

MIE 443 Mechatronics Systems
Contest 1 Report
Team 7

Name	Student Number
Amanat Dhaliwal	1003425377
Jianfei Pan	1003948115
Qiwei Zhao	1003579950
Yuhang Song	1002946510

1.0 Problem Definition and Objective of Contest 1

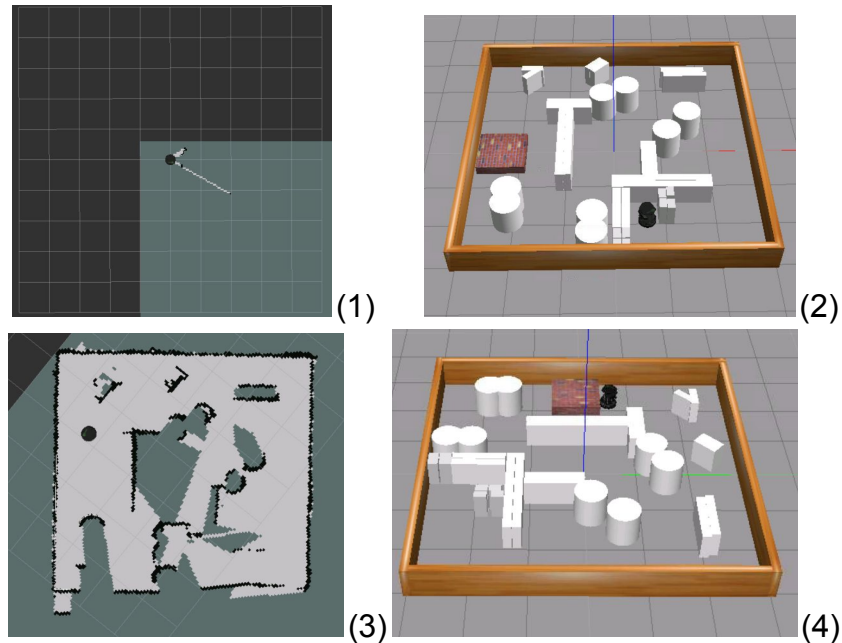


Figure 1. Overview of the Problem(1)(2) and Illustration of the Objective(3)(4)

The goal of contest 1 is to autonomously drive the Turtlebot in an unknown simulated environment, while dynamically mapping out the surrounding within 15 minutes. Specifically, the Turtlebot needs to achieve mapping, localization and planning to successfully complete the task. The turtlebot will be simulated in the Gazebo simulator using the Robotic Operating System (ROS). The Turtlebot will use the depth sensor reading from the Kinect sensor to map the surrounding environment and major obstacles. Moreover, it will use the GMapping package to achieve localization during exploration. Thus, our main goal is to utilize sensor readings and design algorithms so that the turtlebot can navigate safely in various environments and avoid major obstacles. Eventually, the environment and key obstacles are mapped and visualized in RViz in the given 15 minutes.

1.1 Requirements

The requirements for contest 1 are discussed below.

- The layout of the environment is not known to the team before the contest. Therefore, the navigation and obstacle avoidance algorithms need to be robust to cope with unknown environments.
- The contest environment is a 6 x 6 m² 3-D flat ground with walls surrounding the four edges.
- Turtlebot is only required to detect stationary obstacles since they are all static. The obstacles are randomly placed in the area with a red brick wall that cannot be seen by the Turtlebot.

1.2 Constraints

There are some constraints that the Turtlebot must meet in order to complete the tasks successfully.

1. The Turtlebot has a time limit of 15 minutes to traverse, explore, and map the unknown environment.
2. The exploration and mapping processes must be completed entirely autonomously with no human intervention.
3. Sequential control architecture without the aid of sensory readings is prohibited, which means that the navigation and exploration of the environment must be based on the real-time sensor readings.
4. The Turtlebot must not travel faster than 0.25 m/s in the environment and must not travel faster than 0.1 m/s when it is close to walls and other obstacles to ensure good mapping quality.
5. The Turtlebot must stop moving immediately when the 15 minutes time limit is reached.

2.0 Design Strategies

The main goal of contest 1 is to let the Turtlebot mapping the whole environment with the detailed outlines of the obstacles within 15 minutes. Around this goal, the team decides the following strategies that mainly use the bumpers, laser scanner and odometry to win the contest.

2.1 Obstacle Avoidance and Speed Control

In an unknown environment, the Turtlebot needs to know whether there is an obstacle ahead in order to proceed moving forward or to turn. Also, the Turtlebot needs to navigate itself and plot the full map in the Robotic Operation System(ROS). To increase the accuracy of mapping, the Turtlebot has to decrease its velocity to 0.1 m/s when it is too close to the wall. Turtlebot can run faster when it is relatively far from walls or obstacles. In order to avoid obstacles, we want the Turtlebot to stop and turn its heading to avoid a collision, once the distance between Turtlebot and the obstacles are too small. Moreover, we want Turtlebot to minorly adjust its headings if possible to maintain a safe distance to the nearby walls. All in all, the Turtlebot has to utilize its laser scan to scout the surrounding, and various other sensors to ensure appropriate orientation and speed adjustments in the unknown environment.

2.2 Scouting and Mapping

Due to the 15 minutes time limit of contest 1, the Turtlebot needs to explore the new area efficiently. Therefore the Turtlebot needs to track its location and orientation after each step. Hence, the team needs odometry to acquire that crucial information about the Turtlebot's location and orientation in the 6x6 m² contest environment. We can use the Turtlebot's specific orientation at the intersection to prevent the Turtlebot from going back to the explored area multiple times. In addition, we want to build an algorithm for

Turtlebot to deliberately choose which direction it goes at the intersection to increase the exploration efficiency. Moreover, to increase mapping efficiency, our strategy is to let the Turtlebot regularly turn 360 degrees and scout the surroundings to acquire sufficient information about the surroundings as it moves.

2.3 Invisible Wall

Lastly, here is a brick wall or some particular obstacle that the depth sensor cannot detect in the contest environment. This can cause the Turtlebot to get stuck or acquire erroneous localization information and thus devastates the mapping process. Therefore, the Turtlebot needs to use sensory readings from the bumper to check the invisible obstacle through the Boolean condition. Then, the Turtlebot should turn and move away from the invisible obstacle to avoid stagnation. Moreover, the bumper can help the Turtlebot to move out from the stuck point where it hits the obstacle at the blind spot of the depth sensor.

The other sensors, gyroscope, cliff sensors, and RGB camera are not used in contest 1. We think that using bumpers, odometry, and the laser scanner is sufficient to fulfill the requirements and win contest 1. However, it is possible to use these sensors in physical tests or other contests in the future, such as finding objects and picture identification.

3.0 Detailed Robot Design and Implementation

3.1 Sensor Design

3.2.1 Bumpers Design

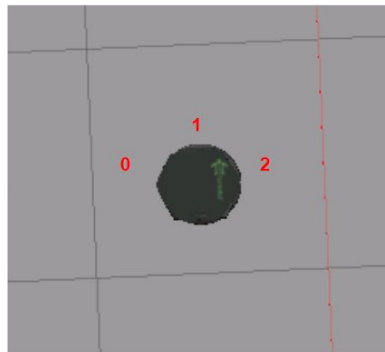


Figure 2 Position of Bumpers

There are three bumper sensors located on the left, front center, and right of the simulated turtlebot that are labeled as 0, 1 and 2 respectively in Figure 2. When any of the bumpers are depressed, the bumper's state is 1, the bumper's state is 0 otherwise.

