Memo

To: Xiaoli Zhang and Mollie Brombaugh

From: Ben Pacheco and Tessa Haws

Team #: N-424

Date: November 10, 2024

Re: Lab 4: Solving the Maze

Problem Statement

As a group, our task in this lab is to collaboratively develop and implement a strategy for a robot to navigate and solve a maze autonomously. The robot must use distance sensors to detect walls and obstacles, systematically explore the entire maze from start to finish, and log its movements. With this recorded data, we will collectively determine and eliminate unnecessary steps, allowing the robot to retrace the maze in the most direct route possible on a second attempt. This lab requires us to apply our skills in sensor calibration, interfacing with infrared and ultrasonic sensors, and algorithm development. Together, we aim to deepen our understanding of robotic navigation, modular programming, and the iterative process of designing practical solutions for real-world challenges.

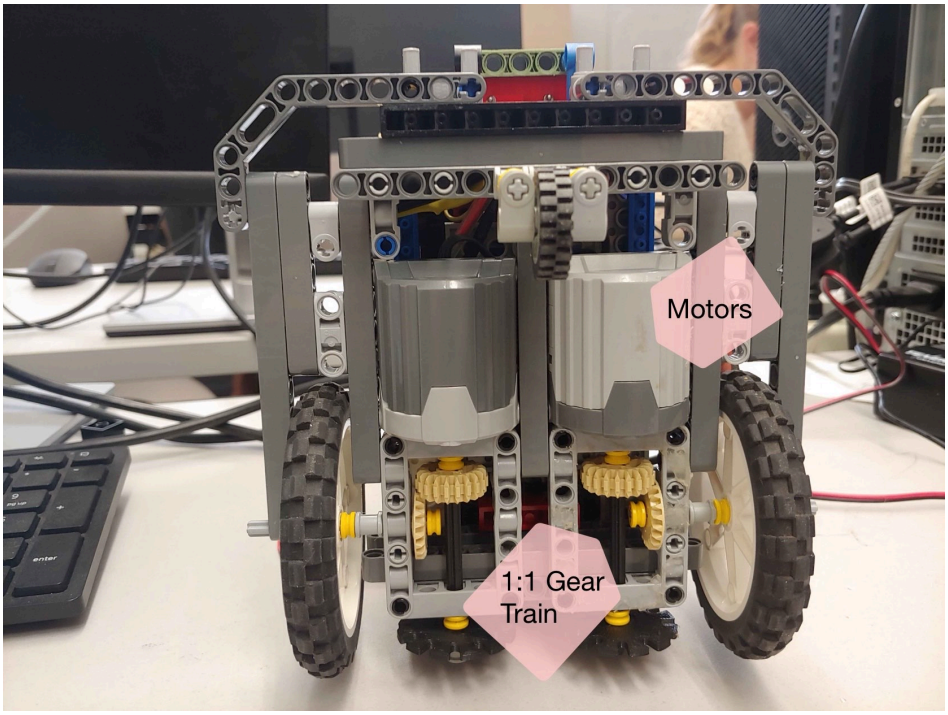


Figure 1. Labeled image the bottom of the car

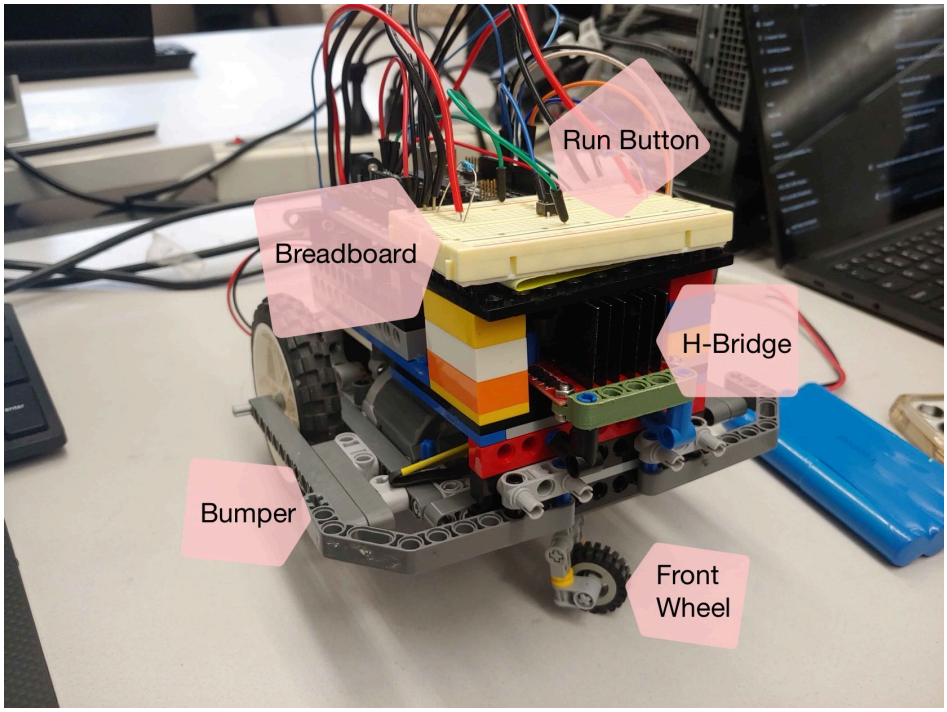


Figure 2. Labeled image of the front of the car

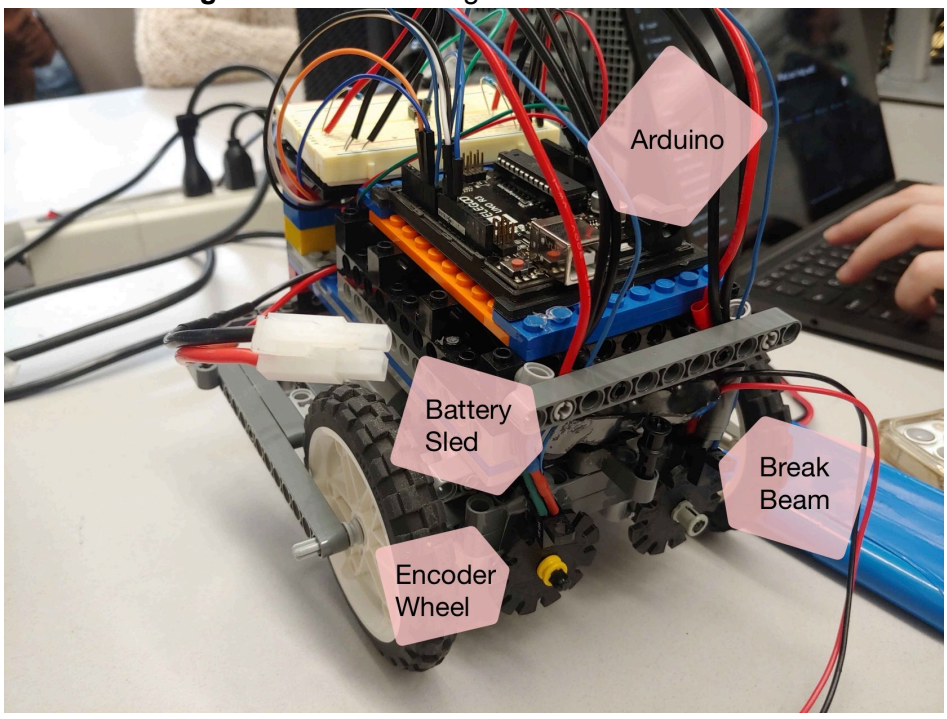


Figure 3. Labeled image of the back of the car

