serial Interface Explained

Our robot uses hierarchical, state-based logic, a form of bang-bang control, to interpret sensor data and command the motors. With this technique, the motors are commanded to one of several distinct states (e.g. straight, sharp left, gentle right) rather than a continuously variable output. The process repeats continuously, ensuring the robot can react to the track in real-time. In every loop, the controller first reads the raw analog values from the four IR reflectance sensors and converts them into simple boolean states by comparing them against their pre-determined calibration thresholds. This tells the controller which sensors are currently detecting the tape. The controller then enters a prioritized if/else if structure to decide on the appropriate action, ordering cases from most to least critical to ensure it handles sharp turns with the highest priority. The highest priority is given to detecting a sharp U-turn; for instance, if the two right-side sensors (S3 and S4) both detect the tape, the controller executes an immediate sharp pivot to the right. If only the single outermost right sensor (S3) detects the tape, it is treated as a standard sharp turn. Conversely, the ideal 'on-center' state occurs when the two middle sensors (S1 and S4) are on the tape, commanding the robot to drive straight. Gentle corrections are made when only one of the middle sensors detects the tape, signaling a slight drift. Finally, if no sensors detect the tape, the controller uses a “LastTurnDirection” variable to continue its last action, effectively "searching" for the line instead of just stopping.

We also implemented a serial command interface. This system allows for the live adjustment of the robot's baseSpeed variable by sending commands from the Arduino IDE's Serial Monitor. The interface is built into the main loop and continuously checks for incoming serial data. We created a simple command protocol where a command starting with the character 's' followed by an integer (e.g. s100) will update the global baseSpeed variable to that new value. The robot then provides feedback in the Serial Monitor, confirming the change. This functionality was very useful for efficient testing, as we could observe the robot's behavior on the physical track and immediately adjust its speed to be faster on straight sections or more controlled during complex turns, dramatically speeding up the tuning process.

Sensor Data Plot:

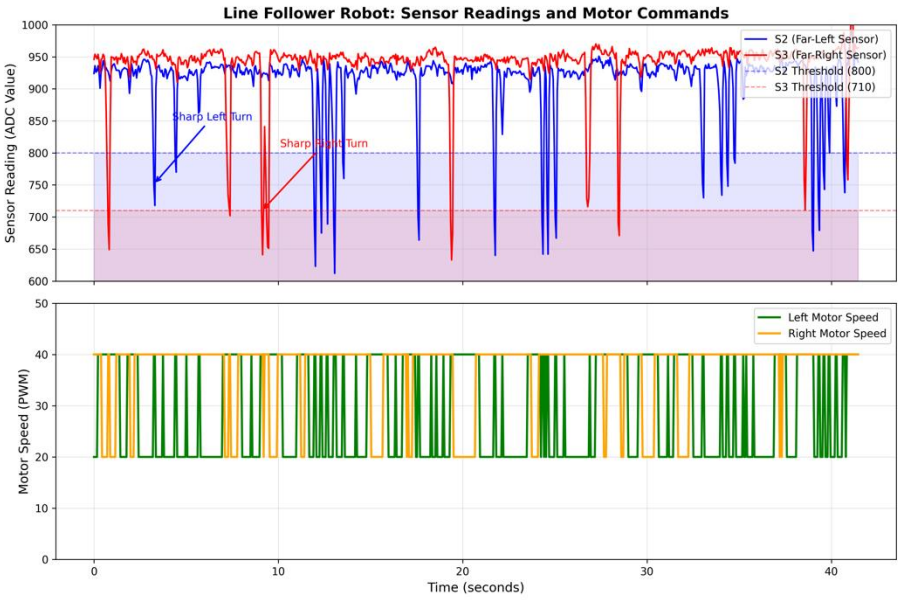


Figure 7: A plot superimposing sensor data and commanded motor speeds over a 45-second trial run. The top panel displays the analog readings from the two outermost IR sensors against their calibrated thresholds, while the bottom panel shows the corresponding PWM speed commands sent to the left and right motors.

To visualize the performance of our control algorithm, we logged sensor and motor data during a short trial run, as shown in Figure 7. The plot clearly demonstrates the cause-and-effect relationship between the robot's sensors and its motors. For example, at approximately 3 seconds, the far-left sensor (S2) detects the tape line, causing its value to drop below the 800 threshold. The controller immediately responds by reducing the left motor's speed to 20 while keeping the right motor at 40, executing a precise left turn.