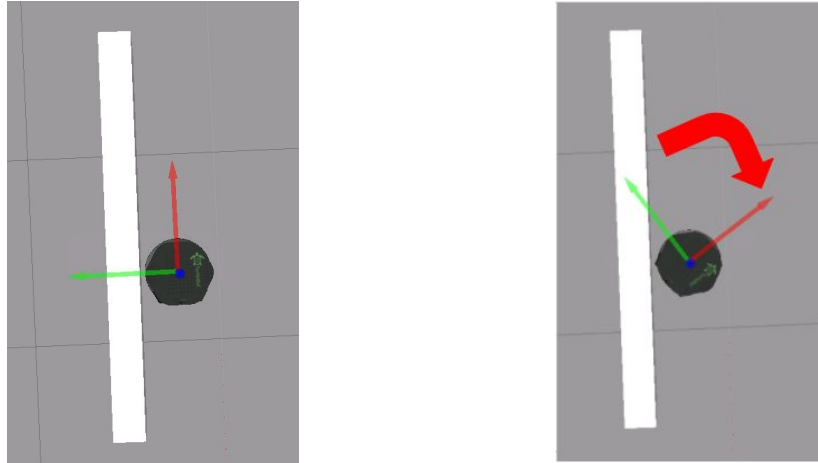


Figure 3 Turning Reaction of Bumper

The use of the bumpers is to detect collision with obstacles and turn the Turtlebot away from the obstacle. Due to the implementation of depth sensors, the Turtlebot will not bump into the visible wall. Therefore, the front bumper will only be pressed where there is an invisible wall ahead. The bumpers on the right and left sides of the Turtlebot are to detect the obstacle at the blind spot of the laser sensor. Figure 3 shows the strategy that the team implements. When the left bumper is pressed, the Turtlebot will receive the feedback from the bumpers and then turn right away from the obstacle. In this way, it minimizes the error in the mapping process and helps Turtlebot escape from obstacles.

3.2.2 Odometry Design

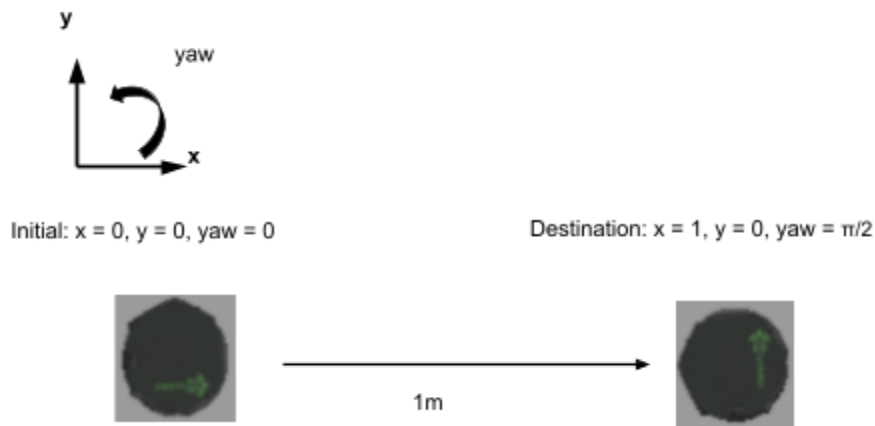


Figure 4 Change of Coordinate and Orientation of The Turtlebot

The use of odometry is to pinpoint the location of the turtlebot in the maze. The initial position of the turtlebot is $x=0, y = 0$, and $\text{yaw} = 0$. The x and y position is measured in meters, and the yaw indicates the orientation of the Turtlebot in radians. Figure 4 shows an example of the Turtlebot's movement. The Turtlebot moves forward and turns 90 degrees to the left. The odometry detects the movement of the Turtlebot and measures the value in meters and radians respectively.

The implementation of odometry is to constantly report the location of the Turtlebot after each step. However, the use of odometry along with the laser scanner allows the Turtlebot to detect the new open space and avoid going back to the explored area. For example, when the Turtlebot reaches a corner, it will rotate 360 degrees to explore all the possible open space and store the corresponding turning angle by calculating the yaw. Once the Turtlebot found out the open space where it came from, it rotated the Turtlebot to a new open space. This method ensures the Turtlebot is able to explore the whole map within 15 minutes. In the future, the team is planning to utilize odometry for path planning as well.

3.2.3 Microsoft Kinect 360 Depth Sensor Design

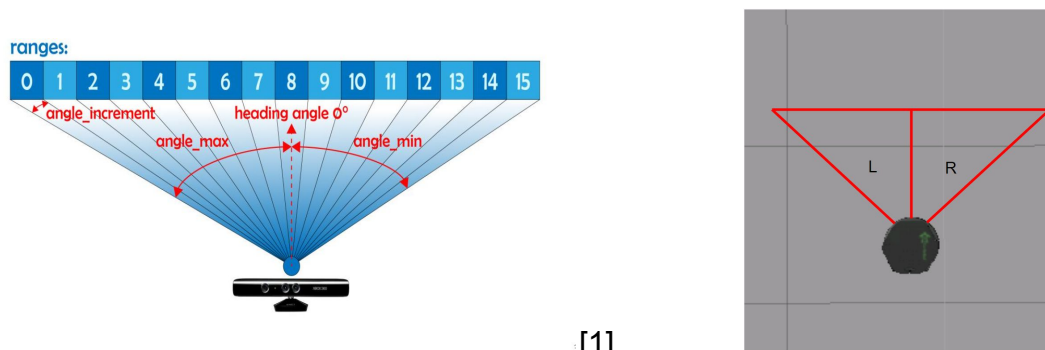


Figure 5 The Laser Array and Strategy of Using Laser Scan

The most important function that Turtlebot must have is vision. Microsoft Kinect 360 is the “eye” of the Turtlebot, as it allows the Turtlebot to detect the distance between the Turtlebot and the obstacles at the front. Specifically, Microsoft Kinect 360 provides a depth sensor and an RGB camera. The depth sensor is the most important sensor as it provides us information about the surrounding obstacles. The RGB camera will be more significant in future contests when we deal with picture identification and classification problems. The depth sensor is the laser scanner that senses a range array of around 600 indexes from left to right. Each index has a measured laser distance. The range of the viewing angle is from -12 degrees to 12 degrees. Increasing the viewing angle makes the Turtlebot have a larger range of detection, but it will decrease the accuracy of the laser scan at the two ends of the index. To precisely measure the position of the obstacles, the Turtlebot will use the smallest laser distance from the indexes. Once the Turtlebot is closing to an obstacle, it will trigger the obstacle avoidance function.

The implementation of the laser scanner is to explore the map and avoid the obstacle. Through the collaboration of laser scanners and different sensors, the Turtlebot can make different decisions under various surroundings, as mentioned previously. The other use of the laser scan is to divide the range array into left and right sections, as shown in Figure 5. If there is an obstacle close to the left side of the Turtlebot, the Turtlebot will turn slightly to the right and move forward. In that way, the Turtlebot is capable of going through a sinuous and narrow corridor.

3.2.4 Other Sensors Choice

Although the team is provided with a gyroscope and 2 cliff sensors in our Turtlebot, they are not used in our contest 1. Cliff sensors are not used because we test our algorithm entirely in the simulation, and we assume there is no change in elevation anywhere on the map. However, in the physical tests, we may add the cliff sensors to make sure the Turtlebot will not run over or fall off the stairs. Gyroscope is not used because the orientation provided by the odometry is enough to determine the orientation and location of the Turtlebot.

3.2.5 Summary

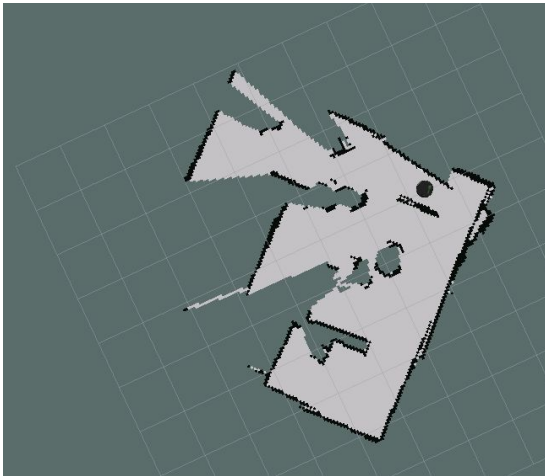


Figure 6. Probability Algorithm



Figure 7. Improved Algorithm

In a nutshell, contest 1 requires the Turtlebot to collaborate with different sensors in order to complete the mapping within 15 minutes. Initially, the team implemented the probability by applying the Bernoulli equation. The Turtlebot sometimes ran through the map and generated an acceptable result, but most of the time the Turtlebot stuck in a specific area and not exploring the full environment, as shown in Figure 6. To solve that, the team put a lot of effort into developing a new algorithm that collaborates with laser scanners and odometry to detect the new open space and prevent the Turtlebot stuck in one area. The bumper is to ensure the Turtlebot would not stick into the wall. As a result, Figure 7 shows the new algorithm that allows the Turtlebot to explore 90% of the map. In addition, the Turtlebot's environment and the obstacle outlines were explicitly depicted in RVIZ, as shown in Figure 7. The following sections have detailed explanations of how we control the Turtlebot, use boolean condition checks to distinguish different conditions, and enable actions to cope with the conditions.

