Feb.11th, 2020

MIE443

Mechatronics Systems: Design and Integration

Contest 1

Team 2

Name	**Student Number**
Zian Zhuang	1002449870
Jinxuan Zhou	1002350582
Rui Cheng Zhang	1002301948
Xu Han	999263434
Tianzhi Huang	1002076807

1.0 Problem Definition

The objective of Contest 1 is to develop and implement an algorithm that allows the TurtleBot to autonomously travel across an unknown environment while mapping its surrounding features such as walls and various obstacles. The robot must effectively utilize sensor feedback received from the Kinect sensor to avoid obstacles and also the ROS gmapping package to develop a map. A time limit of 8 minutes is imposed, as well as a respective travel speed limit of 0.25m/s during navigation and 0.1m/s when approaching obstacles. The contest environment is enclosed within a 4.87 x 4.87m^2 space with stationary objects made of cardboard boxes as obstacles. The layout of the environment is not revealed until the day of the contest, therefore having a flexible strategy with high consistency is critical. The result of the contest will be evaluated based on the percentage of the environment that the Turtlebot succeeded in mapping.

2.0 Overview of Strategies

Prior to the contest, a sample map with specific geographic features was provided for us to test our developed traversal algorithm. Our primary step was to have a functional algorithm that allowed the Turtlebot to achieve the contest objectives in the given sample environment, albeit there was the potential that the program might not be compatible with the actual contest environment.

To maximize the percentage of the map that the robot could observe, we first implemented a ‘self-spinning’ strategy, complemented with a randomized walking pattern. When the Turtlebot was turned on, it would self initialize by rotating 360° and then proceed towards the direction which had the maximum detected distance. The same procedure was executed periodically to randomize the navigation and efficiently reduce the redundancy of the mapped area.

The second consideration was obstacle detection, as we aimed for the Turtlebot to travel in a collision-free manner without relying heavily on the bumper feedback. We investigated the laser callback messages, and developed a strategy to split the scanning region into several segments. We later realized that since the scanning angle did not cover the full frontal view of the robot, it was improbable to avoid obstacle collisions entirely. Hence, we also developed the bumper logic function, which was a simple implementation of a movement back and rotation left or right, explained further in Table 1 below in section 4.1.3.

After the completion of the first two steps, we wanted to integrate high-level control functions which allowed the Turtlebot to check and travel along adjacent walls, thereby increasing its compatibility with unknown environments. Functions that carried out position tracking and error monitoring were also incorporated into our program, in which the Turtlebot would store its plane coordinates every 12 seconds. And if the current position was only 0.5m away from the previously recorded coordinates, it alluded to the immobility of the robot. The program would then be forced to terminate and restart with a 360° spin to prevent any further errors. Additional considerations include path and unique wall checking, which will be described in further details in section 4.

2.1 Contest Results

We received promising results as illustrated in Figure 1, where key geographic features were identified clearly within the limited time frame of 8 mins. The contest environment demonstrated higher complexity than the given sample map, as there were three additional stationary obstacles. The starting point was different in each of the two trials, yet the outline of the map was of great detail. The inner features of the map were shown with a higher visibility in Trial 2, as the navigation of the Turtlebot was more steady with less occurence of wall collision.

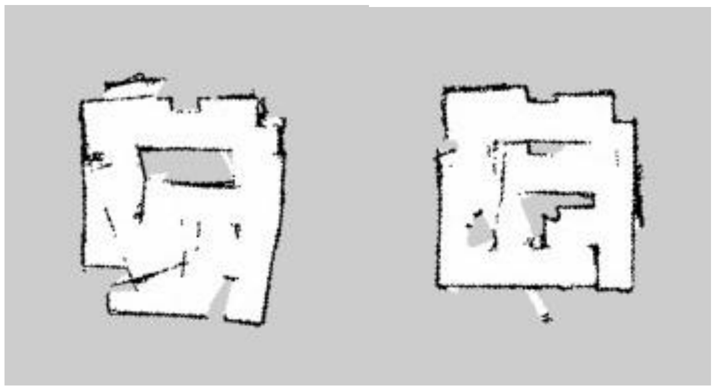


Figure 1: Contest map results (left: trial 1, right: trial 2)

3.0 Sensory Design

As illustrated in Figure 2, the robot is equipped with multiple sensors for various purposes. This section highlights the main sensors utilized by the team in Contest 1 to meet the contest requirements.

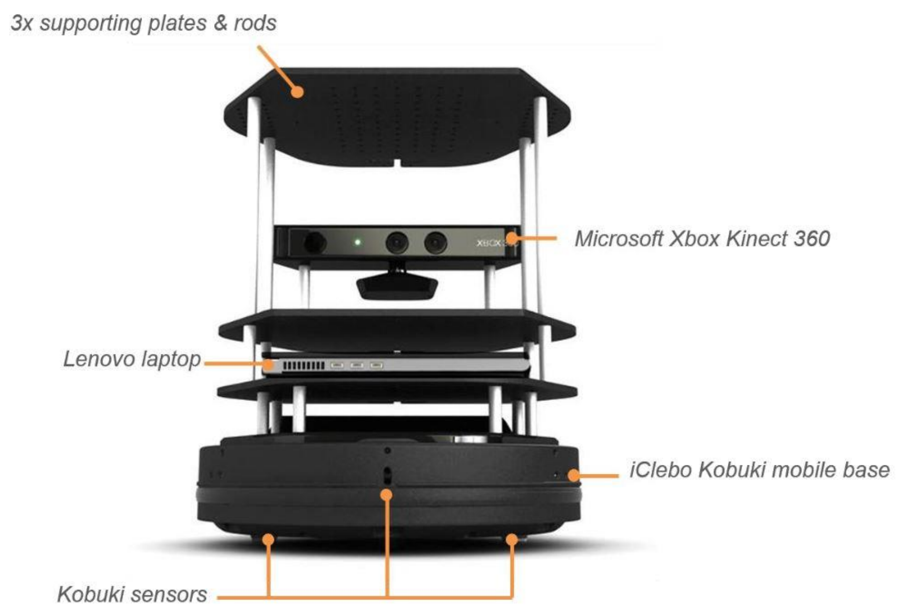

3x supporting plates & rods
Microsoft Xbox Kinect 360
Lenovo laptop
iClebo Kobuki mobile base
Kobuki sensors

Figure 2: Turtlebot schematics

3.1 Bumper Sensor

As shown in Figure 3, the Kobuki mobile base contains 3 bumper sensors. These sensors are placed at the left, center and right side of the base. The bumper sensors are triggered by physical touch and would register a state of 0 or 1. These sensors were used in our algorithm design because they were the main source of information in obstacle collisions. Therefore, the robot could make decisions on the subsequent travel direction based on the bumper responses after collisions.

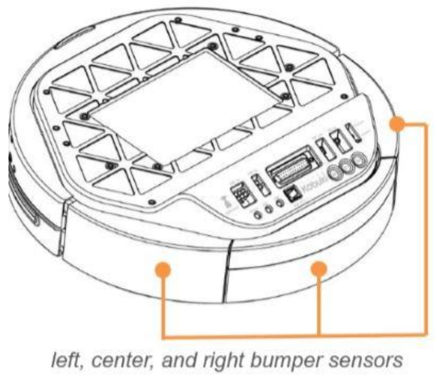


Figure 3: Kobuki mobile base

3.2 Gyro & Motor Encoder

The gyro and motor encoder are part of the Kobuki robot base design for wheel rotation and orientation data. The gyro sensor measures the linear and angular acceleration which assists in computing the odometry information including position and yaw angles. The odometry information is published to ROS and it can be used throughout the map traversal. Our exploration strategy requires accurate position and yaw recordings for precise rotation. Additionally, wheel slips and accumulated error from sensor noise may cause errors in our strategy. Since the contest environment is on flat ground, the wheel slip situation is unlikely to occur in the contest.

3.3 Microsoft 360 Xbox Kinect Sensor

As illustrated in Figure 4, the Microsoft Kinect consists of three main components: a RGB camera, a depth sensor and a microphone array. In this contest, only the depth sensor was utilized to fulfil the contest requirements. As the robot travels in the unknown environment, it relies on the information collected by the depth sensor in order to identify wall patterns and obstacles.

The depth sensor consists of an infrared laser projector combined with a monochrome CMOS sensor. The IR emitter constantly sends out infrared laser which is then reflected off an obstacle and received by the sensor. The vertical viewing angle of the Kinect sensor is 43° and the horizontal viewing angle is 57°.

Our algorithm constantly pulls distance data from the depth sensor to make decisions. The usage of the distance data will be explained in the following sections. In addition, the data collected with the depth sensor is also used in Gmapping to visualize and map the environment.

