MIE 443 Contest 1:

Autonomous Robot Search of an Environment

Yu-Tung Chen - 1005024910
Kevon Seechan - 1004941708
Diego Gomez - 1005139690
Miguel Gomez - 1005138628

February 16th, 2023

1.0 INTRODUCTION

1.1 Objectives

The objective of contest 1 is to develop an algorithm for robot exploration that enables the TurtleBot to independently navigate an unfamiliar environment using sensory data from the Kinect sensor, while utilizing the ROS gmapping libraries to create a dynamic map. The robot must complete the exploration and mapping of the environment within a specified time frame, without any human intervention. The robot's performance is evaluated based on the percentage of the environment mapped within the time limit.

Specifically, the scoring system for the exploration task in contest 1 is determined by two factors. Firstly, the percentage of the environment mapped during the exploration, which carries a weight of 10 marks. Secondly, the mapping of crucial obstacles within the environment, which carries a weight of 5 marks. The map created by the turtleBot will be compared against a ground truth map of the environment to determine its accuracy in mapping the terrain.

1.2 Requirements and Constraints

The TurtleBot will be given 8 minutes to explore and map the environment. It is mandatory that the robot performs the task autonomously without human intervention. To navigate the environment, the robot must rely solely on sensory feedback from the laser, bumper and odometry sensors. It cannot use a predetermined sequence of movements without sensor assistance. To ensure consistency in mapping, the robot must adhere to a speed limit of 0.25m/s when navigating free space and 0.1m/s when in close proximity to obstacles like walls or boxes.

The environment in which the contest will take place is contained within a 4.87x4.87 m^2 area with static objects. The exact layout of the environment will not be revealed to the team in advance. Therefore, the team must create a robust algorithm that can deal with randomized obstacles to ensure maximum marks.

2.0 STRATEGY

To achieve the mentioned objectives and complete the competition, the team decided to employ a ‘state’ or ‘behavior’ based programming approach. This type of programming breaks the robot’s operation into a sequence of events and is programming language independent [1]. This allowed the team to freely create pseudo code without having to consider the C++ syntax.

To develop an appreciation for the TurtleBot’s task, one member of the team was blind folded and guided through the map by the other team members. This revealed that the robot needs to be able to perform 3 main operations, namely: find open space, react to hitting obstacles/walls and drive freely once there is open space. During initial brainstorming, the team developed three ideas for algorithms that can achieve the necessary robot operations. These involved a rotation based algorithm, a wall follower and a weighted random walk. These algorithms were broken down into multiple functions and the work was split amongst team members. All functions were both simulated and tested on the physical robot to ensure proper functionality before being added

to the main functions of the codes. Each algorithm was then tested in the provided contest environment and judged on mapping capabilities and their adherence to the contest rules.

3.0 DETAILED ROBOT DESIGN AND IMPLEMENTATION

This section describes the sensors used and explains the high and low level control of the robot. In total, the team developed 3 different algorithms: biased rotation walk, wall follower and weighted random walk.

3.1 Sensory Design

The robot contains encoders, 3 bumper sensors, 3 cliff sensors, 2 wheel drop sensors, and a kinect laser sensor [2]. Due to the nature of the test environment, the team incorporated the bumper sensors, kinect laser sensor and encoders when developing the algorithms.

3.1.1 Laser Sensors

The range of the Kinect sensor used in the Turtlebot is approximately 50 cm to 5 m with a horizontal resolution of 640 x 480, 45° vertical Field of View (FOV) and 58° horizontal FOV [3]. However, after running tests, the team found multiple limitations associated with the laser. It was observed that the robot's laser can only detect objects at a distance of 1.8m. Additionally, the resolution is not as good as stated in literature, which means that to increase mapping accuracy, the robot must view obstacles from different angles as opposed to a single scan. Lastly, the FOVs is enough for the robot to clearly see large obstacles in front but it prevents the robot from visualizing what is directly to the side or from seeing very small obstacles.

In simulating the RViz software, it was discovered that the actual mapping of the environment only occurs at close distances. As such each algorithm uses the lasers to determine the minimum distance away from an obstacle in order to guide the robot towards or away from the obstacle.

For the wall following algorithm, the laser sensor is used to help identify the robot's position relative to the wall in order to determine the robot's next step. For the other two algorithms, the distances reported by laser sensors are used to determine where there is open space.

3.1.2 Bumper Sensors

Turtlebot2 has 3 built-in bumpers: 1 front and 2 side bumpers. When pressure is applied to the bumper, it retracts a few millimeters and a signal is generated. This signal is accessed by subscribing to "mobile_base/events/bumper" and using a "bumperCallback" function. If a bumper is pressed it returns a '1' value, if unpressed, it returns a '0' value.

For all algorithms the bumper sensors are used to detect when a wall/obstacle is present. The motivation for using the bumpers for this purpose rather than the lasers lies in the simplicity and reliability of the bumper signal. The call back function used to determine which bumper is pressed is the same for all of the algorithms. It begins by storing the state of each of the 3 bumpers in an array. As the robot navigates the environment, the function continuously loops through each item in the array and returns which of the bumpers are pressed.

3.1.3 Odometry Sensors

To collect odometry data, the turtleBot uses a series of encoders. These track changes in position and orientation as the robot maneuvers. This data is accessed by subscribing to the ROS 'odom' topic and using a call back function. The only algorithm that uses the odom data is the 'weighted random walk' algorithm. This is because this algorithm is structured such that it needs to monitor the robot's position. For this purpose, the odometry data is very useful.

3.2 Controller Design

3.2.1 Baised Rotation Walk

The objective of this code is to make an effective algorithm by combining several simple methods and interspersing them (see appendix B for full code). The main section of the code is split into two distinct sections.

The first section of the code occurs in the first half of the allocated exploration time (0 to 240 seconds). In this section, the robot will only perform left turns when it is close to a wall or when a bumper is triggered. This provides a base around which the robot will operate, which is generally sound and will do a fair amount of exploration by itself. However, the main issue with this code is that it tends to move along walls and does not explore open spaces. This is why a right rotation is added every 20 seconds. This element of randomness helps to increase the probability of the robot exploring all areas, although it is not guaranteed that it will.

The second section of the code occurs in the second half of the allocated time. In this section, it simply does the opposite of the first section. Now, the robot only performs right turns when it approaches a wall or when a bumper is triggered. After the first 240 seconds pass, the robot has usually fulfilled its potential with the first section of the code. Having performed left turns for that long, there is likely no new area that the robot will explore (although due to the periodic pseudo random rotations, there is still a chance that the robot will explore a new area). This is why right turns are now being performed (the periodic rotations are now to the left). Some of the unexplored areas (which have generally not been many during the simulations) now have a higher likelihood of being explored. This continues until the time runs out.