4.0 Detailed Robot Design and Implementation

4.1 High-Level Controller Design

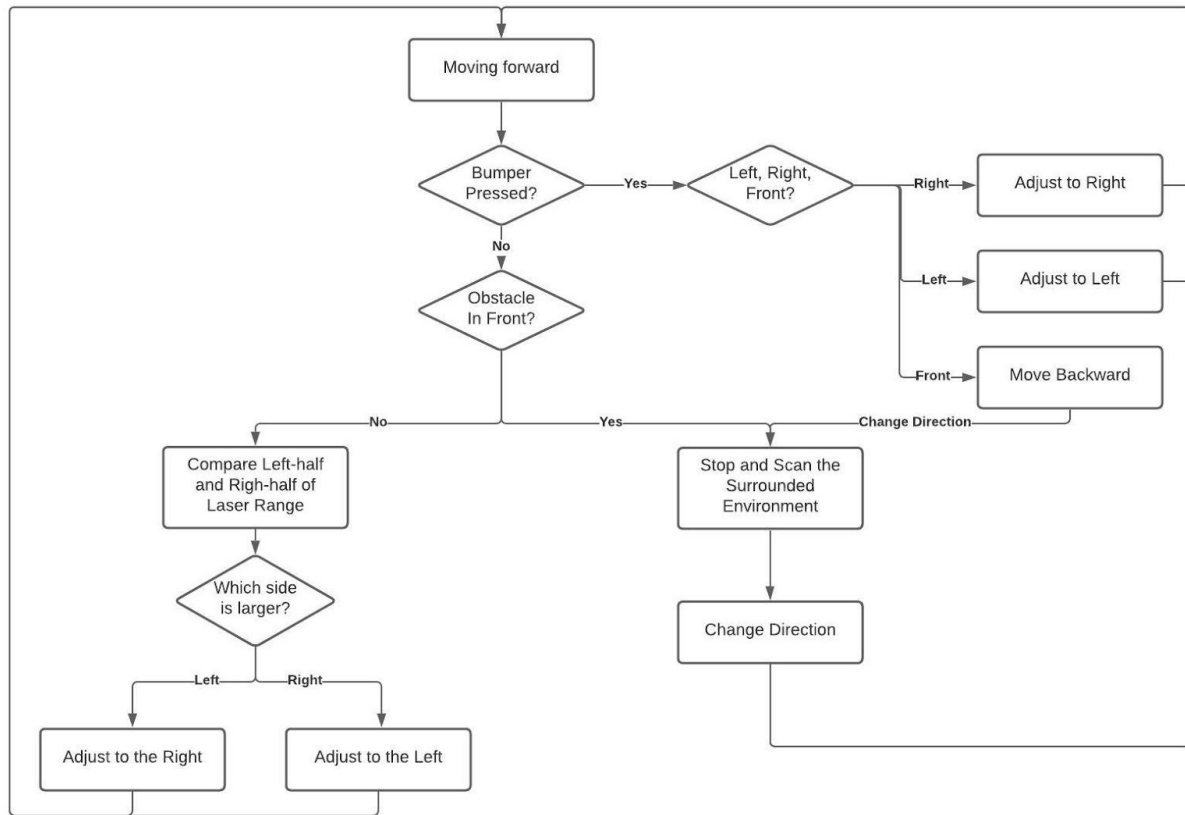


Figure 8. Reactive control flow chart

For the purposes of exploration and mapping of a given landscape using the turtlebot, the team elected to use a hybrid control architecture as demonstrated in Figure 8. Since the environment is static and does not change over time, a mix of reactive control and planning was utilized. Reactive control allows the turtlebot to navigate the environment without colliding into walls and objects using its sensors. While simultaneously using algorithms that allow the robot to make optimal decisions regarding the directional movement to ensure the environment can be mapped in a timely manner.

The reactive aspect of the controller is used to navigate the environment and avoid walls and objects using laser scan and bumper sensors. The team set up a set of rigorous and logical condition checks for the Turtlebot running in the unknown environment. When the robot is within a certain distance away from the wall, it is designed to turn the robot in a deliberately chosen direction. This ensures that the robot does not constantly make turns in a single direction which can lead to the robot moving in a circle and unable to explore the whole environment.

Purely relying on random directional movement when coming in close contact to an object is not a very efficient mode of exploration. Therefore, a separate algorithm is encoded that allows the robot to make better decisions in regards to the direction change upon reaching close to an object or wall. Rather than turning randomly, the robot would rotate in 360 direction and identify the directions that gave the robot ample space to explore. At the same time ensuring that the robot does not travel back in the direction from which it came from. This algorithm allowed the robot to plan before deciding on its new direction that gives it the most room to explore while also ensuring that the robot never unnecessarily moves back and forth in the same location wasting time.

4.2 Low-Level Controller Design/Algorithms

4.2.1 Changing Direction Algorithm

When Turtlebot detects an obstacle too close to itself, it needs to turn its heading to avoid a collision. Moreover, in complicated situations where there are more than two open paths or multiple paths non-orthogonal to each other, Turtlebot needs to detect the directions of the open paths and determine which direction to go. Therefore, based on these considerations, the team has developed a changing direction algorithm that helps Turtlebot determine the proceeding direction when it turns. This is a complicated algorithm that can be divided into two major portions, identifying possible directions and planning which direction to move forward and then move along the selected direction.

Identify possible directions

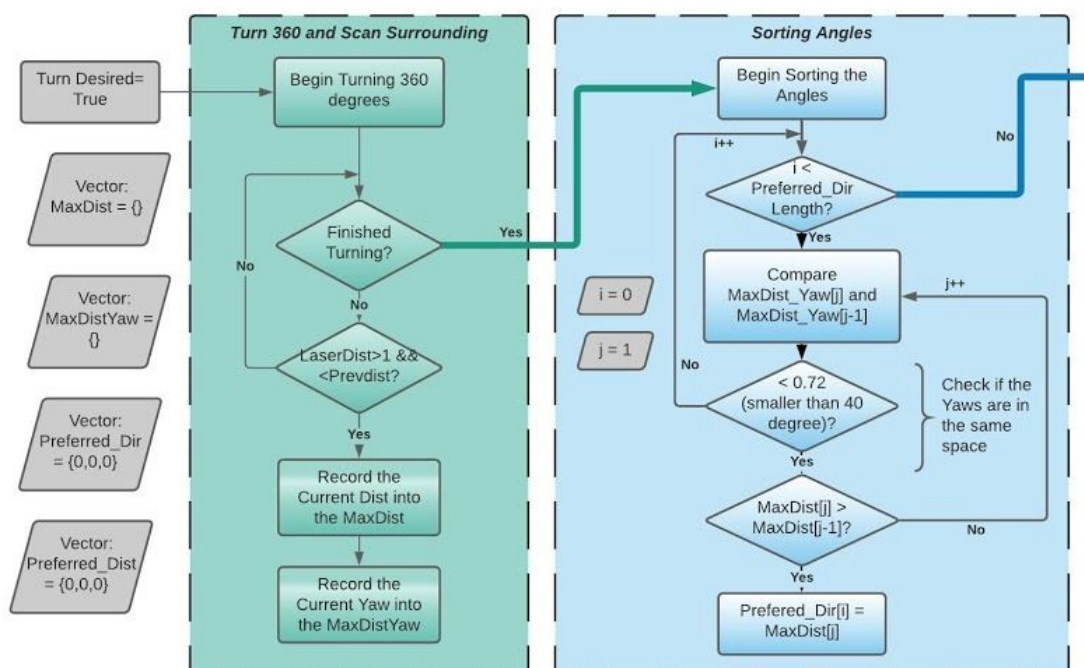


Figure 9. Flowchart of Direction Identification Algorithm

The flowchart in Figure 9 illustrates the algorithm that Turtlebot uses to identify possible open paths and their directions after it runs into a wall or an obstacle. When Turtlebot is about to turn, the Turn Desired state will be activated. As a result, Turtlebot will turn 360 and scout the surrounding first. During turning, the maximum distance values that indicate open spaces and the Turtlebot's yaw facing the direction of maximum distance will be recorded and stored. After turning 360, a vector of maximum distances and their corresponding orientations are acquired. Based on the layout and shape of the obstacles, we usually obtain more maximum distance values than the actual number of open spaces. Thus, the results will be fed into the Sorting Angles mode, where the true directions of open spaces are estimated based on the difference between consecutive yaw values. We will count the very close yaws as one direction that is determined by the maximum distance in these yaws. In this way, we reduce the number of measured open spaces and interpolate the real open spaces that are stored in a new vector. Then, Turtlebot will decide which direction it will turn to based on the vector of interpreted open spaces in the direction planning and movement algorithm, in which Turtlebot will select and turn to an appropriate direction before moving in that direction of open space.

