UNIVERSITY OF TORONTO

DESIGN REPORT

MIE443 - Mechatronics Systems: Design and Integration

Contest #1: Autonomous Robot Search of an Environment

Authors:
Sophie Miller - 1005811392
Jenna Del Fatti - 1006018738
Tiger Luo - 1008306502
Stephanie Beals - 1004450872
Kevin Hsu - 1005367728

Professor:
Goldie Nejat
Head Teaching Assistant:
Aaron Tan

February 13, 2024

Table of Content

1.0 Problem Definition & Introduction

Turtlebot is a low cost personal kit with open source libraries for hobby projects and education [1]. For contest 1, the TurtleBot was used along with ROS to develop an autonomous mapping robot. Robot Operating System (ROS) refers to a set of software libraries and tools that help you build robot applications [2].

1.1 Objective of Contest

The objective of the contest 1 is to develop a combination of algorithms paired with ROS libraries to enable the TurtleBot to autonomously map an environment. Tasked with navigating an unfamiliar environment, the robot's primary goal is to accurately map its surroundings, identifying obstacles and walls along the way. Equipped with an Xbox Kinect sensor, the TurtleBot utilized this tool for mapping and obstacle detection. Furthermore, the robot relied on additional sensory inputs from right, front, and left bumper sensors to enhance its data collection capabilities, essential for refining movement algorithms.

The ultimate challenge for the team was to guide the TurtleBot through the entire map within an eight-minute time frame, aiming to achieve the most accurate representation of the environment possible. As depicted in **Figure 1**, the resulting map showcases a comprehensive view of the area, with gray areas denoting open space and black lines delineating obstacles. **Figure 2** illustrates the test environment utilized by the team for validation purposes, emphasizing the likelihood of encountering more densely populated areas during mapping operations.

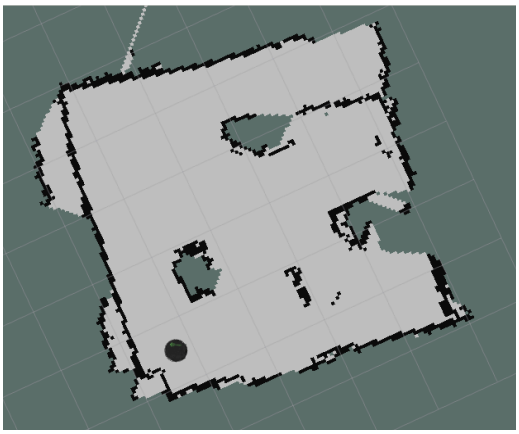

Figure 1. Example of map generated by exploration algorithm

Figure 2. Example of Test Environment

1.2 Design Requirements & Constraints

The design constraints for this project are highlighted in MIE 443 Contest 1 Instructions 2024 [3]. As per the instructions the design constraints are expressed in **Table 1** below.

Table 1. Design Constraints

Constraints
The TurtleBot has a time limit of 8 minutes to both explore and map the environment.
The robot must be completely autonomous .
Robot must navigate through sensor feedback, no hard coding directions
There must be a speed limitation on the robot that will not allow it to go any faster than 0.25m/s when it is navigating the environment and 0.1m/s when it is close to obstacles such as walls.

Throughout the development process the team developed internal constraints in order to optimize our code and ensure the robot functioning optimally.

Table 2. Self Imposed Design Constraints

Self Imposed Constraints
Code must be non-blocking
The number of turns / speed of turns must not cause the robot to lose track of bearings.

2.0 Strategy and Methodology

Our primary design objective is to maximize the robustness and reliability with which the TurtleBot navigates the random environment, while minimizing random unknown behaviours that may lead to errors in data reading (eg. NaN, infinity) and compromised map generation. Considering the contest's unpredictable layout, the robot's start location, and its orientation, coupled with a tight development and testing timeline, our team has adopted a modular approach, integrating two exploration strategies through the development of multiple functions and robot states. We employ a random biased walk aimed at seeking open spaces for our robot exploration to optimize environmental scanning coverage. This strategy was selected for its adaptability regardless of the contest layout therefore ensuring that mapping performance is high in any setting. The functions developed can be implemented independently in various different exploration strategies and combined with a high level

controller design to execute the search algorithm. These additional functions and strategies include navigational logic, avoidance logic, a bumper response, and rotational scan. **Table 3** below introduces and briefly summarizes these strategy functionalities that were implemented to execute our search algorithm. They will be discussed in further detail in the high and low level control descriptions in **Section 3.2.**

Table 3. Summary of Strategies

Strategy 1	**Navigational Logic**
Functionality	Decides which robot state to enter depending on object/corner detection, bumper pressed etc; default state is to explore environment randomly
Strategy 2	**Avoidance Logic**
Functionality	Turns robot in opposite direction of detected obstacle
Strategy 3	**Bumper Response**
Functionality	Stop and turn left, right, or in direction of free path when bumper(s) are activated
Strategy 4	**Rotational Scan**
Functionality	Rotates 360 degrees and moves forward in direction with the most space

3.0 Detailed Robot Design and Implementation

The Turtlebot architecture follows a general structure consisting of high level and low controller design and sensory design as is described in **Figure 3** below. From the bottom up, the low level control contains all principal functions that support robot movements, the sensory design provides information inputted by the environment used to dictate decision making and control, and the high level control contains the mapping and exploration functionality.

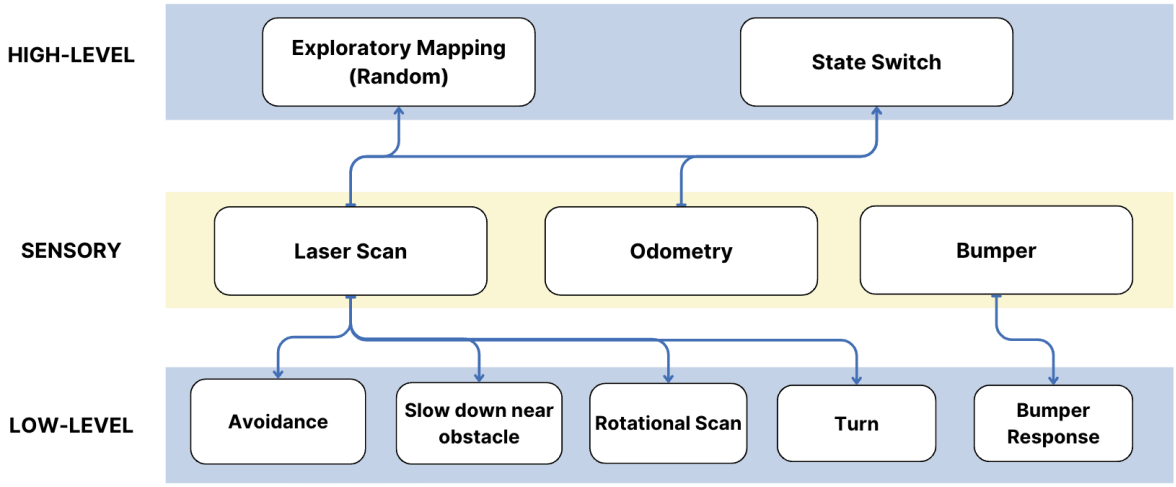

Figure 3. TurtleBot architecture for robot control

3.1 Sensory Design

The TurtleBot 2 robot utilized in this contest is equipped with an iClebo Kobuki mobile base and a Microsoft Xbox Kinect 360 sensor. The iClebo Kobuki is equipped with 3 cliff sensors, 2 wheel drop sensors, 3 front bumpers, a gyro, and wheel encoders for odometry callback. The Kinect sensor has a RGB camera, a depth sensor, and a microphone array [4]. In contest 1, the team implemented a path planning algorithm for exploration that uses 3 sensors on the robot: the laser scan depth sensor, the odometry callback, and the bumper sensors.

