**Rose-Hulman Institute of Technology**

**ECE425-01**

**Introduction to Mobile Robotics**

**Final Project: Navigation Competencies, GUI, and Wireless Communication**

**Team Murphy**

**Jackson Seida & Justin DeWitt**

**02/18/2024**

## Abstract

**Table of Contents**

[Abstract 2](#_Toc158980710)

[Objective 4](#_Toc158980711)

[Theory 5](#_Toc158980712)

[Methods 6](#_Toc158980713)

[Results 7](#_Toc158980714)

[Conclusions & Recommendations 8](#_Toc158980715)

[Questions 9](#_Toc158980716)

[Appendix 12](#_Toc158980717)

## Objective

The final project of this course entails applying concepts learned throughout the quarter to path plan, navigate, wirelessly connect, and create a GUI with the mobile robot. The path planning and navigation includes topological path planning, and metric path planning. Topological path planning means the robot traverses the world based on input commands and sensor feedback. Metric path planning means inputting a world map and having the robot solve for an optimal path, usually using a wavefront algorithm on occupancy grids. After solving for the optimal path, the robot will then navigate to the goal. The robot will be able to complete topological and metric path planning on topological maps and occupancy grids, along with be controlled wirelessly with live sensor data being displayed on a GUI.

## Theory

The theories applied to our robot were that of occupancy grids, wavefront algorithms, and topological maps and found in [1].

An occupancy grid is a matrix of values that represent where pathable areas are, and where non-pathable areas are, i.e. where the walls are at. This can be represented by 0’s in the matrix being where the robot can traverse, and 99’s representing where the robot cannot traverse. The occupancy grid that our robot traversed is shown in figure 1 below.

0 99 99 0

0 0 0 0

0 99 99 0

0 99 0 0

*Figure 1: Occupancy grid traversed by our robot*

A wavefront algorithm can be utilized on an occupancy grid to path plan. A wavefront algorithm works by defining where the robot’s starting point and goal are, then iterating through the occupancy grid until a path between the two is found. An example of this is defining the goal node of the robot as 1, and the robot starting position as 98. Each node of the matrix with a value of 0 is then valued based on how far it is from the goal node. For instance, the nodes next to the goal are labeled 2, then the ones next to the 2 node are labeled 3, and so on. An example of the wavefront algorithm applied to the occupancy grid in figure 1, is shown in figure 2 below.

0 99 99 0 9 99 99 5

0 0 0 0 8 7 6 4

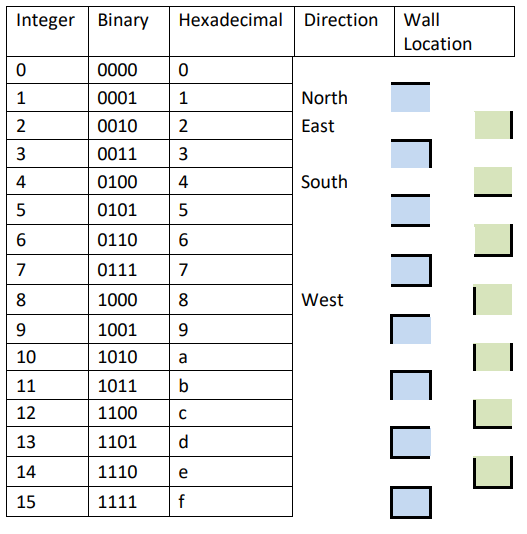
0 99 99 0 9 99 99 3

98 99 1 0 98 99 1 2

*Figure 2: Wavefront algorithm applied to occupancy grid*

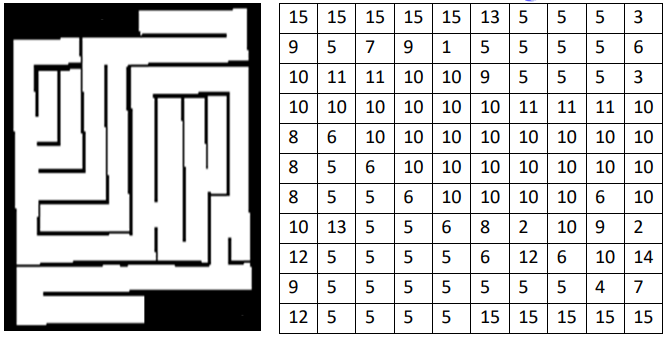
The robot will then navigate to the goal by always travelling to the nodes with the lowest values.

Another method of representing the world to a robot is a topological map. Breaks up the world into 16 different wall orientations corresponding to integer values 0-15. These 16 wall orientations can be seen in figure 3 below.



*Figure 3: Topological grid wall orientation table*

For example, a topological map with its real-world representation is show in figure 4 below.



*Figure 4: Topological map and real-world representation*

To path plan in a topological map, we converted the topological map to an occupancy grid, as discussed in the Methods section.

## Methods

The pseudocode for the following methods can be found in the Appendix.

Topological Path Planning & Execution:

To apply topological path planning and execution to our mobile robot navigation commands were inputted to a function which looks for distinctive features in the physical world. These navigation commands are in the form of a string comprised of characters ‘S’, ‘L’, ‘R’, and ‘T’, standing for start, left, right, and terminate respectively. The start command starts the robot wall following in the direction it is facing. The left and right command has the robot wall following until an opening on the corresponding side is detected, then turning and continuing wall following. The terminate command has the robot continue wall following until walls are detected on the front, left and right sides of the robot.

Metric Map Path Planning & Execution:

The application of a wavefront algorithm, as discussed in the Theory section, was utilized to accomplish metric map path planning and execution on both occupancy grids and topological maps. On an occupancy grid, the robot would move towards the goal by moving into the nodes that had the lowest values. This was accomplished via a function which converted an occupancy post wavefront algorithm to SLRT commands. For instance, this function applied to the wavefront map in figure 2 results in the command SRRRT.

Metric path planning on a topological map was achieved by converting the topological map to an occupancy grid. This was done by converting each topological map node to a 3x3 occupancy grid matrix. An example of this is shown in figure 5 below.

|  |  |  |
| --- | --- | --- |
| 99 | 99 | 99 |
| 99 | 0 | 0 |
| 99 | 0 | 0 |

A blue rectangle with black border

Description automatically generated

9

*Figure 5: Topological map orientation 9 to 3x3 occupancy grid*

This means that a 4x4 topological map is converted to a 12x12 occupancy grid. A wavefront algorithm can then be applied to the occupancy grid and the metric map planning and execution for an occupancy grid can be utilized.

Wireless Communication:

GUI:

## Results

Topological Path Planning & Execution:

The robot was successful in completing topological path planning and execution. This was verified by completing two different world orientations without colliding or getting stuck. These two world orientations can be found in figure 6 below.



*Figure 6: World orientations complete by topological path planning and execution*

The lidar sensors worked well at providing accurate data that allowed the robot to wall follow and detect distinctive features.

Metric Map Path Planning & Execution:

The robot was also successful in completing metric path planning and execution. The robot navigation operated the same as in topological path planning and execution. The difference was the robot first determined its path via the wavefront algorithm vs being given a path via SLRT commands. This metric path planning was applied to the two world orientations shown in figure 6.

Wireless Communication:

GUI:

## Conclusions & Recommendations

## Questions

**1. Were there any issues with the wireless communication? How could you resolve them? If at all.**

**2. What does the state machine, subsumption architecture, flowchart, or pseudocode look like for the path planning, localization, and mapping? (It should be in the appendix of the report).**

See A.1, A.2, and A.3 in the Appendix

**3. How would you implement SLAM on the CEENBot given what you have learned about navigation competencies after completing the final project? If you research solutions, make sure you cite and list references in APA or MLA format.**

**4. What was the strategy for implementing the wavefront algorithm?**

The wavefront algorithm was implemented by checking each of the nodes surrounding a node, north, east, south, and west, for the lowest value then determining if that value is not a wall or a blank space. If that value is not a wall or black space, then set the current node to the lowest value plus one. Iterate this function until no black space nodes are left in the matrix.

**5. Were there any points during the navigation when the robot got stuck? If so, how did you extract the robot from that situation?**

The robot did not get stuck during the navigation because of the robust wall following function that was implemented.

**6. How long did it take for the robot to move from the start position to the goal?**

Depending on the map, it would take anywhere between 20 – 40 seconds to reach the goal. This depends on the amount of turns the robot must make and the amount of ground the robot must cover.

**7. What type of algorithm did you use to select the most optimal or efficient path?**

We used a wavefront algorithm to determine the most optimal path.

**8. How did you represent the robot’s start and goal position at run time?**

The robot’s start and goal position were represented by the integer values 98 and 1 respectively. These values were stored within the occupancy grid matrix.

**9. Do you have any recommendations for improving that robot’s navigation or wavefront algorithm?**

The wavefront algorithm we implemented is not able to detect if the is a possible path to the goal. The algorithm just runs for a max of 100 iterations through the matrix. This means that if there is no valid path to the goal the robot would throw an error when trying to run the wavefrontToSLRT function. The biggest improvement would be to have an error thrown in the wavefront algorithm function that tells the user there is no valid solution.

**10. How did you use the serial monitor and bi-directional wireless communication to represent the map?**

N/A

**11. What type of map did you create and why?**

N/A

**12. What was key in the integration of the localization, mapping, and path planning?**

N/A

## References

[1] Berry, Carlotta A. “Mobile Robotics for Multidisciplinary Study.” *Synthesis Lectures on Control and Mechatronics*, vol. 3, no. 1, 31 Mar. 2012, pp. 1–95.