Planning the direction to move

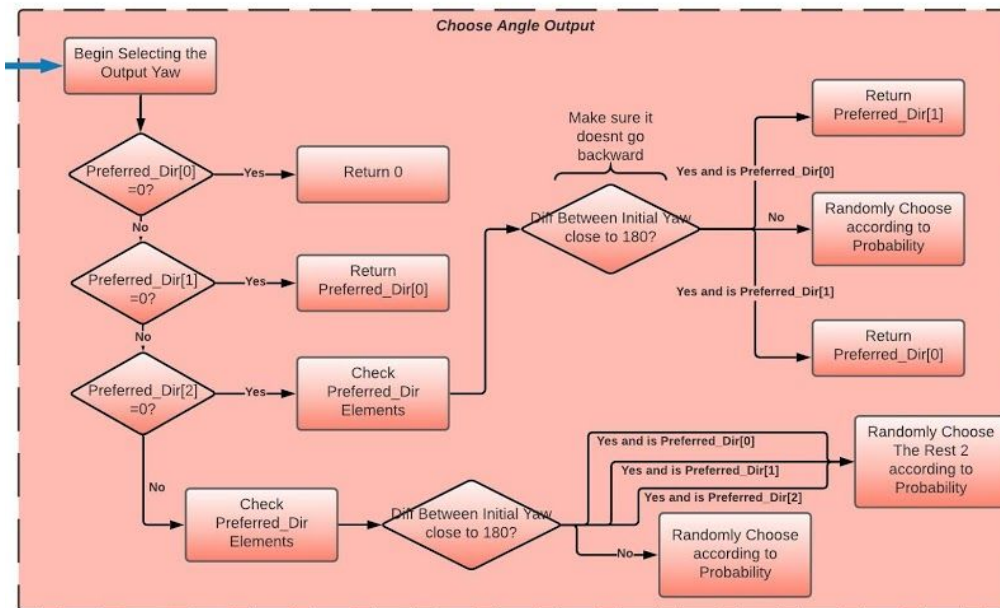


Figure 10. Flowchart of Angle Selection Algorithm

Figure 10 shows the implementation of the Angle Selection algorithm that chooses the best direction for the robot to move to in order to achieve continuous exploration of the map. This algorithm checks the number of available open spaces (identified from the previous algorithm), determines the direction, and moves to the desired orientation

before moving forward. However, there are different scenarios that need to be considered.

Firstly, in the case of only one possible open space, there will be only 1 value in the open space vector. The algorithm will output only one direction, which Turtlebot will adjust its yaw to. Once Turtlebot has changed to the desired yaw, it will start moving forward.

Secondly, in the case of two possible open spaces, the algorithm will check if one of the directions is backward and outputs the direction that is not the backward direction. Thus, Turtlebot avoids going backward and thus improves the exploration efficiency, since Turtlebot will not waste time traversing back and forth along the same path.

Thirdly, in the case of three open spaces, the algorithm will check if one of the directions is backward. Then, similarly, the algorithm will randomly output a direction that is not backward. Then, Turtlebot moves to the randomly selected open space. In the case of more than 3 possible open spaces, the algorithm will select the first 3 open space directions that Turtlebot encounters, which works well in various test environments but is something we can improve in future revisions.

4.2.2 Scouting Algorithm

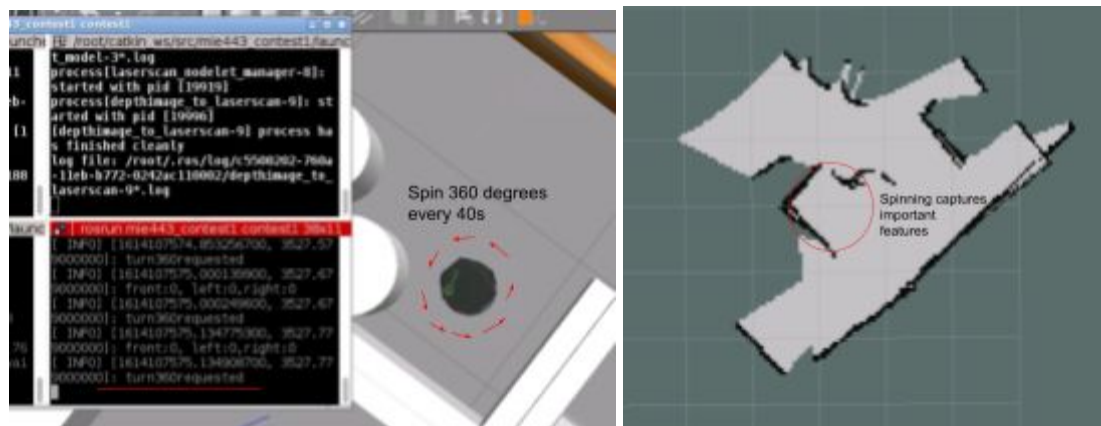


Figure 11. Scouting Algorithm Demonstration

In order to increase mapping efficiency and capture the surroundings effectively, the team designed the scouting algorithm that enables Turtlebot to turn 360 degrees every 60 seconds using the time elapsed variable. Figure 11 demonstrates the effects of turning 360 on the mapping quality in RViz. It shows that the 360 turn allows the RViz to create a much smoother map, with the boundaries of each object defined more clearly. Without this implementation, the RViz generated map did have well-defined object boundaries and missed certain objects entirely on the map. This algorithm also allows the Turtlebot to see objects that it otherwise would have missed during a random exploration of the map. When the scouting state is activated, Turtlebot will record its current yaw value and then turn at a constant pace until a 2π difference is reached.

Then, Turtlebot will stop for a few seconds to process and gather the data before it starts moving in the same direction as before the scouting state is activated.

4.2.3 P-controller for minor Direction Adjustments

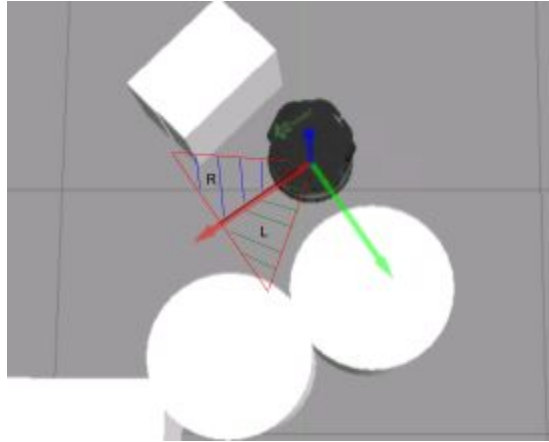


Figure 12. P-controller Algorithm Visualization

During our test, the team has found out that Turtlebot needs to turn multiple times in the meandering narrow path like one illustrated in Figure 12. In this path, Turtlebot will waste a lot of time due to intermittent stoppage and turning. Especially considering the complicated turning algorithm designed by our team, the Turtlebot can waste a majority of available time. Therefore, in order to mitigate this problem and enable Turtlebot to travel smoothly and quickly through the narrow path, the team used the idea of a P-controller in designing our forward movement algorithm. In a P-controller, the sensory outputs are measured, fed back to the inputs, and compared with the desired outputs. In our cause, we want to ensure that Turtlebot moves in the middle of a narrow path, which means that the sum of readings of the left spectrum and right spectrum should be approximately equal.

