Mini Project 3 Report

Section 3–4

Devlin Ih, Kat Canavan, William Skelly

October 14, 2022

Executive summary

In this project we designed a line-following robot using basic hobbyist electronic components and a Python program connected over serial. The design consists of two an infrared (IR) distance sensors mounted to a custom laser-cut chassis. The robot has two powered wheels and a third trailing wheel in the back for stability.

Link to the video: https://youtu.be/ip5WLo_81KU

Mechanical Design

Our mechanical system, shown in Figure 1 consists of a lasercut acrylic chassis, a 3D printed sensor mount, a castoring rear wheel, and two main drive wheels powered by individual geared DC motors.

The lasercut chassis consists of a primary base plate, a breadboard mount, and motor mounts. The rear wheel assembly, Arduino, and sensor mount attach to the base plate via machine screws fastened through holes lasercut into the baseplate. In order to be able to mount the sensors, Arduino, and rear wheel in the ideal locations, and to provide for attachment of the breadboard mount, we designed a new base plate instead of the one given in the kit. This allowed greater fleixibilty of mechanical design for other components.

The breadboard mount consists of two vertical plates of lasercut acrylic that slot into the base plate with Mortis/Tennon joints, and are re-enforced with cyanoacrylate glue. The breadboard fits onto a step in the mount that acts like a shelf and constrains the breadboard in the for-aft axis and vertical axis. To constrain the breadboard in the left-right axis, we tied it down with solid-core copper wire, which gave us the ability for quick removal and adjustment.

We used the rear wheel assembly given with the kit because we felt it would be adequate for our platform. We also used the injection-molded mounting pieces for the motor because

