Memo

To:	Gary Nave, Chukwuebuka Okwor, and Ramon Chavez
From:	Gordon Dina and Coleson Oliver
Team #:	N-423
Date:	05 April 2023
Re:	Lab #4: Solving the Maze

Problem Statement

This lab is an extension of the Lab 3 Maze traversal, but instead of providing the robot with prior information, the robot is expected to use sensing to explore the maze using wall following, and then optimally reduce its exploration traversal to find the best path through the maze.

Methods

The sensor configuration of the robot is again an extension of the configuration for Lab 3 with the addition of two distance sensors.

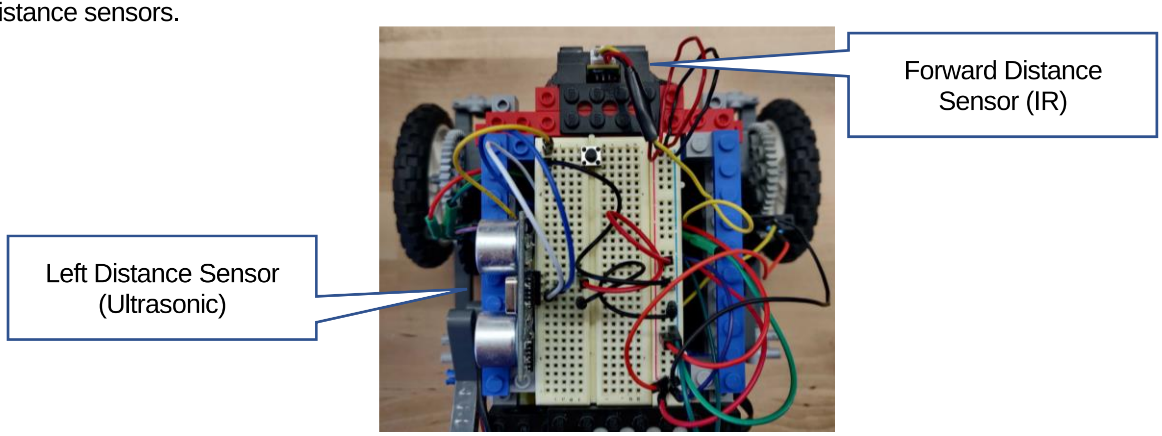


Figure 1: A top-down view of the robot with new sensors identified

The sensor characterization of the forward-facing IR sensor is shown in the figure below.

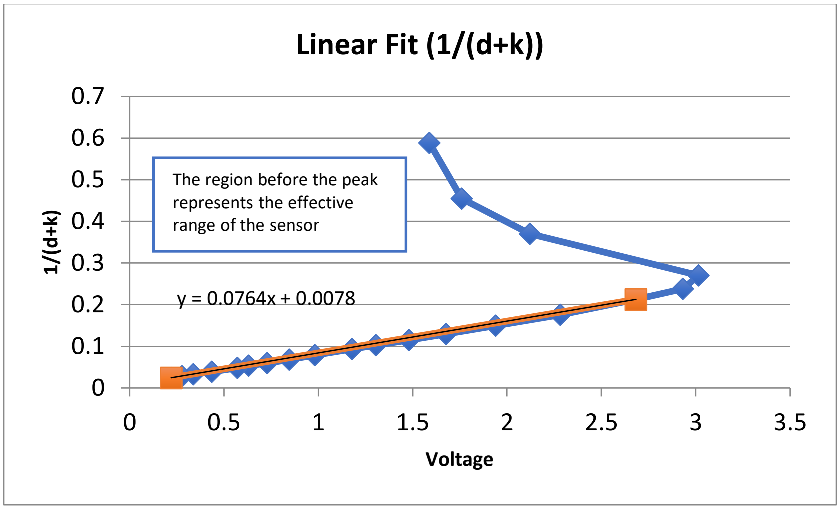


Figure 2: Graph of the calculated characterization of the IR sensor we used (uses information from [Nave 2023])

For the algorithm used to traverse and solve the maze, for the explore phase, the robot first checks the left distance sensor, and if it does not detect a wall, it turns left and drives forward. Otherwise, if there is a wall on its left, it then checks the forward distance sensor, and if there is no wall it drives forward, and otherwise turns right. To solve the maze, the algorithm linearly searches the stored array of moves, and in any location where a double right turn is discovered, it then checks the moves just before and after the double turn. In any case where the moves are rotationally symmetric (or in other words they would cancel each other out), it marks the discovered moves to be skipped. In any case where one of the discovered moves is a turn and the other is to drive forward, it reverses the turn, keeps the move forward and exits the dead-end discovery. Lastly in any case where the moves are to turn in the same direction, it skips both and exits the dead-end discovery. After exiting the dead-end discovery, the algorithm then continues the linear search from where it left off, skipping past any moves that have been marked to skip by previous dead-end discoveries. The advantage of this approach is that it uses only basic C data structures and is light on memory usage, and is also very simple to implement.