Track Video

Below is a link to a video of our robot completing the track:

https://youtube.com/shorts/QMSRv6d-fFA

Reflection

Successes:

A major success in our project was establishing a clear vision for our final robot design early in the process. Having a target physical layout and mechanical structure allowed us to make focused, deliberate decisions from the start. This foresight proved invaluable when we encountered challenges with our initial sensor setup. Our design was intentionally modular and expandable, particularly our 3D-printed sensor housing.

When we realized we needed to upgrade from two to four sensors, the process was incredibly smooth. We simply connected the two new sensors to the Arduino, printed a slightly modified housing, and attached it to the chassis. This ability to iterate quickly and efficiently on our

physical design without a major rebuild was a key factor in our project's success. The serial implementation also helped us debug our code faster.

Additionally, the modularity of the chassis allowed for easy iteration of mechanical design in terms of the placement of the motors, the Arduino-motor shield, breadboard, sensors, and wheels. There were attempts to create a housing system that would have provided a “cleaner” aesthetic, which simply means that that system would have none of its internals on view. This system was facile to prototype and install on the chassis due to this same modularity.

Next Points:

Our initial plan to use only two sensors proved to be a big challenge. While this setup works for gentle curves, it has a critical flaw when navigating sharp turns. With only two sensors, the system can lose sight of the line entirely, leaving the robot "blind" with no information on which way to turn. During initial testing, this became a practical problem, as our robot consistently failed at the same sharp corner. Recognizing this limitation prompted our decision to a four-sensor setup, which provided more granular data and allowed our logic to handle more complex scenarios.

While the four-sensor array was a major improvement, further refinements could be made. One physical improvement would be to adjust the spacing of the two middle sensors to more closely match the width of the tape. This would make the robot's "on-center" detection more precise and reduce the small dead zone where it could drift slightly without triggering a correction. For future iterations, expanding to an even larger array of six or eight sensors could provide near-analog precision, allowing the controller to make smoother, more proportional adjustments rather than the discrete turns of our bang-bang approach, likely improving both speed and accuracy.

Additionally, the implementation of the housing would create a “cleaner” finish for the system, rendering the visual blight of the internals to be hidden by this cover. Finding manners to mount this housing and not impact the overall performance of the system would be a willing challenge to transform the system into a veritable end-product.

Appendix:

