Lab 6: Wall Following

Group 5

Juliette Mitrovich, Sheila Moroney, and Sujani Patel

Lab Summary

The main purpose of this lab was to interface motors and sensing mechanisms with the Arduino to achieve wall following. We found that using an IR sensor placed at a specific angle to the right wall and an ultrasonic sensor facing straight ahead was the best method to achieve wall following. The angle placement of the IR sensor was the most integral part because we needed to figure out the correct angle and distance to place our IR sensor at so our robot could follow the wall, despite the types of corners it came to. Once we found the correct angle, we learned how to create a proportional derivative (PD) controller and used the data from the IR sensor. Through trial and error, we tuned the proportional gain (Kp) and derivative gain (Kd) to achieve straight wall following. By adding a few more conditions, we successfully created a wall following robot.

For this lab, instead of using the smart video kit to follow the wall, we chose to use the robot that we used for the final project. Please see Figure 1 below for the CAD model of the robot used. Please note, for wall following, we only used the base level. We did not incorporate the other levels or ball collection mechanism as it was not necessary for the lab.

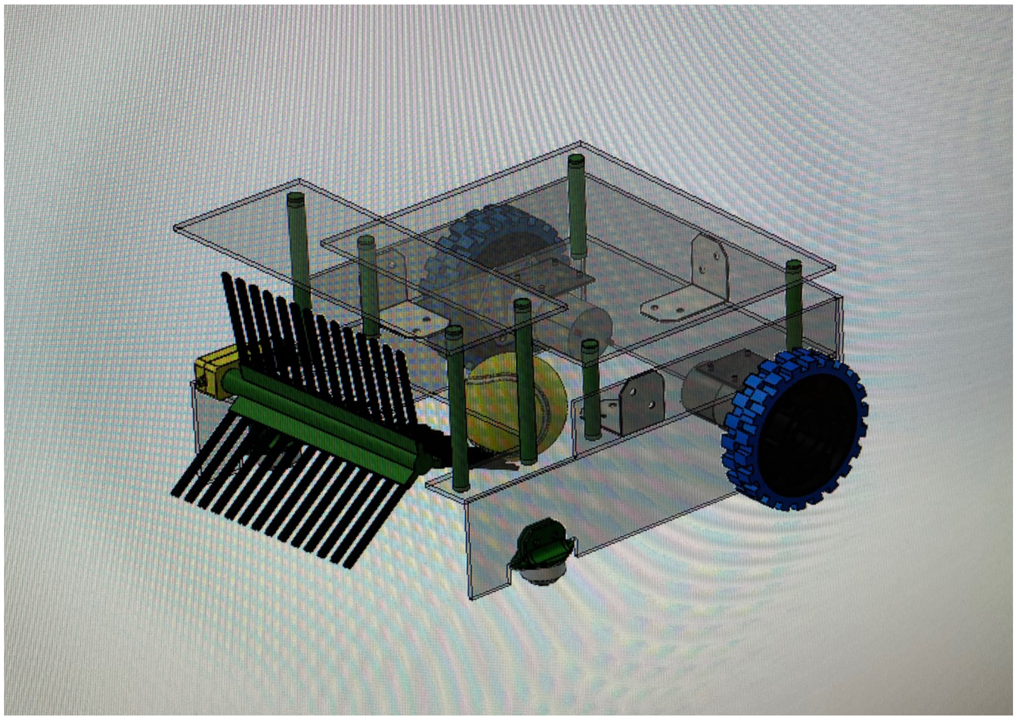
Figure 1. Custom built robot used for testing the wall following code

Speed Control and Steering

To drive our robot, we used two 12-volt, 300 RPM DC motors purchased from Amazon. Because these are simple DC motors, we needed a way to interface them with the Arduino. To do this, we used an L298N, a dual H-Bridge motor-driver which allows for speed and direction control of two DC motors at the same time. On the module, there are two screw terminal blocks for two motors, A and B, another screw terminal to power the H-Bridge/motors, and six input pins to control the speed and direction. Instead of powering the H-Bridge from the Arduino, we chose to power it with a separate power supply.