3.1.1 Laser Scan Depth Sensor

The laser depth sensor is the primary visual perception for the robot that provides a field of view of 60 degrees. It is used for the robot to navigate and map out its environment. The linear and angular speed of the robot is set to a maximum of 0.25 m / s and 22.9° / s, respectively, allowing the mapping algorithm and navigation function using the laser sensor data to construct more accurate maps and make reliable decisions. For robot navigation, the laser readings are used to fulfill three functions: (1) detect obstacles in front of the robot and inform the relative position of the object with respect to the robot, (2) determine the minimum distance from the robot to the obstacle or wall, (3) determine a list of maximum path distances sensed around the robot. The high level concepts are succinctly presented in **Figure 4** below.

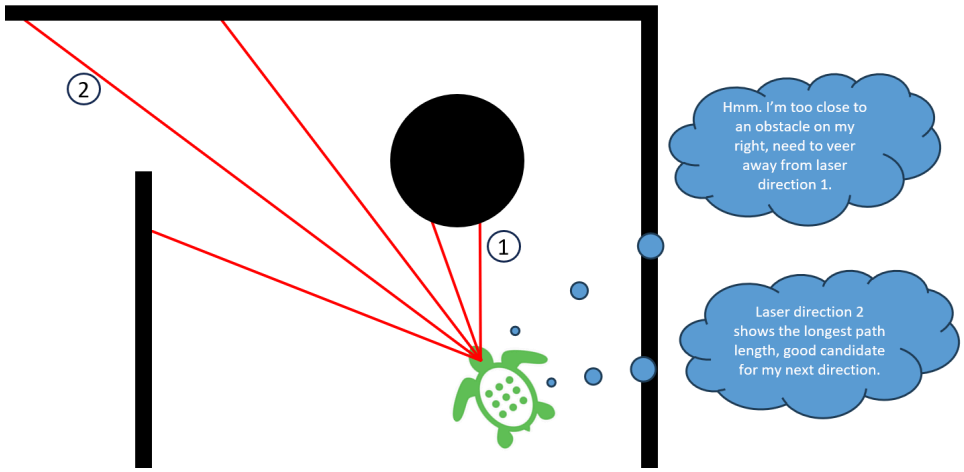

Figure 4. Turtlebot Navigation Decision Making with Laser Readings

The laser data is provided by the ***laserCallback*** function in the attached robot code. The ***laserCallback*** function is subscribed to in ***main*** for detecting obstacles or walls in front of the robot. The laser readings are processed by looping through a range of readings stored in the scanned array (laserVals) while filtering out values that are below or above the reliable laser range. This enables the team to extract relevant information such as max / min laser readings or laser reading indices for precise localization of points of interest. The 3 crucial functions constructed using the laser readings allowed the team to further develop more sophisticated functions such as object detection (objectDetect) in front / left / right, as well as ensuring the robot for maintaining a safe distance between obstacles at all times.

3.1.2 Odometry

The pair of rotary encoders on the motors of the Turtlebot provide data readings on the rotational velocity of each of the wheels. This data is then used with the forward differential kinematics model to generate rotational and linear velocities. The position of the robot after some movement is then estimated by integrating these velocity measurements (over time), using how far the wheels have rotated and the circumference of the wheels to calculate the distance traveled. This yields an estimate of robot position and heading. The odometry data is provided by the ***odomCallback*** function in the attached robot code. The ***odomCallback*** function is subscribed to in ***main*** to set the x and y positions and yaw of the robot to the calculated position values collected by the encoders. This can aid us in our robot's decision making process while informing us how the robot has moved throughout the environment in order to accurately identify obstacles. The value of yaw in the odometry callback is of particular importance as it will be used to execute spins and precisely translate as we navigate the environment.

3.1.3 Bumper Sensors

The iClebo Kobuki base contains three bumpers along the front as shown in **Figure 5**. The bumpers are used to detect if the turtlebot hits an on obstacle and a state of either pressed or not pressed is returned. Understanding which bumper is pressed gives information on the location of obstacles and allows the team to implement movement logic to ensure the turtlebot does not get stuck.

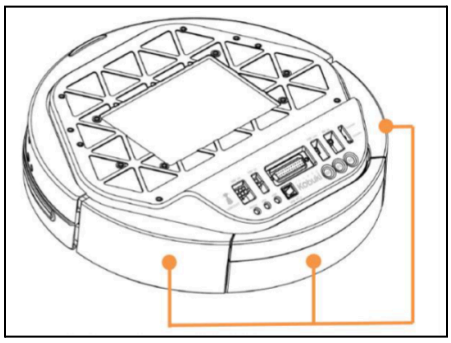

Figure 5. Right, center, and left bumpers on the Kobuki base [5].

As explained in Section 2.0, the navigation logic follows a mixture of different strategies, one of them being the Bumper Response. The motivation behind using the bumper sensors is to ensure the turtlebot does not get stuck in the case it hits a wall or obstacle. Although the laser scan sensor is being used to navigate and avoid objects, there are still multiple cases in which the sensor can fail and a bumper will be hit as shown in **Figure 6**. These include objects being too close, objects being too short, or an object being outside the laser scan range. The bumper sensors are therefore being used as a fail safe to ensure if the turtlebot gets stuck it can carry out specific movements to reorient itself and continue exploring the test space.