Results

For our one recorded run of the explore phase, the robot took roughly 77 seconds to complete the maze from start to finish.

For our three recorded runs of the run phase, the robot took 35.19, 34.82, and 35.65, seconds respectively, for an average of 35.22 seconds to complete the maze from start to finish.

Some manual intervention was necessary for the robot to successfully complete the maze traversal both in the explore phase and the run phase. Inaccuracies in the robot's exact turning angle and drive distance would cause it to run into the walls if unassisted, though the foundations of the drive controller, exploration procedure, and solve algorithm are all individually solid. Overall, the general inaccuracy and need for assistance can be attributed to ageing and inconsistent hardware and electrical components rather than the design of the system.

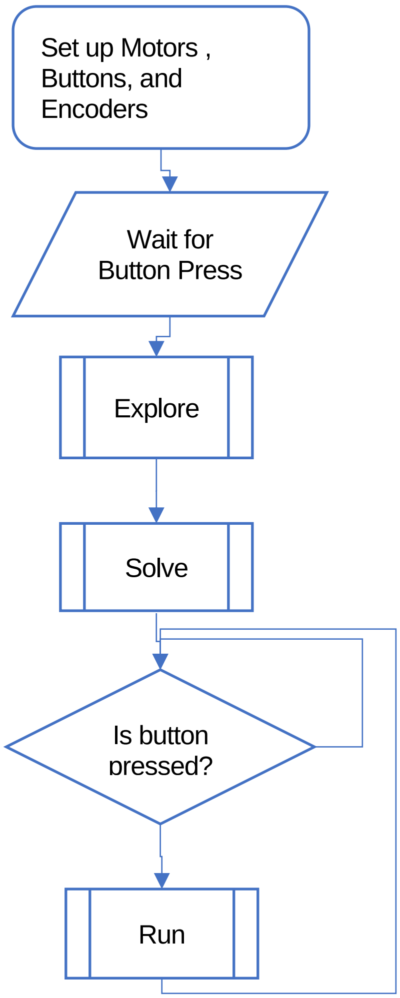
Conclusions

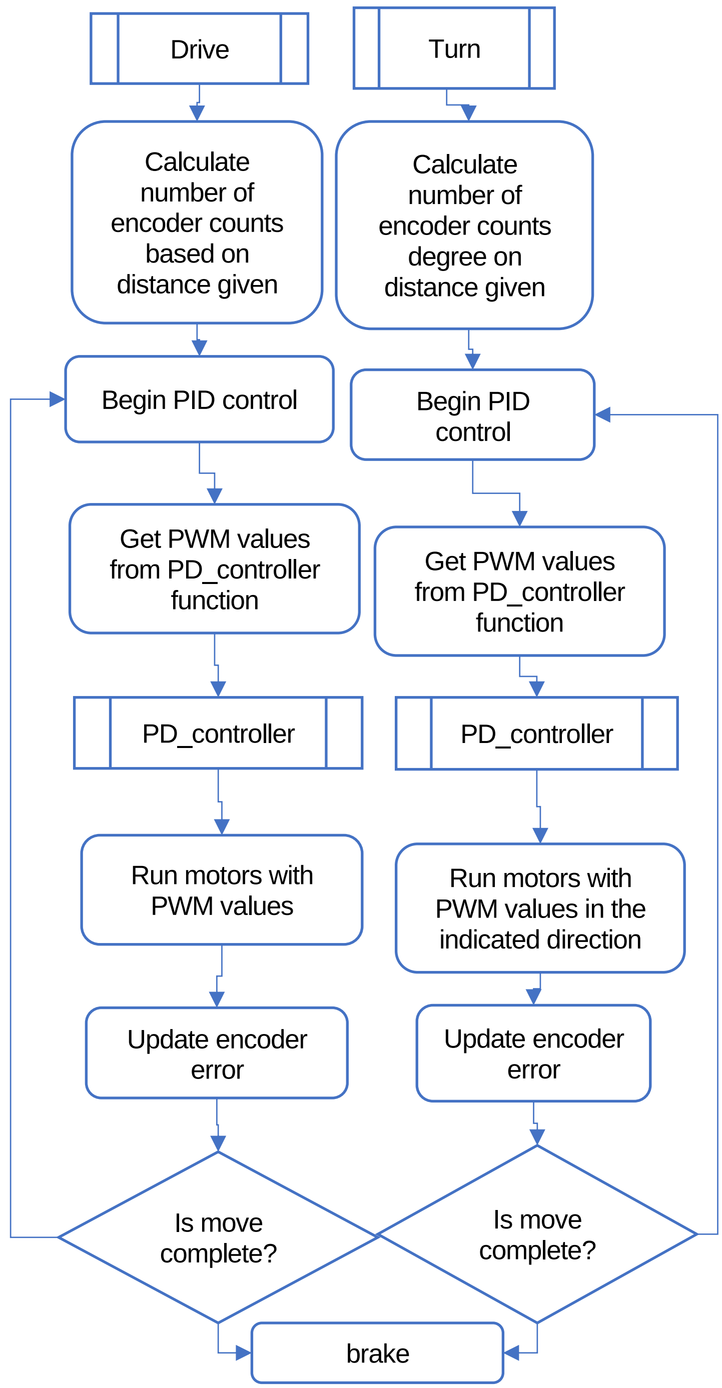
In general, outside of issues with inconsistent hardware and electrical components, the robot performed very well. The software used to explore and solve the maze works flawlessly and uses only basic C structures wherever possible (i.e. outside of the ultrasonic sensor library) for runtime efficiency. The only drawbacks of our design are inherent to the physical construction of the robot itself, particularly that it is nearly as large as the maze squares and does not turn about its center of mass, causing it to accumulate some error as it traverses. This would be the largest target for a change in design, as a smaller robot that turns about its center would have more room for the inherent random error to our hardware and would therefore be more fault tolerant.

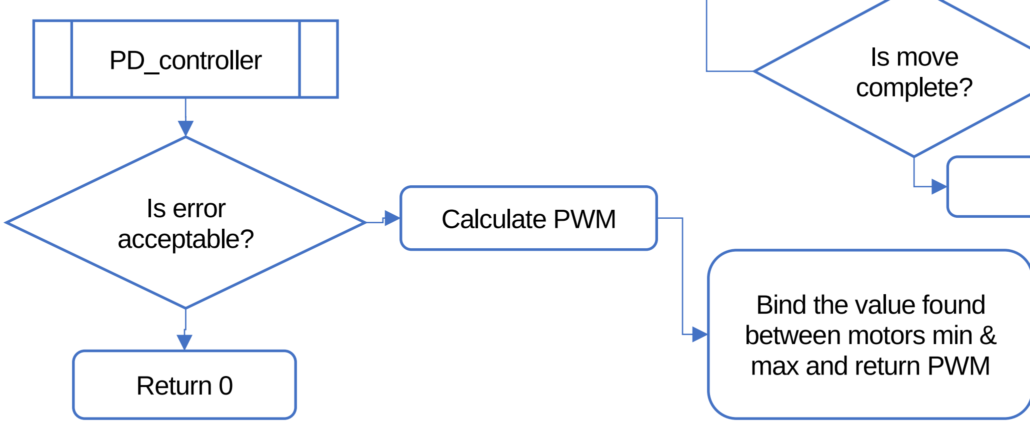
References

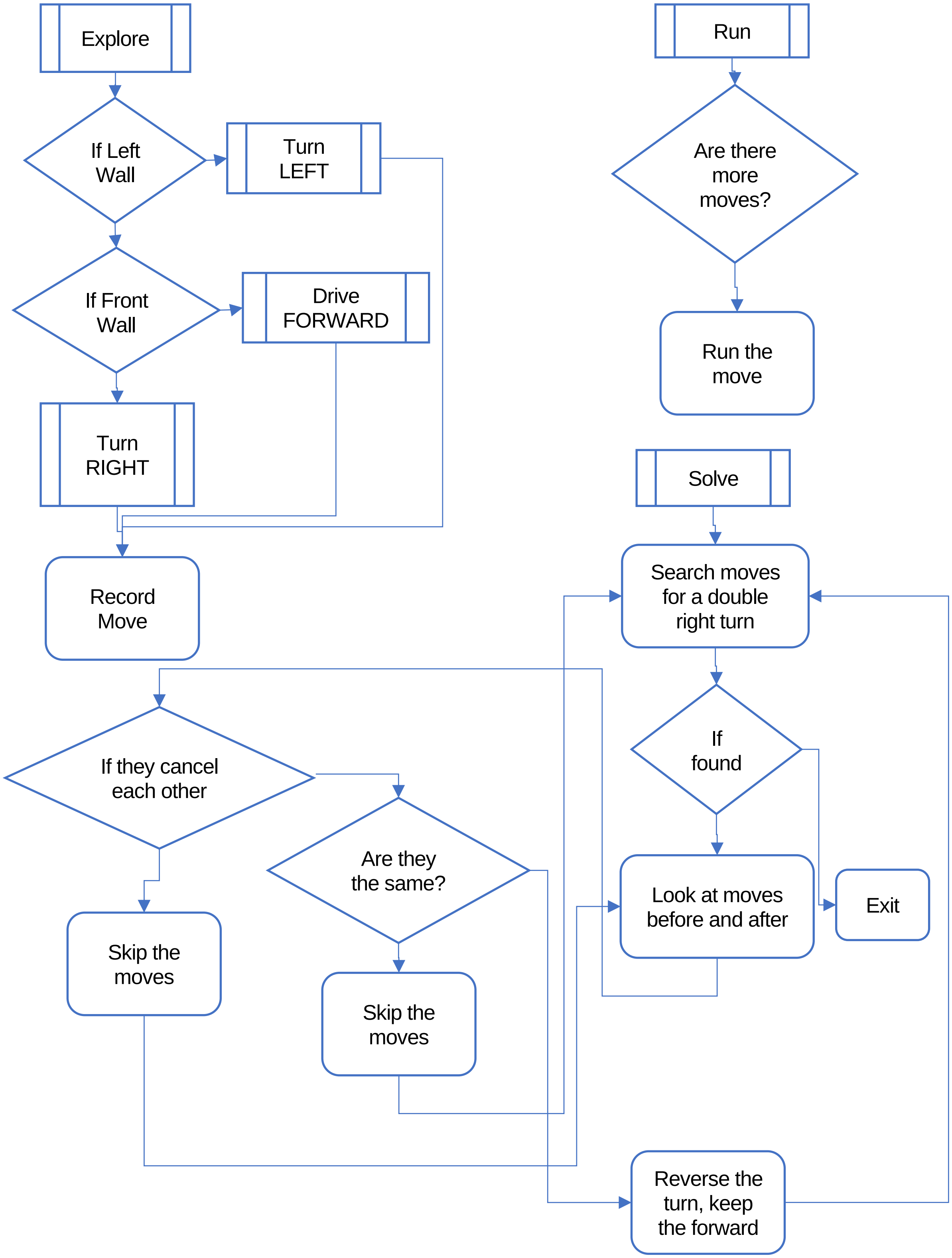
Gary Nave. "Distance Sensors". Colorado School of Mines. Retrieved 05 April 2023

Appendices


Set up Motors , Buttons, and Encoders
Wait for Button Press
Explore
Solve
Is button pressed?
Run


Drive
Turn
Calculate number of encoder counts based on distance given
Calculate number of encoder counts degree on distance given
Begin PID control
Begin PID control
Get PWM values from PD_controller function
Get PWM values from PD_controller function
PD_controller
PD_controller
Run motors with PWM values
Run motors with PWM values in the indicated direction
Update encoder error
Update encoder error
Is move complete?
Is move complete?
brake


PD_controller
Is error acceptable?
Calculate PWM
Return 0
Bind the value found between motors min & max and return PWM


Explore
If Left Wall
Turn LEFT
If Front Wall
Drive FORWARD
Turn RIGHT
Record Move
Run
Are there more moves?
Run the move
Solve
Search moves for a double right turn
If found
Look at moves before and after
Exit
If they cancel each other
Skip the moves
Are they the same?
Skip the moves
Reverse the turn, keep the forward

```
/* Lab 4: maze solving
   ms 20200926
   gkn 20230309
   co 20230310
*/

#include <PinChangeInterrupt.h>
#include <HCSR04.h> // If using any Ultrasonic Distance Sensors

// Copy constants and definitions from Lab 3

// Motor Pin & Variable definitions
#define SHIELD true

#define motorApwm 3
#define motorAdir 12
#define motorBpwm 11
#define motorBdir 13

//Driver Pin variable - any 4 analog pins (marked with ~ on your board)
#define IN1 9
#define IN2 10
#define IN3 5
#define IN4 6

#define A 0
#define B 1

// Move array definitions
#define FORWARD             30.48
#define LEFT                1.0
#define RIGHT              -1.0
#define SKIP                0.0

// Pin definitions
#define pushButton 10
#define EncoderMotorLeft  8
#define EncoderMotorRight 7

volatile unsigned int leftEncoderCount = 0;
volatile unsigned int rightEncoderCount = 0;

// Encoder and hardware configuration constants
#define EncoderCountsPerRev 120.0
```

```
#define DistancePerRev      25.5 // TODO: Measure wheel diameter (8.1
cm)         (Calculated Value 25.44)
#define DegreesPerRev       165 // TODO: Measure distance between wheels (18.8
cm) (Calculated Value 155.58)

// Proportional Control constants
#define kP_A 1.8
#define kP_B 2.0

// Derivative Control constants
#define kD_A 5.0
#define kD_B 1.0

// Controller Distance Tolerance (in encoder ticks)
#define distTolerance 3

// Deadband power settings
// The min PWM required for your robot's wheels to still move
// May be different for each motor
#define minPWM_A 75
#define minPWM_B 75

#define maxPWM_A 202
#define maxPWM_B 230

// Define IR Distance sensor Pins
#define frontIR A0
// #define sideIR  A1

// If using any Ultrasonic - change pins for your needs
#define trig 4
#define echo 5

// if side sensor is Ultrasonic
HCSR04 sideUS(trig, echo);
// if front sensor is Ultrasonic
//HCSR04 frontUS(trig, echo);

// Define the distance tolerance that indicates a wall is present
#define frontWallTol 15 //cm
#define sideWallTol 20 //cm

volatile unsigned int currentMove = 0;
```

```
float moves[50]; // Empty array of 50 moves, probably more than needed, just in
case

void setup() {
  Serial.begin(9600);

  // Set up motors
  Serial.println("Setting up the Motors");
  motor_setup();

  // Space for push button
  Serial.print("Setting up the Push Button: Pin ");
  Serial.print(EncoderMotorLeft);
  Serial.println();
  pinMode(pushButton, INPUT_PULLUP);

  // Valid interrupt modes are: RISING, FALLING or CHANGE
  Serial.print("Setting up the Left Encoder: Pin ");
  Serial.print(EncoderMotorLeft);
  Serial.println();
  pinMode(EncoderMotorLeft, INPUT_PULLUP); //set the pin to input
  attachPinChangeInterrupt(digitalPinToPinChangeInterrupt(EncoderMotorLeft),
indexLeftEncoderCount, CHANGE);
  Serial.print("Setting up the Right Encoder: Pin ");
  Serial.print(EncoderMotorRight);
  Serial.println();
  pinMode(EncoderMotorRight, INPUT_PULLUP);     //set the pin to input
  attachPinChangeInterrupt(digitalPinToPinChangeInterrupt(EncoderMotorRight),
indexRightEncoderCount, CHANGE);
}

void loop()
{
  while (digitalRead(pushButton) == 1); // wait for button push
  while (digitalRead(pushButton) == 0); // wait for button release
  explore();
  run_motor(A, 0);
  run_motor(B, 0);
  solve();
  while (true) { //Infinite number of runs, so you don't have to re-explore
everytime a mistake happens
    while (digitalRead(pushButton) == 1); // wait for button push
    while (digitalRead(pushButton) == 0); // wait for button release
    runMaze();
```

```
    run_motor(A, 0);
    run_motor(B, 0);
  }
}

float readFrontDist() {
  // If IR distance sensor
  int reading = analogRead(frontIR);
  float voltage = reading / 1023.0 * 5.0;
  float m = 0.0764;
  float b = 0.0078;
  float k = 0.7;
  float dist = (1.0 / (m*voltage + b)) - k;

  // if Ultrasonic
  // float dist = frontUS.dist(); //(returns in cm)

  return dist;
}

float readSideDist() {
  // If IR distance sensor
  // int reading = analogRead(sideIR);
  // float dist = // Equation from your calibration;

  // IF Ultrasonic
  float dist_total = 0;
  float DIST_THRESH = 30.0;
  int NUM_DATA_POINTS = 10;
  for (int i = 0; i < NUM_DATA_POINTS; i++) {
    float value = sideUS.dist();
    if (value > DIST_THRESH) value = DIST_THRESH;
    dist_total += value;
  }
  return dist_total/NUM_DATA_POINTS;
}

void explore() {
  while (digitalRead(pushButton) == 1) { //while maze is not solved
    // Read distances
    float side = readSideDist();
    float front = readFrontDist();
    if (side > sideWallTol) {// If side is not a wall
      // turn and drive forward
```

```
      // Record actions
      moves[currentMove] = LEFT;
      currentMove++;
      turn(LEFT, 90);
      delay(1000);
      moves[currentMove] = FORWARD;
      currentMove++;
      drive(FORWARD);
      delay(1000);
    }
    else if (front > frontWallTol) {// else if front is not a wall
      // drive forward
      // Record action
      moves[currentMove] = FORWARD;
      currentMove++;
      drive(FORWARD);
      delay(1000);
    } else {
      // turn away from side
      // Record action
      moves[currentMove] = RIGHT;
      currentMove++;
      turn(RIGHT, 90);
      delay(1000);
    }
  }
  while(digitalRead(pushButton) == 0) {}; // wait for button release
}

void solve() {
  // Write your own algorithm to solve the maze using the list of moves from
explore
  for(int i = 0; i < sizeof(moves)/sizeof(float); i++) {
    if(moves[i] == RIGHT && moves[i+1] == RIGHT) {
      // replace the double right
      moves[i] = SKIP;
      moves[i + 1] = SKIP;
      // initialize lower and upper to start looking just outside of the detected
180 deg turn
      int upper = i + 2;
      int lower = i - 1;
      // loop to remove dead-end segments
      while (upper < sizeof(moves)/sizeof(float) - 1 && lower > 0) { // while in
bounds
```

```
        // skip the skips
        while(moves[lower] == SKIP && lower >= 0) lower--;
        while(moves[upper] == SKIP && upper <= sizeof(moves)/sizeof(float) - 1)
upper++;
        // skip moves that counteract each other
        if(moves[lower] == FORWARD && moves[upper] == FORWARD || moves[lower] ==
moves[upper] * -1.0) {
          moves[lower] = SKIP;
          moves[upper] = SKIP;
        }
        // identify exit of dead-end and correct movement to avoid it
        else {
          if(moves[lower] == FORWARD) {
            moves[upper] *= -1.0; // reverse direction of the turn that took us
in/out of the dead-end
          }
          else if(moves[upper] == FORWARD) {
            moves[lower] *= -1.0; // reverse direction of the turn that took us
in/out of the dead-end
          }
          else if(moves[lower] == moves[upper]) {
            moves[lower] = SKIP;
            moves[upper] = SKIP;
          }
          break; // exit
        }
        // increment counters for next loop
        lower--;
        upper++;
      }
    }
    while(moves[i] == SKIP) i++; // skip the skips
  }
}


void runMaze() {
  for(int i = 0; i < sizeof(moves)/sizeof(float); i++) {
    if(moves[i] == FORWARD) {
      drive(FORWARD);
    }
    else if(moves[i] == RIGHT) {
      turn(RIGHT, 90);
    }
```

```
    else if(moves[i] == LEFT) {
      turn(LEFT, 90);
    }
    else {
      continue;
    }
    delay(1500);
  }
}

// Copy any necessary functions from Lab 3

//////////////////////////////////////////////////////////
int PD_Controller(float kP, float kD, int xerr, int dxerr, float dt, int minPWM,
int maxPWM)
//  gain, deadband, and error, both are integer values
{
  if (xerr <= distTolerance) { // if error is acceptable, PWM = 0
    return (0);
  }

  // Proportional and Derivative control
  int pwm = int((kP * xerr) + (kD * (dxerr/dt)));
  pwm = constrain(pwm,minPWM,maxPWM); // Bind value between motor's min and max
  return(pwm);
}
//////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////////
int drive(float distance)
{
  // create variables needed for this function
  int countsDesired, cmdLeft, cmdRight, xerr_L, xerr_R, dxerr_L, dxerr_R;
  float dt;

  // Find the number of encoder counts based on the distance given, and the
  // configuration of your encoders and wheels
  countsDesired = distance / DistancePerRev * EncoderCountsPerRev;

  // reset the current encoder counts
  leftEncoderCount = 0;
  rightEncoderCount = 0;
```

```
  // we make the errors greater than our tolerance so our first test gets us into
the loop
  xerr_L = distTolerance + 1;
  xerr_R = distTolerance + 1;

  dxerr_L = 0;
  dxerr_R = 0;

  unsigned long t0 = millis();

  // Begin PID control until move is complete
  while (xerr_L > distTolerance || xerr_R > distTolerance)
  {
    dt = float(millis() - t0);
    t0 = millis();

    // Get PWM values from the controller function
    cmdLeft  = PD_Controller(kP_A, kD_A, xerr_L, dxerr_L, dt, minPWM_A,
maxPWM_A);
    cmdRight = PD_Controller(kP_B, kD_B, xerr_R, dxerr_L, dt, minPWM_B,
maxPWM_B);

    // Run motors with calculated PWM values
    run_motor(A, cmdLeft);
    run_motor(B, cmdRight);

    // Update encoder error
    // Error is the number of encoder counts between here and the destination
    dxerr_L = (countsDesired - leftEncoderCount ) - xerr_L;
    dxerr_R = (countsDesired - rightEncoderCount) - xerr_R;

    xerr_L = countsDesired - leftEncoderCount;
    xerr_R = countsDesired - rightEncoderCount;

    // Some print statements, for debugging
    Serial.print(xerr_L);
    Serial.print(" ");
    Serial.print(dxerr_L);
    Serial.print(" ");
    Serial.print(cmdLeft);
    Serial.print("\t");
    Serial.print(xerr_R);
    Serial.print(" ");
    Serial.print(dxerr_R);
```

```
    Serial.print(" ");
    Serial.println(cmdRight);
  }
  run_motor(A, 0);
  run_motor(B, 0);

}
////////////////////////////////////////////////////////////////////////////////

// Write a function for turning with PID control, similar to the drive function
int turn(float direction, float degrees)
{
  // create variables needed for this function
  int countsDesired, cmdLeft, cmdRight, xerr_L, xerr_R, dxerr_L, dxerr_R;
  float dt;

  // Find the number of encoder counts based on the distance given, and the
  // configuration of your encoders and wheels
  countsDesired = degrees / DegreesPerRev * EncoderCountsPerRev;

  // reset the current encoder counts
  leftEncoderCount = 0;
  rightEncoderCount = 0;

  // we make the errors greater than our tolerance so our first test gets us into
the loop
  xerr_L = distTolerance + 1;
  xerr_R = distTolerance + 1;

  dxerr_L = 0;
  dxerr_R = 0;

  unsigned long t0 = millis();

  // Begin PID control until move is complete
  while (xerr_L > distTolerance || xerr_R > distTolerance)
  {
    dt = float(millis() - t0);
    t0 = millis();

    // Get PWM values from controller function
    cmdLeft  = PD_Controller(kP_A, kD_A, xerr_L, dxerr_L, dt, minPWM_A,
maxPWM_A);
```

```
    cmdRight = PD_Controller(kP_B, kD_B, xerr_R, dxerr_L, dt, minPWM_B,
maxPWM_B);

    // Run motors with calculated PWM values in the indicated direction
    run_motor(A, direction * cmdLeft);
    run_motor(B, -direction * cmdRight);

    // Update encoder error
    // Error is the number of encoder counts between here and the destination
    dxerr_L = (countsDesired - leftEncoderCount ) - xerr_L;
    dxerr_R = (countsDesired - rightEncoderCount) - xerr_R;

    xerr_L = countsDesired - leftEncoderCount;
    xerr_R = countsDesired - rightEncoderCount;

    // Some print statements, for debugging
    Serial.print(xerr_L);
    Serial.print(" ");
    Serial.print(dxerr_L);
    Serial.print(" ");
    Serial.print(cmdLeft);
    Serial.print("\t");
    Serial.print(xerr_R);
    Serial.print(" ");
    Serial.print(dxerr_R);
    Serial.print(" ");
    Serial.println(cmdRight);
  }

  run_motor(A, 0);
  run_motor(B, 0);
}

// These are the encoder interrupt funcitons, don't need to edit.
void indexLeftEncoderCount()
{
  leftEncoderCount++;
}
void indexRightEncoderCount()
{
  rightEncoderCount++;
}
```