Arduino Code

```
#include <Adafruit_MotorShield.h>

Adafruit_MotorShield AFMS = Adafruit_MotorShield();
```

```
Adafruit_DCMotor *leftMotor = AFMS.getMotor(3);
Adafruit_DCMotor *rightMotor = AFMS.getMotor(4);

// sensor and controller definitions
#define S1_PIN A0 // middle left sensor
#define S2_PIN A1 // most left sensor
#define S3_PIN A2 // most right sensor
#define S4_PIN A3 // middle right sensor

// sensor calibration thresholds (low value = tape)
#define S1_THRESHOLD 800
#define S2_THRESHOLD 800
#define S3_THRESHOLD 710
#define S4_THRESHOLD 710

// global variables
int baseSpeed = 40;
// remembers the last turn direction for line recovery: -1=left, 0=straight,
1=right
int lastTurnDirection = 0;
// stores the current commanded speed for logging
int leftMotorSpeed = 0;
int rightMotorSpeed = 0;

void setup() {
  Serial.begin(9600);
  AFMS.begin(); // start the motor shield

  // print a header for the csv data
  Serial.println("Timestamp,S2_Value,S3_Value,Left_Speed,Right_Speed");
}

// motor control helper functions

// drives straight by setting both motors to the same speed
void go_straight(int speed) {
  leftMotorSpeed = speed;
  rightMotorSpeed = speed;
  leftMotor->setSpeed(leftMotorSpeed);
  rightMotor->setSpeed(rightMotorSpeed);
  leftMotor->run(FORWARD);
  rightMotor->run(FORWARD);
  lastTurnDirection = 0;
```

```
}

// performs a gentle left turn by slowing down the left motor
void turn_gentle_left(int speed) {
  leftMotorSpeed = speed * 0.5;
  rightMotorSpeed = speed;
  leftMotor->setSpeed(leftMotorSpeed);
  rightMotor->setSpeed(rightMotorSpeed);
  leftMotor->run(FORWARD);
  rightMotor->run(FORWARD);
  lastTurnDirection = -1;
}

// performs a gentle right turn by slowing down the right motor
void turn_gentle_right(int speed) {
  leftMotorSpeed = speed;
  rightMotorSpeed = speed * 0.5;
  leftMotor->setSpeed(leftMotorSpeed);
  rightMotor->setSpeed(rightMotorSpeed);
  leftMotor->run(FORWARD);
  rightMotor->run(FORWARD);
  lastTurnDirection = 1;
}

// performs a sharp left pivot turn by reversing the left motor
void turn_sharp_left(int speed) {
  leftMotorSpeed = speed;
  rightMotorSpeed = speed;
  leftMotor->setSpeed(leftMotorSpeed);
  rightMotor->setSpeed(rightMotorSpeed);
  leftMotor->run(BACKWARD);
  rightMotor->run(FORWARD);
  lastTurnDirection = -1;
}

// performs a sharp right pivot turn by reversing the right motor
void turn_sharp_right(int speed) {
  leftMotorSpeed = speed;
  rightMotorSpeed = speed;
  leftMotor->setSpeed(leftMotorSpeed);
  rightMotor->setSpeed(rightMotorSpeed);
  leftMotor->run(FORWARD);
  rightMotor->run(BACKWARD);
```

```
  lastTurnDirection = 1;
}

// main loop
void loop() {
  // serial tuning interface to change speed live
  if (Serial.available() > 0) {
    char command = Serial.read();
    if (command == 's') {
      int newSpeed = Serial.parseInt();
      if (newSpeed >= 0 && newSpeed <= 255) {
        baseSpeed = newSpeed;
        Serial.print("base speed set to: ");
        Serial.println(baseSpeed);
      }
    }
    while (Serial.available() > 0) { Serial.read(); }
  }

  // read raw analog values from all four sensors
  int s1_raw = analogRead(S1_PIN);
  int s2_raw = analogRead(S2_PIN);
  int s3_raw = analogRead(S3_PIN);
  int s4_raw = analogRead(S4_PIN);

  // convert raw values to simple boolean states (on tape or not)
  bool s1_on_tape = s1_raw < S1_THRESHOLD;
  bool s2_on_tape = s2_raw < S2_THRESHOLD;
  bool s3_on_tape = s3_raw < S3_THRESHOLD;
  bool s4_on_tape = s4_raw < S4_THRESHOLD;

  // main control logic to decide how to move
  // u-turn detection
  if (s3_on_tape && s4_on_tape) {
    turn_sharp_right(baseSpeed);
  }
  else if (s1_on_tape && s2_on_tape) {
    turn_sharp_left(baseSpeed);
  }
  // standard sharp turn
  else if (s3_on_tape) {
    turn_sharp_right(baseSpeed);
  }
```

```
  else if (s2_on_tape) {
    turn_sharp_left(baseSpeed);
  }
  // on center
  else if (s1_on_tape && s4_on_tape) {
    go_straight(baseSpeed);
  }
  // gentle correction
  else if (s4_on_tape) {
    turn_gentle_right(baseSpeed);
  }
  else if (s1_on_tape) {
    turn_gentle_left(baseSpeed);
  }
  // fallback: lost the line so it keeps its input until it finds the line again
  else {
    if (lastTurnDirection == -1) {
      turn_sharp_left(baseSpeed);
    }
    else if (lastTurnDirection == 1) {
      turn_sharp_right(baseSpeed);
    }
    else {
      go_straight(baseSpeed);
    }
  }

  // for data logging: prints data in csv format
  Serial.print(millis());
  Serial.print(",");
  Serial.print(s2_raw); // most left sensor
  Serial.print(",");
  Serial.print(s3_raw); // most right sensor
  Serial.print(",");
  Serial.print(leftMotorSpeed);
  Serial.print(",");
  Serial.println(rightMotorSpeed);

  delay(50);
}
```