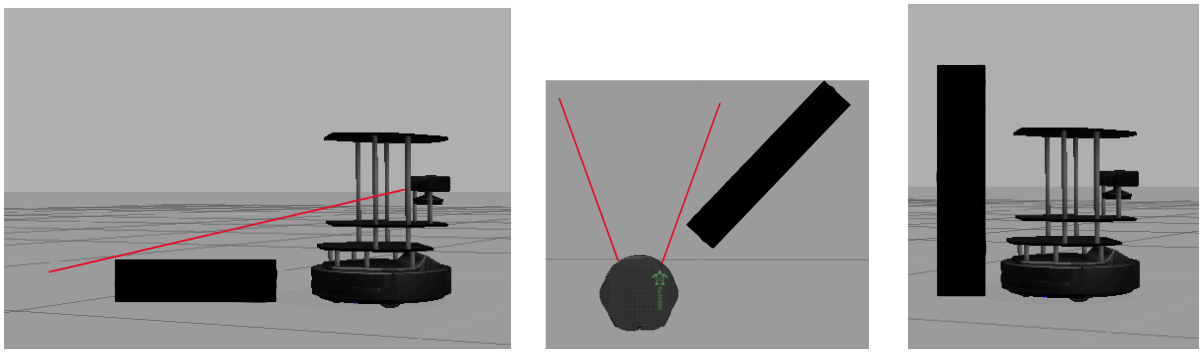

Figure 6. Conditions when the laser scan will not stop the bot from bumping into an object (a) object below laser range (b) object outside laser range angle (c) object too close to sense.

The bumper data is provided in the ***bumperCallback*** function in the attached robot code. The ***bumperCallback*** follows logic to store if a bumper is pressed and which one is pressed. The decision making in the ***bumperCallback*** is shown in **Figure 7**.

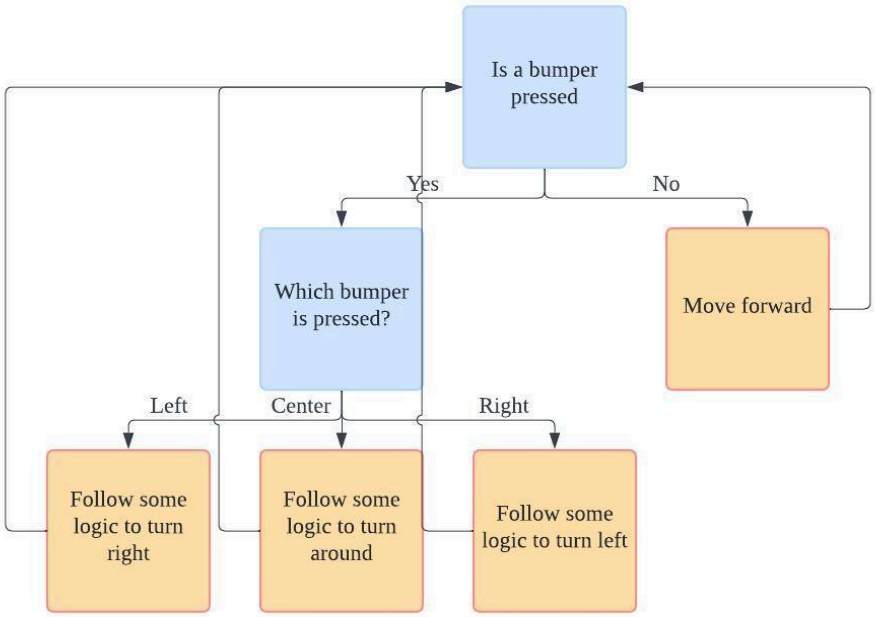

Figure 7. Logic followed in the bumper callback function.

3.2 Controller Design

To control the turtlebot's movements, the team developed and implemented a C++ ROS code which is shown in **Appendix A.** The code makes use of many low level functions which control specific individual movements of the turtlebot. The low level functions are used within a high level state machine to switch between different movement algorithms. The TurtleBot only ever exists in a single state. The state changes are triggered based on sensory feedback from the laser scan, odometry, and bumpers. The implementation of the team's code creates a fully autonomous robot that is able to navigate and map an unknown environment.

3.2.1 Low Level Control

This section outlines the low level control of the turtlebot and the associated functions which control movement. Some of the functions provide supporting movements and features while other functions are associated with states defined in the high level control (**Section 3.2.2**).

3.2.1.1 Angle Add Function: void angularAdd ***(float *summand, float angAddend)***;

When performing turns or other operations with yaw, a global frame of reference is required to know the relative location of where you wish to turn towards. As such, because yaw is only on the domain of (-π, π), a special function is used for adding/subtracting angles. This function ensures the summand falls within the domain by simply subtracting or adding 2π.

3.2.1.2 Path Known: bool pathKnown (***float test_dist, float test_yaw)***;

At various points in the robot's operation, the robot's position will be recorded in a global array. This function takes two arguments, a global yaw and relative distance, and calculates whether or not the end point has already been visited or not. This function is mainly used in the spin around to ensure the same location isn't visited twice.

3.2.1.3 Turn Function: ***bool turn(float turnAmt, int ranIdx);***

The turn function is the most used function. Its functionality is meant to allow for turning a precise angle passed into it as an argument while also being non-blocking for the rest of ROS. This is accomplished by purely using if statements within the function, and having a global array dedicated to its use. The global array stores the starting yaw to compare to the final yaw, which allows the function to be non-blocking. The function returns a boolean, true if it's completed, and false if not. The global array is indexed once the function is completed in case multiple turn functions need to be completed sequentially. This is best demonstrated in the Spin Around function.

3.2.1.4 Spin Around Function: ***void spinAround();***

The Spin Around function provides the primary means of decision making and random walking. It turns in 45 degree segments, taking the normal laser distance of the robot after each turn (this corresponds to laser index 319). It turns a full 360 degrees, and upon completion, parses the stored normal distances, filters out for directions with a distance greater than 1.2m, filters out for locations it has already been, and chooses from the remaining directions at random. This function is perhaps the most complex, requiring many turns and uses of global variables, and as such is carefully indexed through another global variable called stepNo to ensure its non-blocking behavior does not result in erroneous states.

3.2.1.5 Unstuck Function: ***void emergencyUnstuck();***

In the event the spinAround function has no directions which fulfill its requirements, then this function is invoked to attempt to get it moving in a clear direction, simply by turning it until it reads that there is no object detected ahead of it. This detection is calculated during the laserCallback function, and is also used in the avoid function.

3.2.1.6 Avoid Function: ***void avoid()***

This function is considered state 9 and is triggered when an object is detected within 0.7 m of the laser scanner (see **Section 3.2.2.2** for more details). The function works to move the turtlebot in a way that ensures it maneuvers around walls and obstacles by positioning itself to face open space. The avoid function relies on the ***laserCallback*** to detect objects and determine if the detected object is to the left, right, or in front of the turtlebot. Based on the scenario and where objects are sensed, there are four cases of movement the turtlebot will follow. First, if there is an object in the right region but there is space on the left, the bot will

move forward and turn left simultaneously to move towards open space. Similarly, in the second case, if there is an object on the left but there is space on the right, the bot will move forward and turn right simultaneously to face the open space. In the third case, if there are objects detected on both the right and left, but there is nothing detected in front, the bot will move linearly forward. This ensures the turtlebot will be able to move in between two obstacles and successfully move down a narrow corridor. The fourth and final case is if an object is detected in all three regions (left, center, and right). This means the turtlebot is either in a corner or directly facing an obstacle. In this case, the turtlebot is set to state 4 and follows the ***spinAround*** function to orient itself to free space. If any of the four cases is not met, the turtlebot is set to state 0 and the general ***navLogic***, further described in **Section 3.2.2.2,** is followed to enable the bot to keep exploring the space.