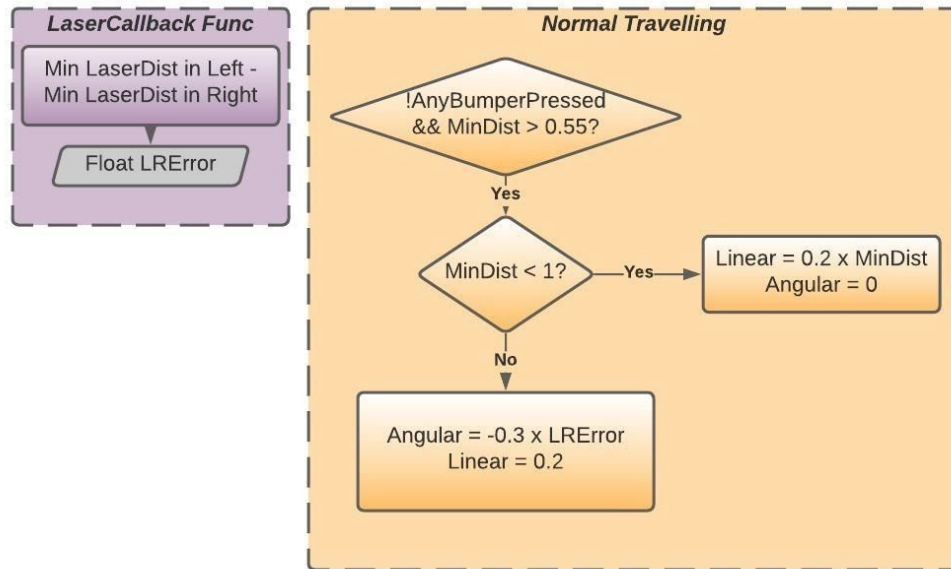


Figure 13. Flowchart of P-controller Algorithm

The team achieves gradual and continuous adjustments of Turtlebot's direction based on the sensor readings. Figure 13 shows the flowchart that demonstrates the basic operation and logic of our P-controller algorithm. Specifically, our algorithm takes in and compares the sum of the left portion and right portion of the laser readings vector. If the magnitude on the left portion is slightly higher than the right portion, this indicates that Turtlebot potentially is closer to the right than the left. In this case, Turtlebot will adjust itself by turning slightly to the left in order to reduce the difference in the sum of readings on the left and right sides of the Turtlebot.

This algorithm has proven to work effectively, as the Turtlebot is able to continuously follow the narrow and relatively sinuous path that is constrained by the walls or obstacles with minor adjustments in its direction. This has dramatically improved the efficiency and reduced the time spent in traversing the curved paths by eliminating the need for frequently checking the desired directions to turn when it is too close to obstacles.

4.2.4 Invisible Wall Algorithm

Based on the problem requirement, there will be some invisible walls that are not high enough to be detected by Turtlebot. Therefore, the team has come up with algorithms to deal with all possible scenarios that can occur when Turtlebot pumps into the invisible walls. Different scenarios and solutions are summarized in Table x below.

	Scenarios	Solution
--	-----------	----------

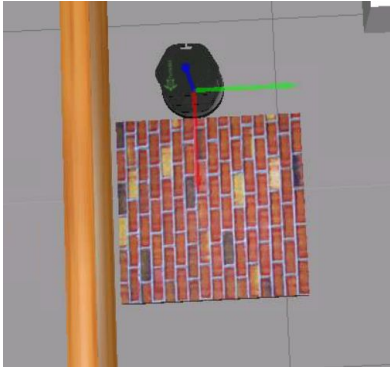
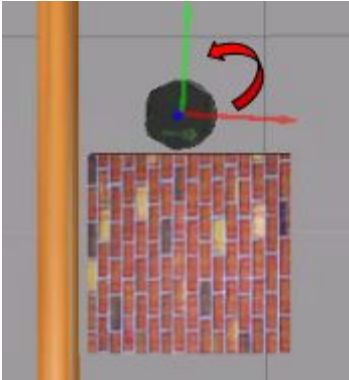
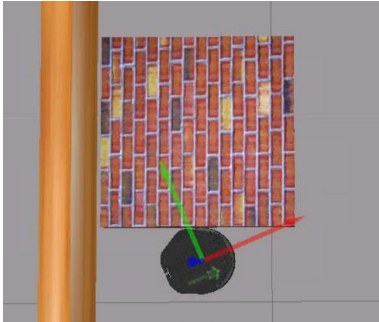
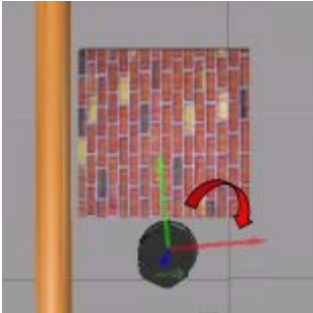
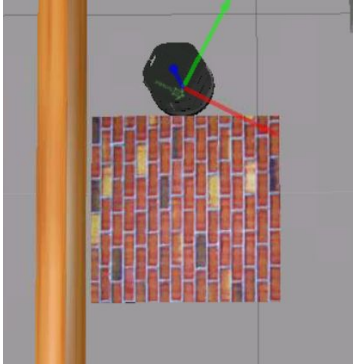
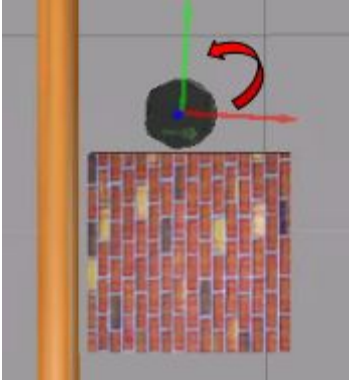
Front Bumper		
Left Bumper		
Right Bumper		

Table 14. Summary of Possible Scenarios and Solutions

The three possible scenarios are dealt with accordingly. When the front bumper is pressed, Turtlebot will turn randomly to the left or right. When the left bumper is pressed, Turtlebot will turn to the right to escape from the obstacle. Lastly, when the right bumper is pressed, Turtlebot will turn to the left to change its heading along the wall. Turtlebot will continuously turn until no bumper is being pressed and then move forward. The specific algorithm that we have developed to solve the invisible wall problem is illustrated in the flowchart in Figure 15.

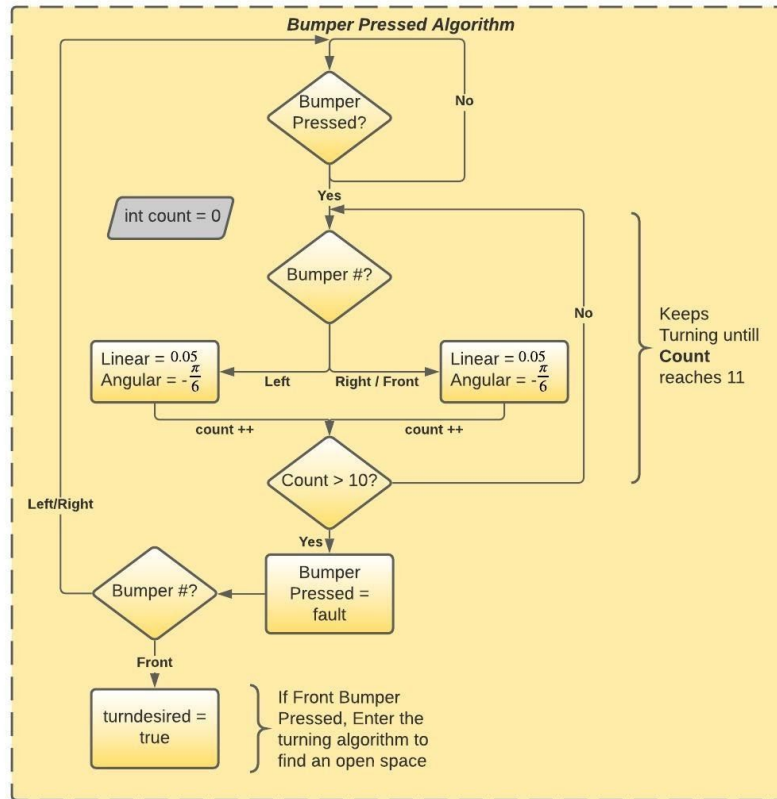


Figure 15 Invisible Wall Algorithm

As shown in the flowchart, whenever bumpers are pressed, the bumper pressed state will be activated and the specific bumper that is pressed is identified. Based on which bumper being pressed, Turtlebot will turn in the desired manner as explained before. We added a counting algorithm to ensure that the adjustment continues for several steps in the while loop so that Turtlebot can fully change its heading after the counts.