## Appendix

A.1 Topological Path Planning Software Design Plan:

Function input is a string of ‘S’, ‘L’, ‘R’, and ‘T’

Iterate through string characters in SLRTchar

If SLRTchar == S

Go forward

If SLRTchar == L and no left wall

Turn left

If SLRTchar == R and no right wall

Turn right

If SLRTchar == T and dead end

Stop

A.2 Metric Path Planning on Occupancy Grid Design Plan:

Function input is a 4x4 Matrix that is the occupancy grid, and an integer representing the starting direction

Convert occupancy grid to wavefront map:

While there are zeros in the map

Iterate through all matrix nodes

Check north, east, south, west nodes for node with lowest non-zero value

Set the current node to lowest surrounding value + 1

Convert wavefront map to SLRT string

While goal is not reached by SLRT command

Start at value 98 in wavefront map

Determine direction of lowest node value in surrounding nodes

If it matches the way the robot is facing, add no value to SLRT string

If it is to the left add L to SLRT string

If it is to the right add R to SLRT string

Update robot direction

Input SLRT string into Topology path planning function

A.3 Metric Path Planning and Following on Topological Map Design Plan:

Function input is a 4x4 Matrix that is the topological map, and an integer representing the starting direction

Convert 4x4 topological map to 12x12 occupancy grid

Each node in the 4x4 map gets converted to a 3x3 part of the 12x12 matrix

Each of the 16 different layout options within the topological map gets converted to a 3x3 occupancy grid

Put the 12x12 occupancy grid into the Metric Path Planning on Occupancy Grid function

A.4 Mapping and Navigation Code:

//include all necessary libraries

#include <Arduino.h>       //include for PlatformIO Ide

#include <AccelStepper.h>  //include the stepper motor library

#include <MultiStepper.h>  //include multiple stepper motor library

#include <RPC.h>

#include <List.hpp>

#include <BasicLinearAlgebra.h>

using namespace BLA;

// Create lists for moving averages

#define SONAR\_ARR\_SIZE 6

int\* frontLidarArr = new int[6];

int\* backLidarArr = new int[6];

int\* leftLidarArr = new int[6];

int\* rightLidarArr = new int[6];

int\* leftSonarArr = new int[SONAR\_ARR\_SIZE];

int\* rightSonarArr = new int[SONAR\_ARR\_SIZE];

// Bool to determine whether to count encoder ticks

bool countTicksL = true;

bool countTicksR = false;

//state LEDs connections

#define redLED 5            //red LED for displaying states

#define grnLED 6            //green LED for displaying states

#define ylwLED 7            //yellow LED for displaying states

#define enableLED 13        //stepper enabled LED

int leds[3] = { 5, 6, 7 };  //array of LED pin numbers

//define motor pin numbers

#define stepperEnable 48  //stepper enable pin on stepStick

#define rtStepPin 50      //right stepper motor step pin

#define rtDirPin 51       // right stepper motor direction pin

#define ltStepPin 52      //left stepper motor step pin

#define ltDirPin 53       //left stepper motor direction pin

//define the Lidar constants

#define LIDAR\_FRONT 0

#define LIDAR\_BACK 1

#define LIDAR\_LEFT 2

#define LIDAR\_RIGHT 3

#define numOfSens 4

//define the behavior constants

#define NO\_BEHAVIOR 0

#define COLLIDE 1

//define the Lidar variables

int16\_t ft\_lidar = 8;

int16\_t bk\_lidar = 9;

int16\_t lt\_lidar = 10;

int16\_t rt\_lidar = 11;

int16\_t lidar\_pins[numOfSens] = {8,9,10,11};

int16\_t lidarDist[numOfSens] = {0,0,0,0};

//define the Sonar constants

#define VELOCITY\_TEMP(temp) ((331.5 + 0.6 \* (float)(temp)) \* 100 / 1000000.0)  // The ultrasonic velocity (cm/us) compensated by temperature

#define SONAR\_RIGHT 0

#define SONAR\_LEFT 1

//define the Sonar variables

int16\_t rt\_trigechoPin = 3;

int16\_t lt\_trigechoPin = 4;

int16\_t trig\_EchoPin[2] = { 3,4 };

int16\_t sonarDist[2] = {0,0};

AccelStepper stepperRight(AccelStepper::DRIVER, rtStepPin, rtDirPin);  //create instance of right stepper motor object (2 driver pins, low to high transition step pin 52, direction input pin 53 (high means forward)

AccelStepper stepperLeft(AccelStepper::DRIVER, ltStepPin, ltDirPin);   //create instance of left stepper motor object (2 driver pins, step pin 50, direction input pin 51)

MultiStepper steppers;                                                 //create instance to control multiple steppers at the same time

#define stepperEnTrue false  //variable for enabling stepper motor

#define stepperEnFalse true  //variable for disabling stepper motor

int pauseTime = 2500;  //time before robot moves

int stepTime = 500;    //delay time between high and low on step pin

int wait\_time = 1000;  //delay for printing data

#define WANDER\_TIME 4000 //time between change of wander wheel speeds in millis

int wanderTimer = 0; //timer to determine when to change wander wheel speeds

//define encoder pins

#define LEFT 0                        //left encoder

#define RIGHT 1                       //right encoder

const int ltEncoder = 18;             //left encoder pin (Mega Interrupt pins 2,3 18,19,20,21)

const int rtEncoder = 19;             //right encoder pin (Mega Interrupt pins 2,3 18,19,20,21)

volatile long encoder[2] = { 0, 0 };  //interrupt variable to hold number of encoder counts (left, right)

int lastSpeed[2] = { 0, 0 };          //variable to hold encoder speed (left, right)

int accumTicks[2] = { 0, 0 };         //variable to hold accumulated ticks since last reset

bool run = false;

struct sensor\_data {

  // this can easily be extended to contain sonar data as well

  int lidar\_front;

  int lidar\_back;

  int lidar\_left;

  int lidar\_right;

  int sonar\_left;

  int sonar\_right;

  int photoresistor\_left;

  int photoresistor\_right;

  // this defines some helper functions that allow RPC to send our struct (I found this on a random forum)

  MSGPACK\_DEFINE\_ARRAY(lidar\_front, lidar\_back, lidar\_left, lidar\_right, sonar\_left, sonar\_right, photoresistor\_left, photoresistor\_right)

} sensors;

// read\_lidars is the function used to get lidar data to the M7

struct sensor\_data read\_sensors() {

  return sensors;

}

// reads a lidar given a pin

int read\_lidar(int pin) {

  int16\_t t = pulseIn(pin, HIGH);

  int d; //distance to  object

  if (t == 0){

    // pulseIn() did not detect the start of a pulse within 1 second.

    //Serial.println("timeout");

    d = 100000; //no object detected

  }

  else if (t > 1850)  {

    //Serial.println("timeout");

    d = 100000; //no object detected

  }

  else  {

    // Valid pulse width reading. Convert pulse width in microseconds to distance in millimeters.

    d = (t - 1000) \* 3 / 40;

    // Limit minimum distance to 0.

    if (d < 0) { d = 0; }

  }

  //   Serial.print(d);

  // Serial.print(" cm, ");

  return d;

}

int movingAverage(int arr[], int arrSize) {

  int sum = 0;

  for (int i = 0; i < arrSize; i++) {

    sum += arr[i];

  }

  return sum / arrSize;

}

int\* shiftArray(int arr[], int arrSize, int newValue) {

  for (int i = arrSize - 1; i > 0; i--) {

    arr[i] = arr[i - 1];

  }

  arr[0] = newValue;

  return arr;

}

void setupM4() {

  // bind a method to return the lidar data all at once

  RPC.bind("read\_sensors", read\_sensors);

}

void loopM4() {

  // update the struct with current lidar data

  struct sensor\_data data;

  float lidarFrontCurr = read\_lidar(8);

  float lidarBackCurr = read\_lidar(9);

  float lidarLeftCurr = read\_lidar(10);

  float lidarRightCurr = read\_lidar(11);

  frontLidarArr = shiftArray(frontLidarArr, 6, lidarFrontCurr);

  backLidarArr = shiftArray(backLidarArr, 6, lidarBackCurr);

  leftLidarArr = shiftArray(leftLidarArr, 6, lidarLeftCurr);

  rightLidarArr = shiftArray(rightLidarArr, 6, lidarRightCurr);

  data.lidar\_front = movingAverage(frontLidarArr, 6);

  data.lidar\_back = movingAverage(backLidarArr, 6);

  data.lidar\_left = movingAverage(leftLidarArr, 6);

  data.lidar\_right = movingAverage(rightLidarArr, 6);

  // float sonarLeftCurr = readSonar(SONAR\_LEFT);

  // float sonarRightCurr = readSonar(SONAR\_RIGHT);

  // leftSonarArr = shiftArray(leftSonarArr, SONAR\_ARR\_SIZE, sonarLeftCurr);

  // rightSonarArr = shiftArray(rightSonarArr, SONAR\_ARR\_SIZE, sonarRightCurr);

  // data.sonar\_left = movingAverage(leftSonarArr, SONAR\_ARR\_SIZE);

  // data.sonar\_right = movingAverage(rightSonarArr, SONAR\_ARR\_SIZE);

  data.photoresistor\_left = analogRead(A0);

  data.photoresistor\_right = analogRead(A1);

  sensors = data;

}

// Helper Functions

//interrupt function to count left encoder tickes

void LwheelSpeed() {

  if (countTicksL) {

    encoder[LEFT]++;  //count the right wheel encoder interrupts

  }

}

//interrupt function to count right encoder ticks

void RwheelSpeed() {

  if (countTicksR) {

    encoder[RIGHT]++;  //count the right wheel encoder interrupts

  }

}

void allOFF() {

  for (int i = 0; i < 3; i++) {

    digitalWrite(leds[i], LOW);

  }

}

//function to set all stepper motor variables, outputs and LEDs

void init\_stepper() {

  pinMode(rtStepPin, OUTPUT);                   //sets pin as output

  pinMode(rtDirPin, OUTPUT);                    //sets pin as output

  pinMode(ltStepPin, OUTPUT);                   //sets pin as output

  pinMode(ltDirPin, OUTPUT);                    //sets pin as output

  pinMode(stepperEnable, OUTPUT);               //sets pin as output

  digitalWrite(stepperEnable, stepperEnFalse);  //turns off the stepper motor driver

  pinMode(enableLED, OUTPUT);                   //set enable LED as output

  digitalWrite(enableLED, LOW);                 //turn off enable LED

  pinMode(redLED, OUTPUT);                      //set red LED as output

  pinMode(grnLED, OUTPUT);                      //set green LED as output

  pinMode(ylwLED, OUTPUT);                      //set yellow LED as output

  digitalWrite(redLED, HIGH);                   //turn on red LED

  digitalWrite(ylwLED, HIGH);                   //turn on yellow LED

  digitalWrite(grnLED, HIGH);                   //turn on green LED

  delay(pauseTime / 5);                         //wait 0.5 seconds

  digitalWrite(redLED, LOW);                    //turn off red LED

  digitalWrite(ylwLED, LOW);                    //turn off yellow LED

  digitalWrite(grnLED, LOW);                    //turn off green LED

  stepperRight.setMaxSpeed(1000);              //set the maximum permitted speed limited by processor and clock speed, no greater than 4000 steps/sec on Arduino

  stepperRight.setAcceleration(500);          //set desired acceleration in steps/s^2

  stepperLeft.setMaxSpeed(1000);               //set the maximum permitted speed limited by processor and clock speed, no greater than 4000 steps/sec on Arduino

  stepperLeft.setAcceleration(500);           //set desired acceleration in steps/s^2

  steppers.addStepper(stepperRight);           //add right motor to MultiStepper

  steppers.addStepper(stepperLeft);            //add left motor to MultiStepper

  digitalWrite(stepperEnable, stepperEnTrue);  //turns on the stepper motor driver

  digitalWrite(enableLED, HIGH);               //turn on enable LED

}

//function prints encoder data to serial monitor

void print\_encoder\_data() {

  static unsigned long timer = 0;                            //print manager timer

  if (millis() - timer > 100) {                              //print encoder data every 100 ms or so

    lastSpeed[LEFT] = encoder[LEFT];                         //record the latest left speed value

    lastSpeed[RIGHT] = encoder[RIGHT];                       //record the latest right speed value

    accumTicks[LEFT] = accumTicks[LEFT] + encoder[LEFT];     //record accumulated left ticks

    accumTicks[RIGHT] = accumTicks[RIGHT] + encoder[RIGHT];  //record accumulated right ticks