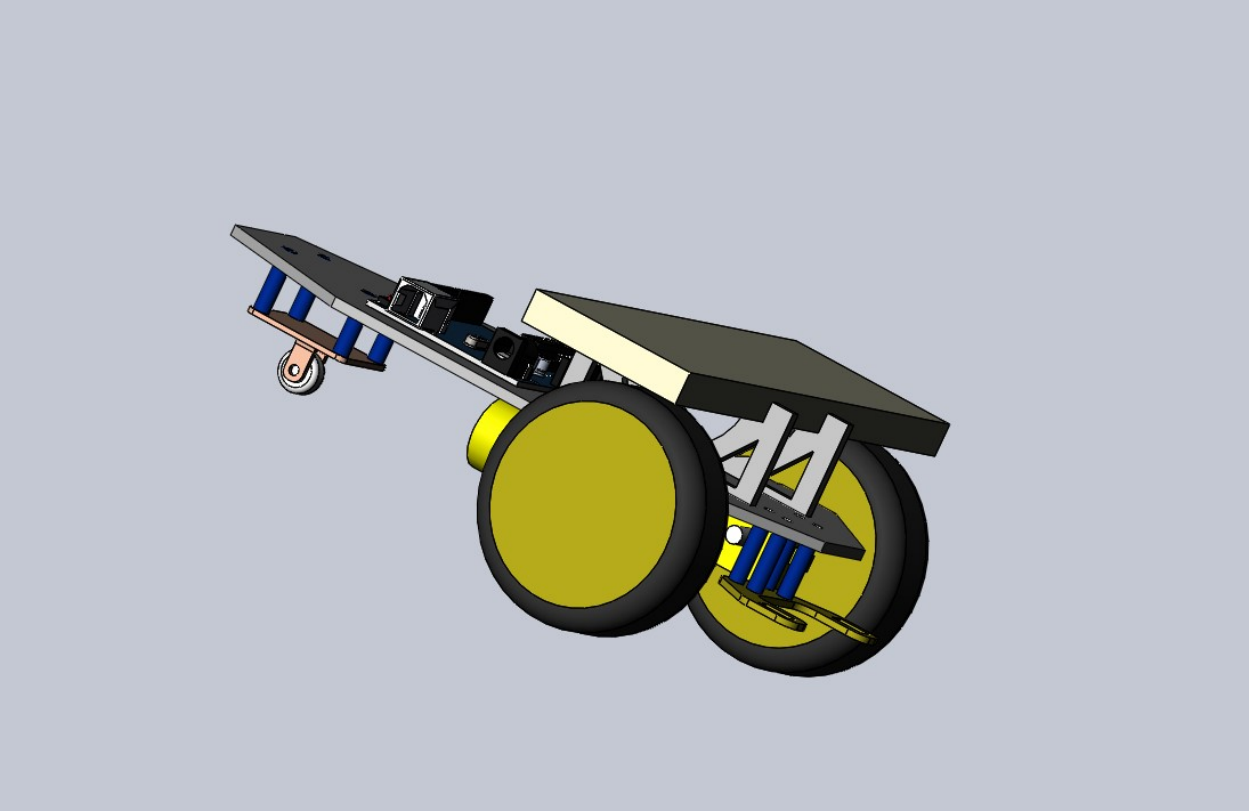

Figure 1: 3D render of custom baseplate assembled with motors, wheels, caster, and breadboard.

we felt they would be adequate, and accordingly we designed the new base plate to fit these parts.

We found that the wheels given in the kit had poor traction and this caused errors in navigation. To solve this problem, we wrapped rubber bands around the wheels. This greatly improved traction and eliminated wheel slippage as a source of navigational error.

The 3D printed sensor mount holds two IR sensors pointing at the ground spaced to be on either side of the tape. We designed the mount to hold the IR sensors with a friction fit so that we could quickly remove and replace the sensor to be able to re-wire our circuit if necessary. We designed the mount as a flat plate bolted to the bottom of the chassis base plate with four 4–40 bolts. Making the mount a flat plate meant that we could 3D print it in under 10 minutes and rapid-prototype multiple iterations. We intentionally chose to 3D print the mount instead of lasercut it so that we could have more control over the friction fit without having to account for kerf of the lasercutter or the slick nature of the acryilic. We also recognized that we might need to revise this part in several iterations to get the right sensor spacing, and 3D printing allowed us to do this without waiting for the shop schedule to align with our project timeline.

Our initial version of the sensor mount bolted directly to the chassis base plate, but we found that the sensors were too high, so we 3D printed 0.75" standoffs to move the sensors closer to the ground. If we were to do this again we'd redesign the sensor mount to be taller and sit closer to the floor.

Electronics and Calibration

This project involved the use of two 12 V DC motors connected to an Adafruit v2.3 Motor Shield as described by Figure 2. The IR sensors themselves were mounted close to the floor while the circuits were implemented on a breadboard. The 200 Ω resistor value on the LED side of the sensor was provided to us however the 30 kΩ resistor on the transistor side required several iterations. Initially we calculated a minimum required resistance of 50 Ω (5 V divided by the 0.1 A on the transistor data sheet) and added another 50 Ω for factor of safety. It quickly became apparent that we needed a much higher resistance value to increase the voltage drop across the resistor and get a greater range of readings from the voltage splitter. We tried recalculating the ideal resistance but eventually realized trial and error was the best method and we eventually found 30 kΩ to be perfect. In the future we may reflect more on the particular function of a resistor (such as in a voltage splitter) before calculating a value.

Before mounting the sensors and breadboard to the robot we conducted several stand-alone tests to get an idea of the range of the sensor readings on the electrical tape (the line) vs the reflective floor. We used this as a baseline when writing the code and in the final version we added a threshold slider (ranging from 0-1023, mirroring the analog range of the sensors) to the GUI so that the robot can be quickly adjusted for different hardware iterations and lighting conditions. Since the resistor is so high and the sensors are so close to the ground, the voltage readings from the sensor on the tape and floor are distinct enough that the threshold