4.2.5 Bernoulli Distribution

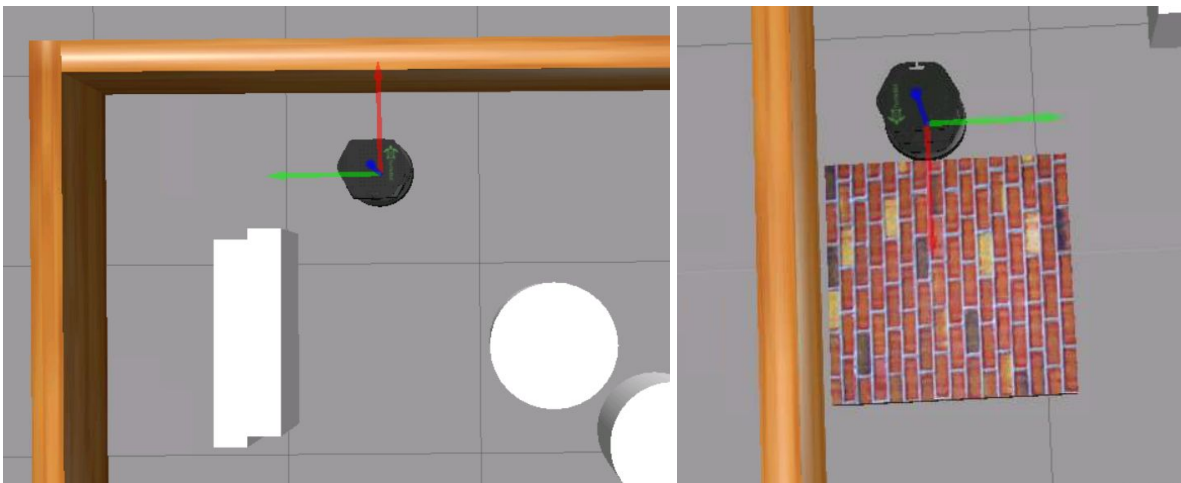


Figure 16. Three Open spaces (Left) and Invisible Wall (Right)

The team has found that there are two cases shown in Figure 16, when Turtlebot needs to decide which direction to turn. Our turning mechanism has been discussed already in the previous session, in which we eluded to a method of random turning when there are more than 1 open space around the Turtlebot. Moreover, when Turtlebot hits the invisible wall and the front bumper is activated, the distance readings from the laser scan still indicate an open space ahead of it. In these cases, randomization needs to be introduced into the algorithm to let the Turtlebot decide which direction to turn due to limited information about the environment.

$$f(k, p) = \begin{cases} p & \text{if } k = 1, \\ 1 - p & \text{if } k = 0. \end{cases}$$

[2]

Figure 17. Probability Mass function of Bernoulli Distribution

Therefore, Bernoulli distribution is used to introduce randomization into our algorithms. Bernoulli distribution is a common discrete probability distribution that provides boolean-valued outcomes. This is extremely important in our application since we only need Turtlebot to turn either left or right, which are exactly two possible outcomes.

```
std::random_device rd;
std::mt19937 mt(rd());
std::bernoulli_distribution dist(0.55); // this is the chance of getting true, between 0 and 1;
```

Figure 18. Bernoulli Distribution Implementation in the Code

The Bernoulli distribution is implemented in our algorithm as shown in Figure 18, where 0.55 indicates the probability of outputting true that instructs Turtlebot to turn left. The probability method has its pros and cons. Thus, when the team finishes the algorithm that enables Turtlebot to always turn and explore new frontiers, the probabilistic approach may be integrated or replaced with a more deterministic method.

5.0 Future Recommendations

If the team had more time to work on algorithms for robot exploration, then we would have liked to implement the following changes.

We will use positional information provided by the odometer to identify and store parts of the map that have already been explored to aid the process of direction selection when Turtlebot is turning. Thus, Turtlebot would choose the direction that allows it to explore the areas that it has not yet explored, leading to a more efficient exploration process.

We will create an algorithm that instructs the robot to perform a 360 turn-based on the current position on the map instead of a random time variable. This could be done using

the odometer on the robot to ensure that all 360 turns performed on the map are occurring at unique positions. Meaning the robot is not constantly performing 360 at similar locations, which wastes valuable testing time and provides no additional information. The odometer would allow us to recognize positions as being unique and performing a 360, and allowing us to generate a map of the environment much more efficiently.

We will add the stuck function in order to prevent the turtlebot from getting stuck in corners or walls. We have noticed that when we place the bot very close to the wall, there is the potential of getting stuck at the back end while turning around.

Lastly, the team would incorporate header files in our main code instead of adding them all on the same file, in order to ensure that the programming code looks much cleaner and easier to understand and debug in the future. This will also allow the team to implement new codes without having to edit the existing main function files, which makes it much easier to implement new codes or reuse codes for different projects.

5.0 Contribution Table

The contribution table is summarized below.

Section	Student Names			
	Eugene Song	Amanat Dhaliwal	Richard Zhao	Jianfei Pan
The problem definition/objective of the contest			MR,RD	MR, RD
Strategy		ET	RD,MR	MR, RD
Detailed Robot Design and Implementation	RD, MR, ET	RS,RD,ET	RD,MR	MR, RD, RS

Future Recommendation		RS,RD,ET		MR
Coding	RD, MR, ET	MI,RS,RD	MI,TS,RS	MI, TS, RS
All		ET		FP, CM, ET

Fill in abbreviations for roles for each of the required content elements. You do not have to fill in every cell. The “all” row refers to the complete report and should indicate who was responsible for the final compilation and final read through of the completed document.

RS – research

RD – wrote first draft

MR – major revision

ET – edited for grammar and spelling

MI – minor revision in codes

TS – test and debug the codes

FP – final read through of complete document for flow and consistency

CM – responsible for compiling the elements into the complete document

OR – other

6.0 References

[1] "weblogin idpz | University of Toronto", Q.utoronto.ca, 2021. [Online]. Available: <https://q.utoronto.ca/courses/198595/files/folder/Tutorials?preview=11980879>. [Accessed: 24- Feb- 2021].

[2] J. Munkhammar, J. Widen, P. Grahn, and J. Ryden, “A Bernoulli distribution model for plug-in electric vehicle charging based on time-use data for driving patterns,” in 2014 IEEE International Electric Vehicle Conference (IEVC), 2014, pp. 1–7, doi: 10.1109/IEVC.2014.7056224.

7.0 Appendix

A1) Full C++ Code

```

#include <ros/console.h>
#include "ros/ros.h"
#include <geometry_msgs/Twist.h>
#include <kobuki_msgs/BumperEvent.h>
#include <sensor_msgs/LaserScan.h>

#include <stdio.h>
#include <cmath>
#include <vector>
#include <chrono>
#include <algorithm>

#include<nav_msgs/Odometry.h>
#include<tf/transform_datatypes.h>

#include <random>
#include <iostream>

#define N BUMPER (3)
#define RAD2DEG(rad) ((rad) *180./M_PI)
#define DEG2RAD(deg) ((deg) *M_PI /180.)

enum Bumper
{
    Left = 0,
    Center = 1,
    Right = 2,
};

uint8_t
bumper[3]={kobuki_msgs::BumperEvent::RELEASED,kobuki_msgs::BumperEvent::RE
LEASED,kobuki_msgs::BumperEvent::RELEASED};

float minLaserDist=std::numeric_limits<float>::infinity();
float minDistL = 100.0;
float minDistR = 100.0;
float angleadjust = 0.0;
float lavg;
float ravg;