Methods

a. Sensor Configuration and Photo

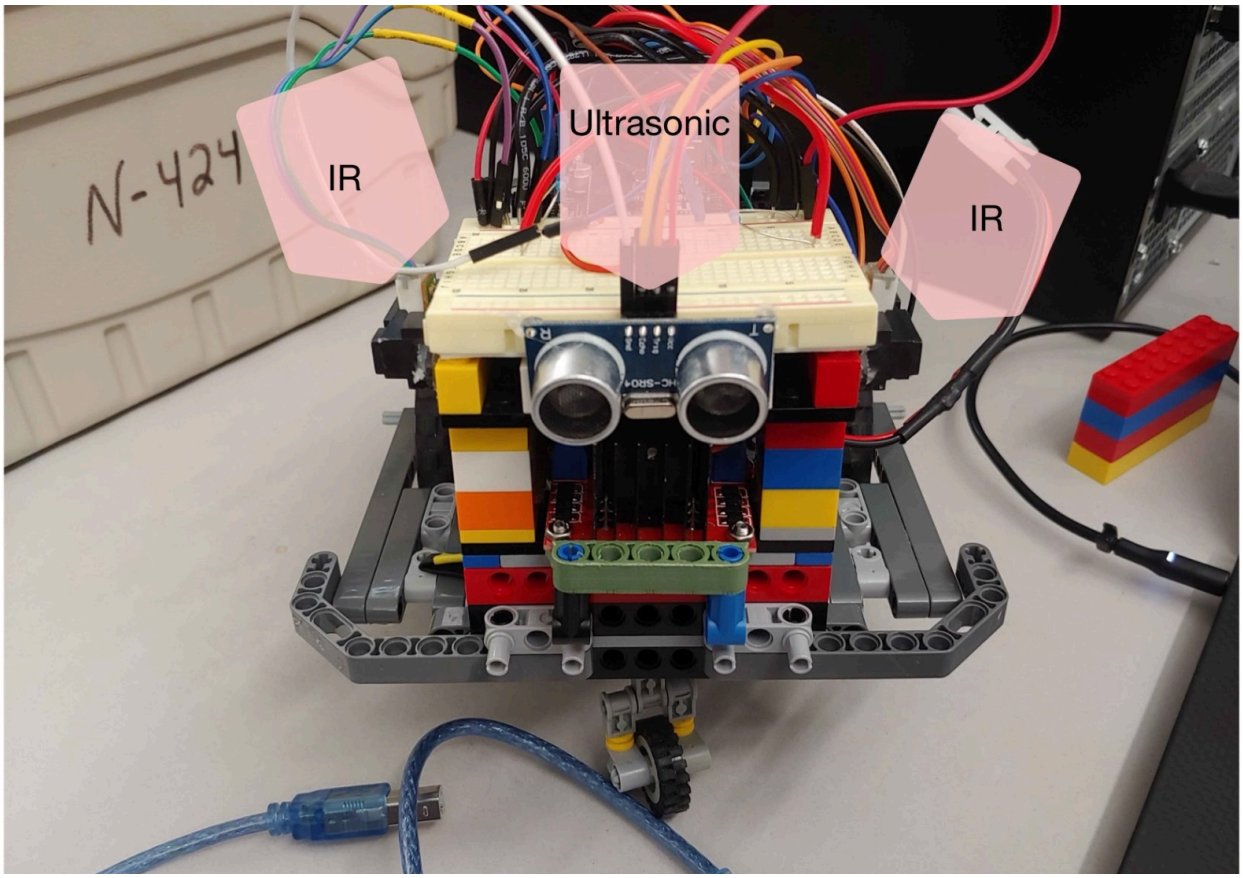


Figure 4. Labeled image of the new sensors

b. Overall code and testing Description

The first step will be integrating the new sensors into the code. The infrared sensors need to be calibrated, and the ultrasonic sensor library added. New functions will be added to return the readings from each sensor. Using this data, the robot can use the existing control system to move through the maze and use logic to determine the best direction to travel in.

c. Reading of path

The simplest version of the path reading algorithm will use the sensors to follow a wall on either the left or right side. This is a less efficient algorithm because it will result in the robot finding a lot of dead ends and needing to backtrack. Some basic research shows how competitive maze-solving robots solve their mazes. A popular and more efficient algorithm is called a waterfall algorithm[1]. Veritasium explores this concept in his video on maze solving, showing how a matrix can be constructed to assign a score to each position in the maze based on its distance from the goal position. An example of such a matrix for our maze is shown in Figure 5.