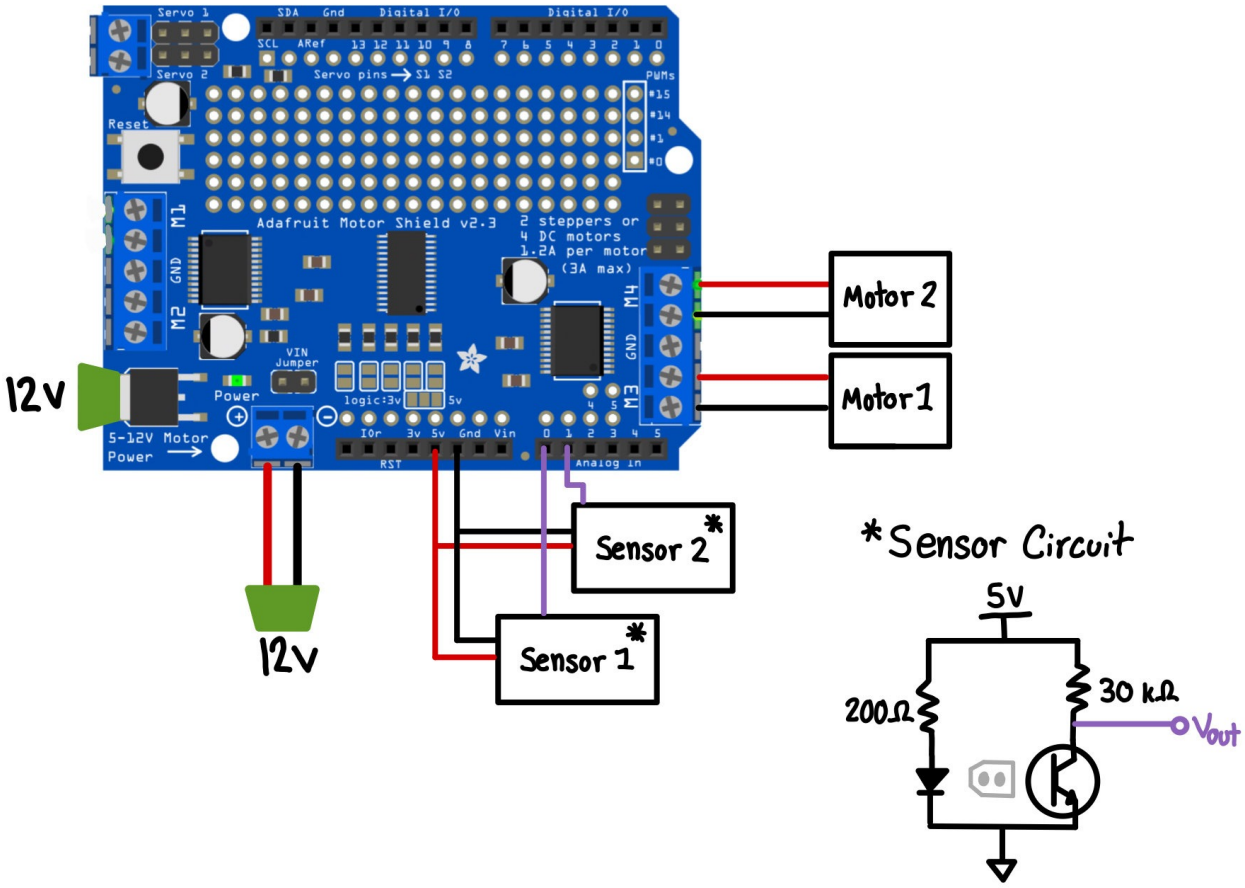

Figure 2: Circuit Diagram.

can be comfortably estimated by holding the robot over each surface and manually checking the readings.

Robot Control

This project took advantage of the Arduino's serial connection to implement all control handling with Python.

Control Architecture

Our robot uses a bang-bang control loop for simplicity. If the sensor reading is above a threshold, the sensor is positioned over tape. The robot has two modes: stopped and following. Stopped is self explanatory. The robot has 4 states in following mode. Figure 3 shows how the motor speeds relate to sensor data.

1. Stopped: Both sensors detect tape. We have traveled a circuit and reached the stop line.
2. Move forward: Both sensors detect floor tile.
3. Turn left: The left sensor detects tape. We need to turn left because the sensor is in the front of the robot.
4. Turn right: The right sensor detects tape. We need to turn right because the sensor is in the front of the robot.