3.2.1.7 Bumper Functions: ***void leftBumper(); void rightBumper(); void centerBumper()***

There are three bumper functions in the code to control the turtlebot's movement depending on which bumper is triggered. If a bumper is triggered at any point throughout the space exploration, the turtlebot is set directly to state 6 for the left bumper, 7 for the center bumper, and 8 for the right bumper.

The left and right bumper functions follow the same logic. If the right bumper is pressed, the turtlebot backs up and turns 30° to the left to turn away from the obstacle. Once the 30° turn is complete, the turtlebot is then set to state 0 to follow its standard navigation logic. Similarly, if the left bumper is pressed, the bot backs up and turns 30° to the right and then set to state 0.

If the center bumper is pressed, the turtlebot first turns 30°. It then turns in 60° increments and takes distance readings from the laser scan at 4 different points. This means the turtle bot is scanning a 180° region opposite the direction to the obstacle as shown in **Figure 8.** The turtlebot then compares the distances and orients itself to the point with most space before setting to state 0 to follow the default navigation logic.

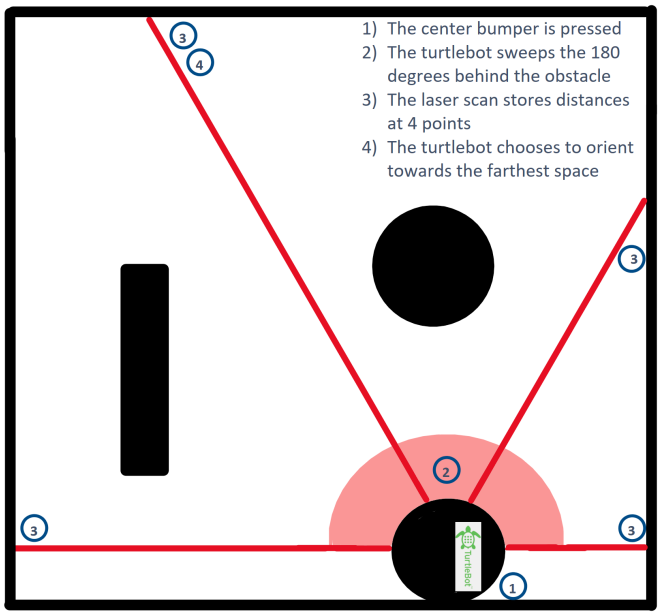

Figure 8. Movement coded in the ***centerBumper*** function

3.2.1.8 Navigational Logic: ***void navLogic();***

This is the primary state the robot will be in, and is fairly simplistic. It simply travels forward, until it gets near an object, at which point it will slow down. If it detects an object ahead of it, it will switch states to attempt to avoid it. Whenever this function is invoked, it 'clears' any global indices or variables used in other states that were being used to keep track of information between ros cycles, so that future states can use the same variables.

3.2.1.9 Long Distance Travel: ***bool longDistTravel();***

This function acts as a failsafe in the event that the robot gets stuck along a corridor. The logic is fairly straightforward: a counter is implemented to track the ROS cycles which run on a 10 Hz frequency. Once the counter reaches 70 cycles (continuously running straight for 7 seconds or around 1.75 meters), it will switch states to run the ***spinAround()*** function to orient to the direction with the most space and continue to move forward. The main purpose of this function is to check if there are other possible routes stemming from a long and straight corridor, another purpose is to prevent the robot from only moving in a continuous straight line in order to best map the environment.

3.2.2 High Level Control

This section outlines the high level control of the turtlebot and the architecture used to control robot movement and functionality. Detailed descriptions of the low level functions used in the state machine are included above in **Section 3.2.1.**

3.2.2.1 State Machine Architecture

The basis of the robot's control is modeled to be a state machine. None of the navigation logic occurs during the main function, instead there is only a switch statement that swaps the rover between different states by invoking different functions. The state machine only has 3 primary states: ***navLogic***, ***avoid***, and ***spinAround*** which correspond to the functions described in the low level control. The main default state is ***navLogic()***, which simply moves the robot forward until it finds an obstacle, after which it moves on to ***avoid()*** once it senses an obstacle in an attempt to avoid it. If it cannot avoid the obstacle by simply going left or right, it moves on to ***spinAround()***, after which it may complete successfully and pick a direction, or it could fail and go into the ***emergencyUnstuck().*** Additional states were planned, however they were ultimately not implemented to maintain robustness. The state machine also has 4 'interrupt' states, one for each of the bumpers, and an emergency unstuck. These states will interrupt any other processes going on, perform their function, then return to ***navLogic***. Global memory has to be reset when an interrupt does occur, and is cleared once again upon returning to navLogic. This state machine architecture is modeled in **Figure 9** below.