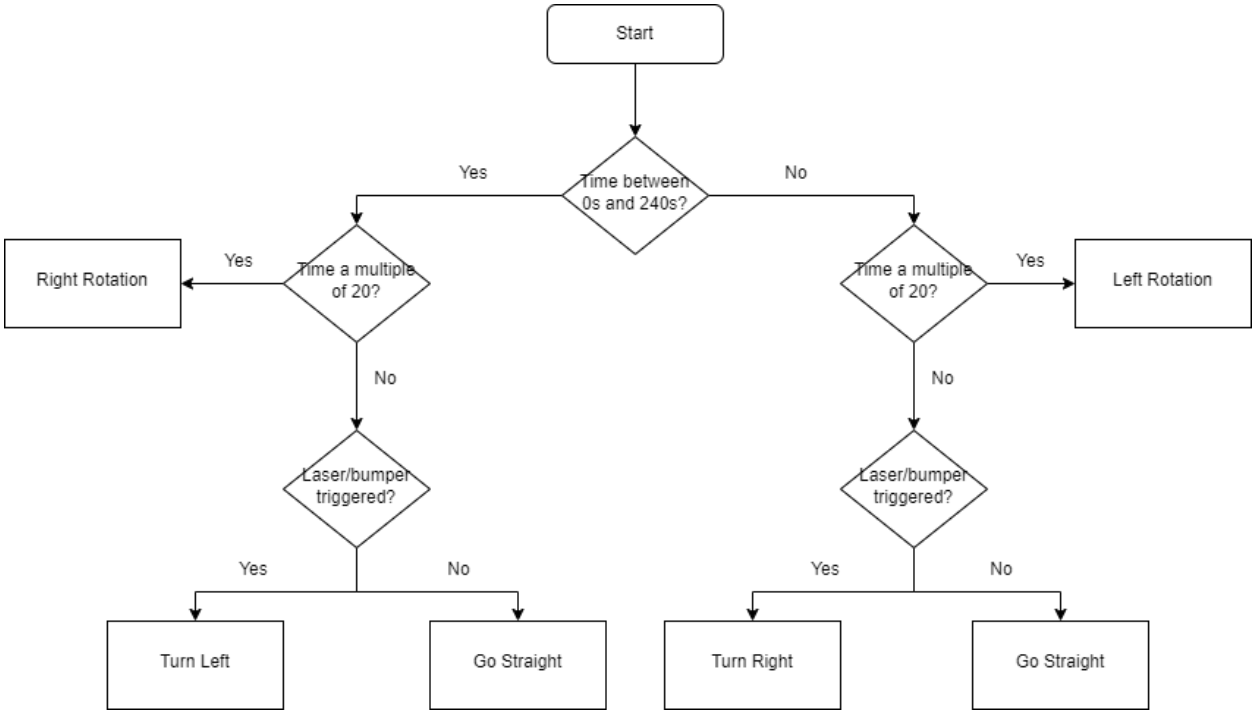

Figure 1: Biased Random Walk FlowChart

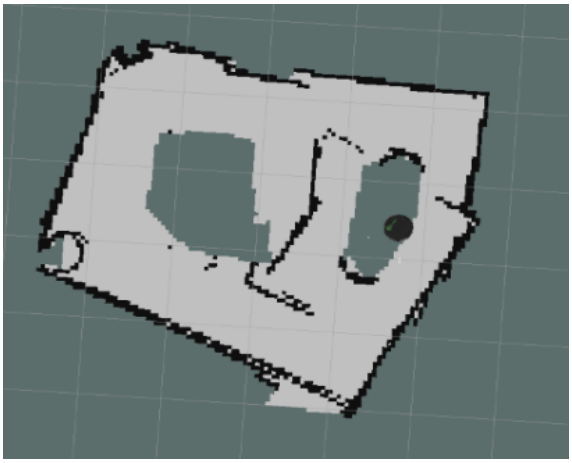

Figure 2: Map generated by the biased rotation walk algorithm

3.2.2 Wall Follower

The purpose of the wall follower algorithm is to allow the Turtlebot to navigate through the maze by following the wall of an obstacle in a set distance (see appendix C for full code). The motivation behind developing a wall following algorithm to the controller design is to ensure that the robot can navigate tight spaces.

The algorithm starts with the robot going straight until the bumper sensor is triggered, indicating a wall is found. The robot will then move backwards so that the laser sensor exits the dead zone. Due to the laser's limited range of view, the robot will not be able to see the wall if faced parallel to the wall. To overcome this problem, the starting position of the robot will then need to be faced slightly towards the wall so that the laser sensor can capture the wall distance. Initially this is done by grouping the laser reading in left, middle, right and taking the average of each group. If the robot is to follow a right wall, it will rotate counterclockwise in a fixed angle, putting the robot to its starting position. The average of the right laser readings will then be taken as the wall distance. It is later revealed through testing that this gives an inaccurate representation of the wall distance in certain situations such as when the obstacle is a cylindrical object. This is because when positioning the robot to its starting position, the robot is always rotated to a fixed angle. If the robot first collides to the wall in an angle, the fixed rotation can cause the robot to over or under rotate. Giving the possibility that the laser is picking up readings other than the distance to the obstacle. Figure 3 gives a visual explanation when the robot rotates.

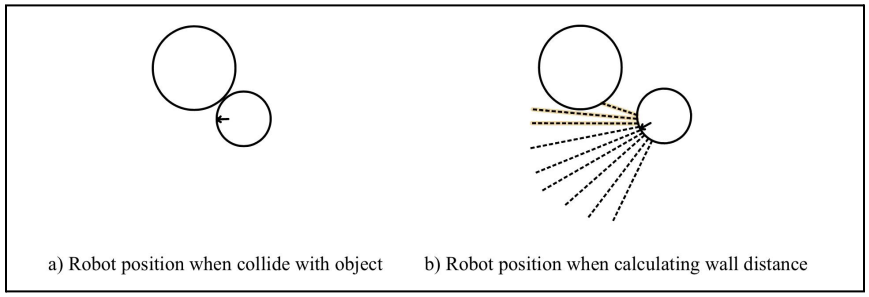

Figure 3: Visual explanation of over rotate

To fix this problem, only the left or right most laser reading is taken as the wall distance. If the robot is to follow an object to its right, the robot will continuously rotate until the rightmost laser value is the smallest value out of all the readings from right to middle. Meaning that the right most laser beam is now perpendicular to the wall. Its value is then taken as the wall distance. Below shows a basic pseudo code of the algorithm.

Algorithm 1: Robot Positioning to Right Wall

```
while (state = positioning) do
    for (i < nLasers/2)                  // nLasers is the total number of laser readings
        if (laser[i] < min)              // laser [i] is the i^th laser reading
            min = laser[i];
    if (laser[0] != min) then
        rotate
```

When determining the wall distance, only half of the readings that are closer to the wall are compared. This is to avoid a situation when the robot thinks the wall distance is 0. For the purpose of this algorithm, 0 is always assigned when NaN is returned. This means that there will be chances that 0 is assigned when in reality the distance is greater than 1.8 m. As this would mostly likely happen to the side of the robot that is opposite to the wall, that side of the laser readings are not used when determining the wall distance.

Once the robot is able to identify a wall to follow and oriented to the right position, it would take in the laser readings to get the distance the robot is away from the wall. As the robot progresses forward, it will constantly take in laser readings so that it can adjust its movement to ensure that the robot does not collide into or drift away from the wall. Below shows a basic pseudo code of the algorithm.

Algorithm 2: Right Wall Follow

```
while (state = right wall follow) do
    if (rightLaser > wallDist) then
        turn right on the spot
    else if (rightLaser < wallDist) then
        turn left and go forward
    else
        go forward and slightly to the left
```

As mentioned previously, in order to get the distance of the wall, the robot needs to be angled slightly towards the wall. Therefore, even when going forward, the robot needs to slightly swerve away from the wall. In case when the robot is getting too close to the wall it will swerve away from the wall even more aggressively.

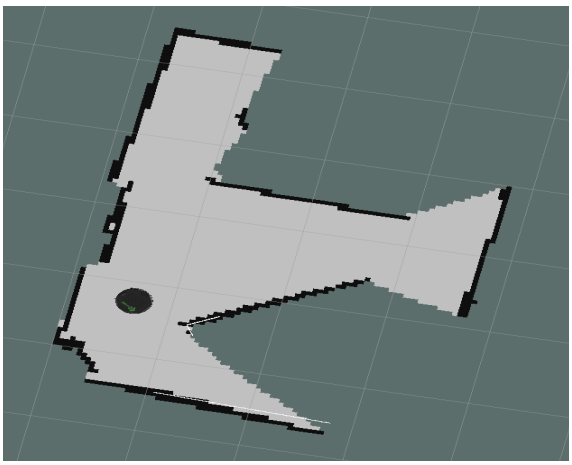

Figure 4: Map generated by wall follower algorithm

3.2.3 Weighted Random Walk

The flowchart below describes the general operation of the weighted random walk (see appendix D for full code).