Firmware

The firmware is limited to reading sensors and setting the speed/direction of the motors. All processing is done on the connected computer through a Python program.

Commands are sent from the Python program to the Arduino over serial. Every command sends a response back over serial. Listing 1 shows the code used to parse commands.

The firmware listens for 6 commands with the following syntax:

1. `setWheelLeft N`: Set speed of left wheel to `N` speed units in the positive direction. Send the value `N` back over serial when completed.
2. `-setWheelLeft N`: Set speed of left wheel to `N` speed units in the negative direction. Send the value `N` back over serial when completed.
3. `setWheelRight N`: Set speed of right wheel to `N` speed units in the positive direction. Send the value `N` back over serial when completed.
4. `-setWheelRight N`: Set speed of right wheel to `N` speed units in the negative direction. Send the value `N` back over serial when completed.
5. `readSensors` Take readings from both reflection sensors. Send the readings over serial in the form `(left, right)`.
6. `STOP` Completely stop the motors. Send the value `0` over serial when completed.

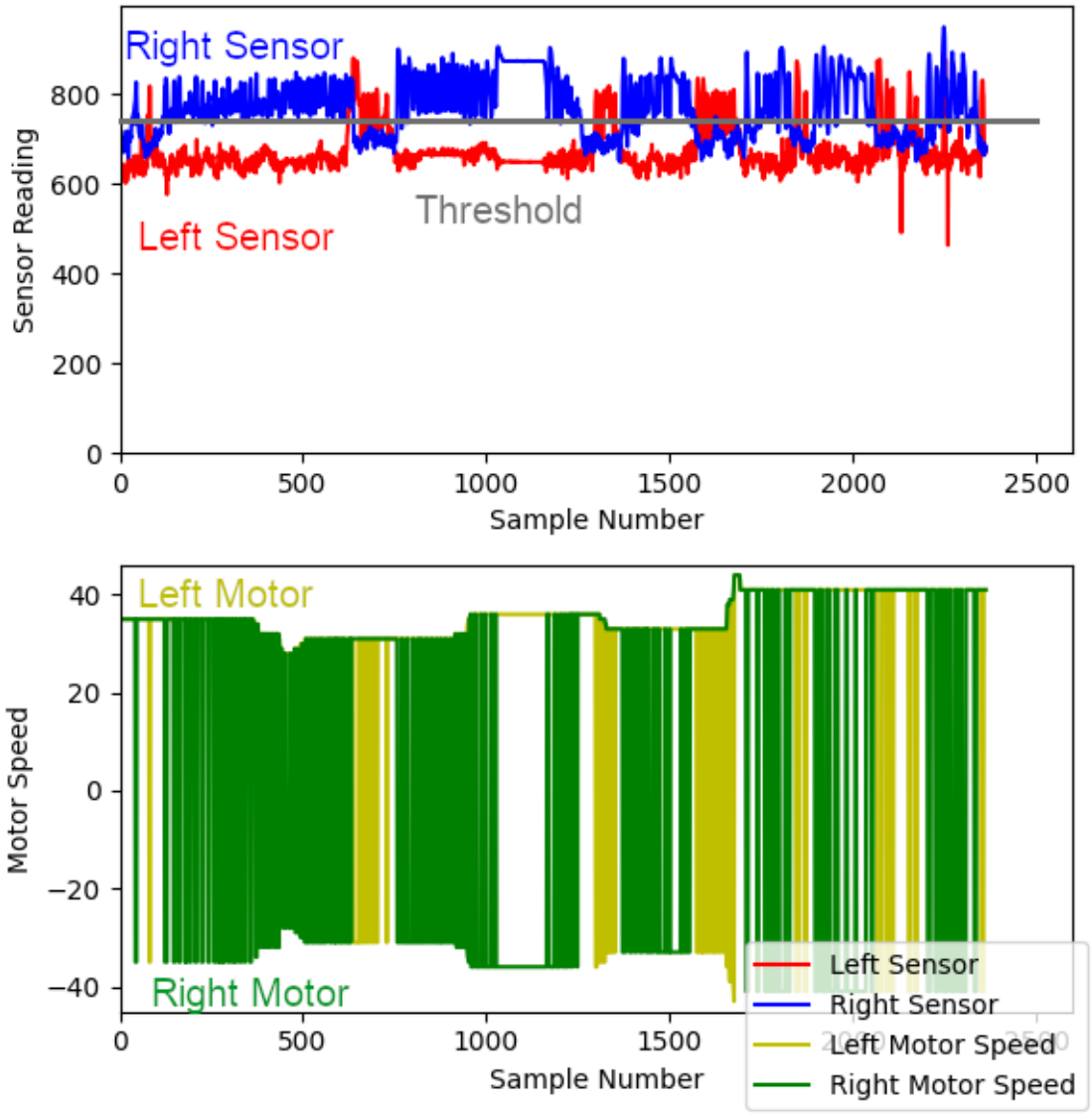

Figure 3: Sensor readings and motor speeds in the bang-bang control loop. Upper plot shows sensor readings while bottom plot shows motor speeds. This data was taken on a clockwise loop of the track. As such, it spends most of its time turning right. This is shown on the upper plot as the blue line spends most of its time above the gray line. On the lower plot, this is shown by the dark green line constantly bouncing around as it switches from rotating in the positive and negative directions. Robot speed was adjusted manually during this loop, shown on the bottom plot.

Listing 1 Arduino code to parse commands over serial and a sample of how that translates to wheel control.

```
void loop() {
  while (Serial.available() == 0) {
    // wait for data available in serial receive buffer
  }
  String command = Serial.readStringUntil('\n');  // read until timeout
  if (command.startsWith("setWheelLeft")) {
    // note: wheel will be in the form "setWheelDirection N" where N is speed
    // extract speed from command string
    uint8_t speed = command.substring(command.indexOf(" ")+1,
                                      command.length()).toInt();
    leftWheel->run(FORWARD);
    leftWheel->setSpeed(speed);
    Serial.println(speed);

  } else if ...
}
```

Software