We started off by understanding how to control the direction of the motors. There are four pins to do this. IN1 and IN2 control motor A and IN3 and IN4 control motor B. For our robot, motor A is the left side motor and motor B is the right-side motor. Using these pins controls the switches on the H-Bridge itself. So, when IN1 is HIGH and IN2 is LOW, the motor moves in one direction and when IN1 is LOW and IN2 is HIGH, the motor moves in the opposite direction. The same applies for IN3 and IN4 for motor B. What direction the motor moves when one pin is high, and the other is low depends on how you connect the power and ground of the motor to the motor screw block terminals. In our case, we plugged in the motor, connected the pins to the Arduino, and tested each case to figure out what made the motor move forwards and backwards. Once we figured out the left motor, we did the same with the right motor. We also made sure to connect the motors in a way so both motors would move forward when the odd pins were HIGH (1 and 3) and move backwards when the even pins were HIGH (2 and 4). Now that we understood how to change the direction of the motors, we moved on to learning how to control the speed.

Controlling the speed of the motors was done with the remaining two input pins, enable A (ENA) and enable B (ENB). By connecting these pins to a pulse width modulation (PWM) pin on the Arduino, we were able to control the speed of the motors. Pulse width modulation values range from 0 to 255, with 255 being full power and 0 being no power. Even though 0 is theoretically no power, the range of what values can move the motor is much smaller. We found that the lowest value that was able to move the wheels/robot was around 80. This was an important discovery because it allowed us to better understand what speeds would make the robot move when developing the wall following code discussed in the next section.

Wall Following

To develop the wall following code, we started off by getting the robot to follow a straight wall. We chose to use one IR sensor placed at an angle rather than using two IR sensors because we did not want to worry about fusing the two sensor signals. The controller we chose to create was a proportional derivative (PD) controller. With a PD controller, you have four variables you must know to calculate the gain. In this case, the gain represents how much you should increase/decrease the motor speeds. The four variables are P, D, Kp, and Kd. P is the proportional term, or the calculated error. For wall following, the error is represented by the difference between how close the robot should be to the wall and how close it is. D is the derivative term, or the calculated error minus the previously calculated error. Kp and Kd are the proportional and derivative gain values respectively. These are variables whose values we chose to achieve the desired performance of the robot. We chose to find said values through trial and error. After many trials, we found the best values to be Kp = 6, and Kd = 110. With these values, the robot followed a straight wall with ease.

Now that the robot was successfully following a straight wall, we moved to trying to get it to make a U-turn, or an outer corner turn. Please see Figure 2 below for a visual representation of a U-turn.

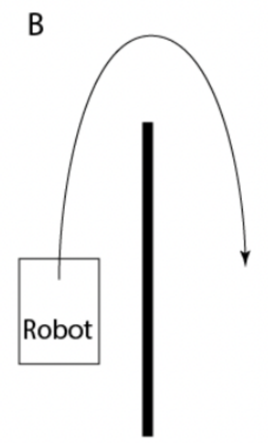

Figure 2. U-turn at the end of a wall

To start off, we decided to just let the robot try and make the outer turn without making any changes to the code. On the first try, the robot took off straight ahead when it no longer saw the wall. After some investigating, we discovered two things. One, when the IR sensor saw no wall, it read an infinitely large value making the gains large. Two, because the gain was large, it was causing the new calculated motor speeds from the gains to go below 0 and above 255. This was problematic because the motors will stall if they are given a value outside of the allowed range. The fix this we limited the maximum value the IR sensor could see to 80cm, we limited the maximum speed of the motors to 255, and we calculated the absolute value of the motor speeds. Once these changes were made, we tested the robot again and it was able to turn the corner. However, the new problem we encountered was the right wheel would get stuck on the corner. To remedy this, we decided to move the IR sensor from the front of the robot to the back. Because the wheels of our robot were at the back, having the sensor at the front of the robot would cause it to detect that there was no wall before the robot had cleared the wall. By moving the sensor back, it would detect that there is no wall when the robot is cleared to turn. Once we made this change, the robot successfully made the outer turn.

The final step of the wall following was to tackle the inner corner turn. Please see Figure 3 below for a visual representation of an inner corner turn.