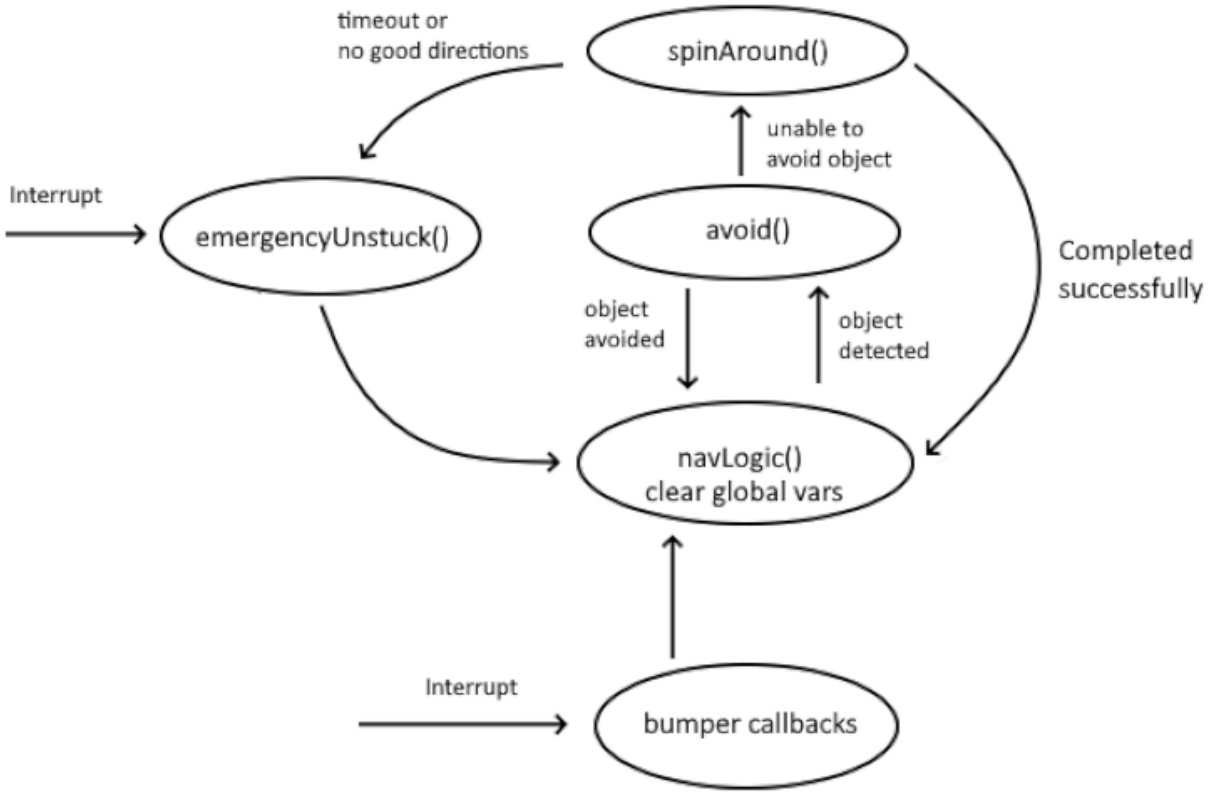

Figure 9. High level representation of state machine

4.0 Future Recommendations for Design Improvements

Due to the time constraint the team was unable to develop all its desired features. Firstly, it was noticed that the robot has a left bias when traveling. This means that the robot never travels straight; the team was hoping to implement a P or PI controller to correct this bias. This bias was unable to be corrected for Contest 1, however it is believed that the offset may contribute to the area explored for mapping. It will be further tested and corrected when developing the code and algorithm for Contest 2.

Secondly, the team was hoping to develop a way to keep track of the co-ordinates the robot visits. This was proposed with the end goal of being able to bias the robots movements towards unexplored areas of the map to maximize the area explored. The team wished to do this as the robot could repeat the path taken therefore missing map areas and obstacles located elsewhere in the map. By improving the position tracking abilities of the robot, it would in theory only travel to unexplored sections of the map by taking into account known locations in its decision making process.

Expanding on this idea, the team would have liked to form our own map or extract information from gmapping. This would have been in efforts of preventing the robot from getting stuck in one zone. The team could have used our own "map", along with the known dimensions of the environment to implement an algorithm that pushes the robot to explore the unseen areas of the map. Furthermore, this base could have served as a base for a frontier-based exploration algorithm, using the information collected from the robot and map in real time as feedback in order to only seek unexplored areas.

For Contest 2 the team also hopes to implement changes in terms of documentation. The team will try to include more comments in the code for readability when passing amongst team members and TAs, as well as keep a more organized git for easier handoff. The nuance in this type of project meant the team did not implement such practices for Contest 1 but will do for Contest 2.

5.0 References

[1] TURTLEBOT2, https://www.turtlebot.com/turtlebot2/#:~:text=TurtleBot%20is%20a%20low%2Dcost,horsepower%20to%20create%20exciting%20applications. (accessed Feb. 12, 2024).

[2] “Robot operating system,” ROS, https://www.ros.org/ (accessed Feb. 12, 2024).

[3] “MIE 443 Contest 1 Instructions 2024,” Quercus , https://q.utoronto.ca/courses/330231/files/folder/Contests/Contest%20%231?preview=29694246.

[4] Microsoft, “Kinect Sensor Components and Specifications.” [Online]. Available: https://msdn.microsoft.com/en-us/library/jj131033.aspx.

[5] G. Nejat et al. “TurtleBot Technical Manual for MIE 443H1S.” [Online]. Available: https://q.utoronto.ca/courses/330231/files/folder/Lab%20Manual?preview=29695423 . (accessed Feb. 11, 2024)

6.0 Attribution Table

The following sections detail the contributions of each group member to the Contest 1 deliverables including the robot code development and report writing.

Table 4. Attribution Table

Section	Jenna Del Fatti	Tiger Luo	Stephanie Beals	Sophie Miller	Kevin Hsu
Report: 1.0			1.0,1.1, 1.2		
Report: 2.0				2.0	
Report: 3.0	3.1.3, 3.2.1.6, 3.2.1.7,	3.2.1, 3.2.2.1		3.0, 3.1.2, 3.2.1.9, 3.2.2	3.1, 3.1.1
Report: 4.0			4.0		
Robot Code Function Development	avoid, bumpers (left,right, center), navLogic	Turn, timeout, angularAdd, spinAround, pathKnown, emergencyUnstuck	bumperCallback	avoid, navLogic	corner Recognition, cornerFollow, longDistTravel