All of the robot's "thinking" is implemented in Python. The Python software communicates with the Arduino through the PySerial library. Listing 2 shows the code used to send serial commands to the Arduino and read a response.

Listing 2 Python code to send a serial command to the Arduino and read a response. After a command is issued over serial, the program waits for a response. The response is returned as a Python string.

```
def send_command(ser: serial.Serial, command: str) -> str:
    message = bytes(command.strip() + "\n", "utf-8")
    ser.write(message)
    data = ser.readline()
    # data is a bytestring, return it as a number.
    return data.decode("utf-8").strip()
```

The bang-bang loop is implemented with the following steps:

1. Send `readSensors` command to Arduino.
2. Parse the returned data and figure out what state to set the robot (move forwards, turn left, turn right).
3. Send `setWheelLeft`/`setWheelRight` commands to Arduino to put the robot in the correct state.

Listing 3 shows the Python implementation for a single iteration of the loop.

In retrospect, implementing the control logic in the Arduino firmware would have resulted in a better robot. The current architecture requires sending a command from the computer

Listing 3 Code implementing a single iteration of the control loop. Pattern matching is used to determine the correct course of action. Lastly, the sensor data is returned as a tuple for plotting. For details on the functions called, see the Appendix.

```
def control_cycle(arduino: serial.Serial, speed: int, threshold: int) -> tuple:
    data = read_sensors(arduino)

    left_over_tape = data[0] > threshold
    right_over_tape = data[1] > threshold

    match (left_over_tape, right_over_tape):
        case (True, True):
            # Both sensors are over tape, we are at start line
            move_straight(arduino, speed)
        case (False, False):
            # Neither sensor is over tape
            move_straight(arduino, speed)
        case (True, False):
            # Left sensor is over tape, turn left as sensor is in front
            turn_left(arduino, speed)
        case (False, True):
            # Right sensor is over tape, turn right as sensor is in front
            turn_right(arduino, speed)
    return data
```

to the Arduino, waiting for a response, processing the response, and finally sending another command to the Arduino. This produces decent latency. As a result, we have to limit the speed we run the robot at to make sure the line will be detected.