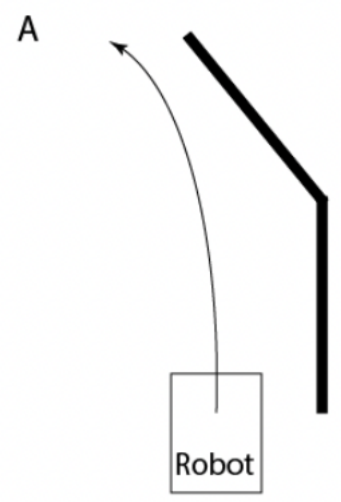

Figure 3. Turn at a corner

The same way we first tested the outer corner, we decided to let the robot attempt the inner corner turn without changing the code to see what would happen. On the first run, the robot started to make the turn, but did not do so fast enough and crashed into the wall. To fix this, we decided to add an ultrasonic sensor to the front of the robot to detect how close it was to a front facing wall. With this sensor value, we could then implement a condition saying, if the ultrasonic sensor detected a forward distance less than 15cm, make a zero-point left turn, where the right wheel moves forward and the left wheel moves backwards. To accomplish the left turn, we created a function with the inputs being the left and right motor speed. Once we implemented this condition and tested the robot again, it made the inner corner turn flawlessly. Now that we had the three possible wall conditions the robot could encounter, we created a track with all of them and let the robot run, and it tracked the wall successfully. For the final code, please see Figure A-1 in Appendix A.

MATLAB Predictive Code

To develop the MATLAB code able to predict the average time the robot can navigate through a maze, we started off by creating a 5x5 maze that generates obstacles randomly. This was done easily with two for loops. Now that we had the maze figured out, we had to figure out a way to have the "robot" navigate through it. Because the robot would start in the known, same spot each time, and the maze is essentially a 5x5 array, our initial thought was to create a program that systematically checked each adjacent grid position to see if there was an obstacle there, always starting to the right direction to simulate right wall following. After attempting this code and collaborating with other students in the class, we came up with a clever way to do this without hard coding each position that needed to be checked. By using sine and cosine, we were able to create the location we needed to check by moving around a unit circle. First, we checked the square to the right. If there was an obstacle there, it would "rotate" by 90^o and check the next square. The robot would do this until it found a square to move to and continue the process until it found the exit. Because we were creating random mazes, there was the possibility that the maze created was not solvable. In this case, if the robot found itself back at the starting location, a new maze would generate, and it would start the whole process over again. As stated in the lab handout, there are 2^23 possible mazes that could be created, and we needed to find that average time it would take for the robot to complete every solvable maze. To do this, every time the robot reached the exit, we had a running sum of the number of mazes that were completed and the amount of time it took for each maze. Once the robot completed 2^{23} number of mazes, we divided the total time for all the completed mazes by the number of mazes completed. In the end, we found that the average completion time for our robot was 12.02 seconds. For the final code used to solve the mazes, please see Figure A-2 below in Appendix A.