```

```
float turning_prob;
float sum;
float lasterror;
float lastavgerror;
float minDistabs = 0;
float prevtime;
std::vector<float> PosXList;
std::vector<float> PosYList;
std::vector<bool> TurnLeftList;
int32_t nLasers=0,desiredNLasers=0,desiredAngle=12;

bool isleftbumperpressed = false;
bool right_bumper_pressed = false;
bool front_bumper_pressed = false;
bool left_bumper_pressed = false;
bool turn360requested = false;
bool turndesired = false;
bool turn = false;
bool maxreach = false;
bool turndesiredold = false;
bool pause3 = false;

int counting_backward_step = 0;
int countsteps = 0;
const int stpsize = 3;

float angular =0.0;
float linear =0.0;
float distinit = 0.0, distprev = 0.0;
float posX=0.0, posY=0.0, yaw =0.0, yawinit = 0, yawdiff = 0, yawprev = 0;
float maxdist = 0.0;
float LError = 0.0;
float changeangle = 0.0;
float currtime = 0;
float LRavgerror = 0.0;
int maxElementIndex = 0;
float maxElement = 0;
```

```

std::random_device rd;
std::mt19937 mt(rd());
std::bernoulli_distribution dist(0.55); // this is the chance of getting
true, between 0 and 1;
std::vector< float > distmax;
std::vector< float > distmaxYaw;
std::vector<float> preferred_dir;
std::vector<float> preferred_dist;
bool turnleft = false;

int Checksimilarity(std::vector<float> &PosXList, std::vector<float>
&PosYList, float &posX, float&posY){
    for(uint32_t i = 0; i<PosXList.size(); ++i) {
        if (fabs(posX-PosXList[i]) < 0.2 && fabs(posY-PosYList[i]) <
0.2){
            return i;
        }
    }
    return PosXList.size();
}

void bumperCallback(const kobuki_msgs::BumperEvent::ConstPtr& msg)
{
    //fill with your code
    // Access using bumper[kobuki_msgs::BumperEvent::{}] LEFT, CENTER, or
RIGHT
    bumper[msg->bumper] =msg->state;

    //uint8_t leftState=bumper[kobuki_msgs::BumperEvent::LEFT];

    //kobuki_msgs::BumperEvent::PRESSED if bumper is
pressed, kobuki_msgs::BumperEvent::RELEASED otherwise
}

int avoidgoingbackward(std::vector<float> &preferred_dir, float &yawinit){
    ROS_INFO("Checking if going backward");
    for (uint32_t i=0; i<preferred_dir.size(); i++){
        if(fabs(fabs(preferred_dir[i]-yawinit)-M_PI)<0.80){ // check if
its the backward direction
            return i;

```

```

    }
}

return preferred_dir.size();
}

float sortingangles(std::vector<float> &distmaxYaw, std::vector<float>
&distmax, std::vector<float> &preferred_dir, std::vector<float>
&preferred_dist, float &yawinit){
    int cnt = 1;
    if (distmax.size()>0){
        preferred_dir[0] = distmaxYaw[0];
        preferred_dist[0] = distmax[0];
    }
    for (uint32_t i=0; i<3; i++){
        if (cnt == distmax.size()){
            if (fabs(distmaxYaw[cnt-1] - distmaxYaw[cnt-2])>0.72&&
preferred_dist[i-1] != distmax[cnt-1]){
                preferred_dist[i] = distmax[cnt-1];
                preferred_dir[i] = distmaxYaw[cnt-1];
            }
        }
        else if(cnt < distmaxYaw.size()){
            while((fabs(distmaxYaw[cnt] - distmaxYaw[cnt-1])<0.72) && cnt
< distmaxYaw.size()){
                if(distmax[cnt]>preferred_dist[i]&&
cnt<distmaxYaw.size()){
                    preferred_dist[i] = distmax[cnt];
                    preferred_dir[i] = distmaxYaw[cnt];
                }
                cnt++;
            }
            cnt++;
        }
    }

    ROS_INFO("preferred_dir      1:%f,2:%f,3:%f,      initial:
%f",preferred_dir[0],preferred_dir[1],preferred_dir[2],yawinit);
    int back = avoidgoingbackward(preferred_dir , yawinit); // check if any
direction is the backward direction
    if (preferred_dir[0] == 0){
        return 0;

    }else if(preferred_dir[1] == 0){ // There is only one open space

```

```

        ROS_INFO("Only One Way to Go!");
        return preferred_dir[0];

    }else if(preferred_dir[1]!=0 && preferred_dir[2] == 0) { // There are
two open spaces
        if(back!= preferred_dir.size()){
            ROS_INFO("In a Corner"); //corner check, make sure it will not
turn back (180 degrees)
            return preferred_dir[!back];
        }else{
            ROS_INFO("Turning Randomly");
            return (turnleft == 1 ? preferred_dir[1] : preferred_dir[0]);
        }
    }else{
        ROS_INFO("3 open spaces");
        if(back!=preferred_dir.size()){
            ROS_INFO("Eliminate 180 one"); // Not turning back
            switch (back){
                case 0: return (turnleft == 1 ? preferred_dir[1] :
preferred_dir[2]); // Randomly chosen a direction to turn
                case 1: return (turnleft == 1 ? preferred_dir[0] :
preferred_dir[2]);
                case 2: return (turnleft == 1 ? preferred_dir[0] :
preferred_dir[1]);
                ROS_INFO("Switch");
            }
        }else{
            return (turnleft == 1 ? preferred_dir[0] : preferred_dir[2]);
// it will randomly turns to first or the thrid direction when none is 180
        }
    }
}

void laserCallback(const sensor_msgs::LaserScan::ConstPtr& msg)
{
    //fill with your code
    minLaserDist=std::numeric_limits<float>::infinity();
    minDistR = 100;
    minDistL = 100;

```

```

    lavg = 0.0;
    ravg = 0.0;
    nLasers=(msg->angle_max-msg->angle_min) /msg->angle_increment;
    desiredNLasers=DEG2RAD(desiredAngle)/msg->angle_increment;
    sum = 0.001;
    int i = 0;

    //ROS_INFO("Size of laser scan array: %i and anglemin: %f and
anglemax: %f", nLasers, msg->angle_max,msg->angle_min);

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;++la
ser_idx){
        minLaserDist=std::min(minLaserDist,msg->ranges[laser_idx]);
    }
}
else {
    for(uint32_t laser_idx=0;laser_idx<nLasers;++laser_idx) {
        minLaserDist=std::min(minLaserDist,msg->ranges[laser_idx]);
    }
}
for(uint32_t laser_idx=20;laser_idx<nLasers/2;++laser_idx){
    minDistL=std::min(minDistL,msg->ranges[laser_idx]);
    if (!std::isnan(msg->ranges[laser_idx])){
        sum += msg->ranges[laser_idx];
        i++;}
}
if(i>0){
    lavg = sum/(i);}
sum = 0.0, i = 0;
for(uint32_t laser_idx=nLasers/2;laser_idx<nLasers-20;++laser_idx) {
    minDistR=std::min(minDistR,msg->ranges[laser_idx]);
    if (!std::isnan(msg->ranges[laser_idx])){
        sum += msg->ranges[laser_idx];
        i++;}
}
if(i>0){
    ravg = sum/(i);}

```



```

        sum = 0.0, i = 0;
        LError = minDistL - minDistR;
        LRavgerror = lavg - ravg;
        minDistabs = msg -> ranges[nLasers/2];
    }

void odomCallback(const nav_msgs::Odometry::ConstPtr&msg)
{
    //fill code
    posX=msg->pose.pose.position.x;
    posY=msg->pose.pose.position.y;
    yaw=tf::getYaw(msg->pose.pose.orientation);
    tf::getYaw(msg->pose.pose.orientation);
    // ROS_INFO("Position: (%f,%f) Orientation:%frad or%fdegrees.", posX,
posY,yaw,RAD2DEG(yaw));
}

float distcal(float px1, float py1, float px2, float py2){
    return sqrt(pow(fabs(px2-px1),2) + pow(fabs(py2-py1),2));
}

//bool any_bumper_pressed()
//{
//    return bumperRight || bumperCenter || bumperLeft;
//}

int main(int argc, char **argv)
{
    ros::init(argc, argv, "maze_explorer");
    ros::NodeHandle nh;

    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();
uint64_t secondsElapsed = 0;

float angular = 0.0;
float linear = 0.0;
int bumper_check = 0;

while(ros::ok() && secondsElapsed <= 900) {
    ros::spinOnce();
    //fill with your code
    // Check if any of the bumpers were pressed.
    bool any_bumper_pressed=false;
    for(uint32_t b_idx=0; b_idx<N BUMPER; ++b_idx)
    {
        any_bumper_pressed|= (bumper[b_idx]
==kobuki_msgs::BumperEvent::PRESSED);
    }

    if (any_bumper_pressed) {

        front_bumper_pressed|= (bumper[1]
==kobuki_msgs::BumperEvent::PRESSED);

        left_bumper_pressed|= (bumper[0]
==kobuki_msgs::BumperEvent::PRESSED);

        right_bumper_pressed|= (bumper[2]
==kobuki_msgs::BumperEvent::PRESSED);
    }

    ROS_INFO("front:%i, left:%i,right:%i", front_bumper_pressed,
left_bumper_pressed,right_bumper_pressed);
    if (PosXList.size()>20) {
        PosXList.clear();
        PosYList.clear();
        TurnLeftList.clear();
    }
}