Sensor Data Plot Code

```
import pandas as pd
import matplotlib.pyplot as plt

# script to generate plot from an external csv file

# set the name of the data file
csv_filename = 'wheel_data.csv'

# define the column headers
column_names = ["Timestamp", "S2_Value", "S3_Value", "Left_Speed", "Right_Speed"]

# set the sensor thresholds from the arduino code
S2_THRESHOLD = 800
S3_THRESHOLD = 710

# use a try block to handle potential file errors
try:
    # load the csv data from specific columns
    # usecols tells pandas to only read columns f through j
    # header equals none because the csv file does not have a header row
    df = pd.read_csv(csv_filename, header=None, usecols=[5, 6, 7, 8, 9],
names=column_names)

    # confirm that the file loaded successfully
    print(f"successfully loaded {len(df)} data points from '{csv_filename}'")

    # convert timestamp from milliseconds to seconds relative to the start
    df['Time_sec'] = (df['Timestamp'] - df['Timestamp'].iloc[0]) / 1000.0

    # extract each data series for plotting
    timestamps = df['Time_sec'].values
    s2_values = df['S2_Value'].values
    s3_values = df['S3_Value'].values
    left_speeds = df['Left_Speed'].values
    right_speeds = df['Right_Speed'].values

    # create a figure with two stacked subplots that share the same x axis
    fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(12, 8), sharex=True)

    # first subplot for sensor readings
    # plot the sensor data over time
```

```
    ax1.plot(timestamps, s2_values, label='s2 (far left sensor)', color='blue',
linewidth=1.5)
    ax1.plot(timestamps, s3_values, label='s3 (far right sensor)', color='red',
linewidth=1.5)

    # draw the threshold lines for visual reference
    ax1.axhline(y=S2_THRESHOLD, color='blue', linestyle='--', alpha=0.5,
linewidth=1,
                label=f's2 threshold ({S2_THRESHOLD})')
    ax1.axhline(y=S3_THRESHOLD, color='red', linestyle='--', alpha=0.5,
linewidth=1,
                label=f's3 threshold ({S3_THRESHOLD})')

    # shade the area where the tape is detected
    ax1.fill_between(timestamps, 600, S2_THRESHOLD, alpha=0.1, color='blue')
    ax1.fill_between(timestamps, 600, S3_THRESHOLD, alpha=0.1, color='red')

    # set the labels and title for the first plot
    ax1.set_ylabel('sensor reading (adc value)', fontsize=12)
    ax1.set_title('line follower robot: sensor readings and motor commands',
                  fontsize=14, fontweight='bold')
    ax1.legend(loc='upper right', fontsize=10)
    ax1.grid(True, alpha=0.3)
    ax1.set_ylim([600, 1000])

    # loop to find and annotate the first right turn
    for i in range(1, len(s3_values)):
        if s3_values[i-1] > 800 and s3_values[i] < S3_THRESHOLD:
            ax1.annotate('sharp right turn',
                         xy=(timestamps[i], s3_values[i]),
                         xytext=(timestamps[i]+1, s3_values[i]+100),
                         arrowprops=dict(arrowstyle='->', color='red', lw=1.5),
                         fontsize=10, color='red')
            break

    # loop to find and annotate the first left turn
    for i in range(1, len(s2_values)):
        if s2_values[i-1] > 850 and s2_values[i] < S2_THRESHOLD:
            ax1.annotate('sharp left turn',
                         xy=(timestamps[i], s2_values[i]),
                         xytext=(timestamps[i]+1, s2_values[i]+100),
                         arrowprops=dict(arrowstyle='->', color='blue', lw=1.5),
                         fontsize=10, color='blue')
```

```
            break

    # second subplot for motor speeds
    # plot the motor speed data over time
    ax2.plot(timestamps, left_speeds, label='left motor speed', color='green',
linewidth=2)
    ax2.plot(timestamps, right_speeds, label='right motor speed', color='orange',
linewidth=2)

    # set the labels for the second plot
    ax2.set_xlabel('time (seconds)', fontsize=12)
    ax2.set_ylabel('motor speed (pwm)', fontsize=12)
    ax2.legend(loc='upper right', fontsize=10)
    ax2.grid(True, alpha=0.3)
    ax2.set_ylim([0, 50])

    # adjust plot layout and save the figure
    plt.tight_layout()
    output_filename = 'line_follower_plot.png'
    plt.savefig(output_filename, dpi=300, bbox_inches='tight')
    plt.show()

# catch errors if the file is not found
except FileNotFoundError:
    print(f"error: could not find file '{csv_filename}'")
    print("please make sure the csv file is in the same directory as this
script.")
# catch any other errors during the process
except Exception as e:
    print(f"error reading or plotting data: {e}")
```