Lastly, parameters such as robot speed and detection threshold can be adjusted in real time through a tkinter based GUI. In addition, sensor readings and motor speeds are plotted in real time with Matplotlib while the robot is in follow mode. Figure 4 shows a screenshot of the GUI. The control loop and the GUI run independently of each other on separate threads. Communication between the two is managed with a producer/consumer architecture. The GUI produces adjustments to a `Queue`, while the control loop consumes new updates on each cycle. The `Queue` is used to avoid race conditions. Listing 4 shows how the GUI produces commands, while Listing 5 shows the consumer interpreting them.

In retrospect, it would have been better to use a consistent style for the producer and consumer. The producer is implemented as a class while the consumer is implemented as a function.

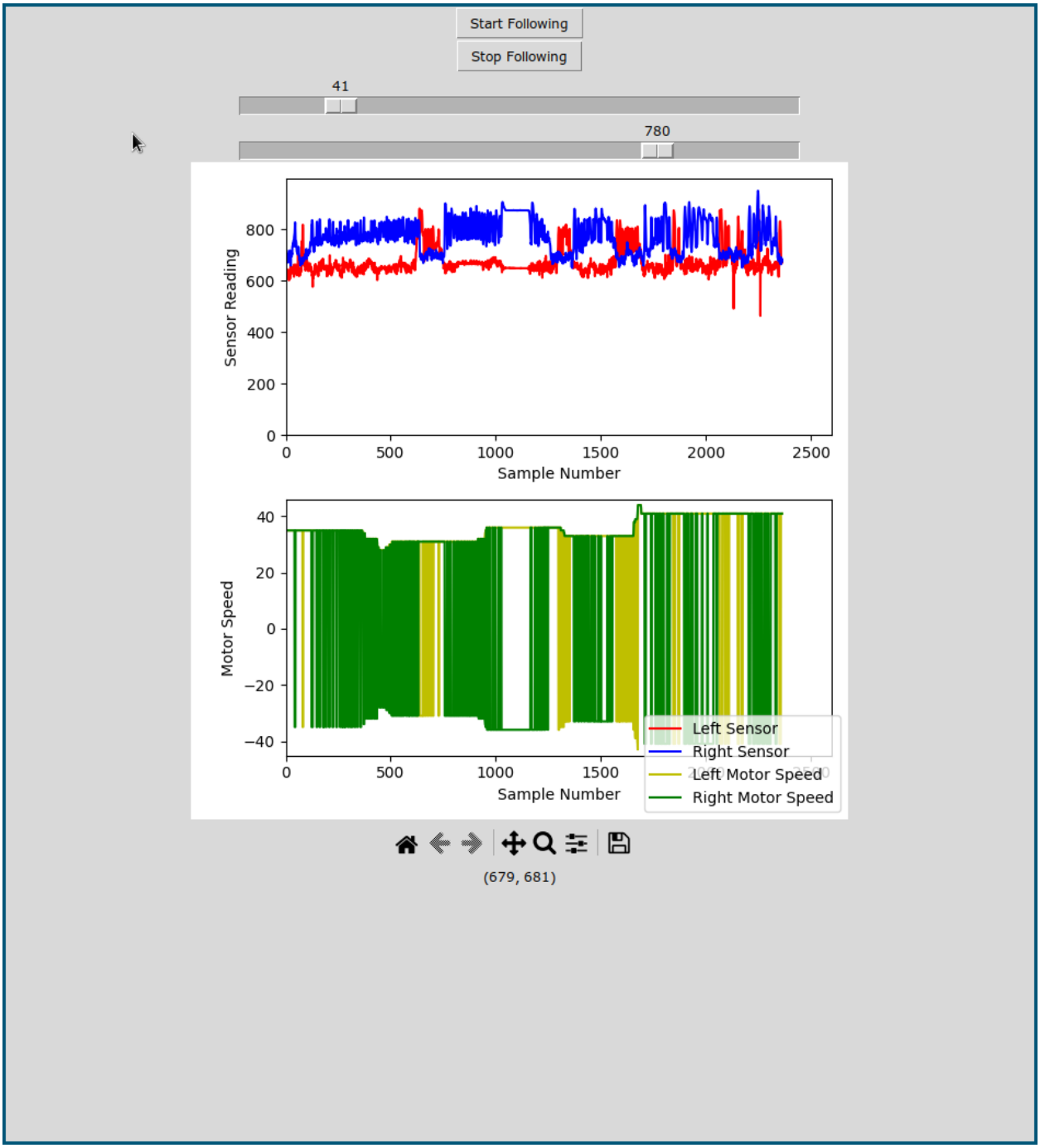

Figure 4: Robot Control GUI

Listing 4 Class implementing the command producing Robot GUI. Each GUI element has an associated event method that adds a command to the queue or completes some other action.

```
class RobotGui:
    def __init__(self, commands: queue.Queue, sensors: SensorData):
        self.commands = commands
        self.sensors = sensors

        self.root = tk.Tk()
        self.root.title("Line Follower GUI")

        start = tk.Button(self.root,
                          text="Start Following",
                          command=self.start)
        start.pack()
        # Additional GUI elements here

    def run(self):
        self.root.mainloop()

    def add_to_queue(self, command: str):
        self.commands.put(command)

    def start(self):
        self.add_to_queue("START")

    # Additional methods associated with GUI elements here.
```

Listing 5 Function implementing control loop while consuming parameter update commands generated from the GUI. Pattern matching is used to process commands and values.

```
def run_robot(arduino: serial.Serial,
              commands: queue.Queue,
              sensors: SensorData, ):
    moving = False
    speed = 30
    threshold = 900
    while True:
        if not commands.empty():
            command = commands.get()