7.0 Appendices

Appendix A: Full C++ ROS Code for Environment Search Algorithm

```
#include <ros/console.h>
#include "ros/ros.h"
#include <geometry_msgs/Twist.h>
#include <kobuki_msgs/BumperEvent.h>
#include <sensor_msgs/LaserScan.h>
#include <nav_msgs/Odometry.h>
#include <tf/transform_datatypes.h>

#include <stdio.h>
#include <cmath>
#include <stdlib.h>

#include <chrono>
#include <thread>

#define N_BUMPER (3)
#define RAD2DEG(rad) ((rad)*180./M_PI)
#define DEG2RAD(deg) ((deg)*M_PI/180.)
#define laserIdxStart nLasers/2-desiredNLasers
#define laserIdxEnd nLasers / 2 + desiredNLasers - 1

using Clock = std::chrono::system_clock;

//defining global variables for odom, laser, and bumper callbacks
float angular = 0.0;
float linear = 0.0;
float posX=0.0, posY=0.0, yaw=0.0;
uint8_t bumper[3]={kobuki_msgs::BumperEvent::RELEASED, kobuki_msgs::BumperEvent::RELEASED, kobuki_msgs::BumperEvent::RELEASED};

float minLaserDist=std::numeric_limits<float>::infinity();
int32_t nLasers=0, desiredNLasers=0, desiredAngle=20;

//float target_x = 0;
//float target_y = 0;

//global variable to start the algorithm with a 360 spin
float spinCount = 0;

//Used in laser callback
float laserVals[639]={0.0};
float laserFirstDiff[638]={0.0};
int minLaserIdx;
float angle_increment;
bool objectDetect[3]={0};

//states defined in main function used to control robot navigation logic
uint8_t state = 0;

bool bumperPressed=false;

void bumperCallback(const kobuki_msgs::BumperEvent::ConstPtr& msg){
    bumper[msg->bumper]=msg->state;

    //if a bumper is pressed the robot enters a corresponding state and function to react
    if (bumper[0]==kobuki_msgs::BumperEvent::PRESSED) {
        state = 6;
        ROS_INFO("Left Bumper Pressed");
        bumperPressed=true;
```

```
    } else if (bumper[1]==kobuki_msgs::BumperEvent::PRESSED) {
        state = 7;
        ROS_INFO("Middle Bumper Pressed");
        bumperPressed=true;
    } else if (bumper[2]==kobuki_msgs::BumperEvent::PRESSED) {
        state = 8;
        ROS_INFO("Right Bumper Pressed");
        bumperPressed=true;
    } else {
        bumperPressed=false;
    }

    //DELETE
    /*for (int i=0; i<3; i++){
        if (bumper[i]==kobuki_msgs::BumperEvent::PRESSED){
            state=9;
            bumperPressed=true;
        }
        else (bumperPressed=false);
    }*/
}
//Go to https://docs.ros.org/en/api/sensor_msgs/html/msg/LaserScan.html
//First entry is on robot right (-ve Y), going CCW around robot

void laserCallback(const sensor_msgs::LaserScan::ConstPtr& msg) {
    //laser callback stores laser scan readings into an array to use values in code

    minLaserDist = std::numeric_limits<float>::infinity();
    nLasers = (msg->angle_max - msg->angle_min) / msg->angle_increment;
    angle_increment = msg->angle_increment;
    desiredNLasers = desiredAngle*M_PI / (180*msg->angle_increment);

    //laser range is split into 3 to determine if an object is on the left, right, or center
    objectDetect[0]=false;
    objectDetect[1]=false;
    objectDetect[2]=false;

    //ROS_INFO("Size of laser scan array: %i and size of offset: %i", nLasers, desiredNLasers);
    int laserValIndex=0;
    if (desiredAngle * M_PI / 180 < msg->angle_max && -desiredAngle * M_PI / 180 > msg->angle_min) {
        for (uint32_t laser_idx = nLasers / 2 - desiredNLasers; laser_idx < nLasers / 2 + desiredNLasers; ++laser_idx){
            if (minLaserDist>msg->ranges[laser_idx]){
                minLaserDist = msg->ranges[laser_idx];
                minLaserIdx=laser_idx;
            }

            laserVals[laser_idx]=msg->ranges[laser_idx];
            if(laser_idx!=0){
                laserFirstDiff[laser_idx]=laserVals[laser_idx]-laserVals[laser_idx-1];
            }
            if (laser_idx<294 && (laserVals[laser_idx]<0.7|| laserVals[laser_idx]==std::numeric_limits<float>::infinity())){
                objectDetect[0]=true;
                //ROS_INFO("Object on left");
            } else if (laser_idx>344 &&  (laserVals[laser_idx]<0.7|| laserVals[laser_idx]==std::numeric_limits<float>::infinity())){
                objectDetect[2]=true;
                //ROS_INFO("Object on right");
            } else if (laser_idx < 344 && laser_idx>294 && (laserVals[laser_idx]<0.7|| laserVals[laser_idx]==std::numeric_limits<float>::infinity())){
                objectDetect[1]=true;
                //ROS_INFO("Object in front");
            }

            laserValIndex++;

            //DELETE
            //if(laserValIndex>300) ROS_INFO("%d", laserValIndex);
        }
    }
    else {
        for (uint32_t laser_idx = 0; laser_idx < nLasers; ++laser_idx) {
            if (minLaserDist>msg->ranges[laser_idx]){
                minLaserDist = msg->ranges[laser_idx];
                minLaserIdx=laser_idx;
            }
            laserVals[laser_idx]=msg->ranges[laser_idx];
            if(laser_idx!=0){
                laserFirstDiff[laser_idx]=laserVals[laser_idx]-laserVals[laser_idx-1];
            }
            laserValIndex++;
        }
    }
```

```
139
140     //ROS_INFO("First entry: %f, mid entry: %f, last entry: %f", laserVals[laserIdxStart], laserVals[300], laserVals[laserIdxEnd]);
141
142     //ROS_INFO("Min dist: %f", minLaserDist);
143
144
145 }
146
147
148 void odomCallback(const nav_msgs::Odometry::ConstPtr& msg){
149     posX=msg->pose.pose.position.x;
150     posY=msg->pose.pose.position.y;
151     yaw=tf::getYaw(msg->pose.pose.orientation);
152     //ROS_INFO("Position: (%f, %f) Orientation: %f rad or %f deg", posX, posY, yaw, RAD2DEG(yaw));
153 }
154
155 void navLogic(); //overall default navigation and conditions to switch states
156 //void bumperFailSafe(); //DELETE
157 //bool orientToNormal(); //DELETE
158 bool turn(float turnAmt, int ranIdx); //takes in an angle amount and turns the robot
159 //bool moveToPt(); //DELETE
160 //void decideDirection(); //decides if 90 degree left or right has most free path: DELETE
161 //void wallFollow(); //once at an obstacle turns 90 degrees left and wall follows: DELETE
162 void avoid(); //turns left, right, or spins to avoid obstacles
163 float idxToAng(int idx); //calculates laser scan angle associated with the array indices
164 bool timeout(uint64_t limit, std::chrono::time_point<std::chrono::system_clock> startPt); //starts a timer to ensure robot does not get stuck in a function
165 void angularAdd(float *summand, float angAddend);
166 void spinAround(); //spin 360 and face direction of most space
167 bool longDistTravel(); //check surronding if travelling straight for meters continuously
168 bool forward(float dist, int ranIdx); //moves robot forward
169 void rightBumper(); //turns left if right bumper pressed
170 void centerBumper(); //turns around and faces direction of most space if center bumper pressed
171 void leftBumper(); //turns right if left bumper pressed
172 bool pathKnown(float test_dist, float mem_yaw);
173 void pickTarget();
174 void emergencyUnstuck();
175
176
177 bool ranOnce[10]={0}; //Global variable for if you want something to run only once.
178 float dynVar[10]={0}; //Global variables for storing things, store anything you want
179 uint8_t dynIdx=0; //Need to start indexing this until you return to navLogic, ie if you have multiple turns in a single function
180 uint8_t stepNo=0;
181
182 float laserMem[8]={0.0};
183 float posXMem[30]={0.0};
184 float posYMem[30]={0.0};
185 int posMemIdx=0;
186
187 bool doLook=false;
188 int prevState=0;
189 uint64_t secondsElapsed = 0;
190
191 int longDistTravelCounter = 0;
192 float turnSpd = 0.4;
193
194 std::chrono::time_point<std::chrono::system_clock> timeoutClk = std::chrono::system_clock::now();
195
196 bool longDistTravel(){
197     longDistTravelCounter++;
198     if (longDistTravelCounter >= 70){
199         longDistTravelCounter = 0;
200         ROS_INFO("Long Distance achieved");
201         //std::cout << "1 meter straight travelling!" << std::endl;
202         state = 4;
203         turnSpd=0.8;
204         return true;
205     } else {
206         //std::cout << "NOT YET!" << std::endl;
207         return false;
208     }
209 }
210
211 int main(int argc, char **argv)
212 {
213     ros::init(argc, argv, "image_listener");
214     ros::NodeHandle nh;
215
```

```
    ros::Subscriber bumper_sub = nh.subscribe("mobile_base/events/bumper", 10, &bumperCallback);
    ros::Subscriber laser_sub = nh.subscribe("scan", 10, &laserCallback);
    ros::Subscriber odom = nh.subscribe("odom", 1, &odomCallback);

    ros::Publisher vel_pub = nh.advertise<geometry_msgs::Twist>("cmd_vel_mux/input/teleop", 1);

    ros::Rate loop_rate(10);

    geometry_msgs::Twist vel;

    // contest count down timer
    std::chrono::time_point<std::chrono::system_clock> start;
    start = std::chrono::system_clock::now();


    //DELETE
    //float distParam=0.8;
    state=0;

    int timeoutCtr=0;
    while(minLaserDist==std::numeric_limits<float>::infinity()&&timeoutCtr<30){
        ros::spinOnce();
        loop_rate.sleep();
        timeoutCtr++;
    }
    int scheduleSpin=240;

    while(ros::ok() && secondsElapsed <= 480) {
        ros::spinOnce();
        auto now =  std::chrono::system_clock::now();
        auto millis = std::chrono::duration_cast<std::chrono::milliseconds>(now-start).count();

        srand(secondsElapsed);



        if (prevState!=state){
            ROS_INFO("State: %d", state);
            if (state==4){
                timeoutClk=std::chrono::system_clock::now();
            }
        }
        prevState=state;



        switch(state){ //4 min for collision avoidance (random walk w/ memory), 4 min for corner travelling
            case 0:
            navLogic();
            for (int i=0; i<10; i++){
                ranOnce[i]=false;
                dynVar[i]=0;
            }
            for (int i=0; i<8; i++){
                laserMem[i]=0;
            }
            stepNo=0;
            dynIdx=0;
            break;

            case 1:
            //ROS_INFO("StepNo: %d", stepNo);
            avoid();
            break;

            case 4:
            spinAround();
            break;

            case 6:
            leftBumper();
            break;

            case 7:
            centerBumper();
            break;

            case 8:
            rightBumper();
            break;