```

```

if(pause3){
    ROS_INFO("Pause 3 seconds");
    if(secondsElapsed - currtime < 3){
        angular = 0;
        linear = 0;    }else{
            pause3 = false;
        }
        goto endofop;}
if (turn360requested){

    while(yaw < yawprev){
        yaw += 2*M_PI;
    }
    yawdiff = fabs(yaw - yawinit);
    ROS_INFO("turn360requested");
    if ((2*M_PI - yawdiff) > 0.1){
        linear = 0;
        angular = M_PI/12;
        yawprev = yaw;
    }
    else{
        currtime = secondsElapsed;
        angular = 0;
        yawprev = 0;
        turn360requested = false;
        pause3 = true;
    }
    goto endofop;
}

if (changeangle !=0){

    while(yaw < yawprev){
        yaw += 2*M_PI;
    }
    ROS_INFO("changeangle");
    if(fabs(changeangle - yaw + 2*M_PI )> 0.1 && yaw < 15){
        angular = M_PI/6;
        linear = 0.0;
    }
}

```

```

        yawprev = yaw;

    }else{
        changeangle = 0;
        currtime = secondsElapsed;
        pause3 = true;
        angular = 0;
        linear = 0.0;
        yawprev = 0;
        turn360requested = false;
    }

    goto endofop;
}

if (!turndesired && !turndesiredold){
    turnleft = dist(mt);
}

if (turndesired && !any_bumper_pressed){
    ROS_INFO("turndesired");
    if (minLaserDist - distprev > 0.05 || minLaserDist < 1
||minLaserDist>6.5){
        distprev = minLaserDist;
        linear = 0;
        if (turnleft){
            angular = M_PI/6;
        }else if(!turnleft){
            angular = - M_PI/6;
        }

        ROS_INFO("Adjusting:%f, prev:%f , angle: current:%f,
init:%f", minLaserDist,distprev,yaw,yawinit);
    }
    else{
        linear = 0;
        angular = 0;
        turndesired = false;
        pause3 = true;
        ROS_INFO("Adjustment finished:%f", minLaserDist);
    }
}

```

```

        goto endofop;}

    if(!turn){
        yawinit = yaw;
    }

    if(secondsElapsed % 100 == 0 && secondsElapsed >0 && !turndesired
&& !turndesiredold){    /// control the time of spinning
        //        ROS_INFO("20 Seconds Passed, Turn 360");
        turn360requested = true;

        goto endofop;
    }

    if (turndesiredold && !any_bumper_pressed){
        ROS_INFO("turndesired_old");

        if(turnleft){
            while(yaw < yawprev && (fabs(yaw-yawinit) > 0.01)){
                yaw += 2*M_PI;
            }else{

                while(yaw > yawprev && (fabs(yaw-yawinit) > 0.01)){
                    yaw -= 2*M_PI;
                }

                yawdiff = fabs(yaw - yawinit);
                ROS_INFO("Adjusting:%f, prev:%f , angle: current:%f, init:%f",
minLaserDist,distprev,yaw,yawinit);
                if ((2*M_PI - yawdiff) > 0.1){
                    if (fabs(M_PI - yawdiff)<M_PI/6 || minLaserDist <1
||minLaserDist>6.5){
                        distprev = minLaserDist;
                        linear = 0;
                        if (turnleft){
                            angular = M_PI/8;
                        }else if(!turnleft){

```

```

        angular = - M_PI/8;
    }

    yawprev = yaw;
    currtime = secondsElapsed;
}
else{
    linear = 0;
    angular = 0;
    turndesiredold = false;
    pause3 = true;
    yawprev = yaw;
    ROS_INFO("Adjustment finished:%f", minLaserDist);
}}else{
    linear = 0;
    angular = 0;
    yawprev = yaw;
    turndesiredold = false;
    distinit = minLaserDist;
    distprev =distinit;
    turndesired = true;
}
goto endofop;
}

if (front_bumper_pressed){ //bumped go backward
ROS_INFO("front_bumper");
    linear = -0.05;
    angular = M_PI/6;
    counting_backward_step++;
    //          ROS_INFO("anguar:%f   linear:%f   MinLaserDist:%f
counting_backward:%i   BACK   BACK   ",   angular,   linear,minLaserDist,
counting_backward_step);

    if (counting_backward_step==14){ // going backward steps
        counting_backward_step=0;
        front_bumper_pressed = false;
        turndesiredold = true;

```

```

        distinit = minLaserDist;
        distprev =distinit;
        goto endofop;
    }
}

else if (left_bumper_pressed){ // bumped turn right
    ROS_INFO("left_bumper");
    linear = -0.05;
    angular = -M_PI/6;
    counting_backward_step++;
    //          ROS_INFO("anguar:%f  linear:%f  MinLaserDist:%f
counting_backward:%i          TURNNING          RIGHT          ",          angular,
linear,minLaserDist,counting_backward_step);
//

    if (counting_backward_step==10){ // going backward steps
        counting_backward_step=0;
        left_bumper_pressed = false;
    }
}

else if (right_bumper_pressed){ //bumped turn left
    linear = -0.05;
    angular = M_PI/6;
    counting_backward_step++;
    //          ROS_INFO("anguar:%f  linear:%f  MinLaserDist:%f
counting_backward:%i          TURNNING          left          ",          angular,
linear,minLaserDist,counting_backward_step);

    if (counting_backward_step==10){ // going backward steps
        counting_backward_step=0;
        right_bumper_pressed = false;
    }
}
}

```

```

else if(!any_bumper_pressed){ // go forward
    ROS_INFO("go forward");

    if (minLaserDist>0.55 && minLaserDist<6.5){

        if (minDistabs < 0.7){
            linear = 0.15;
            angular = -0.3*LRerror;

        }else{
            linear = 0.2;
            angular = -0.3*LRerror;}

        if(angular >M_PI/6 || angular< -M_PI/6){
            angular = 0;
        }

        if(linear >0.2 ){
            linear = 0.2;
        }
        // lastavgerror = LRavgerror;
        // prevtime = secondsElapsed;
        // ROS_INFO("anguar:%f linear:%f LRavgError:%f dError:%f
Lavg:%f Ravg: %f ", angular, linear,LRavgerror,dErroravg,lavg,ravg);

    }

    else if ((minLaserDist<0.55 || minLaserDist>6.5) &&
secondsElapsed >1){
        turndesiredold = true;
//Try

        int check = 0;
        if(PosXList.size()>0){
            check = Checksimilarity(PosXList,PosYList,posX,posY);
            if(check < PosXList.size()){
                turnleft = (turnleft == TurnLeftList[check] ?
!turnleft : turnleft);

                ROS_INFO("Same Place");
            }

```



```

        }
        PosXList.push_back(posX);
        PosYList.push_back(posY);
        TurnLeftList.push_back(turnleft);

//

        distinit = minLaserDist;
        distprev =distinit;
        lasterror = 0;
        yawinit = yaw;
        yawprev = yaw;
        //    if(yawinit < 0){
        //    yawinit = yawinit + 2*M_PI; // start saving the yawinit
        //}
    }
}

endofop:
vel.angular.z = angular;
vel.linear.x = linear;
vel_pub.publish(vel);
if (angular !=0){
    turn =true;
}else{
    turn = false;
}
// The last thing to do is to update the timer.
secondsElapsed =
std::chrono::duration_cast<std::chrono::seconds>(std::chrono::system_clock
::now()-start).count();
    loop_rate.sleep();
}

return 0;
}

```