            # Pattern matching, we require Python 3.10
            match command.split():
                case ["START"]:
                    moving = True
                case ["STOP"]:
                    moving = False
                case ["DONE"]:
                    stop_robot(arduino)
                    break
                case ["SPEED", val]:
                    speed = int(val)
                case ["THRESHOLD", val]:
                    threshold = int(val)
        if moving:
            data = control_cycle(arduino, speed, threshold)
            sensors.update(data)
        else:
            stop_robot(arduino)
```

Appendix

Source code can also be found in the project's Github Repository.

Listing 1 Complete Firmware Source for `LineRobotFirmware.ino`

```
// Firmware for a line following robot
// Olin College of Engineering PIE Miniproject 3

// Adafruit Motorsheild v2
#include <Adafruit_MotorShield.h>

// create motor sheild object
Adafruit_MotorShield AFMS = Adafruit_MotorShield();
// Initialize global variables for DC motors
Adafruit_DCMotor *leftWheel = AFMS.getMotor(3);   // left wheel port 3
Adafruit_DCMotor *rightWheel = AFMS.getMotor(4);  // right wheel port 4

// assign sensor pins
const int RIGHT_SENSE = 0;  // analog pin 0
const int LEFT_SENSE = 1;   // analog pin 1

// function prototypes for organization
void setWheelLeft();
void setWheelRight();
void readSensors();
void stop();

void setup() {
  // initialize serial communication
  Serial.begin(115200);
  // Serial.println("START");
  AFMS.begin();

  // set initial motor speeds to 0
  rightWheel->setSpeed(0);
  leftWheel->setSpeed(0);

  // motors always run forwards
  leftWheel->run(RELEASE);
  rightWheel->run(RELEASE);
}

void loop() {
  while (Serial.available() == 0) {
    // wait for data available in serial receive buffer
  }
  String command = Serial.readStringUntil('\n');  // read until timeout
  if (command.startsWith("setWheelLeft")) {
    // note: wheel will be in the form "setWheelDirection N" where N is speed
    // extract speed from command string
    uint8_t speed = command.substring(command.indexOf(" ")+1, command.length()).toInt();
    leftWheel->run(FORWARD);
    leftWheel->setSpeed(speed);
    Serial.println(speed);
```