    Serial.println("Encoder value:");

    Serial.print("\tLeft:\t");

    Serial.print(encoder[LEFT]);

    Serial.print("\tRight:\t");

    Serial.println(encoder[RIGHT]);

    Serial.println("Accumulated Ticks: ");

    Serial.print("\tLeft:\t");

    Serial.print(accumTicks[LEFT]);

    Serial.print("\tRight:\t");

    Serial.println(accumTicks[RIGHT]);

    encoder[LEFT] = 0;   //clear the left encoder data buffer

    encoder[RIGHT] = 0;  //clear the right encoder data buffer

    timer = millis();    //record current time since program started

  }

}

/\*function to run both wheels to a position at speed\*/

void runAtSpeedToPosition() {

  while (stepperRight.distanceToGo() != 0 || stepperLeft.distanceToGo() != 0) {

    stepperRight.runSpeedToPosition();

    stepperLeft.runSpeedToPosition();

  }

}

/\*function to run both wheels continuously at a speed\*/

void runAtSpeed() {

  while (stepperRight.runSpeed() || stepperLeft.runSpeed()) {}

}

/\*This function, runToStop(), will run the robot until the target is achieved and

   then stop it

\*/

void runToStop() {

  int runNow = 1;

  int rightStopped = 0;

  int leftStopped = 0;

  while (runNow) {

    if (!stepperRight.run()) {

      rightStopped = 1;

      stepperRight.stop();  //stop right motor

    }

    if (!stepperLeft.run()) {

      leftStopped = 1;

      stepperLeft.stop();  //stop ledt motor

    }

    if (rightStopped && leftStopped) {

      runNow = 0;

    }

  }

}

/\*

  INSERT DESCRIPTION HERE, what are the inputs, what does it do, functions used

\*/

void spin(int angle, int dir) {

  int steps = angle \* 5.585;

  if (dir) {

    stepperLeft.move(steps);    //move one full rotation forward relative to current position

    stepperRight.move(-steps);  //move one full rotation forward relative to current position

  } else {

    stepperRight.move(steps);  //move one full rotation forward relative to current position

    stepperLeft.move(-steps);  //move one full rotation forward relative to current position

  }

  runToStop();  //run until the robot reaches the target

}

/\*

  INSERT DESCRIPTION HERE, what are the inputs, what does it do, functions used

\*/

void pivot(int angle, int dir) {

  int steps = angle \* 5.585 \* 2;

  if (dir) {

    stepperLeft.move(steps);  //move steps

  } else {

    stepperRight.move(steps);

  }

  runToStop();  //run until the robot reaches the target

}

/\*

  INSERT DESCRIPTION HERE, what are the inputs, what does it do, functions used

\*/

void turn(int time, int dir) {

  int steps = time \* 500;

  if (dir) {

    stepperLeft.setMaxSpeed(500);

    stepperRight.setMaxSpeed(250);

    stepperLeft.move(steps);       //move one full rotation forward relative to current position

    stepperRight.move(steps / 2);  //move one full rotation forward relative to current position

  } else {

    stepperRight.setMaxSpeed(500);

    stepperLeft.setMaxSpeed(250);

    stepperRight.move(steps);     //move one full rotation forward relative to current position

    stepperLeft.move(steps / 2);  //move one full rotation forward relative to current position

  }

  runToStop();  //run until the robot reaches the target

  stepperRight.setMaxSpeed(1000);

  stepperLeft.setMaxSpeed(1000);

  init\_stepper();

}

/\*

  INSERT DESCRIPTION HERE, what are the inputs, what does it do, functions used

\*/

void forward(int steps) {

  // int steps = distance / 0.034375; // for distance in cm

  stepperRight.move(steps);  //move steps forward relative to current position

  stepperLeft.move(steps);   //move steps forward relative to current position

  runToStop();               //run until the robot reaches the target

}

/\*

  INSERT DESCRIPTION HERE, what are the inputs, what does it do, functions used

\*/

void reverse(int distance) {

  int steps = distance / 0.034375;

  stepperRight.move(-steps);  //move one full rotation reverse relative to current position

  stepperLeft.move(-steps);   //move one full rotation reverse relative to current position

  runToStop();                //run until the robot reaches the target

}

/\*

  INSERT DESCRIPTION HERE, what are the inputs, what does it do, functions used

\*/

void stop() {

  stepperRight.setSpeed(0);  //set right motor speed

  stepperLeft.setSpeed(0);   //set left motor speed

}

//this function will read the left or right sensor based upon input value

uint16\_t readSonar(uint16\_t side) {

  uint16\_t distance;

  uint32\_t pulseWidthUs;

  int16\_t dist, temp, dist\_in;

  pinMode(trig\_EchoPin[side], OUTPUT);

  digitalWrite(trig\_EchoPin[side], LOW);

  digitalWrite(trig\_EchoPin[side], HIGH);  //Set the trig pin High

  delayMicroseconds(10);               //Delay of 10 microseconds

  digitalWrite(trig\_EchoPin[side], LOW);   //Set the trig pin Low

  pinMode(trig\_EchoPin[side], INPUT);                //Set the pin to input mode

  pulseWidthUs = pulseIn(trig\_EchoPin[side], HIGH);  //Detect the high level time on the echo pin, the output high level time represents the ultrasonic flight time (unit: us)

  distance = pulseWidthUs \* VELOCITY\_TEMP(20) / 2.0;  //The distance can be calculated according to the flight time of ultrasonic wave,/

                                                      //and the ultrasonic sound speed can be compensated according to the actual ambient temperature

  dist\_in = 0.394\*distance;    //convert cm to inches

  // Serial.print(dist\_in, DEC);   //print inches

  // Serial.print(" inches ");

  // Serial.print(distance, DEC);  //print cm

  // Serial.println(" cm");

  return distance;

}

/\*

goToAngle rotates the robot to a specified angle

\*/

void goToAngle(int angle) {

  //A wheel travels 27.5cm per revolution

  //A wheel travels 69.1cm per 360 spin

  //There are 800 steps per wheel revolution (quarter stepping)

  //69.1/27.5\*800 = 2010.6 steps per 360 spin

  digitalWrite(grnLED, HIGH);   //turn on green LED

  if (angle == 0) {

    return;

  }

  countTicksL = true;

  countTicksR = true;

  int eCounts = abs(angle / 3.45);

  int speed = 100;

  if (angle < 0) {

    stepperLeft.setSpeed(speed);  //set left motor speed

    stepperRight.setSpeed(-speed);  //set right motor speed

    Serial.println("neg");

  } else {

    stepperLeft.setSpeed(-speed);  //set left motor speed

    stepperRight.setSpeed(speed);  //set right motor speed

    Serial.println("pos");

  }

  while (encoder[RIGHT] - eCounts <= 0 || encoder[LEFT] - eCounts <= 0) {

    stepperRight.runSpeed();

    stepperLeft.runSpeed();

    // Serial.print("Right Encoder: ");

    // Serial.print(encoder[RIGHT]);

    // Serial.print(" ");

    // Serial.print("Left Encoder: ");

    // Serial.println(encoder[LEFT]);

  }

  encoder[RIGHT] = 0;

  encoder[LEFT] = 0;

  digitalWrite(grnLED, LOW);       //turn off green LED

}

/\*

randomWander spins the robot to a random angle then moves it a random amount of

steps forward

\*/

void randomWander() {

  digitalWrite(grnLED, HIGH);      //turn on green LED

    stepperRight.setSpeed(-300);  //set right motor speed

    stepperLeft.setSpeed(-300);   //set left motor speed

  if (millis() - wanderTimer > WANDER\_TIME) {

    spin(random(30, 180), random(0,2));

    wanderTimer = millis();

  }

  runAtSpeed();

  // int angle = random(20, 180);

  // int dir = random(0,2);

  // spin(angle, dir);

  // int distance = random(2000);

  // forward(distance);

}

/\*

collide stops the robot when an object is in front of it

\*/

void collide(void) {

  stepperRight.setSpeed(500);  //set right motor speed

  stepperLeft.setSpeed(500);   //set left motor speed

  sensors = RPC.call("read\_sensors").as<struct sensor\_data>();

  run = true;

  if (sensors.lidar\_front <= 15 || sensors.lidar\_back <= 15 || sensors.lidar\_left <= 15 || sensors.lidar\_right <= 15) {

    run = false;

    digitalWrite(redLED, HIGH);       //turn on red LED

  }

  if (run) {

    runAtSpeed();

    digitalWrite(redLED, LOW);       //turn off red LED

    // Serial.println("run");

  }

}

/\*

runaway avoids all obstacles around the robot

\*/

void runaway(void) {

  int maxSpeed = 300;

  int rightSpeed;

  int leftSpeed;

  int x;

  int y;

  sensors = RPC.call("read\_sensors").as<struct sensor\_data>();

  // Serial.print("left = ");

  // Serial.print(sensors.sonar\_left);

  // Serial.print(" right = ");

  // Serial.println(sensors.sonar\_right);

  if (abs(sensors.lidar\_back) < 30 && abs(sensors.lidar\_front) < 30) {

    x = sensors.lidar\_front - sensors.lidar\_back; // x direction of repulsive vector