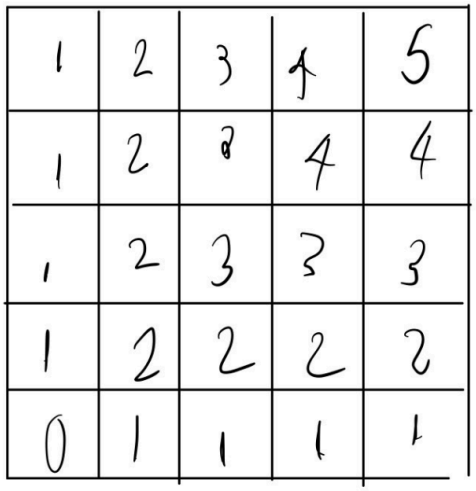


Figure 5. Example of waterfall solve matrix

Then, when multiple paths are available to the robot, the scores of the respective tiles can be compared, and the robot can take the one that gets it closer to the goal. In cases where the scores are the same, the robot can be instructed to favor the rightmost path. The shape and limited complexity of the maze result in this more complex algorithm putting the robot in the same number of dead ends as the right following method, but it is more applicable to different maze shapes and sizes. A combination of the sensor readings and encoder counts from the wheels will tell us when we have traveled the distance, marking one cube and assigning a direction to each direction in the maze, allowing us to generate coordinates for the current position. Using this information, as we explore the maze, we can append our current coordinates to the end of the history matrix to keep track of the route we have taken.

d. Method of eliminating deadends

We have some dead ends included by tracking the entire route we take in exploration. When duplicate coordinates are detected, the robot has visited the same place twice, and a dead end must have occurred. To remove these useless steps before generating our final solve instruction set, we iterate through the coordinate history till we find redundant coordinates with only one coordinate in between; this is the dead end. We can then work outwards from the dead end, pruning each redundant coordinate until the outermost redundant coordinate is found. One of the two outermost redundant coordinates will be kept so the robot still visits the first overlapping square. By iterating through the matrix, all the dead ends can be removed, and the resulting coordinates are the most direct route discovered. From these coordinates, an instruction set can be generated for the robot to execute using our existing control system.

Results

a. Problems you're facing

The constructed robot is having power issues. The Infrared sensors do not consistently read when connected to the battery, resulting in extreme difficulty debugging control logic. No matter what the wall tolerance is set to, the infrared sensors will only detect a wall missing one in five attempts.

b. Solutions attempted

The first solution attempted was to verify the sensor was consistently emitting light. A phone with a camera that doesn't have an infrared sensor was pointed at the infrared sensor emitter to see emitted light. The first sensor did not emit much light, so it was replaced. This did not solve the problem of the sensor failing to detect missing walls. Next, print statements were used inside the code to print out readings while on a test stand to check control logic. In this configuration, the sensors detect the missing wall every time. An eight-sample moving average filter was implemented to limit entrant readings to get an average of the results.

Similarly, when plugged into the computer, this filter gave consistent readings. When the computer connection was removed, and the robot was placed in the maze, it regularly failed to detect missing walls. Another attempted solution was to move pins and verify the sensor's wiring using the provided datasheet. The sensor was wired correctly, and moving to a different analog pin initially provided better results, but after a few attempts, the sensor failed to detect missing walls again. Since dropping battery voltage could affect the ADC readings, the battery was charged to full and re-installed in the robot, resulting in no change in the detection. The code was simplified to limit the chance that an errant loop or control logic influenced the read rate or ability to read. When the robot was only driven until it detected the missing wall, it consistently failed to detect it until it was plugged into a computer again to check serial readings. At that point, it detected the missing wall every time.

c. limiting factors

The inability to use the serial monitor to debug issues is incredibly frustrating. The Arduino's power indicator appears to dimmer when the computer is disconnected, and it switches over to battery power. This adds to the possibility that the battery through the h-bridge is providing inconsistent power to the Arduino, which could cause inconsistencies in the ADC readings. Because the supplied cable is too short to navigate the maze while plugged into the computer and sensors cannot be relied upon to navigate on battery power, it is impossible to build up and debug maze-solving algorithms. The issue could be addressed using different hardware, but further problem-solving would be required to address power issues outside the scope of this lab.

d. IR Sensor Characterization Data and Graphs

Distance (cm)	1/(d+k)	ADC	Voltage	m*V +b	d = 1/(m*V + b) - k	Error	Squared Error
0	#DIV/0!	4	0.019550342	0.005868524	170.1605997	-170.1605997	28954.62967
0.5	1.351351351	280	1.368523949	0.100296676	9.730420112	-9.230420112	85.20065545
1	0.806451613	300	1.46627566	0.107139296	9.093643542	-8.093643542	65.50706579
1.5	0.574712644	320	1.56402737	0.113981916	8.533321556	-7.033321556	49.46761 21
2	0.446428571	420	2.052785924	0.148195015	6.50786532	-4.50786532	20.32084974

3	0.308641975	450	2.19941349	0.158458944	6.070782926	-3.070782926	9.429707777
3.5	0.267379679	630	3.079178886	0.220042522	4.304576162	-0.804576162	0.6473428
4	0.235849057	600	2.93255132	0.209778592	4.526930642	-0.526930642	0.277655901
5	0.190839695	510	2.492668622	0.178986804	5.347004071	-0.347004071	0.120411826
6	0.16025641	430	2.101661779	0.151616325	6.355595844	-0.355595844	0.126448404
7	0.138121547	380	1.857282502	0.134509775	7.194403921	-0.194403921	0.037792885
8	0.121359223	360	1.759530792	0.127667155	7.592868185	0.407131815	0.165756315
9	0.108225108	300	1.46627566	0.107139296	9.093643542	-0.093643542	0.008769113
10	0.09765625	275	1.344086022	0.098586022	9.90342586	0.09657414	0.009326564
12	0.081699346	230	1.124144673	0.083190127	11.780657 2	0.2193428	0.048111264
14	0.070224719	200	0.977517107	0.072926197	13.47249338	0.527506618	0.278263232
16	0.061576355	180	0.879765396	0.066083578	14.89235261	1.107647385	1.22688273
18	0.054824561	160	0.782013685	0.059240958	16.64021319	1.359786811	1.849020171
20	0.049407115	140	0.684261975	0.052398338	18.8445747	1.155425299	1.335007621
25	0.039619651	115	0.562072336	0.043845064	22.56758469	2.432415308	5.916644229
30	0.033068783	95	0.464320626	0.037002444	26.78524205	3.214757948	10.33466867
35	0.028376844	75	0.366568915	0.030159824	32.91669211	2.083307891	4.340171767
40	0.024850895	60	0.293255132	0.025027859	39.71547484	0.284525163	0.080954568

Figure 4. Data Plots for the Right IR Sensor

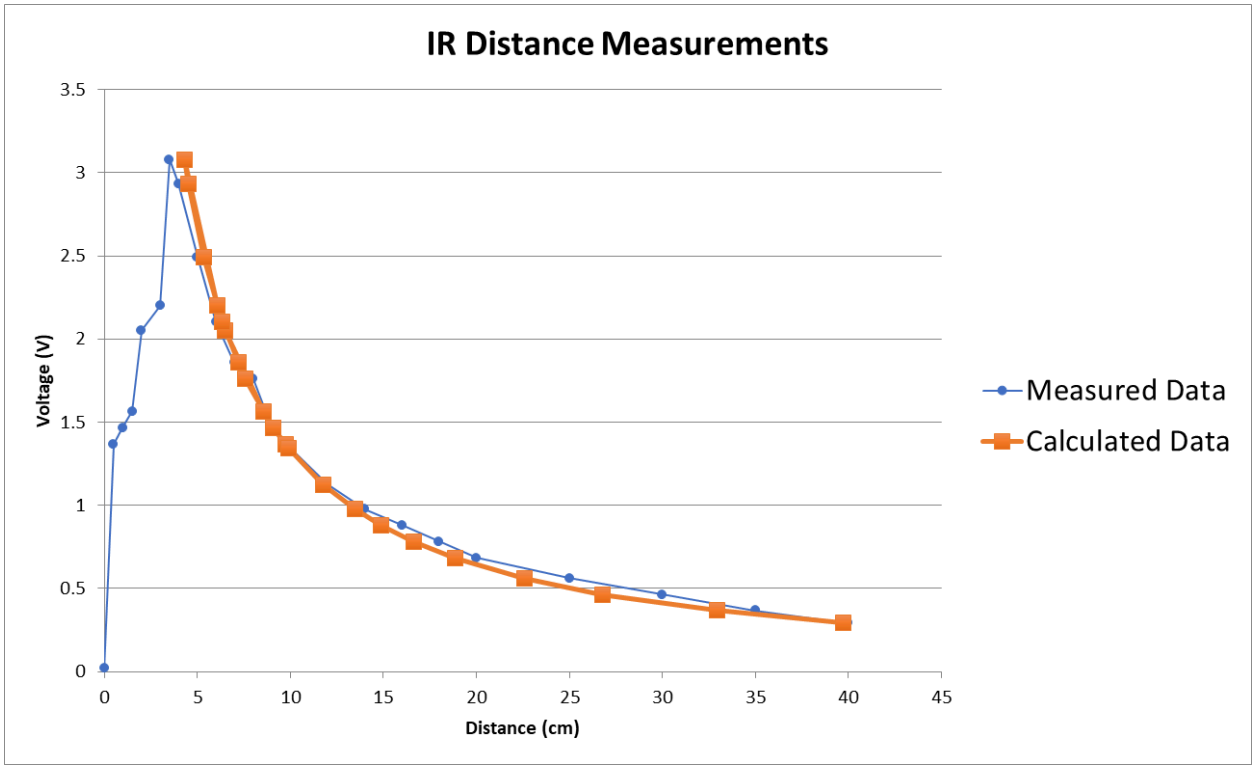


Figure 5. Graph plot for the Right IR Sensor

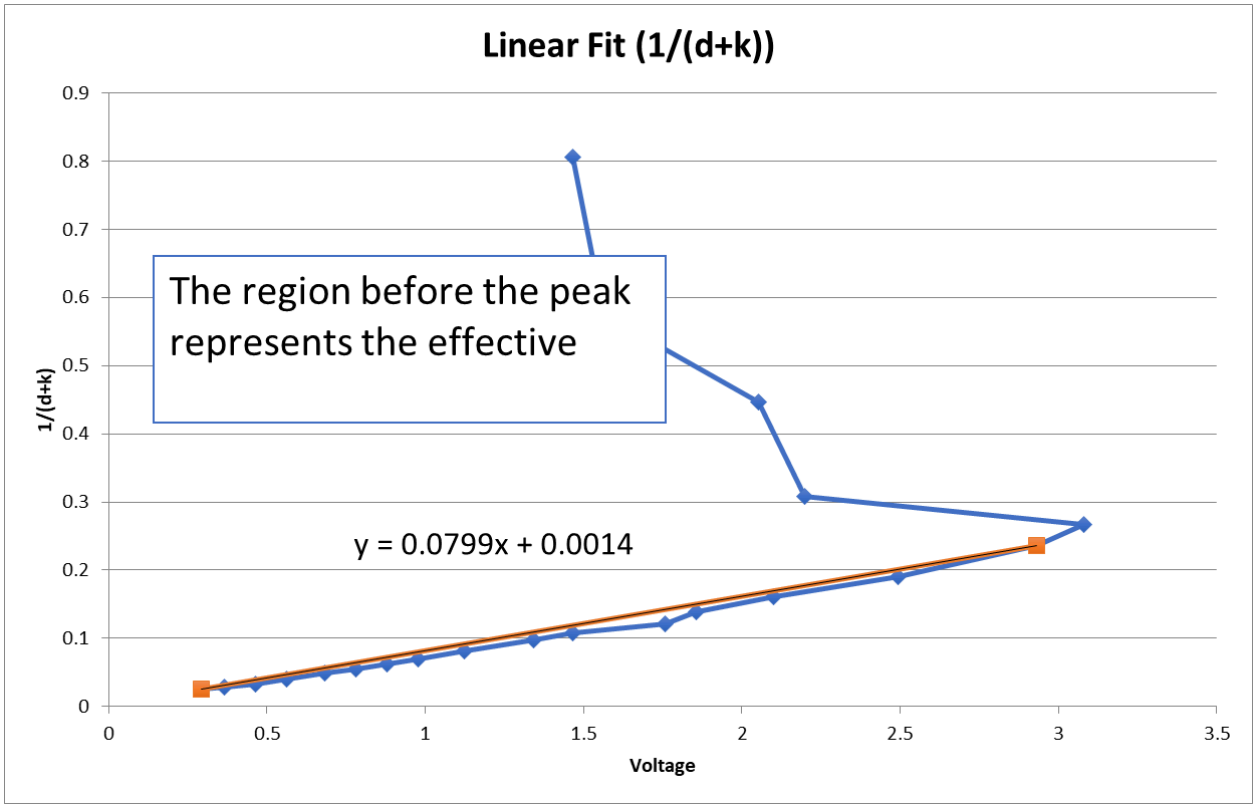


Figure 6. Linear Fit for the Right Sensor

Please note that, for some reason, our Excel document outputs this graph sideways for both the right and left IR sensors.

Distance (cm)	1/(d+k)	ADC	Voltage	m*V +b	d = 1/(m*V + b) - k	Error	Squared Error
0	#DIV/0!	4	0.019550342	0.000792913	1260.93241	-1260.93241	1589950.541
0.5	1.351351351	350	1.710654936	0.065054888	15.1316352	-14.6316352	214.0847487
1	0.806451613	400	1.955034213	0.0743413	13.21147312	-12.21147312	149.1200758
1.5	0.574712644	450	2.19941349	0.083627713	11.7177586	-10.2177586	104.4025909
2	0.446428571	350	1.710654936	0.065054888	15.1316352	-13.1316352	172.439843
3	0.308641975	550	2.688172043	0.102200538	9.544684339	-6.544684339	42.83289309

3.5	0.267379679	600	2.93255132	0.11148695	8.729659666	-5.229659666	27.34934022
4	0.235849057	640	3.128054741	0.11891608	8.169291651	-4.169291651	17.38299287
5	0.190839695	650	3.176930596	0.120773363	8.039971494	-3.039971494	9.241426682
6	0.16025641	640	3.128054741	0.11891608	8.169291651	-2.169291651	4.705826266
7	0.138121547	640	3.128054741	0.11891608	8.169291651	-1.169291651	1.367242965
8	0.121359223	600	2.93255132	0.11148695	8.729659666	-0.729659666	0.532403228
9	0.108225108	550	2.688172043	0.102200538	9.544684339	-0.544684339	0.296681029
10	0.09765625	500	2.443792766	0.092914125	10.52262623	-0.522626228	0.273138174
12	0.081699346	430	2.101661779	0.079913148	12.27358544	-0.273585436	0.074848991
14	0.070224719	380	1.857282502	0.070626735	13.91894418	0.081055817	0.006570045
16	0.061576355	330	1.612903226	0.061340323	16.06249007	-0.062490074	0.003905009
18	0.054824561	300	1.46627566	0.055768475	17.69127746	0.308722541	0.095309607
20	0.049407115	280	1.368523949	0.05205391	18.97085272	1.029147277	1.059144117
25	0.039619651	225	1.099706745	0.041838856	23.66122695	1.338773047	1.792313271
30	0.033068783	190	0.928641251	0.035338368	28.05785498	1.942145022	3.771927285
35	0.028376844	170	0.830889541	0.031623803	31.3817507	3.618249305	13.09172803
40	0.024850895	150	0.7331378 3	0.027909238	35.59043065	4.409569352	19.44430187

Figure 7. Data Plots for the Right IR Sensor

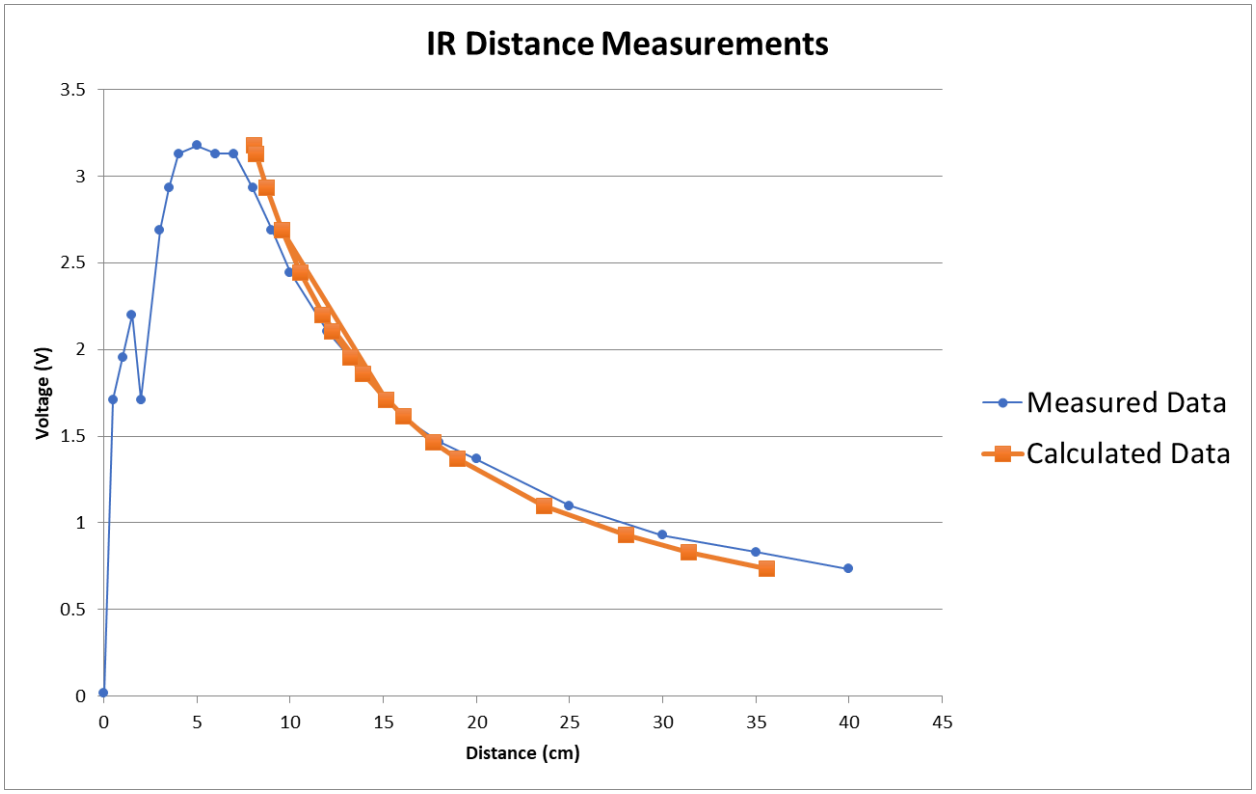


Figure 8. Graph plot for the Left IR Sensor

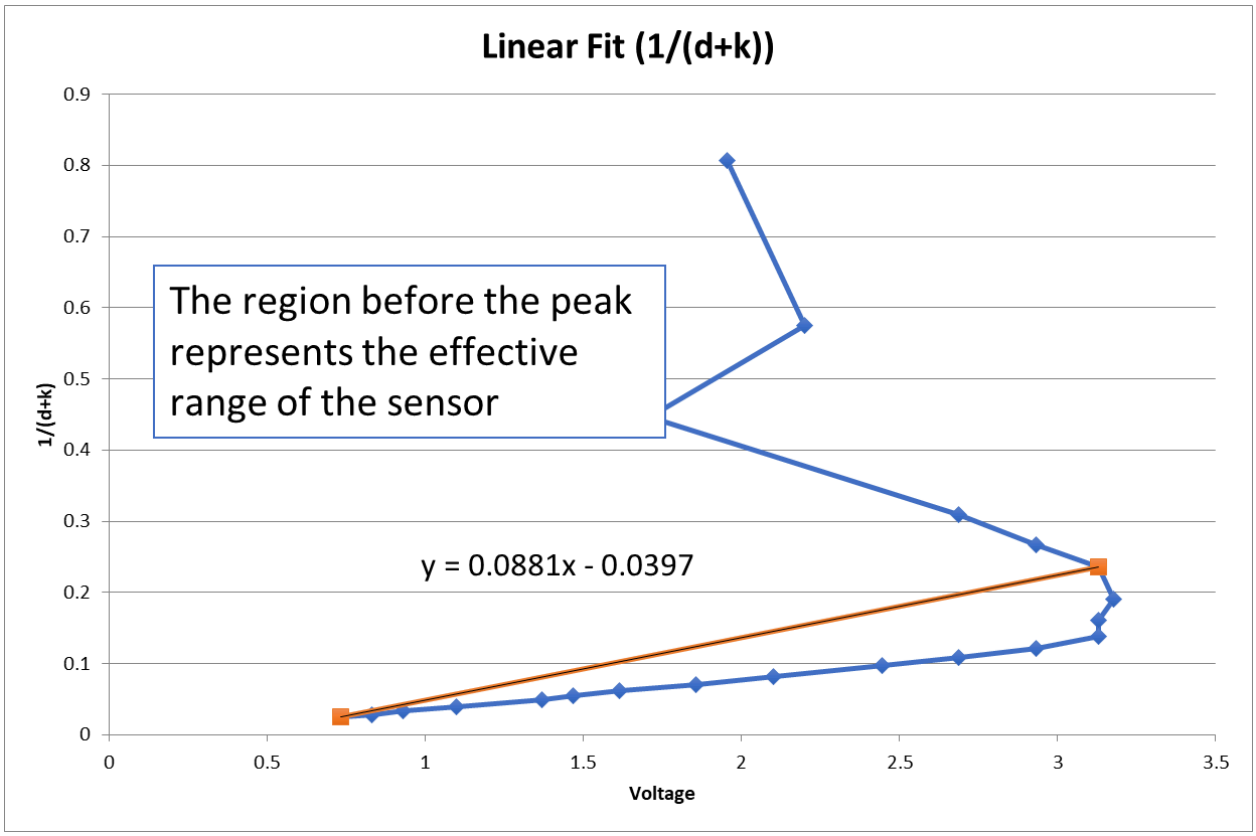


Figure 9. Linear Fit for the Left Sensor

Please note that, for some reason, our Excel document outputs this graph sideways for both the right and left IR sensors.

e. IR Sensor Characterization Equation

The IR sensor characterization equation for the right IR sensor is: dist = 1/(0.07*voltage + 0.0045) - 0.24.

The left IR sensor's characterization equation is dist = 1/(0.038*voltage + 0.00005) - 0.24.

f. Table of results

Due to the issues discussed above, results still need to be obtained.

Conclusions

The control systems built and tuned throughout the previous labs were very successful. Encoders with this low resolution aren't the most reliable units of measurement, but implementing reliable control on top of it resulted in consistent enough control for navigating the maze. The new sensor implementation for this lab ultimately killed our performance. Much of the time spent working on this lab was spent debugging these sensors, with a likely hardware issue resulting in an unsolvable problem. In theory, the maze-solving algorithm we research should be powerful and, when combined with our fast control system, should speed up the maze's exploration and running.

We were surprised by how unreliable the Arduino's ADC is in reading the infrared sensors. Additionally, it was surprising that disconnecting from the computer caused a significant enough change in the system to remove all reliability.

We used all the tools available to debug the system and attempt to resolve the issues we were running into, but no solution resulted. Sensor calibration went well; when connected to the computers, all the sensors read extremely accurately. The final robot construction was solid and introduced little error.

We could have done better implementing the control logic, but more debugging, with the help of the robot, would be needed to achieve a successful implementation. Our wiring also got progressively more messy as the labs progressed, and it would have been much easier to follow if it had been simplified and cleaned up using shorter wires on the breadboard cut to length.

Next time, a fuller test of the supplied hardware may reveal potential issues before spending considerable time trying to code around hardware problems that may not be solvable without replacing broken hardware.

Due to our hardware's limitations, we cannot test a final implementation of our algorithm.

References

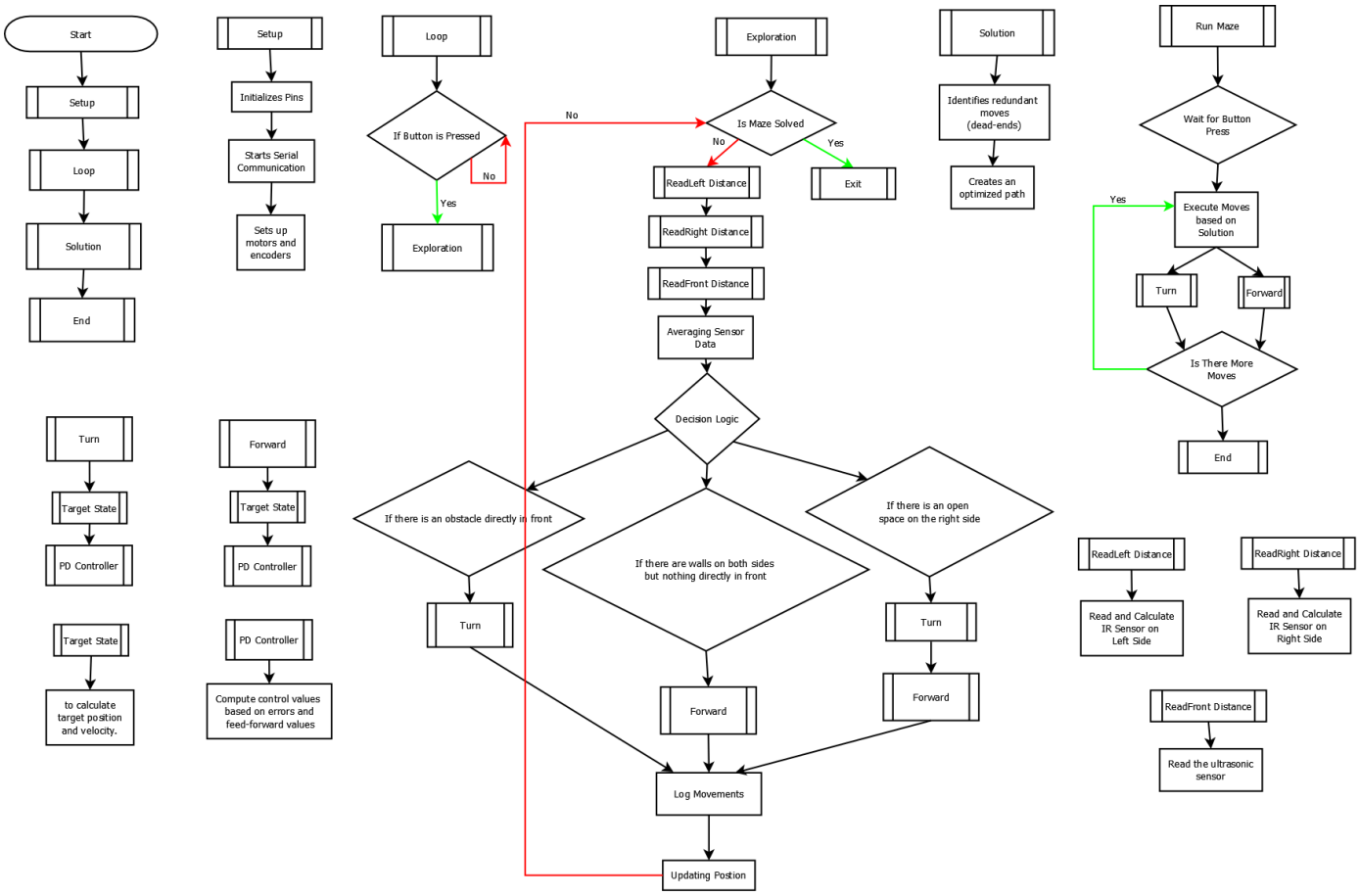
[1] Veritasium video on maze solving competitions - https://www.youtube.com/watch?v=ZMQbHMgK2rw

Appendices

a. Video Link:

Unable to make a video due to issues with our robot.

b. Software Diagram



c. Software

```
/* Lab 4: maze solving
   last updated: gkn 20240327
*/

#include <PinChangeInterrupt.h>
#include <HCSR04.h> // Ultrasonic distance sensor library for HC-SR04
// This code assumes the HC-SR04 Library by gamegine, but many work

// Define pins and constants for hardware setup
#define buttonPin 11

#define EncoderMotorLeft 7
#define EncoderMotorRight 8
```

```
#define A 0
#define B 1

// Motor driver shield settings
#define SHIELD false
#define motorApwm 3
#define motorAdir 12
#define motorBpwm 11
#define motorBdir 13

// Motor driver pins for a Dual H-Bridge setup
#define IN1 9
#define IN2 10
#define IN3 5
#define IN4 6

// Movement constants for direction
#define FORWARD 0
#define LEFT -75
#define RIGHT 85

#define pushButton 11

// Robot states and direction constants
#define STOPPED 0
#define SPEED 1
#define RUN 2
#define SLOW 3
#define NORTH 0
#define EAST 1
#define SOUTH 2
#define WEST 3

// Define IR distance sensor pins
#define LeftIR A2
#define RightIR A5

// Define Ultrasonic sensor pins
#define trig 4
```

```
#define echo 3

// Initialize ultrasonic sensor for the front
HCSR04 frontUS(trig, echo);

// Define distance tolerances for wall detection and collision prevention
#define wallTol 20 //cm
#define crashTol 7 //cm

// Initial state and direction variables
uint8_t currentState = 0;
uint8_t currentDirection = NORTH;

// Score table for maze-solving logic
uint8_t idealScores[5][5] = {
    {1, 1, 1, 1, 1},
    {1, 2, 2, 2, 2},
    {1, 2, 3, 3, 3},
    {1, 2, 3, 4, 4},
    {1, 2, 3, 4, 5}
};

// Movement tracking variables
int moves[50];
int maze[50];
int distance[50];
uint8_t pos[2];

// Structures for velocity and state data
struct velProfile {
  float t1;
  float t2;
  float tf;
  float vel;
};

struct state {
  float x;
  float v;
```

```
};

// PID and speed control variables
float Kp[] = {5.0, 4.75};
float Kd[] = {1, 1};

float maxSpeedA = 93.7;
float maxSpeedB = 112.4;

float cntCm = 0.955;
float DistancePerRev = 28;
float EncoderCountsPerRev = 24;
float l = 9;
float cmDegree = (2*3.5*l)/360;

float desiredMaxSpeed = 40; // Desired speed in cm/s
int desiredDistance = 30; // Test distance in cm

// Encoder variables for left and right motors
volatile unsigned int leftEncoderCount = 0;
volatile unsigned int rightEncoderCount = 0;

int PDdelay = 20;
unsigned long nextPDtime = 0;
int printDelay = 200;
unsigned long nextPrintTime = printDelay;
float rampFraction = 0.3;
struct velProfile profile;

// Setup function for initializing hardware and variables
void setup() {
  pinMode(pushButton, INPUT_PULLUP);
  Serial.begin(9600);
  motor_setup();
  encoder_setup();
  profile = genVelProfile(desiredMaxSpeed, desiredDistance, rampFraction);
}

// Main loop function
```

```
void loop(){
  // Wait for button press to start the maze exploration
  while (digitalRead(pushButton) == 1){
    Serial.print(readLeftDist());
    Serial.print("\t");
    Serial.print(readRightDist());
    Serial.print("\t");
    Serial.println(readFrontDist());
  }

  delay(50); // debounce input
  while (digitalRead(pushButton) == 0); // wait for button release
  delay(50); // debounce input

  explore(); // Start maze exploration
}

// Function to read distance from the left IR sensor
float readLeftDist() {
  float reading = analogRead(LeftIR);
  float voltage = (reading/1024)*5;
  float dist = 1/(0.038*voltage + 0.00005) - 0.24; // Using calibration
equation
  return dist;
}

// Function to read distance from the right IR sensor
float readRightDist() {
  float reading = analogRead(RightIR);
  float voltage = (reading/1024)*5;
  float dist = 1/(0.07*voltage + 0.0045) - 0.24; // Using calibration
equation
  return dist;
}

// Function to read distance from the front ultrasonic sensor
float readFrontDist() {
  float dist = frontUS.dist(); // Distance in cm
  return dist;
```

```
}

// Maze exploration function
void explore() {
  while (digitalRead(pushButton) == 1) { // Loop until the maze is solved
    // Read distances and average sensor inputs
    static uint16_t lastcount = 0;
    static uint8_t ii = 0;
    static uint8_t jj = 0;
    static float rightSide, leftSide, front;
    static float avgTotRight, avgTotLeft, avgTotFront;
    static float rightRead[8], leftRead[8], frontRead[8];

    // Capture samples for averaging sensor inputs
    while(jj < 8){
      rightRead[jj] = readRightDist();
      avgTotRight += rightRead[jj];
      leftRead[jj] = readLeftDist();
      avgTotLeft += leftRead[jj];
      frontRead[jj] = readFrontDist();
      avgTotFront += frontRead[jj];
      jj++;
    }

    // Calculate averages and determine directions based on maze structure

    // Moving logic based on sensor readings
    uint8_t desiredDirection;
    if(currentState == STOPPED && desiredDirection == FORWARD) {
      driveForward(1); // Run acceleration profile
      currentState = RUN;
    }
    else if(currentState == RUN && rightSide < wallTol && leftSide <
wallTol && front > crashTol) {
      driveForward(2); // Continue driving forward
      // Update position based on encoder count and direction
    }
    else {
      // Handle other movement cases
```

```
    }
  }
}

// Function to solve the maze using recorded moves
void solve() {
  // Implement algorithm to solve the maze using explore data
}

// Function to navigate the maze with the shortest path
void runMaze() {
   int j = 0;
  // Wait for button press to start maze run
  while (digitalRead(buttonPin) == 1);
  delay(50); // Debounce delay
  while (digitalRead(buttonPin) == 0); // Wait for button release
  for (int i = 0; i < sizeof(maze)/2; i++) {
    if(moves[i] == FORWARD){
      // Drive forward for specified distance
      j++;
    }
    else {
      run_motor(A, 0);
      turn(30, moves[i]); // Turn in specified direction
    }
    run_motor(A, 0);
    run_motor(B, 0);
  }
  j = 0; // Reset distance index
}

// Function to drive forward following a trapezoidal velocity profile
void driveForward(int forwardState) {
  // Implement forward driving logic based on velocity profile
}

// Function to turn by a specified angle
void turn(float maxAngularSpeed, float degrees) {
    // Convert speed to motor commands and encoder counts
```

```
}

// Function to calculate target state in velocity profile
struct state targetState(float time, struct velProfile vp) {
  // Calculate desired position and velocity at a given time
}

// PD controller function
int pdController(float FF, float Verr, float Xerr, float kp, float kd) {
  int u = FF + kp*Xerr + kd*Verr;
  int cmd = constrain(u, -255, 255);
  return cmd;
}
```