Appendix A
Source code.
Figure A-1. Final wall following code

```
/*
  ME 545 Lab 6 - Wall Following Code
  Group 5 - Juliette Mitrovich, Sheila Moroney, Sujani Patel
  This code allows our custom-built robot to successfully follow a straight wall, outer corners
  and inner corners.
*/
// Libraries
#include <math.h>

///// ULTRASONIC SENSOR /////
// Ultrasoinic sensor pins
const int echoPin_fwd = 32;
const int trigPin_fwd = 33;

// Ultrasonic sensor calculation variables
long duration_fwd;
float distance_fwd;

// Ultrasonic range condition
const float tooClose_fwd = 15; // how close the robot can get to a front facing wall

// FUNCTION //
float senseUltrasonic_frontWall() {
 // calculate the distance in front of the robot (cm)
 digitalWrite(trigPin_fwd, LOW);
 delayMicroseconds(2);
 digitalWrite(trigPin_fwd, HIGH);
 delayMicroseconds(10);
 digitalWrite(trigPin_fwd, LOW);
 duration_fwd = pulseIn(echoPin_fwd, HIGH);
 distance_fwd = duration_fwd * 0.034 / 2; // convert duration to distance
 // limit how far the front sensor can see because we only care about the close values for turning
 if (distance_fwd > 25) {
  distance_fwd = 25;
 }
 return distance_fwd;
}

///// IR SENSOR /////
// IR sensor 1 variables
float IRsensor = A0; // set IRsensor as analog pin 0
float sensorValue; // variable to store the output of IR sensor
```

```
// digital filter coefficients, (fc = 27.77Hz, fs = 125Hz )
const float a1 = -0.0875;
const float b0 = 0.4563;
const float b1 = 0.4563;

// digital filter variables
volatile float Vold = 0; // initialize the first input
volatile float yold = 0; // initialize the first output
volatile float digitalFilter; // initialize variable to store first order digital filter values
volatile float distance_right; // variable to store the distance conversion of the IR sensor

// FUNCTION //
float senseIR() {
  // Collect IR sensor data (through and analog filter
  sensorValue = analogRead(IRsensor);
  // digitially filter IR sensor data
  digitalFilter = (b0 * sensorValue) + (b1 * Vold) - (a1 * yold);
  yold = digitalFilter;
  Vold = sensorValue;
  // Convert filtered IR sensor data to distance
  distance_right = (16569.0 / (digitalFilter + 25)) - 11;
  // limit the max range the sensor can see to avoid large gains
  if (60 < distance_right) {
    distance_right = 60;
  }
  return distance_right;
}

///// PROPORTIONAL DERIVATIVE (PD) CONTROL /////
const float setPoint_side = 55; // always want to be ~20cm from the side wall
// number is > 20 because the sensor is on the back left side and this is a right wall following
robot
// allows you to stay within the correct operating range of the IR sensor (18cm - 150cm)

volatile float P; // proportional term
volatile float D; // derivative term
float Kp = 6.0; // proportional gain
float Kd = 110.0; // derivative gain

volatile float error_side; // variable to store calculated error
volatile float lastError_side; // variable to store the previous error for the derivative term

volatile float motorSpd_gain; // calculated gain from PID control

// FUNCTION //
float PD_control() {
  // find the error and previous error and calculate the necessary gain for the motor speeds
  error_side = setPoint_side - distance_right; // error calculation
  P = error_side;
  D = error_side - lastError_side;
  lastError_side = error_side;
  motorSpd_gain = P * Kp + D * Kd;
  return motorSpd_gain;
}

///// H-BRIDGE AND MOTORS /////
const float avgSpd = 105.0; // set speed motors should move at when going straight
volatile float rightSpd;
volatile float leftSpd;

// Motor pins
const int leftMotor1 = 5; // OUT 3 pin
const int leftMotor2 = 4; // OUT 4 pin

const int rightMotor1 = 3; // OUT 1 pin
const int rightMotor2 = 2; // OUT 2 pin

const int ENB_left = 10; // PWN pin to control left motor speed
const int ENA_right = 9; // PWM pin to control right motor speed

/*
    Pin convention
    Motor A: direction controlled by pins IN1 and IN2
    Motor B: direction controlled by pins IN3 and IN4
    if IN1 is HIGH and IN2 is low (or IN3 and IN4), motor will move one direction
    if you switch it the motor will move in the other direction DEPENDING ON HOW THE
POWER AND GROUND
    WIRES OF THE MOTOR ARE PLUGGED IN TO THE H-BRIDGE
*/

// FUNCTION //
void goFwd(float leftSpd, float rightSpd) {
  // set pins and speed to drive the robot forward (left fwd, right fwd)
  // convention for
  digitalWrite(leftMotor1, LOW);
  digitalWrite(leftMotor2, HIGH);
  digitalWrite(rightMotor1, LOW);
  digitalWrite(rightMotor2, HIGH);
  analogWrite(ENB_left, leftSpd);
  analogWrite(ENA_right, rightSpd);
}
```

```
void turnLeft(float leftSpd, float rightSpd) {
  // set pins and speed to allow robot to do a zero-point left turn (left fwd, right bwd)
  digitalWrite(leftMotor1, HIGH);
  digitalWrite(leftMotor2, LOW);
  digitalWrite(rightMotor1, LOW);
  digitalWrite(rightMotor2, HIGH);
  analogWrite(ENB_left, leftSpd);
  analogWrite(ENA_right, rightSpd);
}

void setup() {
  Serial.begin(115200); // begin serial monitor for debugging

  // set ultrasonic sensor pin modes
  pinMode(echoPin_fwd, INPUT);
  pinMode(trigPin_fwd, OUTPUT);

  // set motor pin modes
  pinMode(rightMotor1, OUTPUT);
  pinMode(rightMotor2, OUTPUT);
  pinMode(leftMotor1, OUTPUT);
  pinMode(leftMotor2, OUTPUT);

  pinMode(IRsensor, INPUT);
}

void loop() {
  // call the neccessary functions for the robot to complete wall following
  senseIR();
  senseUltrasonic_frontWall();
  PD_control();

  // set the speed of the wheels
  // abs() used because the H-bridge does not allow negative speed values
  rightSpd = abs(avgSpd + motorSpd_gain);
  leftSpd = abs(avgSpd - motorSpd_gain);

  // set the max speed of each motor to 255 because PWM range is 0-255
  if (rightSpd > 255.0) {
    rightSpd = 255.0;
  }
  if (leftSpd > 255.0) {
    leftSpd = 255.0;
  }

  // wall following conditions
  if (distance_fwd < tooClose_fwd) {
    // when you come to an inner corner, do a zero point left turn at a set speed
    turnLeft(150, 150);
  }
  if (distance_fwd > tooClose_fwd) {
    // drive forward with these gain values
    Kp = 6.0;
    Kd = 110.0;
    goFwd(leftSpd, rightSpd);
  }

  ///// DEBUGGING /////
  //  Serial.print("Distance R: ");
  //  Serial.print(distance_right);
  //  Serial.print(", ");
  //  Serial.print("Distance fwd: ");
  //  Serial.print(distance_fwd);
  //  Serial.print(", ");
  //  //  Serial.print("Gain: ");
  //  //  Serial.print(motorSpd_gain);
  //  //  Serial.print(", ");
  //  Serial.print("Motor Speeds: ");
  //  Serial.print(leftSpd);
  //  Serial.print(", ");
  //  Serial.print(rightSpd);
  //  Serial.println();
}
```