  } else if (abs(sensors.lidar\_back) < 30) {

    x = 30 - sensors.lidar\_back; // x direction of repulsive vector

  } else if (abs(sensors.lidar\_front) < 30) {

    x = -30 + sensors.lidar\_front; // x direction of repulsive vector

  } else {

    x = 0;

  }

  if (abs(sensors.lidar\_left) < 30 && abs(sensors.lidar\_right) < 30) {

    y = sensors.lidar\_left - sensors.lidar\_right; // x direction of repulsive vector

  } else if (abs(sensors.lidar\_right) < 30) {

    y = 30 - sensors.lidar\_right; // x direction of repulsive vector

  } else if (abs(sensors.lidar\_left) < 30) {

    y = -30 + sensors.lidar\_left; // x direction of repulsive vector

  } else {

    y = 0;

  }

  int angle = atan2(y,x) \* 180 / 3.1415;

  Serial.print("x = ");

  Serial.print(x);

  Serial.print(" y = ");

  Serial.print(y);

  Serial.print(" angle = ");

  Serial.println(angle);

  if (abs(x) > 10 || abs (y) > 10) {

    digitalWrite(ylwLED, HIGH);       //turn on yellow LED

    if (angle > -45 && angle <= 45) {

      rightSpeed = maxSpeed;

      leftSpeed = maxSpeed;

    } else if ((angle > 45 && angle <= 90) || (angle > -135 && angle < -90)) {

      rightSpeed = maxSpeed;

      leftSpeed = -maxSpeed/2;

    } else if ((angle >= -90 && angle <= -45) || (angle > 90 && angle <= 135)) {

      rightSpeed = -maxSpeed/2;

      leftSpeed = maxSpeed;

    } else {

      rightSpeed = -maxSpeed;

      leftSpeed = -maxSpeed;

    }

  } else if (sensors.lidar\_left > 0 && sensors.lidar\_left < 30 && sensors.lidar\_right > 0 && sensors.lidar\_right < 30 && abs(x) < 4 ) {

    digitalWrite(ylwLED, HIGH);       //turn on yellow LED

    rightSpeed = maxSpeed;

    leftSpeed = maxSpeed;

  } else if (sensors.lidar\_front > 0 && sensors.lidar\_front < 30 && sensors.lidar\_back > 0 && sensors.lidar\_back < 30 &&  sensors.lidar\_left > 30 && sensors.lidar\_right > 30) {

    digitalWrite(ylwLED, HIGH);       //turn on yellow LED

    spin(90, 0);

  } else {

    digitalWrite(ylwLED, LOW);       //turn off yellow LED

    rightSpeed = 0;

    leftSpeed = 0;

  }

  // if (abs(x) > 10  || abs (y) > 10) {

  //   if (angle <= 90 && angle >= -90) {

  //     rightSpeed = maxSpeed \* abs((angle + 90)) / 180;

  //     leftSpeed = maxSpeed \* abs((angle - 90)) / 180;

  //   } else {

  //     rightSpeed = -maxSpeed \* abs((angle + 90)) / 180;

  //     leftSpeed = -maxSpeed \* abs((angle - 90)) / 180;

  //   }

  // }

  // float mag = 200;

  // if(angle < 0) {

  // mag \*= -1;

  // angle += 180;

  // }

  // float left\_power = mag \* max(-1, 1 - angle/45);

  // float right\_power = mag \* min(1, 3 - angle/45);

  stepperRight.setSpeed(rightSpeed);  //set right motor speed

  stepperLeft.setSpeed(leftSpeed);   //set left motor speed

  runAtSpeed();

}

/\*

follow follows an object that is in front of the robot

\*/

void follow(void) {

  digitalWrite(redLED, HIGH);       //turn on red LED

  digitalWrite(grnLED, HIGH);       //turn on green LED

  int maxSpeed = 300;

  int rightSpeed;

  int leftSpeed;

  int x = 0;

  int y = 0;

  sensors = RPC.call("read\_sensors").as<struct sensor\_data>();

  // Serial.print("left = ");

  // Serial.print(sensors.sonar\_left);

  // Serial.print(" right = ");

  // Serial.println(sensors.sonar\_right);

  // Determine x direction of attractive vector

  if (sensors.lidar\_back < 30){

    x += -30 + sensors.lidar\_back;

  }

  if (sensors.lidar\_front < 30){

    x += 30 - sensors.lidar\_front;

  }

  if (sensors.sonar\_left < 15) {

    x += 15 - sensors.sonar\_left;

  }

  if (sensors.sonar\_right < 15) {

    x += 15 - sensors.sonar\_right;

  }

  // Determine y direction of attractive vector

  if (sensors.lidar\_right < 30){

    y += -30 + sensors.lidar\_right;

  }

  if (sensors.lidar\_left < 30){

    y += 30 - sensors.lidar\_left;

  }

  if (sensors.sonar\_left < 15) {

    y += 15 - sensors.sonar\_left;

  }

  if (sensors.sonar\_right < 15) {

    y += -15 + sensors.sonar\_right;

  }

  int angle = atan2(y,x) \* 180 / 3.1415;

  Serial.print("x = ");

  Serial.print(x);

  Serial.print(" y = ");

  Serial.print(y);

  Serial.print(" angle = ");

  Serial.println(angle);

  if(abs(y) > 5 || abs(x) > 5) {

    if (angle > -30 && angle < 30 && abs(x) < 25 ) {

      rightSpeed = maxSpeed;

      leftSpeed = maxSpeed;

      Serial.println("Forward");

    } else if (angle > -30 && angle < 30 && abs(x) > 35 ) {

      rightSpeed = -maxSpeed;

      leftSpeed = -maxSpeed;

      Serial.println("Backward");

    } else if (angle >= 30 && angle <= 180) {

      rightSpeed = maxSpeed;

      leftSpeed = -maxSpeed;

      Serial.println("Left");

    } else if (angle <= -30 && angle >= -180) {

      rightSpeed = -maxSpeed;

      leftSpeed = maxSpeed;

      Serial.println("Right");

    } else {

      rightSpeed = 0;

      leftSpeed = 0;

    }

  } else {

    rightSpeed = 0;

    leftSpeed = 0;

  }

  stepperRight.setSpeed(rightSpeed);  //set right motor speed

  stepperLeft.setSpeed(leftSpeed);   //set left motor speed

  runAtSpeed();

}

/\*

smartWander randomly wanders the robot while avoiding obstacles

\*/

#define STATE\_WANDER 0

#define STATE\_COLLIDE 1

#define STATE\_RUNAWAY 2

int state = 0;

void smartWander(void) {

  sensors = RPC.call("read\_sensors").as<struct sensor\_data>();

  switch (state) {

    case STATE\_WANDER:

      digitalWrite(ylwLED, LOW);       //turn off yellow LED

      Serial.println("wander");

      randomWander();

      if (sensors.lidar\_front < 15 || sensors.lidar\_back < 15 || sensors.lidar\_right < 15 || sensors.lidar\_left < 15) {

        state = STATE\_COLLIDE;

      }

      break;

    case STATE\_COLLIDE:

      digitalWrite(grnLED, LOW);       //turn off green LED

      Serial.println("collide");

      collide();

      delay(1000);

      state = STATE\_RUNAWAY;

      break;

    case STATE\_RUNAWAY:

      digitalWrite(redLED, LOW);       //turn off red LED

      Serial.println("runaway");

      runaway();

      if (sensors.lidar\_front > 20 && sensors.lidar\_back > 20 && sensors.lidar\_right > 20 && sensors.lidar\_left > 20) {

        state = STATE\_WANDER;

      }

      break;

    default:

      Serial.println("left state machine");

      break;

  }

}

/\*

smartFollow follows an object that is in front of the robot

\*/

#define STATE\_FOLLOW 3

void smartFollow(void) {

  sensors = RPC.call("read\_sensors").as<struct sensor\_data>();

  switch (state) {

    case STATE\_WANDER:

      digitalWrite(redLED, LOW);       //turn off yellow LED

      digitalWrite(grnLED, LOW);       //turn off yellow LED

      Serial.println("wander");

      randomWander();

      if (sensors.lidar\_front < 15 || sensors.lidar\_back < 15 || sensors.lidar\_right < 15 || sensors.lidar\_left < 15) {

        state = STATE\_COLLIDE;

      }

      break;

    case STATE\_COLLIDE:

      digitalWrite(grnLED, LOW);       //turn off green LED

      Serial.println("collide");

      collide();

      delay(1000);

      state = STATE\_FOLLOW;

      break;

    case STATE\_FOLLOW:

      digitalWrite(redLED, LOW);       //turn off red LED

      Serial.println("follow");

      follow();

      if (sensors.lidar\_front > 20 && sensors.lidar\_back > 20 && sensors.lidar\_right > 20 && sensors.lidar\_left > 20 && sensors.sonar\_left > 20 && sensors.sonar\_left > 20) {

        state = STATE\_WANDER;

      }

      break;

    default:

      Serial.println("left state machine");

      break;

  }

}

/\*

wallFollowBB implements bang bang control in order to follow a wall

\*/

#define NO\_WALL 0

#define LEFT\_WALL 1

#define RIGHT\_WALL 2

#define CENTER\_WALL 3

#define LOST\_WALL 4

#define RANDOM\_WANDER 5

#define BACK\_WALL 6

void wallFollowBB(void) {

  int maxSpeed = 300;

  int rightSpeed;

  int leftSpeed;

  int x = 0;

  int y = 0;

  sensors = RPC.call("read\_sensors").as<struct sensor\_data>();

  Serial.print("left = ");

  Serial.print(sensors.lidar\_left);

  Serial.print(" right = ");

  Serial.println(sensors.lidar\_right);

  if (sensors.lidar\_left < 30 && sensors.lidar\_right < 30) {

    state = CENTER\_WALL;

  } else if (sensors.lidar\_left < 30) {

    state = LEFT\_WALL;

  } else if (sensors.lidar\_right < 30) {

    state = RIGHT\_WALL;

  }

  switch(state) {

    case NO\_WALL:

      rightSpeed = 0;

      leftSpeed = 0;

      break;

    case LEFT\_WALL:

      if (sensors.lidar\_left >= 10 && sensors.lidar\_left <= 15){

        digitalWrite(redLED, LOW);       //turn off red LED

        digitalWrite(ylwLED, LOW);       //turn off yellow LED

        rightSpeed = maxSpeed;

        leftSpeed = maxSpeed;