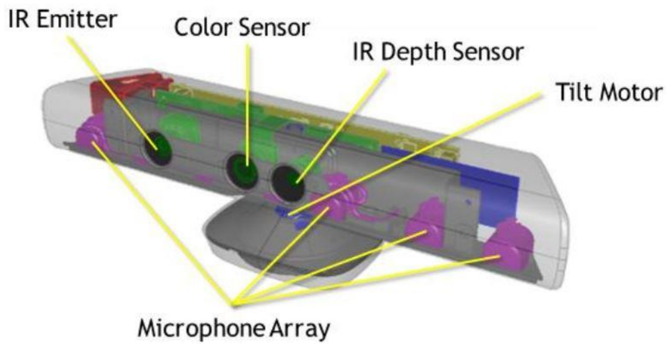


Figure 4: Microsoft 360 Xbox Kinect Sensor

4.0 Controller Design and Strategy

The control design for the turtlebot to achieve the contest 1 objectives has been summarized in the flowsheet below (Figure 5). The strategy relies on wall finding and wall following algorithms which are summarized in section 4.2.1. and section 4.2.2., respectively. The program begins with a 360° spin to find the longest distance, which utilizes the precision rotation function summarized in section 4.1.2. There is a fault check at the beginning of each loop to ensure the robot did not become stuck in one place. The fault check is designed to restart the program and continue exploring the map.

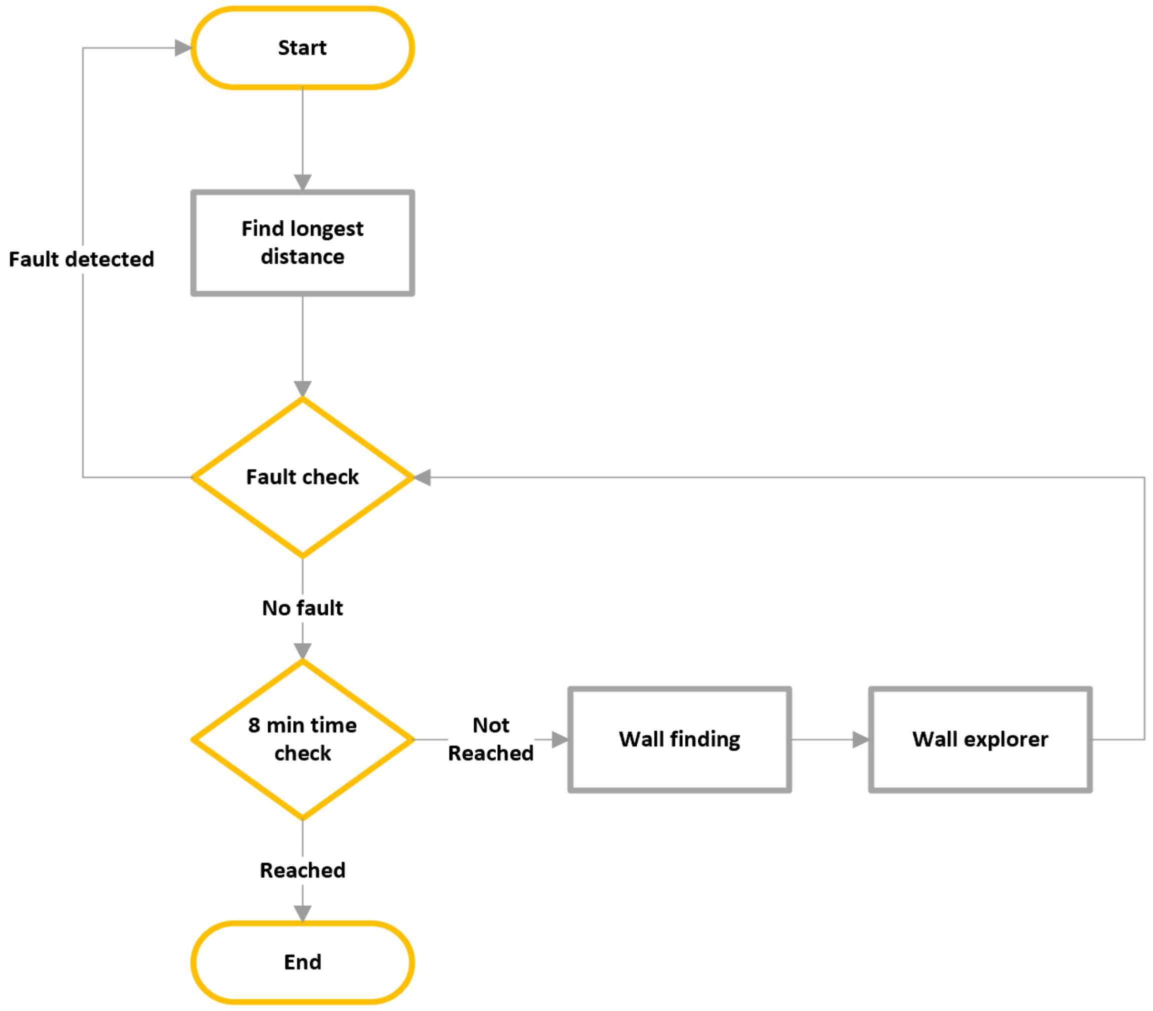


Figure 5: Overall strategy flowchart

4.1 Low-Level Control

Low-level control involves using the modules and packages offered in the ROS environment. The information gathered from the modules, along with output commands obtained in high-level control, was processed. The following section will cover four main modules: Laser, Odometry, Bumper and Gmapping.

4.1.1 Laser Scanning

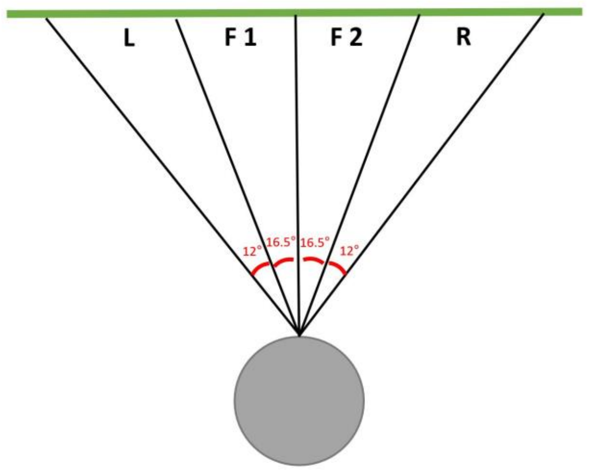


Figure 6: Sections of laser scan

The turtlebot utilizes the Kinect sensor reading through the laser module in the ROS package. The entire laser range was divided into 4 divisions as shown in Figure 6. The team iterated through all the sensor reads in the laser indices to compute the minimum, maximum and average distance within each division. The processed data from the lasers are essential for our high-level controls.

4.1.2 Precise Rotation

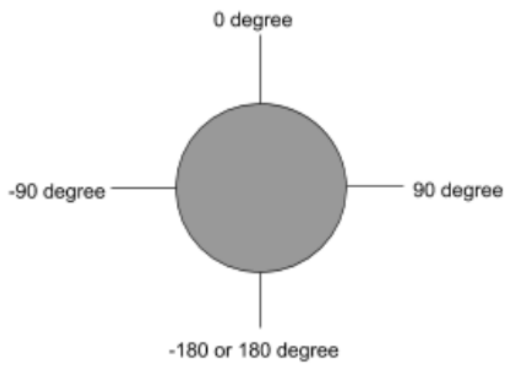


Figure 7: Robot rotation

The rotational function, which takes degrees as input, was developed in order to achieve a precise rotation for the robot throughout the contest. Through the odometry input, the yaw angle ranges from -180° to 180° as shown in Figure 7. In order to accommodate the odometry reading, the function can only take up to 180° degrees turning input on either side. A yaw goal was set with the current yaw angle plus the input turning angle. The yaw goal was converted to the odometer reading in the range of -180° to 180°. After repeated testing, the precision of the rotation was determined to be as low as 1°. In order to achieve a full spin of 360°, two 180° rotation commands serve as input to accomplish the goal. In order to find the maximum detectable distance through the full spins, the max laser distance is collected and recorded through the rotation, which is used at the beginning of the program.