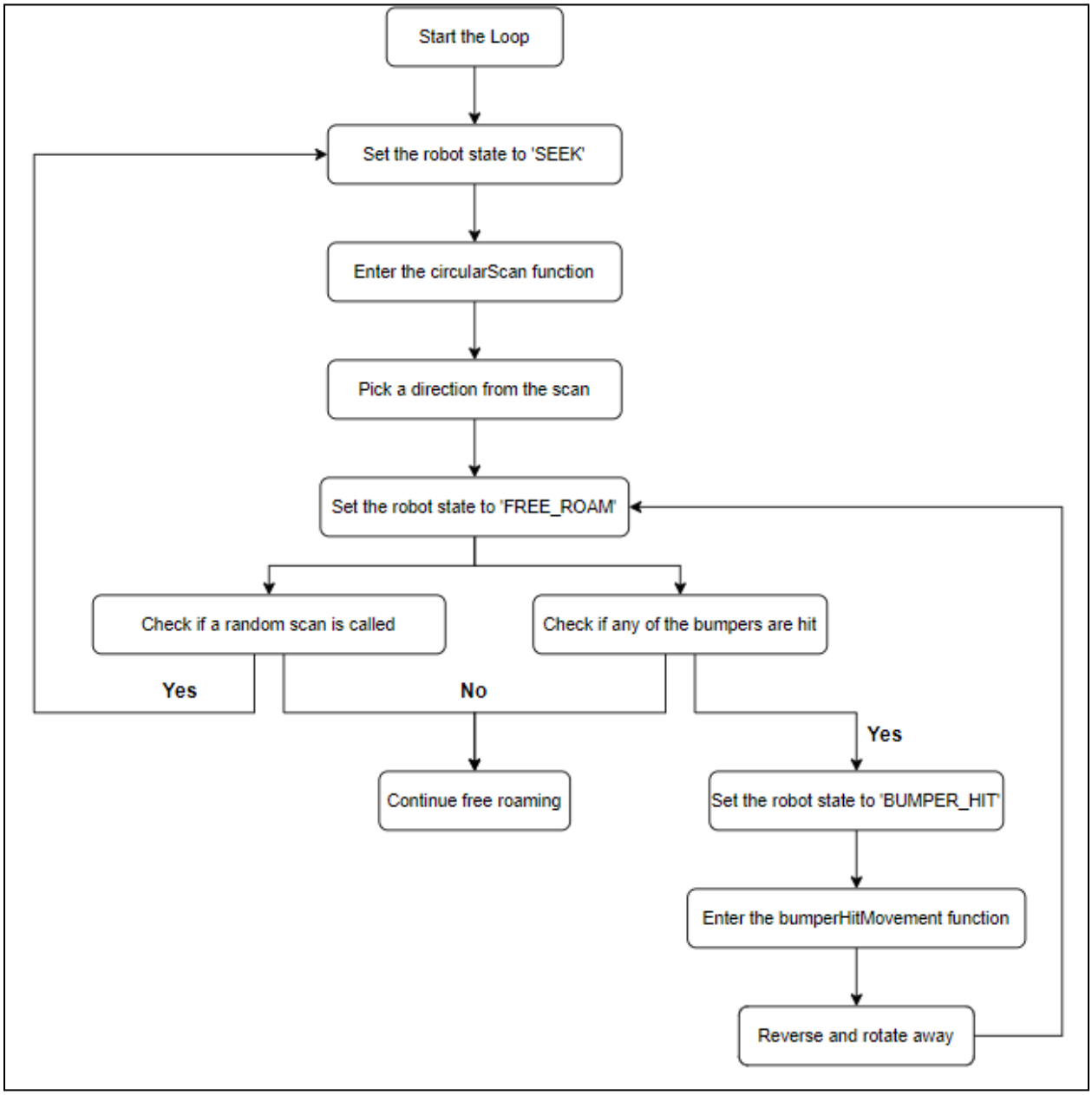

Figure 5: Flowchart explaining the general function of weighted run algorithm

As seen in figure 5, there are 3 robot states: 'SEEK', 'FREE_ROAM' and 'BUMPER_HIT'. The robot reacts to the environment by entering one of these states. Each state has its own function and each function has its own sub functions. The algorithm employs a total of 4 high-level functions which are explained below.

Main Function
This is the main function for the algorithm. It initializes ROS, subscribes to topics for bumper events, laser scan data, and odometry, and publishes twist messages to a cmd_vel_mux topic. It also sets up the contest countdown timer and variables to keep track of states and substates. It enters a loop and switches between different states, calling appropriate functions for each state to control the robot's movement. The loop also updates the timer, publishes velocity commands, and sleeps for a set period of time. The loop continues until the program is terminated or the timer runs out.

circularScan Function
The circularScan function performs a 360-degree scan of the robot's environment using its laser sensor and is called whenever the robot state is set to 'SEEK'. It stops 10 times while rotating and records the distances and yaw obtained for each scan step. Once finished scanning it finds the ideal direction and points the robot in the ideal direction by calling the *findIdealDirection* function and the *newHeading* function. It then sets the robot state to free roam. Note, to accomplish this, it has four substates that are used to control the different parts of the scan.

bumperHitMovement Function
This function is called when the robot's state is set to 'BUMPER_HIT' and implements a behavior for the robot to move away from an obstacle when its bumper is hit. The function takes a substate as an input, which is used to determine the current behavior of the robot.

The function uses a switch statement to handle different substates. When substate is 0, the robot stops and reverses away from the obstacle. When substate is 1, the robot rotates away from the obstacle depending on which bumper was hit. If the right bumper is hit, the robot is told to rotate 45° left. If the middle bumper is hit, the robot is told to rotate 90° to the right. If the left bumper is hit, the robot is told to rotate 45° right. Once finished rotating, the robot state is set to 'FREE_ROAM'. Note that the function uses the trackTurtleBot function to control the robot's movement.

roaming Function
This function is called when the robot's state is set to 'FREE_ROAM' which allows the robot to drive forward until it encounters an obstacle or wall. It uses a switch statement with several states to control the robot's movement. The drive step and velocity are calculated in state 0, and the robot is given driving instructions state 1. If the robot encounters an obstacle, it rotates in state 3 and then returns to state 0. If it does not encounter an obstacle, it calculates a new drive step and velocity and returns to state 0 or 2 depending on whether it is the first drive or not. The function also includes a random choice for which direction the robot should rotate. Additionally, to increase the probability that the robot explores the entire environment, the function has a portion of code that can randomly set the robot into 'SEEK' mode. Currently, the probability that the robot enters 'SEEK' while roaming is 30%, this value can be increased/decreased depending on user preference.

There are also 7 other sub functions that are used to execute and enable the 4 high-level functions. These are explained in detail below.

bumperEventCallback Function
This function is executed when a bumper event is detected. The function takes in a BumperEvent message, which contains information about the state of the bumper that was impacted (pressed or released). The function updates the bumper array to reflect the new state of the impacted bumper, and also sets a variable - 'anyBumperPressed' flag to true if a bumper was pressed. If a bumper is pressed, the function sets the state of the robot to BUMPER_HIT, and also stores the index of the last bumper that was pressed in a variable - 'lastBumperPressed'.

odomCallback Function
This function updates the position and orientation of the robot based on incoming odometry data.

laserCallback Function
This function is executed when a new laser scan message is received. It first sets the minimum laser distance (minLaserDist) to infinity. It then calculates the number of lasers in the scan (nLasers), the number of lasers within the desired angle (desiredNLasers), the beginning and end indices of the lasers to be processed based on the desired angle. The function then loops through the lasers within the desired angle and finds the minimum non-zero reading. Finally, it sets the minimum laser distance to the minimum non-zero reading.

trackTurtleBot Function
This function is used to control the movement of the robot by taking in four input parameters: 'dist', 'rot', 'linVel', and 'rotVel', which represent the desired distance to travel, amount to rotate, linear velocity, and rotational velocity, respectively. The function returns a boolean value indicating whether the robot has completed the driving instructions.