      } else if (sensors.lidar\_left <= 10) {

        digitalWrite(ylwLED, HIGH);       //turn on yellow LED

        rightSpeed = maxSpeed/1.5;

        leftSpeed = maxSpeed;

      } else {

        digitalWrite(redLED, HIGH);       //turn on red LED

        rightSpeed = maxSpeed;

        leftSpeed = maxSpeed/1.5;

      }

      break;

    case RIGHT\_WALL:

      if (sensors.lidar\_right >= 10 && sensors.lidar\_right <= 15){

        digitalWrite(redLED, LOW);       //turn off red LED

        digitalWrite(ylwLED, LOW);       //turn off yellow LED

        rightSpeed = maxSpeed;

        leftSpeed = maxSpeed;

      } else if (sensors.lidar\_right < 10) {

        digitalWrite(ylwLED, HIGH);       //turn on yellow LED

        rightSpeed = maxSpeed;

        leftSpeed = maxSpeed/1.5;

      } else {

        digitalWrite(redLED, HIGH);       //turn on red LED

        rightSpeed = maxSpeed/1.5;

        leftSpeed = maxSpeed;

      }

      break;

    case CENTER\_WALL:

      y = sensors.lidar\_left - sensors.lidar\_right;

      if (y >= -3 && y <= 3) {

        digitalWrite(redLED, LOW);       //turn off red LED

        digitalWrite(ylwLED, LOW);       //turn off yellow LED

        rightSpeed = maxSpeed;

        leftSpeed = maxSpeed;

      } else if (y > 3) {

        rightSpeed = maxSpeed;

        leftSpeed = maxSpeed/2;

      } else {

        rightSpeed = maxSpeed/2;

        leftSpeed = maxSpeed;

      }

      break;

    case RANDOM\_WANDER:

      randomWander();

      break;

  }

  stepperRight.setSpeed(rightSpeed);  //set right motor speed

  stepperLeft.setSpeed(leftSpeed);   //set left motor speed

  runAtSpeed();

}

/\*

wallFollowP implements proportional control in order to follow a wall

\*/

float prop = 0;

void wallFollowP(void) {

  int maxSpeed = 200;

  int frontTurnDist = 15;

  int rightSpeed;

  int leftSpeed;

  int x = 0;

  int y = 0;

  int error = 0;

  float kp = 3;

  sensors = RPC.call("read\_sensors").as<struct sensor\_data>();

  // Serial.print("Sonar left = ");

  // Serial.print(sensors.sonar\_left);

  // Serial.print("Sonar right = ");

  // Serial.println(sensors.sonar\_right);

  if (sensors.lidar\_left < 30 && sensors.lidar\_right < 30) {

    state = CENTER\_WALL;

  } else if (sensors.lidar\_left < 30) {

    state = LEFT\_WALL;

  } else if (sensors.lidar\_right < 30) {

    state = RIGHT\_WALL;

  }

  switch(state) {

    case NO\_WALL:

      rightSpeed = 0;

      leftSpeed = 0;

      break;

    case LEFT\_WALL:

      if (sensors.lidar\_left >= 10 && sensors.lidar\_left <= 15){

        digitalWrite(redLED, LOW);       //turn off red LED

        digitalWrite(ylwLED, LOW);       //turn off yellow LED

      } else if (sensors.lidar\_left <= 10) {

        digitalWrite(ylwLED, HIGH);      //turn on yellow LED

      } else {

        digitalWrite(redLED, HIGH);      //turn on red LED

      }

      error = min(sensors.lidar\_left - 12.5, 12);

      prop = kp \* error;

      rightSpeed = maxSpeed + prop;

      leftSpeed = maxSpeed - prop;

      break;

    case RIGHT\_WALL:

      if (sensors.lidar\_right >= 10 && sensors.lidar\_right <= 15){

        digitalWrite(redLED, LOW);       //turn off red LED

        digitalWrite(ylwLED, LOW);       //turn off yellow LED

      } else if (sensors.lidar\_right < 10) {

        digitalWrite(ylwLED, HIGH);       //turn on yellow LED

      } else {

        digitalWrite(redLED, HIGH);       //turn on red LED

      }

      error = min(sensors.lidar\_right - 12.5, 12);

      prop = kp \* error;

      rightSpeed = maxSpeed - prop;

      leftSpeed = maxSpeed + prop;

      break;

    case CENTER\_WALL:

      y = sensors.lidar\_left - sensors.lidar\_right;

      if (y >= -3 && y <= 3) {

        digitalWrite(redLED, HIGH);       //turn on red LED

        digitalWrite(ylwLED, HIGH);       //turn on yellow LED

        digitalWrite(grnLED, HIGH);       //turn on green LED

      } else {

        digitalWrite(redLED, LOW);       //turn off red LED

        digitalWrite(ylwLED, LOW);       //turn off yellow LED

        digitalWrite(grnLED, LOW);       //turn off green LED

      }

      error = min(y, 12);

      prop = kp \* error;

      rightSpeed = maxSpeed + prop;

      leftSpeed = maxSpeed - prop;

      break;

  }

  if (sensors.lidar\_front < 15) {

    if (state == LEFT\_WALL) {

      collide();

      delay(1000);

      spin(90, 1);

    } else {

      collide();

      delay(1000);

      spin(90, 0);

    }

  }

  Serial.print("left = ");

  Serial.print(leftSpeed);

  Serial.print(" right = ");

  Serial.println(rightSpeed);

  stepperRight.setSpeed(rightSpeed);  //set right motor speed

  stepperLeft.setSpeed(leftSpeed);   //set left motor speed

  stepperRight.runSpeed();

  stepperLeft.runSpeed();

}

/\*

wallFollowPD implements proportional/derivative control in order to follow a wall

\*/

float pd = 0;

float lastError = 0;

bool loved = false;

void wallFollowPD(void) {

  int maxSpeed = -300;

  int frontTurnDist = 10;

  int rightSpeed;

  int leftSpeed;

  float x = 0;

  float y = 0;

  float error = 0;

  float kp = 10;

  float kd = 1;

  float kp\_back = 200;

  sensors = RPC.call("read\_sensors").as<struct sensor\_data>();

  if (!loved) {

    if (sensors.lidar\_left < 20 && sensors.lidar\_right < 20) {

      state = CENTER\_WALL;

    } else if (sensors.lidar\_left < 30) {

      state = LEFT\_WALL;

    } else if (sensors.lidar\_right < 30) {

      state = RIGHT\_WALL;

    }

  }

  lightState(state, sensors);

  switch(state) {

    case NO\_WALL:

      rightSpeed = 0;

      leftSpeed = 0;

      break;

    case LEFT\_WALL:

      error = min(sensors.lidar\_left - 12.5, 12);

      pd = kp \* error + kd \* (error - lastError);

      if (sensors.lidar\_back <= frontTurnDist) {

        pd -= kp\_back \* (frontTurnDist - sensors.lidar\_back);

      }

      rightSpeed = maxSpeed - pd;

      leftSpeed = maxSpeed + pd;

      break;

    case RIGHT\_WALL:

      error = min(sensors.lidar\_right - 12.5, 12);

      pd = kp \* error + kd \* (error - lastError);

      if (sensors.lidar\_back <= frontTurnDist) {

        pd -= kp\_back \* (frontTurnDist - sensors.lidar\_back);

      }

      rightSpeed = maxSpeed + pd;

      leftSpeed = maxSpeed - pd;

      break;

    case CENTER\_WALL:

      error = sensors.lidar\_left - sensors.lidar\_right;

      //Serial.print("error = ");

      //Serial.print(error);

      if (abs(error) <= 1) {

        pd = 0;

      } else {

        pd = kp \* error + kd \* (error - lastError);

      }

      rightSpeed = maxSpeed - pd;

      leftSpeed = maxSpeed + pd;

      break;

  }

  // Serial.print("Sonar left = ");

  // Serial.print(sensors.sonar\_left);

  // Serial.print("back = ");

  // Serial.print(sensors.lidar\_back);

  // Serial.print("left = ");

  // Serial.print(leftSpeed);

  // Serial.print(" right = ");

  // Serial.println(rightSpeed);

  stepperRight.setSpeed(rightSpeed);  //set right motor speed

  stepperLeft.setSpeed(leftSpeed);   //set left motor speed

  stepperRight.runSpeed();

  stepperLeft.runSpeed();

  lastError = error;

}

/\*

wallFollowStates implements PD control in order to follow a wall, along with

random wander when all walls are lost, and avoid when the robot gets too

close to a wall

\*/

bool timerStarted = false;

int wallTimer = 0;

void wallFollowStates (void) {

  int maxSpeed = -200;

  int frontTurnDist = 10;

  int rightSpeed;

  int leftSpeed;

  float x = 0;

  float y = 0;

  float error = 0;

  float kp = 20;

  float kd = 1;

  float kp\_back = 200;

  int lostTimer = 0;

  sensors = RPC.call("read\_sensors").as<struct sensor\_data>();

  if (sensors.lidar\_left < 30 && sensors.lidar\_right < 30) {

    state = CENTER\_WALL;

    wallTimer = millis();

  } else if (sensors.lidar\_left < 40) {

    state = LEFT\_WALL;

    wallTimer = millis();

  } else if (sensors.lidar\_right < 40) {

    state = RIGHT\_WALL;

    wallTimer = millis();

  } else {

    if (millis() - 4000 > wallTimer) {

      state = RANDOM\_WANDER;

    }

  }

  lightState(state, sensors);

  switch(state) {

    case NO\_WALL:

      rightSpeed = 0;

      leftSpeed = 0;

      break;

    case LEFT\_WALL:

      error = min(sensors.lidar\_left - 12.5, 12);

      pd = kp \* error + kd \* (error - lastError);

      if (sensors.lidar\_back <= frontTurnDist) {

        pd -= kp\_back \* (frontTurnDist - sensors.lidar\_back);

      }

      rightSpeed = maxSpeed - pd;

      leftSpeed = maxSpeed + pd;

      break;

    case RIGHT\_WALL:

      error = min(sensors.lidar\_right - 12.5, 12);

      pd = kp \* error + kd \* (error - lastError);

      if (sensors.lidar\_back <= frontTurnDist) {

        pd -= kp\_back \* (frontTurnDist - sensors.lidar\_back);