```

```
            case 9:
            emergencyUnstuck();

            default:
            navLogic();
        }


        vel.angular.z = angular;
        vel.linear.x = linear;
        vel_pub.publish(vel);
        secondsElapsed = std::chrono::duration_cast<std::chrono::seconds>(std::chrono::system_clock::now()-start).count();
        loop_rate.sleep();
    }

    for (int i=0; i<posMemIdx; i++){
        ROS_INFO("Memory: %d, %f, %f", i, posXMem[i], posYMem[i]);
    }
    return 0;
}

void navLogic(){
    //navLogic decides which states to enter, reactive states are the first few if statements in order of priority, proactive later
    //Last state is default, move in a straight line forward
    //Need some way to remember last state
    float distParam=0.7;
    float farDistParam=1.2;
    //ROS_INFO("MinLaserDist: %f", minLaserDist);

    //before doing anything else do a 360 scan to face most space
    if (spinCount==0) {
        state = 4;
        spinCount=1;
        longDistTravelCounter = 0;
    } else if (minLaserDist<distParam || minLaserDist==std::numeric_limits<float>::infinity()){
        linear=0;
        state=1;
        longDistTravelCounter = 0;
    } else if (objectDetect[1]==true){
        linear=0.10;
        angular=0.0;
        state = 0;
    } else if (minLaserDist<farDistParam && minLaserDist>distParam) {
        linear = 0.15;
        angular = 0;
        longDistTravelCounter = 0;
    } else {
        linear=0.25;
        angular=0.0;
        state=0;
        longDistTravel();
    }

    return;
}

void spinAround(){
    //the spinAround function spins the robot 360 degrees and uses the laser scan to orient itself to the index with the most space
    //returns to state 0 when done to move forward
    //There seems to be a 15 degree overshoot after doing the full turn around. Maybe consider accounting for it?

    volatile int minIndex=0;
    //ROS_INFO("Step: %d", stepNo);
    if (timeout(40, timeoutClk)){
        ROS_INFO("Spin around timeout, invoking unstuck");
        state=9;
    }

    if(stepNo<8 && !turn(DEG2RAD(-45),stepNo)){
        return;
    } else if (stepNo<8){
        laserMem[stepNo] = laserVals[319];

        //DELETE
        //laserMem[stepNo]=minLaserDist;

        if(minLaserDist == std::numeric_limits<float>::infinity()){
            laserMem[stepNo]=0;
        }
        ROS_INFO("Mem: %d, %f", stepNo, laserMem[stepNo]);
```

```
        stepNo++;
    }
    if (stepNo==8){
        int goodIndices=0;
        int goodArr[8]={0};
        for (int i=0; i<8; i++){
            if (laserMem[i]>1.2){
                float temp_yaw=dynVar[0];
                angularAdd(&temp_yaw, DEG2RAD(-45.0*(i+1)));
                if (!pathKnown(laserMem[i], temp_yaw)){
                    goodArr[goodIndices]=i;
                    goodIndices++;
                } else ROS_INFO("Path known: %d", i);
            }
        } ROS_INFO("Good Indices: %d", goodIndices);
        if (goodIndices<2){
            if (goodIndices==0){
                state=9;
                return;
            }
            for (int i=1; i<8; i++){
                if(laserMem[minIndex]<laserMem[i]){
                    //ROS_INFO("Curr max: %f, Comp: %f", laserMem[minIndex], laserMem[i]);
                    minIndex=i;
                }
            }
        } else {
            int randIndex=rand()%goodIndices;
            minIndex=goodArr[randIndex];
            dynVar[8]=minIndex;
        }
        ROS_INFO("maxIndex: %d", minIndex);
        stepNo++;
    }
    if (stepNo==9){
        minIndex=dynVar[8];
        if(!turn(DEG2RAD(-45*(minIndex+1)),9) && minIndex!=7){
            turnSpd=0.6;
            //ROS_INFO("Ang diff: %d", -45*(minIndex+1));
            return;
        } else {
            ROS_INFO("Adding to memory, %d: %f, %f", posMemIdx, posX,posY);
            posXMem[posMemIdx]=posX;
            posYMem[posMemIdx]=posY;
            posMemIdx++;
            state=0;
            return;
        }
    }
}

void avoid (){
    //the avoid function turns left if an object is sensed on the right, turns right if an object is on the left, moves forward if in a corridor,
    //and calls the spinAround function if there's objects to the left right and center
    //returns state 0 to move forward

    if (objectDetect[0] == true && objectDetect[2] ==false){
        angular = 0.4;
        linear = 0.1;
    } else if((objectDetect[2] == true) && objectDetect[0] == false) {
        angular = -0.4;
        linear = 0.1;
    } else if (objectDetect[0]==true && objectDetect[2]==true && objectDetect[1]==false){
        linear=0.25;
        angular=0.0;
    } else if (objectDetect[0] == true && objectDetect[1] == true && objectDetect[2] == true) {
        state =4;
        turnSpd=0.4;
    } else {
        state =0;
    }
}