Figure A-2. Final source code for task 4 maze solving

```
%% ME 545 Lab 6
% Group 5: Juliette Mitrovich, Sheila Moroney, Sujani Patel
% Maze solving
clc
clear
close all
%%
maze = zeros(7);
for i = 2:(size(maze)-1)
    for j = 2:(size(maze)-1)
        maze(i,j) = round(rand);
    end
end
maze(2,6:7) = 0.5;
maze(6,2) = 0.5;
% imshow(maze,'InitialMagnification',20000);

% try to solve the maze
direction = 0; % way the robot should move (0,90,180,270)
robot_spd = 1; % how fast the robot moves
compass = [-sind(direction), cosd(direction)]; % convert direction to north,
south, east, west
location = [6,2];
laps = 1;
laps_prev = laps;
moved = 0;
solve = 0;
sum_mazes = 0;
sum_time = 0;
while solve == 0
    direction = direction - 90;
    compass = [-sind(direction), cosd(direction)]; % look right
    check = location + compass; % location of square you're looking at
    % if there's no empty square, move 90º and check again
    if maze(check(1),check(2)) == 0
        direction = direction + 90;
        compass = [-sind(direction), cosd(direction)];
        check = location + compass;
        if maze(check(1),check(2)) == 0
            direction = direction + 90;
            compass = [-sind(direction), cosd(direction)];
            check = location + compass;

            if maze(check(1),check(2)) == 0
                direction = direction + 90;
                compass = [-sind(direction), cosd(direction)];
                check = location + compass;
            end
        end
    end
    % if there's an empty square move to it
    if maze(check(1),check(2)) > 0
        location = check;
        if laps_prev ~= laps
```

```
            moved = 0;
            laps_prev = laps;
        end
        moved = moved + 1;
        maze(location(1),location(2)) = 0.6;

        %         figure(laps)
        %         imshow(maze,'InitialMagnification',20000)
        maze(location(1),location(2)) = 1;
    end
    % if you arrive at the start again, generate a new maze
    if location == [6,2]
        maze = zeros(7);
        for i = 2:(size(maze)-1)
            for j = 2:(size(maze)-1)
                maze(i,j) = round(rand);
            end
        end
        maze(2,6:7) = 0.5;
        maze(6,2) = 0.5;
        laps = laps + 1;
        %         close all
        %         figure(laps)
        %         imshow(maze,'InitialMagnification',20000);
        %         fprintf('REDOING MAP\n')
    end
    %     pause(0.25);
    % If you complete the maze, keep generating mazes until you've generated
    % every possible one
    if location == [2,7]
        maze = zeros(7);
        for i = 2:(size(maze)-1)
            for j = 2:(size(maze)-1)
                maze(i,j) = round(rand);
            end
        end
        maze(2,6:7) = 0.5;
        maze(6,2) = 0.5;
        location = [6,2];
        laps = laps + 1;
        %         close all
        %         figure(laps)
        %         imshow(maze,'InitialMagnification',20000);
        %         fprintf('REDOING MAP\n')
        sum_mazes = sum_mazes + 1;
        sum_time = sum_time + moved*robot_spd;
        %         fprintf('REDOING MAP\n')
        %         fprintf('\n \n \nGood Job! Your robot solved it in %f
seconds \n',moved*robot_spd)
    end

    if laps == 2^23
        solve = 1;
        fprintf('\n \n \nGood Job! Your average time was %0.2f seconds
\n',sum_time/sum_mazes)
```

```
        end
    end
```