      }

      rightSpeed = maxSpeed + pd;

      leftSpeed = maxSpeed - pd;

      break;

    case CENTER\_WALL:

      error = sensors.lidar\_left - sensors.lidar\_right;

      Serial.print("error = ");

      Serial.print(error);

      if (abs(error) <= 3) {

        pd = 0;

      } else {

        pd = kp \* error + kd \* (error - lastError);

      }

      rightSpeed = maxSpeed - pd;

      leftSpeed = maxSpeed + pd;

      break;

    case RANDOM\_WANDER:

      randomWander();

      if (sensors.lidar\_back < 10) {

        spin(90, 0);

      }

      break;

  }

  // Serial.print("Sonar left = ");

  // Serial.print(sensors.sonar\_left);

  // Serial.print("back = ");

  // Serial.print(sensors.lidar\_back);

  Serial.print("left = ");

  Serial.print(leftSpeed);

  Serial.print(" right = ");

  Serial.println(rightSpeed);

  stepperRight.setSpeed(rightSpeed);  //set right motor speed

  stepperLeft.setSpeed(leftSpeed);   //set left motor speed

  stepperRight.runSpeed();

  stepperLeft.runSpeed();

  lastError = error;

}

/\*

goToGoalAvoidObs goes to a specific goal location while being able to avoid objects in its path

\*/

#define NO\_OBSTACLE 0

#define SIDE\_1 1

#define SIDE\_2 2

#define SIDE\_3 3

#define POST\_OBSTACLE 4

int gtgWall = 0;

int gtgState = NO\_OBSTACLE;

bool hasTurned = false;

void goToGoalAvoidObs(int x, int y) {

  int angle;

  angle = atan2(y, x)\*180/3.1415;

  // Serial.println("Angle: ");

  // Serial.println(angle);

  goToAngle(angle);

  delay(1000);

  digitalWrite(grnLED, LOW);       //turn off green LED

  double distance = sqrt(pow(x,2) + pow(y,2));

  // Serial.println("Dist: ");

  // Serial.println(distance);

  int eCounts = distance / 10.8 \* 40;

  Serial.print("eCount: ");

  Serial.println(eCounts);

  int speed = -300;

  int turnDelay = 4000;

  int changeStateDelay = 10000;

  int turnTimer = 0;

  int obsCount = 0;

  countTicksR = false;

  encoder[LEFT] = 0;

  encoder[RIGHT] = 0;

  while (eCounts - encoder[LEFT] >= 0) {

    sensors = RPC.call("read\_sensors").as<struct sensor\_data>();

    Serial.print("Ecounts Left: ");

    Serial.println(eCounts - encoder[LEFT]);

    if (sensors.lidar\_back < 10 && gtgState == NO\_OBSTACLE){

      gtgState = SIDE\_1;

      hasTurned = false;

    } else if (sensors.lidar\_right > 40 && sensors.lidar\_left > 40 && gtgState == SIDE\_1) {

      gtgState = SIDE\_2;

      turnTimer = millis();

      hasTurned = false;

    } else if (sensors.lidar\_right > 40 && sensors.lidar\_left > 40 && millis() - turnTimer > changeStateDelay && gtgState == SIDE\_2) {

      gtgState = SIDE\_3;

      turnTimer = millis();

      hasTurned = false;

    } else if (obsCount\*2 <= encoder[RIGHT] && gtgState == SIDE\_3) {

      gtgState = POST\_OBSTACLE;

      hasTurned = false;

    }

    if(sensors.lidar\_right < 40) {

      gtgWall = RIGHT\_WALL;

    } else if (sensors.lidar\_left < 40) {

      gtgWall = LEFT\_WALL;

    }

    if (gtgState == NO\_OBSTACLE) {

      Serial.println("State: NO\_OBSTACLE");

      countTicksL == true;

    }

    if (gtgState == SIDE\_1) {

      Serial.println("State: SIDE\_1");

      countTicksL = false;

      if (!hasTurned) {

        if (gtgWall == LEFT\_WALL) {

          spin(90, 0);

        } else {

          spin(90, 1);

        }

        hasTurned = true;

        countTicksR = true;

      }

    }

    if (gtgState == SIDE\_2) {

      Serial.println("State: SIDE\_2");

      obsCount = encoder[RIGHT];

      Serial.print("ObsCount: ");

      Serial.println(obsCount);

      if (!hasTurned && millis() - turnTimer > turnDelay) {

        countTicksR = false;

        if (gtgWall == LEFT\_WALL) {

          spin(90, 1);

        } else {

          spin(90, 0);

        }

        countTicksL = true;

        hasTurned = true;

      }

    }

    if (gtgState == SIDE\_3) {

      Serial.println("State: SIDE\_3");

      if (!hasTurned && millis() - turnTimer > turnDelay) {

        if (gtgWall == LEFT\_WALL) {

          spin(90, 1);

        } else {

          spin(90, 0);

        }

        hasTurned = true;

        countTicksR = true;

        countTicksL = false;

      }

    }

    if (gtgState == POST\_OBSTACLE) {

      Serial.println("State: POST\_OBSTACLE");

      if (!hasTurned) {

        if (gtgWall == LEFT\_WALL) {

          spin(90, 0);

        } else {

          spin(90, 1);

        }

        hasTurned = true;

        countTicksL = true;

      }

    }

    stepperLeft.setSpeed(speed);   //set left motor speed

    stepperRight.setSpeed(speed);  //set right motor speed

    stepperRight.runSpeed();

    stepperLeft.runSpeed();

  }

  encoder[RIGHT] = 0;

  encoder[LEFT] = 0;

}

/\*

lightState updates the leds on the robot

\*/

void lightState(int lightState, struct sensor\_data sensors) {

  switch (lightState) {

    case NO\_WALL:

      digitalWrite(redLED, LOW);       //turn off red LED

      digitalWrite(ylwLED, LOW);       //turn off yellow LED

      digitalWrite(grnLED, LOW);       //turn off green LED

      break;

    case LEFT\_WALL:

      if (sensors.lidar\_left >= 10 && sensors.lidar\_left <= 15){

        digitalWrite(grnLED, HIGH);       //turn on green LED

        digitalWrite(redLED, LOW);       //turn off red LED

        digitalWrite(ylwLED, HIGH);       //turn on yellow LED

      } else if (sensors.lidar\_left <= 10) {

        digitalWrite(ylwLED, HIGH);      //turn on yellow LED

        digitalWrite(grnLED, LOW);       //turn off green LED

        digitalWrite(redLED, LOW);       //turn off red LED

      } else {

        digitalWrite(redLED, HIGH);      //turn on red LED

        digitalWrite(ylwLED, LOW);       //turn off yellow LED

        digitalWrite(grnLED, LOW);       //turn off green LED

      }

      break;

    case RIGHT\_WALL:

      if (sensors.lidar\_right >= 10 && sensors.lidar\_right <= 15){

        digitalWrite(redLED, HIGH);       //turn on red LED

        digitalWrite(ylwLED, HIGH);       //turn on yellow LED

        digitalWrite(grnLED, LOW);       //turn off green LED

      } else if (sensors.lidar\_right < 10) {

        digitalWrite(ylwLED, HIGH);       //turn on yellow LED

        digitalWrite(grnLED, LOW);       //turn off green LED

        digitalWrite(redLED, LOW);       //turn off red LED

      } else {

        digitalWrite(redLED, HIGH);       //turn on red LED

        digitalWrite(ylwLED, LOW);       //turn off yellow LED

        digitalWrite(grnLED, LOW);       //turn off green LED

      }

      break;

    case CENTER\_WALL:

      if (sensors.lidar\_left - sensors.lidar\_right >= -3 && sensors.lidar\_left - sensors.lidar\_right <= 3) {

        digitalWrite(redLED, HIGH);       //turn on red LED

        digitalWrite(ylwLED, HIGH);       //turn on yellow LED

        digitalWrite(grnLED, HIGH);       //turn on green LED

      } else {

        digitalWrite(redLED, LOW);       //turn off red LED

        digitalWrite(ylwLED, LOW);       //turn off yellow LED

        digitalWrite(grnLED, LOW);       //turn off green LED

      }

      break;

      case RANDOM\_WANDER:

        digitalWrite(redLED, LOW);       //turn off red LED

        digitalWrite(ylwLED, LOW);       //turn off yellow LED

        digitalWrite(grnLED, HIGH);       //turn off green LED

        break;

  }

  if (sensors.lidar\_back <= 18) {

    digitalWrite(redLED, HIGH);       //turn on red LED

    digitalWrite(ylwLED, LOW);       //turn off yellow LED

    digitalWrite(grnLED, HIGH);       //turn on green LED

  }

}

/\*

fear uses the photoresistors to turn the robot away from light when sensed

\*/

void fear(void) {

  digitalWrite(redLED, HIGH);       //turn on red LED

  digitalWrite(ylwLED, HIGH);       //turn on yellow LED

  digitalWrite(grnLED, HIGH);       //turn on green LED

  sensors = RPC.call("read\_sensors").as<struct sensor\_data>();

  int rightSpeed = 0;

  int leftSpeed = 0;

  if (sensors.photoresistor\_right > 950) {

    rightSpeed = -4\* (sensors.photoresistor\_right - 950);

  }

  if (sensors.photoresistor\_left > 950) {

    leftSpeed = -4 \* (sensors.photoresistor\_left - 950);

  }

  stepperLeft.setSpeed(leftSpeed);   //set left motor speed

  stepperRight.setSpeed(rightSpeed);  //set right motor speed

  stepperRight.runSpeed();

  stepperLeft.runSpeed();

}

/\*

aggression uses the photoresistors to turn the robot towards the light

\*/

void aggression(void) {

  digitalWrite(redLED, HIGH);       //turn on red LED

  digitalWrite(ylwLED, HIGH);       //turn off yellow LED

  digitalWrite(grnLED, LOW);       //turn on green LED

  sensors = RPC.call("read\_sensors").as<struct sensor\_data>();

  int rightSpeed = -200;

  int leftSpeed = -200;

  if (sensors.photoresistor\_right > 950) {

    rightSpeed += 4 \* (sensors.photoresistor\_right - 950);

  }

  if (sensors.photoresistor\_left > 950) {

    leftSpeed += 4 \* (sensors.photoresistor\_left - 950);