```

```
void rightBumper() {
    //the rightBumper function turns the robot left by 30 degrees if the right bumper is hit
    //returns state 0 to move forward
    if(!turn(DEG2RAD(30),1)){
        turnSpd=0.4;
        linear=0;
        return;
    } else {
        state=0;
        return;
    }
}

void leftBumper() {
    //the leftBumper function turns the robot right by 30 degrees if the left bumper is hit
    //returns 0 to move forward
    if(!turn(DEG2RAD(-30),1)){
        turnSpd=0.4;
        linear=0;
        return;
    } else {
        state=0;
        return;
    }
}

void centerBumper() {
    //the centerBumper function spins the robot and scans the 180 degrees behind the direction of the obstacle and faces the
    //direction withmost space
    //returns state 0 to move forward

    volatile int minIndex=1;
    //ROS_INFO("Step: %d", stepNo);
    turnSpd=0.4;

    if(stepNo==0 && !turn(DEG2RAD(-30),stepNo)){
        return;
    } else if (stepNo==0){
        ROS_INFO("Finished 45 degree turn");
        stepNo++;
    }

    if (stepNo>0 && stepNo<5 && !turn(DEG2RAD(-60),stepNo)) {
        return;
    } else if (stepNo>0 && stepNo<5){
        laserMem[stepNo] = laserVals[319];

        if(minLaserDist == std::numeric_limits<float>::infinity()){
            laserMem[stepNo]=0;
        }
        ROS_INFO("Mem: %d, %f", stepNo, laserMem[stepNo]);
        stepNo++;
    }

    if (stepNo==5){
        for (int i=2; i<5; i++){
            if(laserMem[minIndex]<laserMem[i]){
                //ROS_INFO("current max: %f, compared: %f", laserMem[minIndex], laserMem[i]);
                minIndex=i;
            }
        } //ROS_INFO("max index: %d", minIndex);
        if(!turn(DEG2RAD(60*(4-minIndex)),6) && minIndex!=5){
            turnSpd=0.6;
            return;
        } else {
            state=0;
            return;
        }
    }
    return;

}

void emergencyUnstuck(){
    if (objectDetect[1]!=false){
        angular=0.2;
        linear=0;
    } else {
        state=0;
        return;
    }
}
```

```
532
533 bool pathKnown(float test_dist, float test_yaw){
534     float margin=0.20;
535     /*
536     float displace_x=test_dist*cos(test_yaw) + posX;
537     float displace_y=test_dist*sin(test_yaw) + posY;
538     ROS_INFO("Path tested: %f, %f, %f", displace_x, displace_y, RAD2DEG(test_yaw));
539
540     for (int i=0; i<posMemIdx; i++){
541         if (displace_x<posXMem[i]+margin && displace_x>posXMem[i]-margin && displace_y<posYMem[i]+margin && displace_y>posYMem[i]-margin){
542             ROS_INFO("Path known");
543             return true;
544         }
545     }*/
546
547     for (int i=0; i<posMemIdx; i++){
548         float displace_x=posXMem[i]-posX;
549         float displace_y=posYMem[i]-posY;
550         float disp_yaw=atan2(displace_y, displace_x);
551         float displace_dist=sqrt((displace_x*displace_x)+(displace_y*displace_y));
552         if (disp_yaw<test_yaw+7 && disp_yaw>test_yaw-7){
553             if (displace_dist+margin>test_dist && displace_dist-margin<test_dist){
554                 return true;
555             }
556         }
557     }
558     return false;
559 }
560
561 void pickTarget(){
562     float max_x=0.0;
563     float max_y=0.0;
564     float min_x=0.0;
565     float min_y=0.0;
566     for (int i=0; i<posMemIdx; i++){
567         max_x=std::max(max_x, posXMem[i]);
568         max_y=std::max(max_y, posYMem[i]);
569         min_x=std::min(min_x, posXMem[i]);
570         min_y=std::min(min_y, posYMem[i]);
571     }
572     return;
573 }
574
575 bool timeout(uint64_t limit, std::chrono::time_point<std::chrono::system_clock> startPt){ //Non-blocking timer
576     //Create a time point at when your timer starts and pass it in to this function (see 'now' or 'start' for examples)
577     //Returns true if the limit has passed since the timer start was initialized
578     //Runs in milliseconds
579     auto now = std::chrono::system_clock::now();
580     int timePassed=std::chrono::duration_cast<std::chrono::seconds>(now-startPt).count();
581     if (timePassed<limit)return false;
582     else return true;
583 }
584
585 float idxToAng(int idx){ //Takes the laser index and returns the angle. Can be negative
586     idx=idx-nLasers;
587     return idx*angle_increment;
588 }
589
590 void angularAdd(float *summand, float angAddend){
591     *summand=*summand+angAddend;
592     while(*summand>M_PI){
593         *summand=*summand-2*M_PI;
594     }
595     while(*summand<0-M_PI){
596         *summand=*summand+2*M_PI;
597     }
598     return;
599 }
600
601 bool turn(float turnAmt, int ranIdx){ //CCW is +ve, turnAmt is between -pi to pi
602 //the turn function takes in an angle and spins the robot that angle
603     if (!ranOnce[ranIdx]){
604         //ROS_INFO("Init dynVar");
605         ranOnce[ranIdx]=true; //You have to remember in the CALLER to reset the corresponding ranOnce
606         dynVar[ranIdx]=yaw;
607     }
```

```
608     float goal=dynVar[ranIdx];
609     angularAdd(&goal, turnAmt);
610     if (yaw > goal+0.08 || yaw < goal-0.08) { //Generall the leniency should be angular*2/10
611         //ROS_INFO("yaw: %f, goal, %f", yaw, goal);
612         if (turnAmt<0){
613             angular=-turnSpd;
614         } else {
615             angular=turnSpd;
616         }
617         linear = 0.0;
618         return false;
619     }
620     else return true;
621 }
622
623 bool forward(float dist, int ranIdx){
624     //the forward function moves the robot forward
625     if (!ranOnce[ranIdx]){
626         ranOnce[ranIdx]=true;
627         dynVar[ranIdx]=laserVals[319];
628     }
629     float goal=dynVar[ranIdx]-dist;
630     if (laserVals[319]>goal){
631         angular=0.0;
632         linear=0.2;
633         return false;
634     }return true;
635 }
636
```