Listing 2 Complete listing for `arduino.py`

```
"""
Connect to an Arduino running PIE MP3 Firmware.
"""

import ast
import serial
import serial.tools.list_ports as list_ports

BAUDRATE = 115200
ARDUINO_TIMEOUT = 3  # seconds

# List of Arduino IDs provided by Brad
# https://github.com/bminch/PIE/blob/main/Serial_cmd.py
ARDUINO_IDS = ((0x2341, 0x0043), (0x2341, 0x0001),
               (0x2A03, 0x0043), (0x2341, 0x0243),
               (0x0403, 0x6001), (0x1A86, 0x7523))


def guess_port() -> str:
    """
    Try to find an Arduino connected to the computer.

    Returns:
        A string representing the port of the first Arduino found. Returns the
        empty string if no Arduino is found.
    """
    devices = list_ports.comports()
    for device in devices:
        if (device.vid, device.pid) in ARDUINO_IDS:
            return device.device
    return ""


def send_command(ser: serial.Serial, command: str) -> str:
    """
    Send a command to an Arduino connected over serial.

    Args:
        ser: A serial connection to an Arduiono.
        command: A string representing a command to send to the Arduino.

    Returns:
        A string representing the received message from the Arduino.
    """
    message = bytes(command.strip() + "\n", "utf-8")
    ser.write(message)
    data = ser.readline()
    # data is a bytestring, return it as a number.
    return data.decode("utf-8").strip()
```

Listing 3 Complete source listing for `control_loop.py`.

```
"""
Control loop for line following robot.
"""

import serial
import time

from arduino import (guess_port,
                     send_command,
                     parse_tuple,
                     BAUDRATE,
                     ARDUINO_TIMEOUT,)

DEFAULT_THRESHOLD = 780
DEFAULT_SPEED = 35


def move_straight(arduino: serial.Serial, speed: int) -> tuple[int, int]:
    """
    Move forward at a given speed.
    """
    send_command(arduino, f"setWheelLeft {speed}")
    send_command(arduino, f"setWheelRight {speed}")
    return (speed, speed)


def turn_right(arduino: serial.Serial, speed: int) -> tuple[int, int]:
    """
    Turn right at given speed.
    """
    send_command(arduino, f"setWheelLeft {speed}")
    send_command(arduino, f"-setWheelRight {speed}")
    return (speed, -speed)


def turn_left(arduino: serial.Serial, speed: int) -> tuple[int, int]:
    """
    Turn left at given speed.
    """
    send_command(arduino, f"-setWheelLeft {speed}")
    send_command(arduino, f"setWheelRight {speed}")
    return (-speed, speed)


def read_sensors(arduino: serial.Serial) -> tuple[int, int]:
    """
    Read sensors.
    """
    return parse_tuple(send_command(arduino, "readSensors"))
```

Listing 4 Complete source listing for `robot_gui.py`.

```
"""
Tkinter based GUI for control loop
"""

# Standard library stuff
import threading
from threading import Thread
import queue
import time
import tkinter as tk

# PyPI
import serial

# Plotting
from matplotlib.backends.backend_tkagg import (FigureCanvasTkAgg,
                                               NavigationToolbar2Tk, )
from matplotlib.backend_bases import key_press_handler
from matplotlib.figure import Figure

# Custom Modules
from arduino import (guess_port,
                     BAUDRATE,
                     ARDUINO_TIMEOUT, )
from control_loop import (control_cycle,
                          stop_robot,
                          DEFAULT_SPEED,
                          DEFAULT_THRESHOLD, )


class SensorData:
    """
    Class to store the sensor data between threads.
    """
    # Don't care about locking or race conditions because this isn't for anything
    # computationally important, just for visualizing the current sensor readings.

    def __init__(self, data: tuple[int, int]):
        self.data = data
        self.history = [(0, 0)]
        self.speeds = [(0, 0)]

    def update(self, data, speeds):
        self.history.append(data)
        self.speeds.append(speeds)
        self.data = data


class RobotGui:
```