  }

  stepperLeft.setSpeed(leftSpeed);   //set left motor speed

  stepperRight.setSpeed(rightSpeed);  //set right motor speed

  stepperRight.runSpeed();

  stepperLeft.runSpeed();

}

/\*

love uses the photoresistors to turn the robot towards the light when sensed

\*/

void love(void) {

  digitalWrite(redLED, HIGH);       //turn on red LED

  digitalWrite(ylwLED, LOW);       //turn off yellow LED

  digitalWrite(grnLED, HIGH);       //turn on green LED

  sensors = RPC.call("read\_sensors").as<struct sensor\_data>();

  int rightSpeed = 0;

  int leftSpeed = 0;

  // Serial.print("Left Sensor: ");

  // Serial.print(sensors.photoresistor\_left);

  // Serial.print("   ");

  // Serial.print("Right Sensor: ");

  // Serial.println(sensors.photoresistor\_right);

  if (sensors.photoresistor\_left > 800) {

    rightSpeed = -2\* (sensors.photoresistor\_left - 800);

  }

  if (sensors.photoresistor\_right > 800) {

    leftSpeed = -2 \* (sensors.photoresistor\_right - 800);

  }

  stepperLeft.setSpeed(leftSpeed);   //set left motor speed

  stepperRight.setSpeed(rightSpeed);  //set right motor speed

  stepperRight.runSpeed();

  stepperLeft.runSpeed();

}

/\*

explorer uses the photoresistors to turn the robot away from light

\*/

void explorer(void) {

  digitalWrite(redLED, HIGH);       //turn on red LED

  digitalWrite(ylwLED, LOW);       //turn off yellow LED

  digitalWrite(grnLED, HIGH);       //turn on green LED

  sensors = RPC.call("read\_sensors").as<struct sensor\_data>();

  int rightSpeed = -200;

  int leftSpeed = -200;

  if (sensors.photoresistor\_left > 950) {

    rightSpeed += 4 \* (sensors.photoresistor\_left - 950);

  }

  if (sensors.photoresistor\_right > 950) {

    leftSpeed += 4 \* (sensors.photoresistor\_right - 950);

  }

  stepperLeft.setSpeed(leftSpeed);   //set left motor speed

  stepperRight.setSpeed(rightSpeed);  //set right motor speed

  stepperRight.runSpeed();

  stepperLeft.runSpeed();

}

/\*

loveAvoid uses the love behavior, but has obstacle avoidance when the robot gets too close

to obstacles

\*/

void loveAvoid(void) {

  sensors = RPC.call("read\_sensors").as<struct sensor\_data>();

  if (sensors.lidar\_back < 10 || sensors.lidar\_right < 15 || sensors.lidar\_left < 15) {

    runaway();

  } else {

    love();

  }

}

/\*

homing utlizes a state machine to have the robot wall follow until a light is sensed,

then move towards the light, turn 180 degrees, go back to the wall, and wall follow

\*/

#define STATE\_WALLFOLLOW 0

#define STATE\_LOVE 1

#define STATE\_AFTERLOVE 2

#define LOVE\_LIGHT 900

int homingState = STATE\_WALLFOLLOW;

void homing(void) {

  sensors = RPC.call("read\_sensors").as<struct sensor\_data>();

  Serial.print("Left Sensor: ");

  Serial.print(sensors.photoresistor\_left);

  Serial.print("   ");

  Serial.print("Right Sensor: ");

  Serial.print(sensors.photoresistor\_right);

  if (homingState == STATE\_WALLFOLLOW && !loved && (sensors.photoresistor\_right > LOVE\_LIGHT || sensors.photoresistor\_left > LOVE\_LIGHT)) {

    homingState = STATE\_LOVE;

    loved = true;

  } else if (homingState == STATE\_LOVE && sensors.lidar\_back < 10) {

    homingState = STATE\_AFTERLOVE;

    delay(3000);

    spin(180, 0);

  } else if (homingState == STATE\_AFTERLOVE && sensors.lidar\_back < 10) {

    if (state == RIGHT\_WALL) {

      spin(135, 1);

    } else if (state == LEFT\_WALL)  {

      spin(135, 0);

    }

    homingState = STATE\_WALLFOLLOW;

  }

  switch (homingState) {

    case STATE\_WALLFOLLOW:

      wallFollowPD();

      Serial.println("  State: Wall Follow");

      break;

    case STATE\_LOVE:

      love();

      Serial.println("  State: Love");

      break;

    case STATE\_AFTERLOVE:

      stepperLeft.setSpeed(-300);   //set left motor speed

      stepperRight.setSpeed(-300);  //set right motor speed

      stepperRight.runSpeed();

      stepperLeft.runSpeed();

      Serial.println("  State: After Love");

      break;

  }

}

int charIndex = 0;

int rightSpeed = 0;

int leftSpeed = 0;

int turnTimer = 0;

bool timerSet = false;

bool terminated = false;

/\*

topologicalPath converts a string comprised of S, L, R, and T into actions to

reach a goal.

\*/

void topologicalPath(String SLRTstring) {

  sensors = RPC.call("read\_sensors").as<struct sensor\_data>();

  char SLRTchar = SLRTstring.charAt(charIndex);

  switch(SLRTchar) {

    case 'S':

      //Serial.println("S");

      wallFollowPD();

      charIndex++;

      digitalWrite(redLED, LOW);       //turn off red LED

      digitalWrite(ylwLED, LOW);       //turn off yellow LED

      digitalWrite(grnLED, LOW);       //turn off green LED

      break;

    case 'L':

      if (sensors.lidar\_right > 25) {

        //Serial.println("L");

        forward(-750);

        spin(90, 0);

        forward(-600);

        charIndex++;

      } else {

        wallFollowPD();

      }

      digitalWrite(redLED, LOW);       //turn off red LED

      digitalWrite(ylwLED, HIGH);      //turn on yellow LED

      digitalWrite(grnLED, LOW);       //turn off green LED

      break;

    case 'R':

      if (sensors.lidar\_left > 25) {

        //Serial.println("R");

        forward(-750);

        spin(90, 1);

        forward(-600);

        charIndex++;

      } else {

        wallFollowPD();

      }

      digitalWrite(redLED, LOW);       //turn off red LED

      digitalWrite(ylwLED, LOW);       //turn off yellow LED

      digitalWrite(grnLED, HIGH);      //turn on green LED

      break;

    case 'T':

      if (sensors.lidar\_right < 20 && sensors.lidar\_left < 20 && sensors.lidar\_back < 15) {

        //Serial.println("T");

        rightSpeed = 0;

        leftSpeed = 0;

        terminated = true;

      } else if (!terminated) {

        wallFollowPD();

      }

      digitalWrite(redLED, HIGH);       //turn on red LED

      digitalWrite(ylwLED, LOW);      //turn on yellow LED

      digitalWrite(grnLED, LOW);       //turn off green LED

      break;

  }

}

#define GOAL\_NUM 1

#define ROBOT\_NUM 98

bool hasZero = true;

/\*

mapToWavefront12 converts a 12x12 occupancy grid to a wavefront map

\*/

void mapToWavefront12(BLA::Matrix<12, 12> &map) {

  int iter = 0;

  int maxIter = 100;

  while (hasZero && iter < maxIter) {

    hasZero = false;

    iter++;

    for (int i = 0; i < map.Rows; i++) {

      for (int j = 0; j < map.Cols; j++) {

        int lowestBorder = 100;

        if (map(i, j) == 0) {

          hasZero = true;

        }

        if (map(i, j) != 1 && map(i, j) < 98) {

          if (i != 0) {

            if (map(i-1, j) > 0 && map(i-1, j) < 98 && map(i-1, j) < lowestBorder){

              lowestBorder = map(i-1, j);

            }

          }

          if (i != map.Rows - 1) {

            if (map(i+1, j) > 0 && map(i+1, j) < 98  && map(i+1, j) < lowestBorder){

              lowestBorder = map(i+1, j);

            }

          }

          if (j != 0) {

            if (map(i, j-1) > 0 && map(i, j-1) < 98 && map(i, j-1) < lowestBorder){

              lowestBorder = map(i, j-1);

            }

          }

          if (j != map.Cols - 1) {

            if (map(i, j+1) > 0 && map(i, j+1) < 98 && map(i, j+1) < lowestBorder){

              lowestBorder = map(i, j+1);

            }

          }

        }

        if (lowestBorder != 100) {

          map(i, j) = lowestBorder + 1;

        }

      }

    }

  }

}

/\*

mapToWavefront converts a 4x4 occupancy grid to a wavefront map

\*/

void mapToWavefront(BLA::Matrix<4, 4> &map) {

  while (hasZero) {

    hasZero = false;

    for (int i = 0; i < map.Rows; i++) {

      for (int j = 0; j < map.Cols; j++) {

        int lowestBorder = 100;

        if (map(i, j) == 0) {

          hasZero = true;

        }

        if (map(i, j) != 1 && map(i, j) < 98) {

          if (i != 0) {

            if (map(i-1, j) > 0 && map(i-1, j) < 98 && map(i-1, j) < lowestBorder){

              lowestBorder = map(i-1, j);

            }

          }

          if (i != map.Rows - 1) {

            if (map(i+1, j) > 0 && map(i+1, j) < 98  && map(i+1, j) < lowestBorder){

              lowestBorder = map(i+1, j);

            }

          }

          if (j != 0) {

            if (map(i, j-1) > 0 && map(i, j-1) < 98 && map(i, j-1) < lowestBorder){

              lowestBorder = map(i, j-1);

            }

          }

          if (j != map.Cols - 1) {

            if (map(i, j+1) > 0 && map(i, j+1) < 98 && map(i, j+1) < lowestBorder){

              lowestBorder = map(i, j+1);

            }

          }

        }

        if (lowestBorder != 100) {

          map(i, j) = lowestBorder + 1;

        }

      }

    }

  }

}

int nodeState = 0;

int dirState = 0;

bool foundEnd = false;

#define NORTH 0

#define EAST 1

#define SOUTH 2

#define WEST 3

/\*

wavefrontToSLRT12 converts a 12x12 wavefront map to a SLRT string

\*/

String wavefrontToSLRT12(Matrix<12, 12> wavefront, int startingDir) {

  String SLRTstring = "S";

  dirState = startingDir;

  while(!foundEnd) {

    for (int i = 0; i < wavefront.Rows; i++) {

      for (int j = 0; j < wavefront.Cols; j++) {

        int lowestBorder = 1000;

        if (wavefront(i, j) == 98) {

          if (i != 0) {

            if (wavefront(i-1, j) < lowestBorder){

              lowestBorder = wavefront(i-1, j);

              nodeState = NORTH;