4.1.3 Bumper Activation

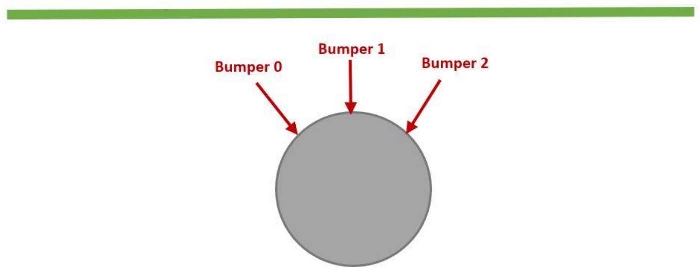
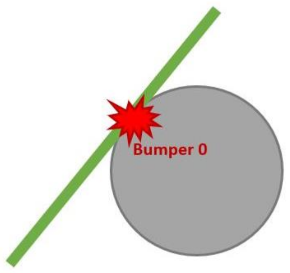
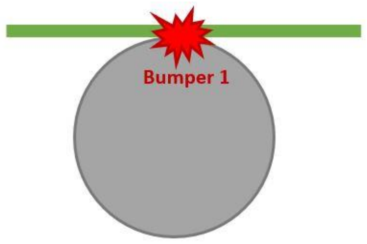
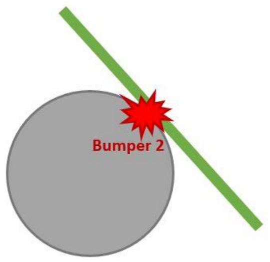


Figure 8: Bumper configuration

Since the Kinect sensor was set to have a viewing span of 57°, it would not be able to effectively detect obstacles at its left or right side. Therefore, the activation of bumper sensors was utilized as signals for the Turtlebot to change its orientation and steer away from the site of collision. The robot was equipped with three bumpers --- left as 'bumper 0', front as 'bumper 1', and right as 'bumper 2', as seen in Figure 8. Based on the type of bumper signals, the Turtlebot carried out its defense strategy correspondingly.

Table 1: Bumper response

Bumper Signal Type	**Response**
	Retreat and rotate clockwise
	**During ‘ wall following’:** Retreat and then rotate based on the side of which wall is at
	During ‘obstacle avoidance’: Move backwards
	Retreat and rotate counter-clockwise

4.1.4 Gmapping

Gmapping is used to visualize the map and its inner features. The gmapping package provides laser-based SLAM (Simultaneous Localization and Mapping). With this package, a 2-D occupancy grid map is created from laser and pose data collected by our mobile robot.

4.2 High-level Control

The high-level control utilizes the data structures and logic from low-level control and performs additional computations and logic for the overall traversal algorithm to command the Turtlebot during Contest 1. The main topics covered in this section include our wall finding strategy, wall following strategy, obstacle avoidance conditions, time and fault checking and path tracking.

4.2.1 Wall Finding Strategy

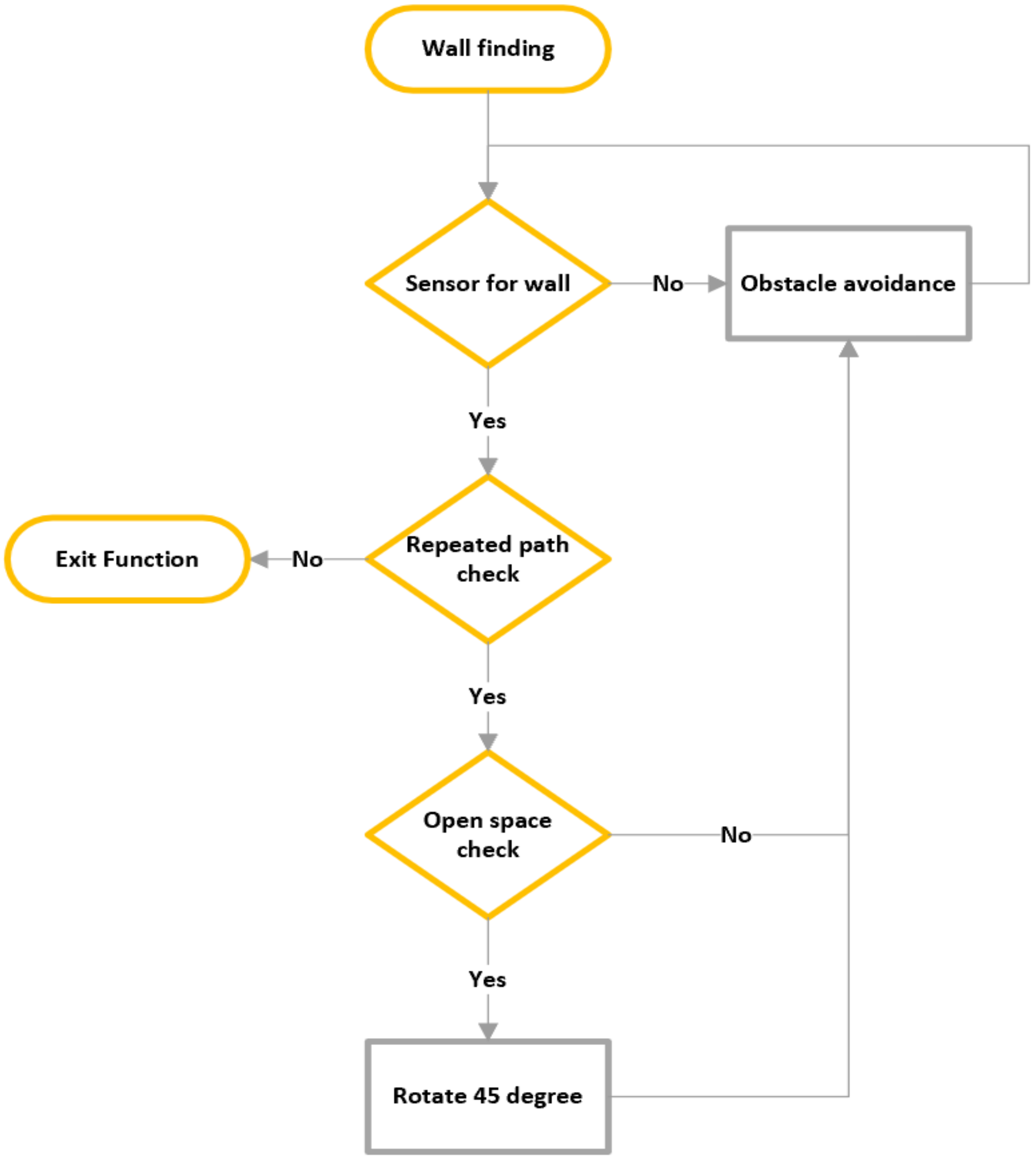
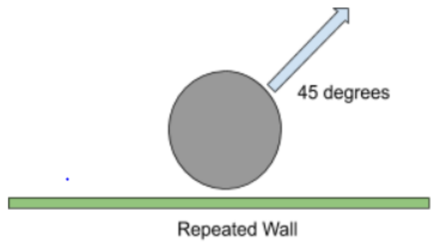
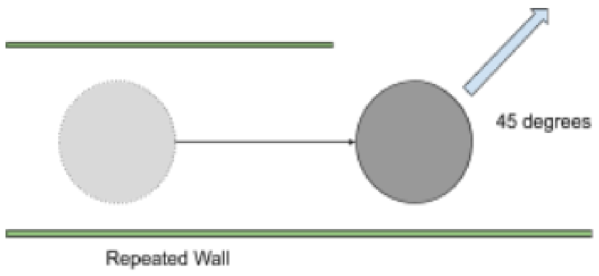


Figure 9: Wall finding function flowchart

The map exploration strategy centers around the wall following algorithm. Therefore, it is crucial to identify a correct wall in order to transition to wall following. The overall structure of the wall finding function is summarized in the flowchart Figure 9. The wall finding strategy has two main parts: identify a wall and check whether the wall has been followed before. Before any wall is detected, the robot will navigate in the open space through the obstacle avoidance algorithm summarized in Section 4.2.3. Based on the proximity of the adjacent walls, a right or left wall would be chosen and robot would proceed into a wall following command in Section 4.2.2. However, in order to maximize the efficiency of exploration, a repeated wall should not be followed again. Therefore, the array of position vectors is checked to confirm whether the wall has been followed before. The detailed path check strategy and position vector array utilization will be summarized in Section 4.2.5. When the path check algorithm shows that a repeated wall is within 0.8m from the robot, the program will navigate the robot off of the repeated path with a forced turn angle of 45 degrees. However, should the robot encounter a narrow passage, the program will force a turn after the robot proceeds to open space through the obstacle avoidance algorithm. The two scenarios are summaries in Table 2. This will ensure that the robot will not move in a zig-zag pattern and bump into the wall in a narrow passage, ensuring we find a new wall and continue the exploration.

Table 2: Action in open space

Scenarios	**Response**
	Rotate 45 degrees to open space and activate obstacle avoidance.
	Move within the narrow passage until open space is detected. Rotate 45 degrees and activate obstacle avoidance.

4.2.2 Wall Following Strategy

Once an unexplored wall is found, the robot will enter either the left-wall-following or right-wall-following algorithm, depending on which condition is satisfied.

The robot begins to move once the wall following strategy is executed and its start location (x,y coordinates) is recorded. The speed and direction of the robot depend on the sensory information collected at each moment, as illustrated in Figure 10. In the right-wall-following algorithm, the robot would first check if the minimum distance in the F1 area is smaller than the threshold (0.6m). If so, the robot will turn left until F1 is clear. If not, the robot will examine the minimum distance collected in R area, and the results are divided into 3 cases. If the distance between 0.6m and 0.7m, the robot is following the wall properly and travelling within the designed range. In this case, the robot will continue moving forward. If this distance is greater than 0.7m, it means the robot is deviating from the designed wall-following path and drifting away from the wall. Then the algorithm will make the robot steer right and force it closer to the right wall. This is particularly important when the wall pattern requires the robot to make a right turn. Finally, if the detected distance is smaller than 0.6m, it means the robot is moving too close to the wall and might collide into the wall if not corrected properly. The program will then make the robot steer left to move away from the wall. The algorithm for left-wall-following contains the exact same logic, only with all the directions reversed.

As the robot is exploring a wall, it also records and stores its coordinates every second. After following a wall for 20 seconds, the program will continuously compare the robot’s current location with its starting location for that wall. If it finds that its location is within 0.5m of the starting location, it concludes that the exploration of the current wall is completed, and it will exit the function and go back to wall finding by moving towards the direction with the greatest distance sensed. As explained in the previous section, these recorded locations will also be used later in finding a new wall when the current wall is completed. This path tracking strategy will be explained in more detail in Section 4.2.5.

While the robot follows the walls, it also spins 360° every 30 seconds to maximize the coverage of the scanned area.

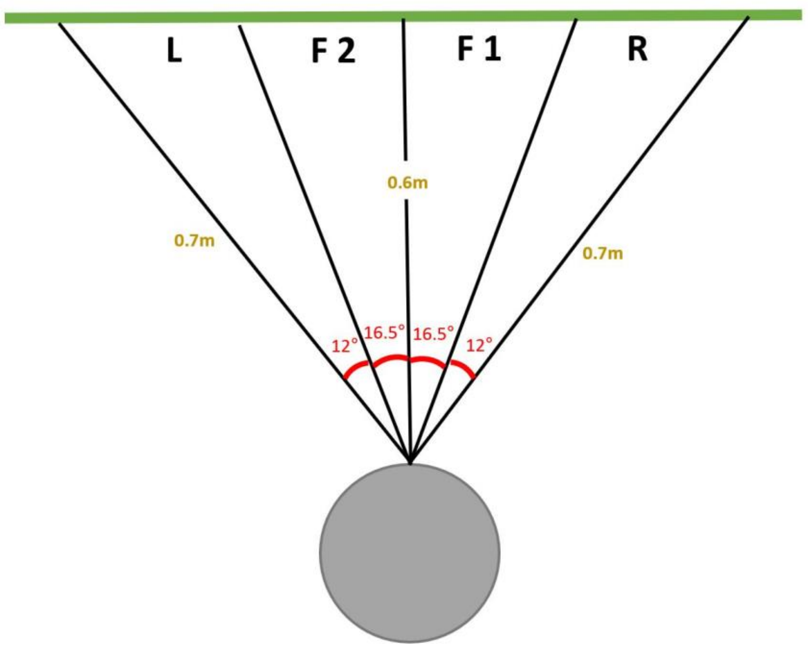


Figure 10: Sensor data during wall following

4.2.3 Obstacle Avoidance

An algorithm designed to detect obstacles is used as a preventative measure for the robot to navigate in a collision-free manner. During testing, it was realized that an ‘infinity’ feedback would be obtained once the sensor was in extreme proximity to surfaces. To resolve this issue, the Turtlebot was designed to self-rotate until the sensor had a clear viewing zone with an optimal distance of above 0.5 away from the obstacles.

The Kinect sensor was configured to have a total viewing angle of 57° which was then divided into three uneven segments --- left (L), front (F), and right (R), as shown in Figure 11. The front segment was assigned with a critical distance of 0.55m, while the other two were 0.65m. The robot would only proceed forwards if the minimum front distance was at least double its critical distance, 0.55m. Table 3 below depicts the Turtlebot’s responses when the sensor readings fell below the critical distance.

Table 3: Bumper response

Laser Segments	Condition	Response
Left	< 0.65m	Rotate Clockwise
Front	< 0.55m	Rotate Counter-clockwise
Right	< 0.65m	Rotate Counter-clockwise

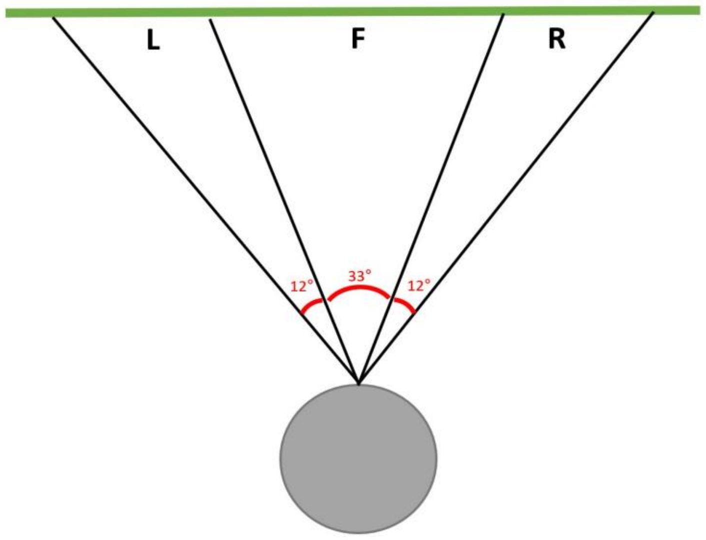


Figure 11: Sensor data during obstacle avoidance

4.2.4 Time Based Strategy

To ensure the continuous functionality of the Turtlebot, certain precautions were taken and incorporated into the algorithm. For situations where the robot remained stagnated within a while loop for 120 seconds, it was programmed to break out of the loop and restart itself. On the other hand, there was a probability that the robot was incapable of moving freely and failed to acknowledge that it became stuck at certain locations within the map. The solution was to employ and concurrently store the plane coordinates of the robot every 12 seconds. If the distance between the current set of coordinates and the last set of coordinates was less than 0.5m, it would indicate that the robot was unable to steer away from that specific location. Similarly, the program would then restart itself. In addition to the above two cases, the overall running time of the algorithm was also being monitored. After the program was kept on for 480s (8 mins), it would be terminated as the contest time limit has been reached.

4.2.5 Path Tracking

Our path tracking strategy is a strategy we employed to completely “mark” walls that have been followed previously to prevent repeated followings of the same wall in our wall follow algorithm. In other words, once the robot comes to an obstacle, it will have the information available to decide whether it has already traversed the entirety of this obstacle. If so, the robot will ignore it and continue exploring other areas of the map.

The main data structure for our path checking strategy is a vector of pairs called “path”. The pairs store the floating-point values of posX and posY, a simple cartesian coordinate based on the global map. We also keep a separate cartesian pair coordinate: “wall_start_pos”, which stores the single x and y point at which we entered the wall follow algorithm.

During the duration of the wall follow algorithm, we add the current position into our “path” vector every second. This way, we have a list of coordinates for the entire path we follow. Additionally, during the wall follow algorithm, we continue to check our current position with “wall_start_pos”. Once we reach “wall_start_pos”, we know that the wall/obstacle has been traversed entirely, and we are free to explore elsewhere.

To avoid the corner case of when the robot has just started following the wall and immediately exits, thinking it has reached “wall_start_pos”, we only start checking against the “wall_start_pos” after a reasonable amount of time has passed for the robot to have made traversal progress (20 seconds).

As was mentioned before, once our robot encounters a new obstacle, it will simply compare its current position with the positions stored in “path”. We perform a distance calculation against each point in “path”, and if any distances fall within our threshold (0.8 meters), we ignore the obstacle and continue exploring. The process above is shown in the figure below.

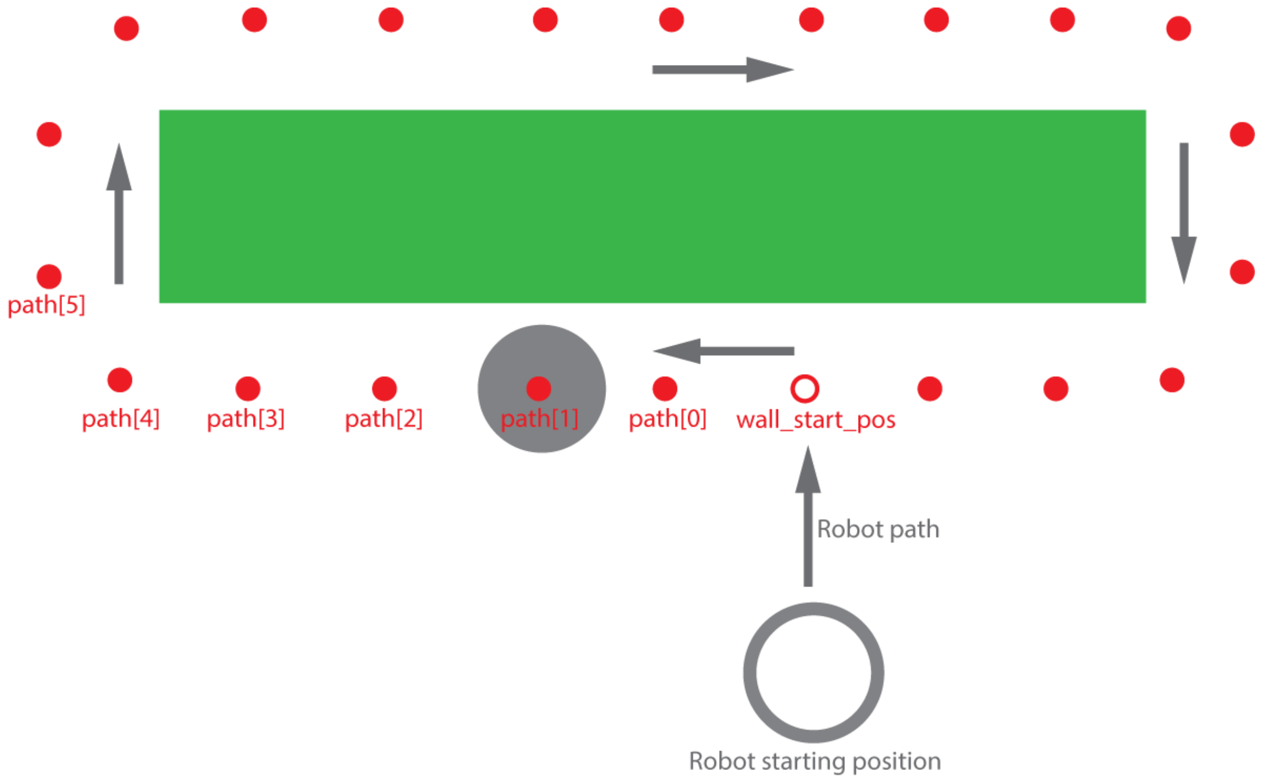


Figure 12: Position tracking

5.0 Future Recommendations

5.1 Current issues:

Through completing the contest, the team has identified some issues which decreased map quality. When the robot turns within the wall following algorithm, the robot loses track of the wall and results in a sharp turn. Since the wall following algorithm requires the information from the wall, the limited sensor reading range will result in the robot bumping into the corner wall. The resulting map has shown that the bumper activation can significantly decrease the quality of the map in corners. Furthermore, the team has set the linear speed and turning speed to 0.25 m/s and 0.1 m/s, respectively. The "wall following algorithm" switches from rotation to linear movement repetitively, which resulted in high accelerations during the contest. The abrupt movements have resulted in an incorrect reading for both odometry and gmapping drift. The drift in the movement has been reflected significantly in trail 1, as shown in the bottom left corner. This issue can be fixed by decreasing the linear velocity to match with the turning speed, and it will help to increase the map quality.

5.2 Would you use different methods or approaches based on the insight you now have?

One different approach we identified would be using average distance instead of minimum distance and dividing the laser sight range into more sections.

We initially implemented our laser sight system into 3 general areas: front, right, and left. We then calculated the minimum distance for each section to make traversal decisions with the robot. In our recent implementations, we started to divide the sections into smaller areas for more precision, and were able to achieve much better results from better and more accurate decision making

However, we believe the best approach would be to further divide the sections. In addition, instead of simply taking the minimum distance of each section, we would take an average of the distance to use in our decision logic. Doing so would mitigate possible cases of sensor errors with erroneous distance values, and provide us with more consistent information.

5.3 What would you do if you had more time?

If we had more time to work on our current project, we would try and implement frontier exploration. Our current implementation of wall follow would eventually guarantee the complete traversal of the map provided we have enough time. However, since the objective of the contest is for a complete mapping of the environment, there is no need to closely follow each wall and obstacle surface. Since the range of the laser sensor is relatively long, we feel that 360-degree spins in certain key areas of the map would be able to achieve all contest objectives. Therefore, we feel a frontier exploration type of algorithm would be much more efficient in distance travelled, and thus time taken to explore the entirety of the map.

Like our current algorithm in determining a direction heading, we would perform a complete rotation to determine the map of the immediate area. We would then access the gmapping data with the occupancy grid and find unoccupied grid locations beyond a certain threshold distance. These locations would be added to a stack.

Using the obstacle avoidance logic to traverse to each marked grid location in the stack, we would perform complete rotations at each point, adding each new grid location to the stack and removing our current location (marking it as “visited”). We believe using a stack to traverse the map, similar to a depth-first search on the accessible locations in the map, would be a much more efficient way of traversal.

The algorithm would then repeatedly visit locations in the stack until the stack is empty. With the complete 360 degree rotations at our identified points, the map would be completely scanned by the end of the stack.

6.0 Attribution Table

Section	**Student Names**				
	Zian Zhuang	**Jinxuan Zhou**	**Xu Han**	**Rui Cheng Zhang**	**TianZhi Huang**
Robot Design					
Wall following	RD, MR, DB	MR,DB	MR,DB	MR,DB	MR,DB
Wall exploration	RD, MR, DB	MR,DB	MR,DB	MR,DB	MR,DB
Obstacle Avoidance	RD, MR, DB	MR,DB	MR,DB	MR,DB	MR,DB
Low Level Controls	RD, MR, DB	RD, MR, DB	RD, MR, DB	RD, MR, DB	RD, MR, DB
Report					
1.0 Problem	MR,ET	MR,ET	MR,ET	MR,ET	RD,MR,ET
2.0 Overall Strategy	MR,ET	MR,ET	MR,ET	MR,ET	RD,MR,ET
3.0 Sensor Design	MR,ET	RD,MR,ET	MR,ET	RD,MR,ET	RD,MR,ET
4.0 Controller Design	MR,ET	MD,MR,ET	RD,MR,ET	RD,MR,ET	RD,MR,ET
5.0 Future Recommandations	RD,MR,ET	MR,ET	RD,MR,ET	RD,MR,ET	MR,ET
Overall	ET	ET	ET	ET	ET

RS	– major research
RD	– wrote first draft
MR	– major revision
DB	- Debug and Testing
ET	– edited for grammar and spelling

Appendix

```
LIBRARIES
==============================================================================

#include <ros/console.h>

#include "ros/ros.h"

#include <geometry_msgs/Twist.h>

#include <tf/transform_datatypes.h>

#include <kobuki_msgs/BumperEvent.h>

#include <sensor_msgs/LaserScan.h>

#include <nav_msgs/Odometry.h>

#include <stdio.h>

#include <cmath>

#include <chrono>

#include <vector>

#define N_BUMPER (3)

#define RAD2DEG(rad) ((rad)*180. / M_PI)

#define DEG2RAD(deg) ((deg)*M_PI / 180.)

FUNCTIONS
==============================================================================

// CALLBACKS

void bumperCallback(const kobuki_msgs::BumperEvent::ConstPtr &msg);

void laserCallback(const sensor_msgs::LaserScan::ConstPtr &msg);

void odomCallback(const nav_msgs::Odometry::ConstPtr &msg);

// BASICS

float calcDistance(float x1, float y1, float x2, float y2);

void run(ros::Publisher vel_pub, geometry_msgs::Twist vel);

void stop(ros::Publisher vel_pub, geometry_msgs::Twist vel);
```

```
bool is_idle();

void rotate_deg(ros::Publisher vel_pub, geometry_msgs::Twist vel, float angle);

void face_max_dist(ros::Publisher vel_pub, geometry_msgs::Twist vel);

void take_action(ros::Publisher vel_pub, geometry_msgs::Twist vel);

// ALGORITHM

void avoid_obstacle(ros::Publisher vel_pub, geometry_msgs::Twist vel);

void wall_follow_R(ros::Publisher vel_pub, geometry_msgs::Twist vel);

void wall_follow_L(ros::Publisher vel_pub, geometry_msgs::Twist vel);

// STRATEGY

void wall_finder(ros::Publisher vel_pub, geometry_msgs::Twist vel);

void wall_explorer(ros::Publisher vel_pub, geometry_msgs::Twist vel);

GLOBAL-VARIABLES
================================================================================

// SPEED VARIABLES [angular_max = +/- M_PI/6 | linear_max = +/- 0.25 (navigating) | +/- 0.1 (close
to obstacle)]

float linear = 0.0, angular = 0.0;

// ODOM VARIABLES

float posX = 0.0, posY = 0.0, posX_prev = 0.0, posY_prev = 0.0;

float yaw = 0.0, yaw_deg = 0.0, max_dist_yaw_deg = 0.0;

float rotate_precision = 1.0; // DEFAULT 1.0 DEG

std::pair<float, float> wall_start_pos;

std::vector<std::pair<float, float>> path;

// BUMPER VARIABLES

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

bool any_bumper_pressed = false;

// LASER VARIABLES

float maxLaserDist = 0.0;
float maxLaserDist_temp = 0.0;
float minLaserDist = std::numeric_limits<float>::infinity();
float minRightDist = std::numeric_limits<float>::infinity();
float minFrontDist = std::numeric_limits<float>::infinity();
float minFrontLeftDist = std::numeric_limits<float>::infinity();
float minFrontRightDist = std::numeric_limits<float>::infinity();
float minLeftDist = std::numeric_limits<float>::infinity();
int32_t nLasers = 0, desiredNLasers = 0, desiredAngle = 1; // DEFAULT 1 DEG FoV
// TIME VARIABLES
std::chrono::time_point<std::chrono::system_clock> start;
uint64_t secondsElapsed = 0;
uint64_t idle_checktime = 0;
// STATE VARIABLES
bool fault = false;
bool while_trap = false;
bool forbid_spin = false;
bool right_rule = true;
bool wall_following = false;
SUBSCRIBER-CALLBACK
==============================================================================
void bumperCallback(const kobuki_msgs::BumperEvent::ConstPtr &msg)
{ // 0 - LEFT; 1 - CENTER; 2 - RIGHT
    //uint8_t leftState = bumper[kobuki_msgs::BumperEvent::LEFT];
    //uint8_t centerState = bumper[kobuki_msgs::BumperEvent::CENTER];
    //uint8_t rightState = bumper[kobuki_msgs::BumperEvent::RIGHT];
    //ROS_INFO("Left: %u Center: %u Right: %u", leftState, centerState, rightState);
    any_bumper_pressed = false;
    bumper[msg->bumper] = msg->state;
    for (uint32_t b_idx = 0; b_idx < N_BUMPER; ++b_idx)
    {
        any_bumper_pressed |= (bumper[b_idx] == kobuki_msgs::BumperEvent::PRESSED);
    }
}
void laserCallback(const sensor_msgs::LaserScan::ConstPtr &msg)
{
    maxLaserDist = 0.0;
    minLaserDist = std::numeric_limits<float>::infinity();
    minRightDist = std::numeric_limits<float>::infinity();
    minFrontDist = std::numeric_limits<float>::infinity();
    minFrontLeftDist = std::numeric_limits<float>::infinity();
    minFrontRightDist = std::numeric_limits<float>::infinity();
    minLeftDist = std::numeric_limits<float>::infinity();
    nLasers = (msg->angle_max - msg->angle_min) / msg->angle_increment;
    desiredNLasers = DEG2RAD(desiredAngle) / msg->angle_increment;
    if (DEG2RAD(desiredAngle) < msg->angle_max && -DEG2RAD(desiredAngle) > msg->angle_min)
    { // DESIRED FoV < SENSOR FoV
        for (uint32_t laser_idx = nLasers / 2 - desiredNLasers; laser_idx < nLasers / 2 + desiredNLasers;
++laser_idx)
        { // FIND MIN & MAX IN DESIRED FoV
            minLaserDist = std::min(minLaserDist, msg->ranges[laser_idx]);
            if (msg->ranges[laser_idx] != std::numeric_limits<float>::infinity())
            { // EXCLUDE INF
                maxLaserDist = std::max(maxLaserDist, msg->ranges[laser_idx]);
            }
        }
    }
```

```
else
{ // SENSOR FoV < DESIRED FoV
    for (uint32_t laser_idx = 0; laser_idx < nLasers; ++laser_idx)
    { // FIND MIN & MAX IN SENSOR FoV
        minLaserDist = std::min(minLaserDist, msg->ranges[laser_idx]);
        if (msg->ranges[laser_idx] != std::numeric_limits<float>::infinity())
        { // EXCLUDE INF
            maxLaserDist = std::max(maxLaserDist, msg->ranges[laser_idx]);
        }
    }
}
for (uint32_t laser_idx = 501; laser_idx <= 639; ++laser_idx)
{ // 139
    minLeftDist = std::min(minLeftDist, msg->ranges[laser_idx]);
    //if (!std::isnan(msg->ranges[laser_idx]))
    //{
    //    totalLeftDist = totalLeftDist + msg->ranges[laser_idx];
    //    LeftDist_count = LeftDist_count + 1.0;
    //}
}
for (uint32_t laser_idx = 139; laser_idx <= 500; ++laser_idx)
{ // 362
    minFrontDist = std::min(minFrontDist, msg->ranges[laser_idx]);
}
for (uint32_t laser_idx = 320; laser_idx <= 500; ++laser_idx)
{ // 181
    minFrontLeftDist = std::min(minFrontLeftDist, msg->ranges[laser_idx]);
    }
    for (uint32_t laser_idx = 139; laser_idx <= 319; ++laser_idx)
    { // 181
        minFrontRightDist = std::min(minFrontRightDist, msg->ranges[laser_idx]);
    }
    for (uint32_t laser_idx = 0; laser_idx <= 138; ++laser_idx)
    { // 139
        minRightDist = std::min(minRightDist, msg->ranges[laser_idx]);
    }
}
void odomCallback(const nav_msgs::Odometry::ConstPtr &msg)
{
    posX = msg->pose.pose.position.x;
    posY = msg->pose.pose.position.y;
    yaw = tf::getYaw(msg->pose.pose.orientation);
    yaw_deg = RAD2DEG(yaw);
    //ROS_INFO("Position: (%f, %f) Orientation: %f rad or %f degrees.", posX, posY, yaw, yaw_deg);
}
FUNCTIONS-VERIFIED
=====================================================================
float calcDistance(float x1, float y1, float x2, float y2)
{
    return std::sqrt(std::pow(x1 - x2, 2) + std::pow(y1 - y2, 2));
}
void run(ros::Publisher vel_pub, geometry_msgs::Twist vel)
{
    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();
    if (secondsElapsed > 480)
    { // SYSTEM TOTAL RUNTIME REACHED
        fault = true;
    }
}
void stop(ros::Publisher vel_pub, geometry_msgs::Twist vel)
{
    angular = 0.0;
    linear = 0.0;
    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();
    if (secondsElapsed > 480)
    { // SYSTEM TOTAL RUNTIME REACHED
        fault = true;
    }
}
bool is_idle()
{                                                  // EVERY 12 SEC MUST MOVE AT LEAST 0.5 M
    if ((secondsElapsed - idle_checktime) >= 12)                  // (10 - 12)
    {                                                 // DEFAULT 12 SECS FOR AT MOST 2 TIMES BUMPER
TRIGGERS
        if (calcDistance(posX, posY, posX_prev, posY_prev) <= 0.5) // (0.3 - 0.7)
        {                                             // DEFAULT 0.5 M
            ROS_INFO("IDLE --> PROGRAM REBOOTED.");
            posX_prev = posX;
            posY_prev = posY;
            idle_checktime = secondsElapsed;
            return true;
        }
        else
        {
            posX_prev = posX;
            posY_prev = posY;
            idle_checktime = secondsElapsed;
            return false;
        }
    }
    else
    {
        return false;
    }
}
void rotate_deg(ros::Publisher vel_pub, geometry_msgs::Twist vel, float angle)
{ // ROTATE +CCW -CW (RANGE: 0 - 180)
    ros::spinOnce();
    float yaw_goal = yaw_deg + angle;
    if (yaw_goal < -180.0)
    { // SET YAW_GOAL BETWEEN -180 - +180
        yaw_goal = yaw_goal + 360.0;
    }
    else if (yaw_goal > 180.0)
    {
        yaw_goal = yaw_goal - 360.0;
    }
    if (angle > 0.0)
    { // SET ROTATE DIRECTION
        angular = M_PI / 6;
        linear = 0.0;
    }
    else
    {
        angular = -M_PI / 6;
        linear = 0.0;
    }
    while (rotate_precision < abs(yaw_goal - yaw_deg) && abs(yaw_goal - yaw_deg) < (360 -
rotate_precision))
    { // ROTATE TILL YAW_DEG = YAW_GOAL
        run(vel_pub, vel);
        ros::spinOnce();
        if (maxLaserDist > maxLaserDist_temp)
        { // UPDATE MAX DIST YAW
            max_dist_yaw_deg = yaw_deg;
            maxLaserDist_temp = maxLaserDist;
        }
    }
    posX_prev = posX;
    posY_prev = posY;
    idle_checktime = secondsElapsed;

    stop(vel_pub, vel);

}

void face_max_dist(ros::Publisher vel_pub, geometry_msgs::Twist vel)

{

    maxLaserDist_temp = 0.0;

    rotate_deg(vel_pub, vel, -181);

    rotate_deg(vel_pub, vel, -181);

    angular = M_PI / 6;

    linear = 0.0;

    while (rotate_precision < abs(max_dist_yaw_deg - yaw_deg) && abs(max_dist_yaw_deg - yaw_deg)
< (360 - rotate_precision))

    { // ROTATE TILL YAW = MAX_DIST_YAW

        run(vel_pub, vel);

        ros::spinOnce();

    }

    posX_prev = posX;

    posY_prev = posY;

    idle_checktime = secondsElapsed;

    stop(vel_pub, vel);

}

void take_action(ros::Publisher vel_pub, geometry_msgs::Twist vel)

{ // RUN WITH BUMPER PRIORITY 2 SEC BACK OFF TIME

    std::chrono::time_point<std::chrono::system_clock> bumper_pressed_start;

    bumper_pressed_start = std::chrono::system_clock::now();

    if (any_bumper_pressed)

    {

        if (bumper[0] == 1 && bumper[2] == 1)
```

```
{                   // LR or LFR
    linear = -0.13; // DEFAULT -0.12
    angular = 0.0;
    if (wall_following)
    { // TEST VALUE
        if (right_rule)
        {
            linear = -0.12;      // DEFAULT -0.12
            angular = M_PI / 10; // DEFAULT PI/10
        }
        else
        {
            linear = -0.12;       // DEFAULT -0.12
            angular = -M_PI / 10; // DEFAULT -PI/10
        }
    }
}
else if (bumper[0] == 1)
{                        // L or LF
    linear = -0.12;      // DEFAULT -0.12
    angular = -M_PI / 6; // DEFAULT -PI/6
}
else if (bumper[2] == 1)
{                       // R or RF
    linear = -0.12;     // DEFAULT -0.12
    angular = M_PI / 6; // DEFAULT PI/6
}
```

```
        else if (bumper[1] == 1)
        {                   // F
            linear = -0.13; // DEFAULT -0.12
            angular = 0.0;
            if (wall_following)
            { // TEST VALUE
                if (right_rule)
                {
                    linear = -0.12;      // DEFAULT -0.12
                    angular = M_PI / 10; // DEFAULT PI/10
                }
                else
                {
                    linear = -0.12;       // DEFAULT -0.12
                    angular = -M_PI / 10; // DEFAULT -PI/10
                }
            }
        }
        while     (std::chrono::duration_cast<std::chrono::seconds>(std::chrono::system_clock::now()     -
bumper_pressed_start).count() <= 1)
        { // DEFAULT 2 SEC BACK OFF TIME
            run(vel_pub, vel);
            ros::spinOnce();
        }
    }
    else
    {
        run(vel_pub, vel);
    }
}

void avoid_obstacle(ros::Publisher vel_pub, geometry_msgs::Twist vel)
{                         // FRONTDIST 0.55; SIDEDIST 0.65
    float FrontDist = 0.55; // DEFAULT 0.55 M
    float SideDist = 0.65;  // DEFAULT 0.65 M
    float ErrDist = std::numeric_limits<float>::infinity();
    forbid_spin = false;
    ros::spinOnce();
    float Left = minLeftDist, Front = minFrontDist, Right = minRightDist; // A1: USE minDist
    //float Left = avgLeftDist, Front = avgFrontDist, Right = avgRightDist; // A2: USE avgDist
    if (Right == ErrDist && Left == ErrDist)
    { // RIGHT & LEFT TOO CLOSE
        linear = 0.0;
        angular = M_PI / 6;
        forbid_spin = true;
    }
    else if (Right == ErrDist)
    { // RIGHT TOO CLOSE
        linear = 0.0;
        angular = M_PI / 6;
        forbid_spin = true;
    }
    else if (Left == ErrDist)
    { // LEFT TOO CLOSE
        linear = 0.0;
        angular = -M_PI / 6;
        forbid_spin = true;
    }
    else if (Front == ErrDist)
    { // FRONT TOO CLOSE
        linear = 0.0;
        angular = M_PI / 6;
        forbid_spin = true;
    }
    else if (Left >= SideDist && Front >= FrontDist && Right >= SideDist)
    { // NONE
        if (Left < (2 * SideDist) || Front < (2 * FrontDist) || Right < (2 * SideDist))
        {
            linear = 0.2;
            angular = 0.0;
        }
        else
        {
            linear = 0.25;
            angular = 0.0;
        }
    }
    else if (Left < SideDist && Front > FrontDist && Right < SideDist)
    { // RL
        linear = 0.15;
        angular = 0.0;
        forbid_spin = true;
    }
    else if (Right < SideDist)
    { // RFL RF R
        linear = 0.0;
        angular = M_PI / 6;
        forbid_spin = true;
    }
    else if (Left < SideDist)
    { // FL L
        linear = 0.0;
        angular = -M_PI / 6;
        forbid_spin = true;
    }
    else if (Front < FrontDist)
    { // F
        linear = 0.0;
        angular = M_PI / 6;
        forbid_spin = true;
    }
    else
    { // UNKOWN
        linear = 0.0;
        angular = 0.0;
        forbid_spin = true;
    }
    take_action(vel_pub, vel);
}
```

```
void wall_follow_R(ros::Publisher vel_pub, geometry_msgs::Twist vel)
{                       // CRITDIST 0.6; WALLDIST 0.6 - 0.7
    float wallDist = 0.7; // DEFAULT 0.7
    float critDist = 0.6; // DEFAULT 0.6
    float ErrDist = std::numeric_limits<float>::infinity();
    forbid_spin = false;
    ros::spinOnce();
    if (minFrontRightDist <= critDist || minFrontRightDist == ErrDist)
    {
        linear = 0;
        angular = M_PI / 6;
    }
    else if ((minFrontRightDist > critDist && minRightDist < critDist) || (minFrontRightDist > critDist &&
minRightDist == ErrDist))
    {
        linear = 0.12;      // DEFAULT 0.12
        angular = M_PI / 14; // DEFAULT PI/14
        forbid_spin = true;
    }
    else if (minFrontRightDist > critDist && minRightDist > wallDist)
    {
        linear = 0.12;       // DEFAULT 0.12
        angular = -M_PI / 14; // DEFAULT -PI/14
        forbid_spin = true;
    }
    else if (minFrontRightDist > critDist && minRightDist <= wallDist && minRightDist >= critDist)
    {
        linear = 0.25;
        angular = 0.0;
    }
    take_action(vel_pub, vel);
}
void wall_follow_L(ros::Publisher vel_pub, geometry_msgs::Twist vel)
{                       // CRITDIST 0.6; WALLDist 0.6 - 0.7
    float wallDist = 0.7; // DEFAULT 0.7
    float critDist = 0.6; // DEFAULT 0.6
    float ErrDist = std::numeric_limits<float>::infinity();
    forbid_spin = false;
    ros::spinOnce();
    if (minFrontLeftDist <= critDist || minFrontLeftDist == ErrDist)
    {
        linear = 0;
        angular = -M_PI / 6;
    }
    else if ((minFrontLeftDist > critDist && minLeftDist < critDist) || (minFrontLeftDist > critDist &&
minLeftDist == ErrDist))
    {
        linear = 0.12;       // DEFAULT 0.12
        angular = -M_PI / 14; // DEFAULT -PI/14
        forbid_spin = true;
    }
    else if (minFrontLeftDist > critDist && minLeftDist > wallDist)
    {
        linear = 0.12;      // DEFAULT 0.12
        angular = M_PI / 14; // DEFAULT PI/14
        forbid_spin = true;
    }
    else if (minFrontLeftDist > critDist && minLeftDist <= wallDist && minLeftDist >= critDist)
    {
        linear = 0.25;
        angular = 0.0;
    }
    take_action(vel_pub, vel);
}
void wall_finder(ros::Publisher vel_pub, geometry_msgs::Twist vel)
{ // FRONTDIST 0.6 ; SIDEDIST 0.7 || PATH CHECK 0.8 || TURN EVERY 15 SEC || PERIODIC SPIN
EVERY 40 SEC
    std::chrono::time_point<std::chrono::system_clock> finder_start;
    finder_start = std::chrono::system_clock::now();
    uint64_t finder_duration = 0;
    uint64_t finder_duration_turn = 0;
    uint64_t finder_duration_rotate = 0;
    bool wall_found = false;
    while (wall_found == false)
    {
        finder_duration                                                                    =
std::chrono::duration_cast<std::chrono::seconds>(std::chrono::system_clock::now()          -
finder_start).count();
        if (finder_duration > 120)
        { // CHECK IF STUCK IN WHILE DEFAULT 120 SEC
            ROS_INFO("STUCK IN FINDER --> PROGRAM REBOOTED.");
            while_trap = true;
        }
        if (is_idle() == true || while_trap == true || fault == true)
        { // CHECK PROGRAM FAULT
            fault = true;
            break;
        }
        if (minLeftDist < 0.7 || minFrontDist < 0.6 || minRightDist < 0.7) // FRONTDIST 0.6; SIDEDIST 0.7
        {                                                   // CONDITION 1: CHECK IF CLOSE TO WALL SO
WALL FOLLOWING ALGO CAN APPLY
            for (auto it = path.begin(); it != path.end(); it++)
            { // CONDITION 2: CHECK IF REPEATED WALL
                if (calcDistance(posX, posY, it->first, it->second) <= 0.8)
                { // PATH CHECK RANGE DEFAULT 0.8 M
                    if ((finder_duration - finder_duration_turn) > 15)
                    { // TRIGGER TURN EVERY 15 SEC IF OPEN AREA AVAILABLE
                        ROS_INFO("Found repeated wall");
                        if (minLeftDist < 0.7 && minFrontDist > 0.6 && minRightDist > 0.7)
                        { // RIGHT AREA OPEN
                            ROS_INFO("Found repeated wall - right area open");
                            rotate_deg(vel_pub, vel, -45);
                            finder_duration_turn                                              =
std::chrono::duration_cast<std::chrono::seconds>(std::chrono::system_clock::now()            -
finder_start).count();
                        }
                        else if (minRightDist < 0.7 && minFrontDist > 0.6 && minLeftDist > 0.7)
                        { // LEFT AREA OPEN
                            ROS_INFO("Found repeated wall - left area open");
                            rotate_deg(vel_pub, vel, 45);
                            finder_duration_turn                                              =
std::chrono::duration_cast<std::chrono::seconds>(std::chrono::system_clock::now()            -
finder_start).count();
```

```
                }
            }
            goto found_repeated_wall;
        }
    }
    // BOTH CONDITIONS MEET, A NEW WALL IS FOUND!
    if (minFrontRightDist < 0.6 || minRightDist < 0.7)
    { // USE RIGHT HAND RULE
        right_rule = true;
    }
    else
    { // USE LEFT HAND RULE
        right_rule = false;
    }
    ROS_INFO("Found new wall");
    wall_found = true;
}
else
{ // IF CONDITIONS FAIL, DO OBSTACLE AVOIDANCE TILL CONDITIONS MEET
found_repeated_wall:;
    avoid_obstacle(vel_pub, vel);
    if ((finder_duration - finder_duration_rotate) > 35 && forbid_spin == false)
    { // PERIODIC SPIN EVERY 40 SEC & SPIN ALLOWED
        ROS_INFO("Periodic Rotation");
        rotate_deg(vel_pub, vel, 181);
        rotate_deg(vel_pub, vel, 181);
```

```
                finder_duration_rotate                                                    =
std::chrono::duration_cast<std::chrono::seconds>(std::chrono::system_clock::now()                   -
finder_start).count();

            }

        }

    }

}

void wall_explorer(ros::Publisher vel_pub, geometry_msgs::Twist vel)

{ // CHECK IF RETURN TO STARTING POSITION AFTER 20 SECONDS || PATH CHECK 0.5 ||
PERIODIC SPIN EVERY 30 SEC

    std::chrono::time_point<std::chrono::system_clock> explorer_start;

    explorer_start = std::chrono::system_clock::now();

    uint64_t explorer_duration = 0;

    uint64_t explorer_duration_path = 0;

    uint64_t explorer_duration_rotate = 0;

    bool wall_complete = false;

    // REFRESH IDLE STATE

    posX_prev = posX;

    posY_prev = posY;

    idle_checktime = secondsElapsed;

    // RECORD STARTING POSITION

    wall_start_pos.first = posX;

    wall_start_pos.second = posY;

    ROS_INFO("Start Wall Following");

    ROS_INFO("Starting Position: (%f, %f)", wall_start_pos.first, wall_start_pos.second);

    while (wall_complete == false)

    {
        explorer_duration                                                                                   =
std::chrono::duration_cast<std::chrono::seconds>(std::chrono::system_clock::now()                         -
explorer_start).count();

        if (explorer_duration > 180)

        { // CHECK IF STUCK IN WHILE DEFAULT 180 SEC

            ROS_INFO("STUCK IN EXPLORER --> PROGRAM REBOOTED.");

            while_trap = true;

        }

        if (is_idle() == true || while_trap == true || fault == true)

        { // CHECK PROGRAM FAULT

            fault = true;

            break;

        }

        wall_following = true;

        if (right_rule)

        { // FOLLOW THE RIGHT HAND WALL

            wall_follow_R(vel_pub, vel);

        }

        else

        { // FOLLOW THE LEFT HAND WALL

            wall_follow_L(vel_pub, vel);

        }

        if ((explorer_duration - explorer_duration_path) >= 1)

        { // RECORD POSITION EVERY SECOND

            path.push_back(std::make_pair(posX, posY));

            explorer_duration_path = explorer_duration;

            // ROS_INFO("Recorded Path Position: (%f, %f)", posX, posY);

        }
        if (explorer_duration > 20)
        { // CHECK IF RETURN TO STARTING POSITION AFTER 20 SECONDS
            if (calcDistance(posX, posY, wall_start_pos.first, wall_start_pos.second) <= 0.5)
            { // IF RETURN TO STARTING POINT WITHIN 0.5, FIND MAX DIST TO EXIT
                ROS_INFO("Complete Wall Following");
                face_max_dist(vel_pub, vel);
                wall_complete = true;
                break;
            }
        }
        if ((explorer_duration - explorer_duration_rotate) > 25 && forbid_spin == false)
        { // PERIODIC SPIN EVERY 30 SEC & SPIN ALLOWED
            ROS_INFO("Periodic Rotation");
            rotate_deg(vel_pub, vel, 181);
            rotate_deg(vel_pub, vel, 181);
            explorer_duration_rotate                                                    =
std::chrono::duration_cast<std::chrono::seconds>(std::chrono::system_clock::now()       -
explorer_start).count();
        }
    }
    ROS_INFO("Exiting Position: (%f, %f)", posX, posY);
    wall_following = false;
}
MAIN-BLOCK
=============================================================================
int main(int argc, char **argv)
{
    // INITIALIZATION
    ros::init(argc, argv, "maze_explorer");
    ros::NodeHandle nh;
    // SUBSCRIBER - BUMPER, SCAN, ODOM
    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);
    // PUBLISHER - VELOCITY
    ros::Publisher vel_pub = nh.advertise<geometry_msgs::Twist>("cmd_vel_mux/input/teleop", 1);
    geometry_msgs::Twist vel;
    // CONTEST TIMER
    ros::Rate loop_rate(10);
    start = std::chrono::system_clock::now();
    // STOP WHEN TERIMINATION SIGNAL RECEIVED OR 480 SECS PASSED
    while (ros::ok() && secondsElapsed <= 480)
    {
        fault = false;      // SYSTEM NO ERROR
        while_trap = false; // SYSTEM NO ERROR
        face_max_dist(vel_pub, vel);
        while (!fault)
        { // CONTROL LOGIC
            // CHECK SUBSCRIBED MSGS
            ros::spinOnce();
            wall_finder(vel_pub, vel);
            wall_explorer(vel_pub, vel);
        }
        // UPDATE TIMER
        secondsElapsed                                                                            =
std::chrono::duration_cast<std::chrono::seconds>(std::chrono::system_clock::now() - start).count();
```

```
        loop_rate.sleep();
    }
    stop(vel_pub, vel);
    return 0;
}
```