The function first checks if the input parameters are identical to the previous ones. If so, it sets a flag repeat to true and monitors the robot's progress using the previous values of position, orientation, linear velocity, and rotational velocity. It updates the remaining distance and rotation to travel and checks if the robot has reached the desired distance and rotation. If so, it sets the linear and rotational velocities to 0, indicating that the driving is complete.

If the input parameters are different from the previous ones, the function assigns new driving instructions by setting the remaining distance and rotation to travel, the current position and orientation, and the linear and rotational velocities. It then prints a message with the new driving instructions. Overall, the function keeps track of the robot's progress and controls its movement based on the input parameters.

driveAngle Function
This function takes in two values minSpinVal and maxSpinVal as parameters. These values are currently set to 450 and 1400 and were determined via testing. It calculates a random angle value between minSpinVal and maxSpinVal based on the beta probability distribution using a formula with two constants alpha and beta. To bias the robot towards minSpinVal (encouraging the robot

to keep driving forward), alpha was set to be larger than beta. It returns the calculated angle value.

findIdealDirection Function
This function takes in an array of distances minDist[] and an array of yaw positions yawPos[], and a float value power. It calculates a score for each element in minDist[] by raising the distance value to the power factor, and normalizing the scores so that the sum of all scores equals 1. It then selects a random index based on the normalized scores and returns the corresponding yaw position from yawPos[]. This yaw position represents the ideal direction to move in based on the distances measured by the sensor. This function biases the robot to select directions where the minDist[] values are larger (where the robot sees open space).

newHeading Function
This function determines whether to call the trackTurtleBot function to rotate the robot to a new heading. It takes in a targetDirection and rotationVelocity as inputs. It keeps track of the previous target direction and rotation velocity, as well as the amount of turning and the turning direction, using static variables. If the target direction and rotation velocity haven't changed, it just calls trackTurtleBot with the same parameters. If they have changed, it calculates the amount of turning and the turning direction, and calls trackTurtleBot with the updated parameters. The function returns a boolean value indicating whether the heading has changed.

Full explanation of weighted random walk algorithm
The algorithm works in the following way, the robot's state is initially set to 'SEEK' and as such, the robot enters the *circularScan* function. During the scan, the robot stores distance and yaw values. From these values, the code uses the *findIdealFunction* and *newHeading* function to point the robot in the 'ideal' direction. The state of the robot is then set to 'FREE_ROAM'. The code then enters the *roaming* function which allows the robot to drive around freely unless it encounters a wall or an obstacle. If it does, the bumpers are triggered, the robot state is set to 'BUMPER_HIT' and the *bumperHitMovement* function is called. Once this function has adjusted the robot away from the hit, the robot state is reset to 'FREE_ROAM' and it continues driving around. Additionally, when roaming, the robot has a 30% chance of randomly entering the 'SEEK' state which will in turn call the *circularScan* function again. These series of steps continue until the timer runs out.

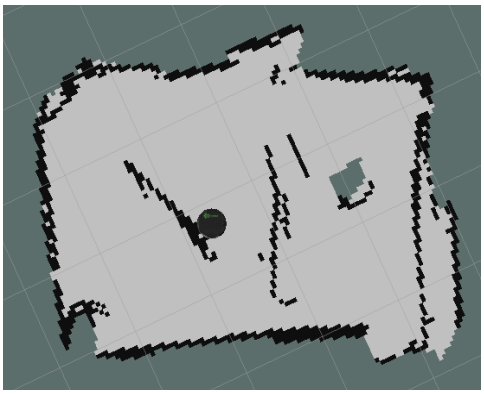

Figure 6 - Map generated by weighted random walk algorithm

3.2.4 Final Algorithm

After multiple rounds of simulation testing and physical testing, the team decided to use the weighted random walk algorithm for contest 1. This algorithm was selected for three main reasons. Firstly, it created the best map when using gazebo and the RViz software. This map can be seen in figure 6 above. In comparing figure 6 with the maps generated by the other algorithms in figures 4 and 2, it is clear that the weighted random walk has the strongest mapping capabilities.

Secondly, it performed the best in real world testing, it was able to navigate the robot throughout the majority of the environment and effectively react to walls/obstacles. The other algorithms struggled under real world conditions and occasionally got stuck in infinite loops.

Finally, the weighted random walk algorithm is more robust than the other algorithms as it is 'state' based and abides by all of the contest rules. In particular, the code is able to effectively switch between fast and slow driving speeds when navigating the environment and reacting to bumper hits.

4.0 FUTURE RECOMMENDATIONS

Due to the limitation on time and the sensor equipment, the algorithm implemented is not 100% efficient and reliable. While a random walk algorithm is a decent starting point, there is a lack of certainty and a degree of unpredictability. A number of improvements can be made if given more time and better equipment.

One of the improvements is to implement the wall follower algorithm. The algorithm was not selected for contest 1 since the team did not have enough time to perfect the algorithm. If the team was able to incorporate the wall following algorithm, it would eliminate the possibility that the robot gets stuck in tight spaces. Additionally, more complex exploration algorithms such as a frontier search should be implemented for a more efficient search. Even if the wall follower algorithm is incorporated with random walk, it still cannot visit unexplored areas intentionally, making the system less efficient. The frontier search will allow the robot to "see" areas that it has not yet explored which the robot can then aim directly toward unexplored areas instead of wasting time wandering aimlessly.

In terms of hardware limitation, additional sets of laser sensors can be implemented on the side of the robot to compensate for the limitation of its FOV. This can be beneficial especially for the wall follower algorithm since the robot would be able to see the wall distance without having to keep turning towards the wall.

Improvement that could be done for the future contests would be to utilize the lab sessions better and perform more testing on the actual robot. Due to the team's limited experience with ROS and C++, the majority of the lab sessions were used to understand how to operate the system instead of testing the functionality of the robot and sensor.

Lastly, the code could be made more modular for smoother integration and easier debugging. Each of the functions created for the weighted random walk algorithm can be further broken down into smaller functions and assigned to their own header files. This will allow for the work to be split amongst team members more effectively and allow for an easier implementation of additional strategies and algorithms into the main code.

5.0 REFERENCES

[1] Team, B. (n.d.). *Behavioral Programming: Behavioral Programming*. https://www.wisdom.weizmann.ac.il/~bprogram/more.html#:~:text=Behavioral%20Programming%20(BP)%20is%20an,live%20sequence%20charts%20(LSC).

[2] Clearpath Robotics. (2021, November 16). *TurtleBot 2 - Open source personal research robot*. https://clearpathrobotics.com/turtlebot-2-open-source-robot/

[3] Geospatial Modeling & Visualization. (n.d.). *Microsoft Kinect – Hardware | Geospatial Modeling & Visualization*. https://gmv.cast.uark.edu/scanning/hardware/microsoft-kinect-resourceshardware/#:~:text=The%20Kinect%20sensor%20is%20limited,63%20cm%20(25%20in)

6.0 APPENDIX

Appendix A - Contribution Table

Team Member	Task
Yu-Tung Chen	Responsible for developing, testing, and tuning of the wall follower algorithm. Worked on sections 3.1.1, 3.2.1, 4.0 of the report.
Kevon Seechan	Responsible for developing, testing and tuning weighted random walk algorithms. Completed sections 2.0, 3.2.3, and 3.2.4 of the report.
Miguel Gomez	Responsible for developing, testing and tuning biased rotation walk algorithms. Completed section 3.2.1 of the report. Responsible for testing code when course laptops were not available.
Diego Gomez	Responsible for generating simulated maps of each algorithm when course laptops were not available. Completed sections 1.1, 1.2, 3.1.1, 3.1.2 and 3.1.3 of the report.

Appendix B - Biased Rotation Walk 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 <chrono>

// Added for the ODOM demo
#include <nav_msgs/Odometry.h>
#include <tf/transform_datatypes.h>
float posX = 0.0, posY = 0.0, yaw = 0.0;
#define N_BUMPER (3)
#define RAD2DEG(rad) ((rad) * 180. / M_PI)
#define DEG2RAD(deg) ((deg) * M_PI / 180.)


// Added for BUMPER demo
uint8_t bumper[3] = {kobuki_msgs::BumperEvent::RELEASED,
kobuki_msgs::BumperEvent::RELEASED, kobuki_msgs::BumperEvent::RELEASED};


// Added for LASER demo
float minLaserDist = std::numeric_limits<float>::infinity();
int32_t nLasers=0, desiredNLasers=0, desiredAngle=15;


// SECOND EXAMPLE: Bumper
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;
}

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

```
{
    //fill with your code
    // nLasers = (msg->angle_max - msg->angle_min) / msg->angle_increment;
    // desiredNLasers = DEG2RAD(desiredAngle)/msg->angle_increment;
    // ROS_INFO("Size of laser scan array: %i and size of offset: %i",
nLasers, desiredNLasers);

    minLaserDist = std::numeric_limits<float>::infinity();
    nLasers = (msg->angle_max - msg->angle_min) / msg->angle_increment;
    desiredNLasers = desiredAngle*M_PI / (180*msg->angle_increment);
    ROS_INFO("Size of laser scan array: %i and size of offset: %i",
nLasers, desiredNLasers);

    if (desiredAngle * M_PI / 180 < msg->angle_max && -desiredAngle * M_PI
/ 180 > msg->angle_min) {
        for (uint32_t laser_idx = nLasers / 2 - desiredNLasers; laser_idx
< nLasers / 2 + desiredNLasers; ++laser_idx){
            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]);
        }
    }

}

// FIRST EXAMPLE: Odom
void odomCallback (const nav_msgs::Odometry::ConstPtr& msg)
{
    // // //fill with your 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: %f rad or %f degrees.",
posX, posY, yaw, RAD2DEG(yaw));
}
```

```
int main(int argc, char **argv)
{
    ros::init(argc, argv, "image_listener"); //can change the node name to
anything
    ros::NodeHandle nh;

    // 2 subscribers
    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", 10, &odomCallback);

    // 1 publisher
    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;

    while(ros::ok() && secondsElapsed <= 240) {
        ros::spinOnce();

        // ROS_INFO("Postion: (%f, %f) Orientation: %f degrees Range: %f",
posX, posY, RAD2DEG(yaw), minLaserDist);
        // 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 (/*posX < 0.5 && */yaw < M_PI / 12 && !any_bumper_pressed &&
minLaserDist > 0.6) {
            angular = 0.0;
            linear = 0.2;
        }
        else if (yaw < M_PI / 2 /*&& posX > 0.5*/ && !any_bumper_pressed
&& minLaserDist > 0.3) {
            angular = M_PI / 6;
            linear = 0.0;
        }
        while (any_bumper_pressed) {
            angular = M_PI/3;
        }

        if (minLaserDist > 1. && !any_bumper_pressed) {
            linear = 0.1;
            if (yaw < 17 / 36 * M_PI || posX > 0.6) {
                angular = 0;
            }
            else if (yaw < 19 / 36 * M_PI || posX < 0.4) {
                angular = 0;
            }
            else {
                angular = 0;
            }
        }
        else {
            angular = M_PI / 4;
            linear = 0.0;
        }

        if (secondsElapsed % 20 == 0){
            angular = -M_PI / 2;
        }

        vel.angular.z = angular;
        vel.linear.x = linear;
        vel_pub.publish(vel);

        // 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();
    }

    while(ros::ok() && 240 < secondsElapsed <= 480) {
        ros::spinOnce();

        // ROS_INFO("Postion: (%f, %f) Orientation: %f degrees Range: %f",
posX, posY, RAD2DEG(yaw), minLaserDist);
        // 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 (/*posX < 0.5 && */yaw < M_PI / 12 && !any_bumper_pressed &&
minLaserDist > 0.6) {
            angular = 0.0;
            linear = 0.2;
        }
        else if (yaw < M_PI / 2 /*&& posX > 0.5*/ && !any_bumper_pressed
&& minLaserDist > 0.3) {
            angular = -M_PI / 6;
            linear = 0.0;
        }
        while (any_bumper_pressed) {
            angular = -M_PI/3;
        }

        if (minLaserDist > 1. && !any_bumper_pressed) {
            linear = 0.1;
            if (yaw < 17 / 36 * M_PI || posX > 0.6) {
                angular = 0;
            }
            else if (yaw < 19 / 36 * M_PI || posX < 0.4) {
                angular = 0;
            }
```

```
            else {
                angular = 0;
            }
        }
        else {
            angular = -M_PI / 4;
            linear = 0.0;
        }

        if (secondsElapsed % 20 == 0){
            angular = M_PI / 2;
        }

        vel.angular.z = angular;
        vel.linear.x = linear;
        vel_pub.publish(vel);

        // 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;
}
```

Appendix C - Wall Follower Code

```cpp
#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 <chrono>

#include <nav_msgs/Odometry.h>
#include <tf/transform_datatypes.h>
float posX = 0.0, posY = 0.0, yaw = 0.0;
#define N_BUMPER (3)
#define RAD2DEG(rad) ((rad) * 180. / M_PI)
#define DEG2RAD(deg) ((deg) * M_PI / 180.)

float angular = 0.0;
float linear = 0.0;
float start_pos_x;
float start_pos_y;
int side = -1;

// variable for bumper
uint8_t bumper[3] = {kobuki_msgs::BumperEvent::RELEASED, kobuki_msgs::BumperEvent::RELEASED, kobuki_msgs::BumperEvent::RELEASED};
bool left_bumper = (bumper[0] == kobuki_msgs::BumperEvent::PRESSED);
bool middle_bumper = (bumper[1] == kobuki_msgs::BumperEvent::PRESSED);
bool right_bumper = (bumper[2] == kobuki_msgs::BumperEvent::PRESSED);

// variable for laser
float minLaserDist = std::numeric_limits<float>::infinity();
float minLaserDistLeft = std::numeric_limits<float>::infinity();
int32_t nLasers=0, desiredNLasers=0, desiredAngle=5;
float laserDist = std::numeric_limits<float>::infinity();
std::vector<double> laserReadings;

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;
}
```

```cpp

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

    minLaserDist = std::numeric_limits<float>::infinity();
    minLaserDistLeft = std::numeric_limits<float>::infinity();

    laserReadings.clear();

    //calculate # of lasers
    nLasers = (msg->angle_max - msg->angle_min) / msg->angle_increment;
    auto ranges = msg->ranges;


    // Set value to 0 if NaN is returned
    for (int i = 0; i < nLasers; i++) {
        if(std::isnan(ranges.at(i))){
            laserReadings.push_back(0);
        } else {
            laserReadings.push_back(ranges.at(i));
        }
    }
    ROS_INFO_STREAM("LASER READING LEFT: " << laserReadings.at(nLasers - 1));
    ROS_INFO_STREAM("LASER READING RIGHT: " << laserReadings.at(0));

    //finding the min value for left and right
    for (uint32_t laser_idx = 0; laser_idx < nLasers/2; ++laser_idx) {
        minLaserDist = std::min(minLaserDist, msg->ranges[laser_idx]);
    }
    for (uint32_t laser_idx = nLasers/2; laser_idx < nLasers; ++laser_idx) {
        minLaserDistLeft = std::min(minLaserDistLeft, msg->ranges[laser_idx]);
    }
    ROS_INFO_STREAM("Min LASER DIST: " << minLaserDist << "Min LASER DIST Left: " << minLaserDistLeft);


}

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);
    tf::getYaw(msg->pose.pose.orientation);
}

```

```
int wall_side_bumper(){

    //checks which side the wall is at
    if (bumper[1] == kobuki_msgs::BumperEvent::PRESSED){
        return 1;
    }else if (bumper[0] == kobuki_msgs::BumperEvent::PRESSED){
        return 0;
    }else {
        return 2;
    }
}

bool find_wall() {

    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){

        linear = 0.2;
        angular = 0.0;
        return false;
    }

    linear = 0.0;
    angular = 0.0;
    side = wall_side_bumper();
    ROS_INFO("wall found!");

    return true;
}

```

```
int wall_side_bumper(){

    //checks which side the wall is at
    if (bumper[1] == kobuki_msgs::BumperEvent::PRESSED){
        return 1;
    }else if (bumper[0] == kobuki_msgs::BumperEvent::PRESSED){
        return 0;
    }else {
        return 2;
    }
}

bool find_wall() {

    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){

        linear = 0.2;
        angular = 0.0;
        return false;
    }

    linear = 0.0;
    angular = 0.0;
    side = wall_side_bumper();
    ROS_INFO("wall found!");

    return true;
}

double  laser_reading(int side) {

    if(side == 0){
        return laserReadings.at(nLasers - 1);
    } else if (side == 2){
        return laserReadings.at(0);
    } else {
        return laserReadings.at(0);
    }
}
```

```cpp
int main(int argc, char **argv) {
    ros::init(argc, argv, "image_listener"); //can change the node name to anything
    ros::NodeHandle nh;

    // 2 subscribers
    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);

    // 1 publisher
    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;


    int bumper_counter = 0;
    bool wall_found = false;

    float start_pos_x;
    float start_pos_y;
    float start_yaw;
    double wall_distance;
    int counter = 0;
    int flag = 0;
    int back_up_count = 10;
    bool rotated = false;
```

```cpp
    while (ros::ok() && secondsElapsed <= 480) {
        ros::spinOnce();

        if (!wall_found) {
            wall_found = find_wall();
            counter = 0;
        } else {
            if (back_up_count > 0) {
                linear = -0.2;
                angular = 0.0;
                back_up_count--;
            } else if (!rotated){
                if (side == 1 || side == 2){
                    if (laserReadings[0] > minLaserDist){
                        linear = 0.0;
                        angular = M_PI/4;
                        ROS_INFO_STREAM("turning");

                    }
                }else {
                    if (laserReadings[nLasers-1] > minLaserDistLeft){
                        linear = 0.0;
                        angular = - M_PI/4;
                        ROS_INFO_STREAM("turning");
                    }
                }

                if (laser_reading(side) == minLaserDist){
                    linear = 0.0;
                    angular = 0.0;
                    rotated = true;
                    wall_distance = minLaserDist;
                    ROS_INFO_STREAM("I HIT WALL " << side <<" distance: "<<wall_distance);
                }
```

```
            }else{
                if (laser_reading(side) > wall_distance - 0.01 && laser_reading(side) < wall_distance + 0.01){
                    ROS_INFO_STREAM("GOING STRAIGHT: "<< "wall " << side <<" " << laser_reading(side)<<" away ");
                    if (counter > 200){
                        wall_found = false;
                        rotated = false;
                        back_up_count = 10;
                        break;
                    }
                    linear = 0.2;
                    angular = 0.3;
                    counter++;
                }
                else if(laser_reading(side) > wall_distance + 0.01){
                    ROS_INFO_STREAM("TOO FAR: "<< "wall " << side <<" " << laser_reading(side)<<" away ");
                    if (counter > 200){
                        wall_found = false;
                        rotated = false;
                        back_up_count = 10;
                        break;
                    }
                    if(side == 2 || side == 1){
                        angular = -0.3;
                        linear = 0.0;
                    }else {
                        angular = 0.3;
                        linear = 0.0;
                    }
                    counter++;
                }else{
                    ROS_INFO_STREAM("TOO CLOSE: "<< "wall " << side <<" " << laser_reading(side)<<" away ");
                    if (counter > 200){
                        wall_found = false;
                        rotated = false;
                        back_up_count = 10;
                        break;
                    }
                    if(side == 2 || side == 1){
                        angular = M_PI/2;
                        linear = 0.2;
                    }else {
                        angular = -M_PI/4;
                        linear = 0.2;
                    }
                    counter++;
                }
```

```
            }
        }
        vel.angular.z = angular;
        vel.linear.x = linear;
        vel_pub.publish(vel);

        // Update timer.
        secondsElapsed = std::chrono::duration_cast<std::chrono::seconds>(std::chrono::system_clock::now()-start).count();
        loop_rate.sleep();


    }

    return 0;
}
```

Appendix D - Weighted Random Walk Code

```
//standard libraries
#include <ros/console.h>
#include "ros/ros.h"
#include <stdio.h>
#include <cmath>
#include <chrono>

//for collecting sensor data
#include <geometry_msgs/Twist.h>
#include <kobuki_msgs/BumperEvent.h>
#include <sensor_msgs/LaserScan.h>
#include <nav_msgs/Odometry.h>
#include <tf/transform_datatypes.h>

//to convert radians and degrees
#define RAD2DEG(rad) ((rad) * 180. / M_PI)
#define DEG2RAD(deg) ((deg) * M_PI / 180.)


//for odometry
float posX = 0.0, posY = 0.0, yaw = 0.0;

//for bumpers and bumper events
const uint8_t N_BUMPER = 3;
uint8_t bumper[N_BUMPER] = {kobuki_msgs::BumperEvent::RELEASED,
kobuki_msgs::BumperEvent::RELEASED, kobuki_msgs::BumperEvent::RELEASED};
bool anyBumperPressed = false;
uint8_t lastBumperPressed = 0;
const float reverseLinDist = 0.1;
const float reverseAngDist = M_PI/4;

//for lasers
float minLaserDist = std::numeric_limits<float>::infinity();
int32_t nLasers=0, desiredNLasers=0, desiredAngle=5;
float avgLaserDist = 0;

//robot states
const int8_t SEEK = 0;
```

```
const int8_t FREE_ROAM = 1;
const int8_t BUMPER_HIT = 2;
int8_t state;

//for movement
float angular_vel = 0.0, linear_vel = 0.0;
const float rot_slow = (M_PI/12);
const float rot_fast = (M_PI/4);
const float drive_slow = 0.1;
const float drive_fast = 0.25;

//to stay away from walls
const float safeWallDist = 0.1; //used to ensure we slow down near walls
const float startSlowingDownDist = 0.2; //set at 0.2 to give the robot
enough time to slow down
const float maxDrive = 1.2; //adjust this to increase/decrease the max
step the robot takes

//for drving functions
float preX = 0.0, preY = 0.0, preYaw = 0.0;
float remDist = 0.0, remRot = 0.0, currentLinear = 0.0;
float currentAngular = 0.0, distToTravel = 0.0, amtToRotate = 0.0;
bool drivingComplete = false;
bool repeat = false;
bool headingChange = false;
float turnClockwise = 0.0, turnAntiClockwise = 0.0;

//for looking around
const uint8_t numScanSteps = 10;
const float amtRotPerStep = (2*M_PI) / numScanSteps;
float bias = 2.0; //adjust this value to bais how much the robot likes
larger distances

//for roaming around
const u_int8_t lookAround = 30; //increase or decrease this value (0-100)
to adjust the %chance the robot does a random scan while roamin
const u_int8_t minSpin = 45;  //should be enough to turn away from
obstacles
const u_int8_t maxSpin = 140; //can be increased, 140 found to be a good
value after testing
```

```
//Data Collection
//this laser function returns the minimum laser distance
void laserCallback(const sensor_msgs::LaserScan::ConstPtr& msg){
   minLaserDist = std::numeric_limits<float>::infinity();
   avgLaserDist = 0;

   nLasers = (msg->angle_max - msg->angle_min) / msg->angle_increment;
   desiredNLasers = desiredAngle*M_PI / (180*msg->angle_increment);
   //ROS_INFO("Size of laser scan array: %i and size of offset: %i",
nLasers, desiredNLasers);

   //calculate beginning and end of for loop based on desired angle
   int32_t beginIndex = 0, endIndex = nLasers;
   //ROS_INFO("endIndex: %i", endIndex);

   if (desiredAngle * M_PI / 180 < msg->angle_max && -desiredAngle * M_PI
/ 180 > msg->angle_min) {
       beginIndex = nLasers / 2 - desiredNLasers;
       endIndex = nLasers / 2 + desiredNLasers;
   }

   float min_dist = std::numeric_limits<float>::infinity();
   float sum_dist = 0;
   int count = 0;
   //loop through and find minimum and non zero values to get an average
   for(int i = beginIndex; i < endIndex; i++){
       float reading = msg->ranges[i];
       // Check if the reading is within a certain range
       if (reading >= msg->range_min && reading <= msg->range_max) {
           min_dist = std::min(min_dist, reading);
           sum_dist += reading;
           count++;
           //ROS_INFO("MIN_DIST Value: %.2fm", min_dist);
       }
   }
   //avg distance was calculated for debugging purposes, is not actually
used.
   avgLaserDist = count > 0 ? sum_dist / count: 0;
```

```
   //set minLaserDist based on min_dist
   minLaserDist = min_dist;
   //if(minLaserDist == std::numeric_limits<float>::infinity())
minLaserDist = 0;

   //ROS_INFO("Min Laser Distance: %.2fm, Avg Laser Distance: %.2fm",
minLaserDist, avgLaserDist);
}

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);
   //ROS_INFO("Position: (%f, %f) Orientation: %f rad or %f degrees.",
posX, posY, yaw, RAD2DEG(yaw));
}

//this function tracks the turtlebots movement and assigns new drive
instructions
bool trackTurtleBot(double dist, double rot, double linVel, double
rotVel){
   //variables to keep track of previous values
   static double preDist = 0, preLinVel = 0, preRot = 0, preRotVel = 0;
   repeat = false;

   //check that if the call is a repeat
   if((dist == preDist) && (linVel == preLinVel) && (rot == preRot) &&
(preRotVel == rotVel)){
       repeat = true;
   }

   //update the previous values
   preDist = dist;
   preLinVel = linVel;
   preRot = rot;
   preRotVel = rotVel;

   drivingComplete = false;
```

```
   //if it is a repeat call, monitor the driving
   if(repeat == true){
       angular_vel = currentAngular;
       linear_vel = currentLinear;
       distToTravel = sqrt(pow(posX - preX, 2) + pow(posY - preY, 2));
       preX = posX;
       preY = posY;

       remDist = remDist - distToTravel;
       if(remDist <= 0.0) linear_vel = 0;

       amtToRotate = std::abs(yaw - preYaw);
       if (amtToRotate > M_PI){
           //ensures the amount to rotate is within pi and -pi
           amtToRotate = (M_PI - abs(yaw)) + (M_PI - abs(preYaw));
       }

       remRot = remRot - amtToRotate;
       if(remRot <= 0.0) angular_vel = 0;
       preYaw = yaw;

       currentAngular = angular_vel;
       currentLinear = linear_vel;

       if((angular_vel == 0.0) && (linear_vel == 0.0)) drivingComplete =
true;

       if(drivingComplete){
           ROS_INFO("Driving Finished");
       }
   }
   //if it is a new call, assign new drive instructions
   else{
       //the remaining dist and rotation becomes the values that the
function is called with
       remDist = fabs(dist);
       remRot = fabs(rot);

       preX = posX;
       preY = posY;
```

```
      preYaw = yaw;

      linear_vel = linVel;
      angular_vel = rotVel;

      currentLinear = linear_vel;
      currentAngular = angular_vel;

      ROS_INFO("New Driving Instructions - Dist:%.2f, LinVel: %.2f, rot:
%.0f, rotVel: %.0f", remDist, linear_vel, remRot, angular_vel);
   }
   return drivingComplete;

}

//function to point the robot in a new direction
bool newHeading(float targetDirection, float rotationVelocity) {
   static float previousDirection = 0;
   static float previousVelocity = 0;
   static float amountToTurn = 0;
   static float turnSpeed = 0;
   bool headingChange = false;

   if (previousDirection == targetDirection && previousVelocity ==
rotationVelocity) {
      // If the target direction and rotation velocity haven't changed,
just call trackTurtleBot with the same parameters
      headingChange = trackTurtleBot(0, amountToTurn, 0, turnSpeed);
   } else {
      // If the target direction or rotation velocity have changed,
update the stored values
      previousDirection = targetDirection;
      previousVelocity = rotationVelocity;

      // Calculate the amount to turn and the direction to turn
      float deltaDirection = targetDirection - yaw;
      //ensures that the change is always within pi and -pi
      while (deltaDirection > M_PI) deltaDirection -= 2*M_PI;
      while (deltaDirection < -M_PI) deltaDirection += 2*M_PI;
```

```
       bool turnClockwise = deltaDirection < 0;
       amountToTurn = fabs(deltaDirection);

       // Set the turn speed based on the direction to turn
       turnSpeed = (turnClockwise ? -1 : 1) * rotationVelocity;

       // Call trackTurtleBot with the calculated amount to turn and
rotation velocity
       headingChange = trackTurtleBot(0, amountToTurn, 0, turnSpeed);
   }

   return headingChange;
}


//Obstacle Interaction Functions
//function to check bumpers
void bumperEventCallback(const kobuki_msgs::BumperEvent::ConstPtr& msg){
   bumper[msg->bumper] = msg->state;

   if (msg->state == kobuki_msgs::BumperEvent::RELEASED){
       ROS_INFO("%d released", msg->bumper);
   } else if (msg->state == kobuki_msgs::BumperEvent::PRESSED) {
       anyBumperPressed = true;
       ROS_INFO("%d bumper impact detected.", msg->bumper);
       state = BUMPER_HIT;
       lastBumperPressed = msg->bumper;
   }
}

//function to react to a bumper being pressed
void bumperHitMovement(u_int8_t &subState) {
   //check the subState pass in from the main function
   switch(subState) {
   //when substate = 0, reverse away from the obstacle and increment the
substate by 1
       case 0:
           if(trackTurtleBot(0, 0, 0, 0)){
               subState++;
               ROS_INFO("Bumper Hit detected, stoping robot");
```

```
            }
            break;
    //when substate = 1, rotate the robot depending on which bumper got
pressed.
        case 1:
            if(trackTurtleBot(reverseLinDist, 0, (-drive_fast), 0)){
                ROS_INFO("Done reversing from hit.");
                subState++;
            }
            break;
        case 2:
            switch(lastBumperPressed){
                case 0:
                    //rotate 45 degrees to the right
                    if(trackTurtleBot(0, reverseAngDist, 0, (-rot_fast))){
                        ROS_INFO("Done rotating from left hit bumper.");
                        state = FREE_ROAM;
                    }
                    break;

                case 1:
                    //rotate 90 degrees
                    if(trackTurtleBot(0, (reverseAngDist*2), 0, rot_fast)){
                        ROS_INFO("Done rotating from middle hit bumper.");
                        state = FREE_ROAM;
                    }
                    break;

                case 2:
                    //rotate 45 degrees to the left
                    if(trackTurtleBot(0, reverseAngDist, 0, rot_fast)){
                        ROS_INFO("Done rotating from right hit bumper.");
                        state = FREE_ROAM;
                    }
                    break;
            }
            break;
    }
}
```

```
//LOOK AROUND FUNCTIONS
//function to look around in a circle, turns the robot 360 degrees in 10
steps
//once the scan is complete, this function determines the direction to
point the robot in
float findIdealDirection(const float yawPos[], const float minDist[],
float power){
   //to find the ideal direction, randomly select a heading based on the
probabilities proportional to the distances.
   float sumDists = 0;
   int idealIndex = 0;
   float scores[numScanSteps];
   float randomValue = 0;
   float sumScores = 0;

   //calculate the sum of all the distances returned from minDist[],
applies a power factor to increase the chance of choosing a large distance
   for(int i = 0; i < numScanSteps; i++){
       sumDists += pow(minDist[i], power);
   }

   //if all the distances are 0, sumDists=0 and there is an error
   if (sumDists == 0){
       ROS_INFO("Scan unsuccessful");
       return yaw;
   }

   //give each scan step a score
   for(int i = 0; i < numScanSteps; i++){
       scores[i] = pow(minDist[i], power) / sumDists;
   }

   //select a random step based on the scores
   randomValue = sumDists * (rand() / (float)RAND_MAX);
   for(int i = 0; i < numScanSteps; i++){
       sumScores += scores[i];
       if (sumScores >= randomValue){
           idealIndex = i;
           break;
```

```
        }
    }

    return yawPos[idealIndex];
}


void circularScan(uint8_t &subState) {
    static float yawPositions[numScanSteps];
    static float minDistances[numScanSteps];
    static float idealDirection;
    static uint8_t scanSteps = 0;
    static bool scanCompleted = false;

    switch(subState) {
        case 0:
            // Initialize scan
            scanSteps = 0;
            subState = 1;
            ROS_INFO("Beginning 360 Scan of Environment");
            break;

        case 1:
            // Rotate the robot and store the distances and yaw obtained
for each scan step
            scanCompleted = trackTurtleBot(0, amtRotPerStep, 0, rot_fast);

            if (scanCompleted) {
                // Data collection completed for this scan step
                yawPositions[scanSteps] = yaw;
                minDistances[scanSteps] = minLaserDist;
                scanSteps++;
                ROS_INFO("Step#:%d, Distance:%.2f, Yaw:%.0f", scanSteps,
minLaserDist, yaw);

                // Stop the robot at each step
                if (scanSteps < 10) {
                    trackTurtleBot(0, 0, 0, 0);
                }
                else {
```

```
                    // All data has been collected, find the ideal
direction to drive
                    idealDirection = findIdealDirection(yawPositions,
minDistances, bias);
                    subState = 2;
                    ROS_INFO("After scan, ideal direction found to be
%.0f", RAD2DEG(idealDirection));
                }
            }
            break;

        case 2:
            // Point the robot in the ideal direction and set to free roam
            scanCompleted = newHeading(idealDirection, rot_fast);

            if (scanCompleted) {
                state = FREE_ROAM;
            }
            break;

        default:
            // Invalid substate, do nothing
            break;
    }
}




//Functions for Roaming Around
//this function tries minimize robot turning after each drive step to bais
the robot to travel forward
float driveAngle(float minSpinVal, float maxSpinVal){
    float range = maxSpinVal - minSpinVal;
    float alpha = 5.0; // Choose a value for alpha (larger than beta to
skew angle to minSpinVal)
    float beta = 2.0; // Choose a value for beta
    float u = static_cast<float>(rand()) / RAND_MAX;
    //calculates the angle based on the beta probability distribution
    float angle = minSpinVal + range * pow(u, alpha) * pow(1 - u, beta);
```

```
   return angle;
}


int8_t roaming(u_int8_t &subState) {
   static float driveStep = 0, driveVel = 0;
   static bool firstDrive = true;

   switch (subState) {
       //when subState is 0, calculate the drive step and velocity
       case 0: {
           driveStep = std::max(0.0f, minLaserDist - safeWallDist);
           driveStep = std::min(maxDrive, driveStep);
           //ensure that the slow speed is set if we are close to
obstacles/walls
           driveVel = ((minLaserDist - driveStep) < startSlowingDownDist)
? drive_slow : drive_fast;
           subState++;
           break;
       }

       case 1: {
           ROS_INFO("Free Roaming forward %.2f m at a velocity of %.2f
m/s", driveStep, driveVel);
           bool finishedDriving = trackTurtleBot(driveStep, 0, driveVel,
0);
           if (finishedDriving) {
               //calculate a random value
               u_int8_t diceRoll = rand() % 100;
               ROS_INFO("diceRoll: %i", diceRoll);
               //if this random value is less than lookAround, take a new
360 scan
               if (diceRoll < lookAround) {
                   state = SEEK;
                   return state;
               }
               else {
                   //calculate the new drive step and velocity if the
robot decides to take another step
                   if (minLaserDist < (startSlowingDownDist)) {
```

```
                    //if there is an obstacle ahead, calculate a new
drive step based on minSpin and maxSpin
                    driveStep = DEG2RAD(driveAngle(minSpin, maxSpin));
                }
                else {
                    //if no obstacle present, allow no spin in the
calculation
                    driveStep = DEG2RAD(driveAngle(0, maxSpin));
                }
                //random choice of drive velocity between fast
clockwise or anti clockwise
                driveVel = ((rand() % 2) == 1) ? rot_fast :
(-rot_fast);
                if (firstDrive) {
                    firstDrive = false;
                    subState = 2;
                }
                else {
                    subState = 3;
                }
            }
        }
        break;
    }

    case 2: {
        //send the new driving instructions
        bool finishedDriving = trackTurtleBot(driveStep, 0, driveVel,
0);
        if (finishedDriving) {
            firstDrive = true;
            subState++;
        }
        break;
    }

    case 3: {
        //rotates the robot if an obstacle is in front
        if (minLaserDist < (startSlowingDownDist)) {
            driveStep = 1.5 * driveStep;
```

```
                driveVel = -driveVel;
                subState = 4;
            }
            else {
                subState = 0;
            }
            break;
        }

        case 4: {
            //sends the rotate instructions
            bool finishedDriving = trackTurtleBot(0, driveStep, 0,
driveVel);
            if (finishedDriving) {
                subState = 0;
            }
            break;
        }
    }
}



int main(int argc, char **argv)
{
    //initilise ROS
    ros::init(argc, argv, "image_listener");
    ros::NodeHandle nh;

    //subscribe to bumper, laser and odometry topics
    ros::Subscriber bumper_sub = nh.subscribe("mobile_base/events/bumper",
10, &bumperEventCallback);
    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;

//variables to keep track of states and subStates
u_int8_t preState = 0;
u_int8_t subState = 0;
u_int8_t loopCount = 0;

//get one set of calls before starting to loop
ros::spinOnce();
loop_rate.sleep();

//set the state to SEEK so that the robot scans its environment first
state = SEEK;

//start the loop
ROS_INFO("STARTING THE LOOP");
while(ros::ok() && secondsElapsed <= 480) {
    ros::spinOnce();
    loopCount++;

    //check what the state is
    if(preState != state){
        preState = state;
        subState = 0;
        loopCount = 0;
    }

    //tell the robot to act based on the current state
    switch (state) {
        case BUMPER_HIT:
            bumperHitMovement(subState);
        break;
        case FREE_ROAM:
            roaming(subState);
        break;
        case SEEK:
            circularScan(subState);
```

```
            break;
        }


        //publish velocity command
        vel.angular.z = angular_vel;
        vel.linear.x = linear_vel;
        vel_pub.publish(vel);

        // 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;
}
```