            }

          }

          if (i != wavefront.Rows - 1) {

            if (wavefront(i+1, j) < lowestBorder){

              lowestBorder = wavefront(i+1, j);

              nodeState = SOUTH;

            }

          }

          if (j != 0) {

            if (wavefront(i, j-1) < lowestBorder){

              lowestBorder = wavefront(i, j-1);

              nodeState = WEST;

            }

          }

          if (j != wavefront.Cols - 1) {

            if (wavefront(i, j+1) < lowestBorder){

              lowestBorder = wavefront(i, j+1);

              nodeState = EAST;

            }

          }

          if (dirState + 1 == nodeState || (dirState == WEST && nodeState == NORTH)){

            SLRTstring += "R";

          } else if (dirState - 1 == nodeState || (dirState == NORTH && nodeState == WEST)) {

            SLRTstring += "L";

          }

          if (lowestBorder == 1) {

            SLRTstring += "T";

            foundEnd = true;

            return SLRTstring;

          }

          wavefront(i, j) = 100;

          switch(nodeState) {

            case NORTH:

              wavefront(i - 1, j) = 98;

              break;

            case EAST:

              wavefront(i, j + 1) = 98;

              break;

            case SOUTH:

              wavefront(i + 1, j) = 98;

              break;

            case WEST:

              wavefront(i, j - 1) = 98;

              break;

          }

          dirState = nodeState;

        }

      }

    }

  }

}

/\*

wavefrontToSLRT converts a 4x4 wavefront map to a SLRT string

\*/

String wavefrontToSLRT(Matrix<4,4> wavefront, int startingDir) {

  String SLRTstring = "S";

  dirState = startingDir;

  while(!foundEnd) {

    for (int i = 0; i < wavefront.Rows; i++) {

      for (int j = 0; j < wavefront.Cols; j++) {

        int lowestBorder = 1000;

        if (wavefront(i, j) == 98) {

          if (i != 0) {

            if (wavefront(i-1, j) < lowestBorder){

              lowestBorder = wavefront(i-1, j);

              nodeState = NORTH;

            }

          }

          if (i != wavefront.Rows - 1) {

            if (wavefront(i+1, j) < lowestBorder){

              lowestBorder = wavefront(i+1, j);

              nodeState = SOUTH;

            }

          }

          if (j != 0) {

            if (wavefront(i, j-1) < lowestBorder){

              lowestBorder = wavefront(i, j-1);

              nodeState = WEST;

            }

          }

          if (j != wavefront.Cols - 1) {

            if (wavefront(i, j+1) < lowestBorder){

              lowestBorder = wavefront(i, j+1);

              nodeState = EAST;

            }

          }

          if (dirState + 1 == nodeState || (dirState == WEST && nodeState == NORTH)){

            SLRTstring += "R";

          } else if (dirState - 1 == nodeState || (dirState == NORTH && nodeState == WEST)) {

            SLRTstring += "L";

          }

          if (lowestBorder == 1) {

            SLRTstring += "T";

            foundEnd = true;

            return SLRTstring;

          }

          wavefront(i, j) = 100;

          switch(nodeState) {

            case NORTH:

              wavefront(i - 1, j) = 98;

              break;

            case EAST:

              wavefront(i, j + 1) = 98;

              break;

            case SOUTH:

              wavefront(i + 1, j) = 98;

              break;

            case WEST:

              wavefront(i, j - 1) = 98;

              break;

          }

          dirState = nodeState;

        }

      }

    }

  }

}

/\*

tMapToOMap converts a 12x12 topological map to a occupancy grid

\*/

Matrix<12,12> tMapToOMap(Matrix<4,4> tMap) {

  Matrix<12,12> oMap = 0;

  for (int i = 0; i < tMap.Rows; i++) {

    for (int j = 0; j < tMap.Cols; j++) {

      int currNode = tMap(i, j);

      switch (currNode) {

        case 0:

          break;

        case 1:

          setNorthWall(oMap, i\*3, j\*3);

          oMap(i\*3 + 2, j\*3) = 99;

          oMap(i\*3 + 2, j\*3 + 2) = 99;

          break;

        case 2:

          setEastWall(oMap, i\*3, j\*3);

          oMap(i\*3, j\*3) = 99;

          oMap(i\*3 + 2, j\*3) = 99;

          break;

        case 3:

          setNorthWall(oMap, i\*3, j\*3);

          setEastWall(oMap, i\*3, j\*3);

          break;

        case 4:

          setSouthWall(oMap, i\*3, j\*3);

          oMap(i\*3, j\*3) = 99;

          oMap(i\*3, j\*3 + 2) = 99;

          break;

        case 5:

          setNorthWall(oMap, i\*3, j\*3);

          setSouthWall(oMap, i\*3, j\*3);

          break;

        case 6:

          setEastWall(oMap, i\*3, j\*3);

          setSouthWall(oMap, i\*3, j\*3);

          break;

        case 7:

          setNorthWall(oMap, i\*3, j\*3);

          setEastWall(oMap, i\*3, j\*3);

          setSouthWall(oMap, i\*3, j\*3);

          break;

        case 8:

          setWestWall(oMap, i\*3, j\*3);

          oMap(i\*3, j\*3 + 2) = 99;

          oMap(i\*3 + 2, j\*3 + 2) = 99;

          break;

        case 9:

          setNorthWall(oMap, i\*3, j\*3);

          setWestWall(oMap, i\*3, j\*3);

          break;

        case 10:

          setEastWall(oMap, i\*3, j\*3);

          setWestWall(oMap, i\*3, j\*3);

          break;

        case 11:

          setNorthWall(oMap, i\*3, j\*3);

          setEastWall(oMap, i\*3, j\*3);

          setWestWall(oMap, i\*3, j\*3);

          break;

        case 12:

          setSouthWall(oMap, i\*3, j\*3);

          setWestWall(oMap, i\*3, j\*3);

          break;

        case 13:

          setNorthWall(oMap, i\*3, j\*3);

          setSouthWall(oMap, i\*3, j\*3);

          setWestWall(oMap, i\*3, j\*3);

          break;

        case 14:

          setEastWall(oMap, i\*3, j\*3);

          setSouthWall(oMap, i\*3, j\*3);

          setWestWall(oMap, i\*3, j\*3);

          break;

        case 15:

          setNorthWall(oMap, i\*3, j\*3);

          setEastWall(oMap, i\*3, j\*3);

          setSouthWall(oMap, i\*3, j\*3);

          setWestWall(oMap, i\*3, j\*3);

          break;

      }

    }

  }

  return oMap;

}

/\*

Used within the tMapToOMap function to set the matrix nodes corresponding with a wall

on the north side of the 3x3 grid

\*/

void setNorthWall(BLA::Matrix<12, 12> &oMap, int row, int col) {

  oMap(row,col) = 99;

  oMap(row,col+1) = 99;

  oMap(row,col+2) = 99;

}

/\*

Used within the tMapToOMap function to set the matrix nodes corresponding with a wall

on the South side of the 3x3 grid

\*/

void setSouthWall(BLA::Matrix<12, 12> &oMap, int row, int col) {

  oMap(row+2,col) = 99;

  oMap(row+2,col+1) = 99;

  oMap(row+2,col+2) = 99;

}

/\*

Used within the tMapToOMap function to set the matrix nodes corresponding with a wall

on the West side of the 3x3 grid

\*/

void setWestWall(BLA::Matrix<12, 12> &oMap, int row, int col) {

  oMap(row,col) = 99;

  oMap(row+1,col) = 99;

  oMap(row+2,col) = 99;

}

/\*

Used within the tMapToOMap function to set the matrix nodes corresponding with a wall

on the East side of the 3x3 grid

\*/

void setEastWall(BLA::Matrix<12, 12> &oMap, int row, int col) {

  oMap(row,col+2) = 99;

  oMap(row+1,col+2) = 99;

  oMap(row+2,col+2) = 99;

}

/\*

oGridMetricPath plans a path based on a 4x4 occupancy grid and executes navigation

\*/

void oGridMetricPath(Matix<4, 4> oMap, int startingDir) {

  mapToWavefront(oMap);

  String SLRTstring = wavefrontToSLRT(wavefront, startingDir);

  while (true) {

    topologicalPath(SLRTstring);

  }

}

/\*

topMapMetricPath plans a path based on a 4x4 topological map and executes navigation

\*/

void topMapMetricPath(Matix<4, 4> tMap, int startingDir) {

  Matrix<12, 12> oMap = tMapToOMap(Matrix<4,4> tMap);

  mapToWavefront12(oMap);

  String SLRTstring = wavefrontToSLRT12(wavefront, startingDir);

  while (true) {

    topologicalPath(SLRTstring);

  }

}

void setup() {

  RPC.begin();

  if(HAL\_GetCurrentCPUID() == CM7\_CPUID) {

    // if on M7 CPU, run M7 setup & loop

    setupM7();

    while(1) loopM7();

  } else {

    // if on M4 CPU, run M7 setup & loop

    setupM4();

    while(1) loopM4();

  }

}

// loop() is never called as setup() never returns

void loop() {}

//// MAIN

String SLRTcommand = "";

void setupM7() {

  int baudrate = 9600;  //serial monitor baud rate'

  init\_stepper();       //set up stepper motor

  attachInterrupt(digitalPinToInterrupt(ltEncoder), LwheelSpeed, CHANGE);  //init the interrupt mode for the left encoder

  attachInterrupt(digitalPinToInterrupt(rtEncoder), RwheelSpeed, CHANGE);  //init the interrupt mode for the right encoder

  for (int i = 0; i<numOfSens;i++){

    pinMode(lidar\_pins[i],INPUT);

  }

  Serial.begin(baudrate);  //start serial monitor communication

  delay(1000);

  Serial.println("Robot starting...Put ON TEST STAND");

  delay(pauseTime);  //always  wait 2.5 seconds before the robot moves

}

void loopM7() {

  //Uncomment to read Encoder Data (uncomment to read on serial monitor)

  // print\_encoder\_data();   //prints encoder data

  // delay(10000);               //wait to move robot or read data

}

A.5 Wireless Communication Design Plan:

A.6 Wireless Communication Tutorial:

A.7 Wireless Communication Code: