Contest 1 Report

MIE443 Project

Savo Bajic - 1003051485
Maximilian Glidden - 1002277396
Catherine Kucaba - 1003278026

1.0 Problem Definition and Design Requirements

The objectives and requirements for this portion of the project are outlined in the Contest 1 Manual [1]. The primary goal of this contest is to develop an exploration algorithm that allows a robot to autonomously navigate an unknown environment within a certain time limit. Due to the Covid-19 pandemic, this project is completed only using software, as it would typically involve a physical robot navigating a real environment.

The robot is simulated using TurtleBot and it will navigate an environment created in gazebo. While traversing the area, the robot will use the ROS gmapping package to dynamically create a map from information provided by sensors on the TurtleBot, including a Kinect sensor and front bumper sensors. Figure 1 shows an example of a map generated in a practice environment by a robot that was controlled by a person via teleops. The white area is open space while the grey areas are the obstacles or walls. Although created by a person, this map provides an idea for what the exploration algorithm should aim to achieve. Figure 2 shows a rudimentary example of a map generated by the exploration algorithm in the same practice environment as Figure 1 (and within the time limit). The team will be scored based on the percentage of the total environment mapped in addition to detecting and mapping key obstacles within the time limit.

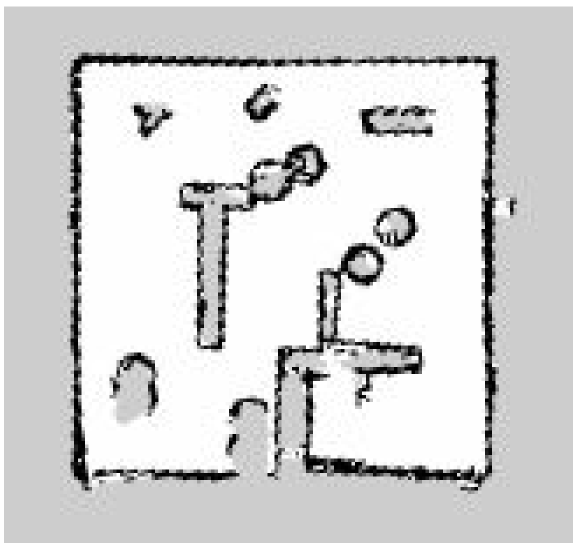

Figure 1: Example of a map generated by a human-operated robot

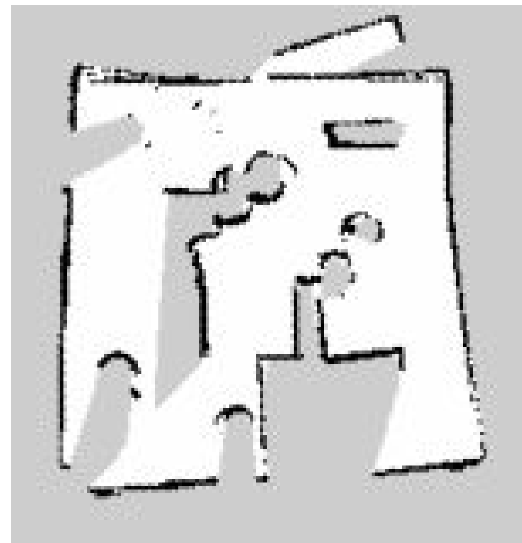

Figure 2: Example of a map generated by the exploration algorithm

The concepts required for this project can be applied to many other industries that use and rely on robots to execute tasks. For example, a similar scenario is presented for search and rescue robots used after natural disasters, as they must navigate unknown and hazardous terrain to search for missing or injured people. Other, less extreme, examples include robots used in the service and medical industries to assist, help, and treat humans, such as by escorting people to certain locations or navigating environments to drop off items/packages.

1.1 Requirements and Constraints

For this contest, there are some design requirements that the team must take into account when developing the navigation algorithm for the robot. These requirements can be found in the Contest 1 Manual [1] and are also presented below. It is noted that some of the design requirements are due to the limitations of the robot and its sensors in the simulation.

1. The robot must explore and map the environment within a maximum time limit of 15 minutes.
2. The robot must stop moving when the 15 minutes are completed or it has finished exploring and mapping the environment.
3. The robot must autonomously navigate the environment (i.e., no human intervention will be provided).
    a. It is noted that human intervention is required to set up the simulation environment, launch gmapping, and run the code. These tasks are not included in the above statement.
4. The robot must use sensory feedback provided by TurtleBot to navigate.
    a. Fixed sequence movements cannot be generated without the help of sensors.
5. The robot must not move faster than*:
    a. 0.25 m/s when navigating the environment (linear speed)
    b. 0.1 m/s when near obstacles (linear speed)
    c. $\pi/6$ rad/s when rotating (angular speed)
6. The robot cannot use code, algorithms, or libraries that directly implement a key portion of the algorithm.
7. The contest environment will be a three-dimensional simulated environment with static obstacles contained in a 6x6 m^2 area.
    a. It is noted that the red brick wall cannot be sensed by the robot.
8. The layout of the contest environment is not known to the team.
9. There are two trials per team, with the best trial counting towards the final score.

*Note: A speed limit is set in order to reduce errors in mapping.

2.0 Methods and Strategy

Given the uncertainty in environmental layout for the contest, and the limited development time, the team chose to implement a modular, behaviour-based design. The reasoning behind this approach was that regardless of the exploration strategy used, the behaviours required to achieve the contest requirements would be similar. For example, regardless of the exploration method, the robot must be able to scan its environment, and retreat from obstacles when a bumper is pressed. These behaviours could be implemented as their own modules and functions, which could be combined with a high-level control architecture. This gave the team flexibility in designing the high-level exploration strategy for the robot.

Initially, the team was aiming to implement a frontier-based approach to exploration. Due to time constraints and unfamiliarity with elements of gmapping and the ROS localization suite, the team instead opted to use a weighted random walk method for exploration, utilising a reward function fed by the turtlebot's depth camera to help guide the robot towards open spaces. Compared to the initial approach, this method was easier to implement, which was important given the limited development timeframe. Compared to a true random walk, the weighted walk guides the robot to travel further and faster, exposing it to more of its environment, which was important given the time requirements for the contest.

3.0 Detailed Robot Design and Implementation

3.1 Sensory Design

The simulated Turtlebot platform is equipped with a variety of sensors to aid in navigation, including touch bumpers, a depth camera, cliff sensors, and odometry data. Given that the environment the robot is exploring in Contest 1 is flat and enclosed, cliff sensors do not produce much useful information; there are no edges or changes in elevation for the robot to detect. The touch sensors, camera, and odometry were therefore used extensively to aid in exploration.

3.1.1 Bumpers

The Turtlebot has a set of three bumpers located on the front, left, and right sides. These bumpers publish their state (1 when pressed, 0 otherwise), which in turn is fed to the robot's high level control architecture. The primary motivation for using bumpers was to aid the robot in obstacle avoidance. The environment contains some obstacles that are invisible to the depth camera; for example because they are too low to be detected in the camera's field of view. The bumpers allow for these obstacles to be detected upon collision. During development, the team also identified issues with the depth camera failing to detect objects closer than 45 cm to the robot.

The bumpers facilitated the addition of local obstacle avoidance during traversal, when the camera could not detect obstacles in the motion path. Specifically, a set of obstacle avoidance behaviours were created with the bumper states as a sensory input. These behaviours take precedence over all other traversal behaviours in the robots control architecture. When a bumper is pressed, the bumper callback function records which bumper was triggered, and updates a global state variable. At the beginning of a control loop iteration, the value of each callback

function is evaluated, and the robot state is checked. If the robot is in the ‘bumper state,’ the *bumperMove* function is called. This function receives an index denoting the most recently triggered bumper, and executes a reverse movement, followed by a rotation. The direction and magnitude of rotation is dependent on which bumper detected a hit. After executing an evasive action, the robot returns to an *explore* state, and proceeds accordingly. A consequence of using the bumper for obstacle avoidance is that the bumper hit can be used for mapping. The *gmapping* package adds bumper contact to the map by default, which improves the fidelity of the map in situations where obstacles are not detected by the camera.

3.1.2 Depth Camera

The Turtlebot is equipped with a Microsoft Kinect camera system, which integrates both an RGB and depth camera. The depth camera can be used to approximate the functionality of a laser scanner, which generates a distance at each point in a one-dimensional array from the depth camera input. The motivation for using the depth camera in this way was to detect obstacles at longer ranges, while simultaneously building a map of the environment. Compared to the bumpers, the camera sees more of the environment, and does not require contact with an obstacle. It therefore made sense to use the camera as the primary method for exploring the environment and making movement decisions.

The camera was integrated as a laser scanner through the *laserCallback* function. During a function call, a desired field of view is divided into a series of discrete points. The range value is evaluated at each point, and an average and minimum distance is computed from the set of points. At the beginning of a loop iteration, the minimum distance passed by *laserCallback* is stored, and used to set maximum step sizes for movements. In the robot’s scan state, an array is used to store a minimum distance for each of a discrete set of orientations, which is then passed to the control logic governing exploration to generate a weighted set of scores for movement in various directions. This helps the robot decide which direction to move in, with a preference for open spaces, meeting the requirements for autonomous exploration and decision making.

3.1.3 Odometry

Encoders on the motors of the turtlebot provide information on the rotational velocity of each wheel, which is used with the forward differential kinematics model to produce linear and rotational velocities. Integrating these velocities over time produces an estimate of the robot’s position and heading; this is the odometry data supplied by the *odomCallback* function.

Odometry data was used to close the control loop with respect to movement. Inputting movement commands with motor velocities and durations alone is an example of open-loop control. The robot does not track how far it has actually travelled, nor can it easily estimate its position in the world frame. By checking the odometry data at the beginning of each loop iteration, the turtlebot can adjust itself to reach the desired destination. A consequence of this is that the main control loop can be iterated repeatedly during movement commands by comparing the odometry callback to the desired output of the movement command, and setting a *doneMoving* state accordingly. This allows for more frequent updates from all of the sensors (given that the values of the callback functions are evaluated only at the beginning of each control loop iteration).

3.2 Controller Design

The overarching control architecture used for the first contest was a behaviour-based controller, implemented using a finite state machine. Combinations of a set of behaviours were encoded in various states (“scan”, “explore”, etc.), with transitions between states triggered by sensory input and the code of other states. The robot can only exist in a single state at a given time, but this state may encompass multiple sub-behaviours called substates within the team’s code. It is important to note that by having functions that are repeatedly called instead of taking control within them, this allows for more fluid transitions between the behaviour states, notably if a bumper is struck.

The main behaviour states are: Bumper, Explore (the main/default state), and Scanning. The Bumper State, detailed in Section 3.2.4, handles the bumper “interrupts”. This means that when this state is achieved, a bumper has been pressed on the robot. The program adjusts the robot to avoid or move away from the obstacle it impacted in order to release the bumper and allow it to either explore or scan the surroundings. The Explore State is the main state the robot takes while navigating the environment. In this state, the robot traverses the world randomly, while primarily sticking to straight lines until obstacles are encountered. Occasionally, the robot will conduct a scan to map out features of the environment as it explores. The third state is the Scanning State, which performs a complete rotation and records the minimum distance at regular intervals using the laser scan sensor. Once the scan is complete, the program aligns the robot with a heading based on a weighted decision that favours directions with larger distances from any obstacles, before handing control back to the explore code. The details of the code used in the algorithm is explained in the sections below.

3.2.1 Contest 1 Main Code

The main code for the robot. It initializes the ROS systems the robot depends on, such as subscribing to different sensor data streams and configuring the control loop. Before entering the control loop to execute the robot’s behaviour.

The main control loop begins by processing all the ROS callbacks to acquire the most recent set of sensor data. It then calls on the different behaviours depending on the state (*state*) it is currently in and keeps track of their sub states (*subState*). If the state is ever changed the robot resets the sub state to zero to properly initialize the newly selected state. These behaviours merely configure the robot’s motion or future state, but never actually do any publishing within themselves so they must be repeatedly called to complete their task. This is intentional so the callbacks can be properly processed by the system with every loop iteration.

For example, if the explore state had its own internal loop for updating ROS callbacks and publishing the motion it needs, the team may forget to include code to handle a bumper event in that function and thus have undesired behaviour from the robot overall.

It finishes each loop cycle by publishing the current motion to the robot to perform it, and keeping track of the time elapsed so the robot stops at 15 minutes.

3.2.2 Globals Header

This header is used to simplify the production of other code files by including all the common libraries and global constants and variables that may be needed in code in one file, so the other files only have to include "global.h" and they will have access to them all.

All global variables are declared here using the *extern* prefix (e.g. *linear*) to signify in files that include the global header, that they are declared as global in another .cpp file more specific to that variable (i.e. *movement.cpp*). Most global constants, such as *EXPLORE_STATE*, are declared directly in the file, since they are used in multiple places in multiple files.

Other than libraries and global constants and variables, there are also two *#define* statements to create the *DEG2RAD* and *RAD2DEG* "functions" in code across the system.

3.2.3 Movement Code

The robot's movement is controlled using velocities, so achieving controlled displacements (e.g. moving 0.87m forwards) needs to be implemented in software, this is the primary purpose of the movement library, ultimately simplifying the control of the robot for other functions.

All movement done by the robot is handled by these functions. The process is the same for any motion, first the velocities and displacements (angular and linear) desired must be set using the *setMotion*, then the motion is maintained by calling *monitorMotion* until it is complete. Motion is maintained by ensuring that the velocities are kept at the specified levels, and checking that the robot's displacement requirements have been met using the robot's odometry. Once either linear or angular displacement requirements have been met, the velocity of that axis of motion is set to zero to prevent further overshoot. Once both displacements have been met, the motion is determined to be completed and a value of true is returned by *monitorMotion* (it returns false if the motion is still on-going).

Using odometry is helpful to the team because it ensures that the robot has truly performed the desired displacement in comparison to a time-based system that predicts when the motion should be completed. Odometry is handled by the callback *odomCallback* which updates the robot's estimated position (*posX*, *posY*) and orientation (*yaw*) in the world.

To further simplify the coding for motion for the team, two other functions were designed for motion, *travel* and *setHeading*. The function *travel* takes in the desired displacement and velocities and handles calling *setMotion* and *monitorMotion* within itself which simplifies code elsewhere. It achieves this by monitoring the input parameters and comparing them to the previous ones passed in. Should it be called with new parameters it will know that a new motion is requested and set it up using *setMotion* otherwise it will call *monitorMotion* and pass the status of the motion (complete or not) back to where it was called.

The *setHeading* function is used to align the robot with a desired heading (yaw) value. It works similarly to *travel* by monitoring the parameters passed into it for changes to see if new motion is desired or it needs to simply monitor the motion. It however, contains code to determine the shortest way to align the robot with the desired heading based on its current heading as part of its

set up code after new parameters have been detected, before calling *travel* to begin this motion. When sustaining motion it just calls *travel* with the parameter determined when setting up the alignment. It returns the value returned to it by *travel* allowing the user to know when alignment to the new heading is complete.

3.2.4 Bumper Code

The bumper header (see Appendix B), and thus the bumper code, includes the *globals* and *movement* headers, as it only relies on some global variables and the functions stored in *movement* in order to move the robot once a bumper registers that it has been struck/hit by an obstacle. The callback functions are called with each iteration of the while loop in the main code via the *ros::SpinOnce()* command. This means that the code is constantly checking to see if any new data has been published by the bumpers to indicate that one of the three bumpers has been struck while the robot was either exploring or scanning. Thus, the bumper code acts as an interrupt to shift the program away from what it was currently doing to react to the hit quickly and efficiently.

The operation of the bumper code (and, by extension, the portion of the main code that calls the *bumperMove* function) is outlined below. See Appendix A and C to view the main code and the bumper code. As mentioned previously, the main code checks the *bumperCallback* function to determine if any of the three bumpers have been pressed. If not, the robot will continue to either explore (see Section 3.2.6 Exploration Code) or scan (see Section 3.2.7 Scanning Code). If it is determined that a bumper has been pressed, the *bumperCallback* function executes to check which bumper was last pressed (in the event multiple ones were pressed due to the nature of the collision) with the use of the *bumperName* function and then outputs this information to the user (i.e., “Right bumper impact detected.”).

The program then sets the state to *BUMPER_STATE* to handle the motion to move the robot away from the obstacle. This results in the main code calling the *bumperMove* function to execute the movement commands. Once called, this function moves the robot backwards 0.1 m at a rate of 0.25 m/s, so that it is no longer in contact with the obstacle. It then rotates a certain direction according to which bumper was struck:

- If the left bumper was struck, the robot rotates π/4 rad clockwise at a rate of π/6 rad/s.
- If the centre bumper was struck, the robot rotates π/2 rad counterclockwise at a rate of π/6 rad/s.
- If the right bumper was struck, the robot rotates π/4 rad counterclockwise at a rate of π/6 rad/s.

Once the movement is complete, the state of the robot is set to *EXPLORE_STATE* to move the program into the explore code. Lastly, the bumper code outputs to the user that the bumper correction has been completed.

3.2.5 Laser Code

The laser header only includes the *globals* header for global variables and one function used for laser callback. The laser code only handles reading data that is published by the laser scanner sensor on the robot. With each iteration of the while loop in the main code, the callback function for the laser scanner is called, although it is not used in the main code. Instead, the code is used extensively in the exploring and scanning codes to help navigate and map the surroundings.

The *laserCallback* function first reads in data published from the laser scanner sensor and determines the number of laser beams available. It then narrows down the selection of values to a desired range around the centre, which is determined by a desired angle set by the user prior to running the code. The size of the laser scan array (number of laser beams available), the size of the offset (number of lasers stored given the desired range), and the width of the laser scan array are output to the user.

Next, the function finds the minimum distance value in the narrowed range as well as taking the average to the valid distances. If the scanner cannot make an accurate measurement due to the limitations of the sensor in the simulator (i.e., it enters a "dead zone"), a distance of infinity is returned. Through testing, the lowest valid measurement observed was approximately 0.45 m. Due to this issue, if the minimum value is infinity or less than 0.46 m, the program sets it to zero so the rover treats it as a contact. Therefore, in the exploration code, when the robot approaches an obstacle, if the minimum distance is calculated to be less than 0.46 m or infinity, the robot will stop and rotate either clockwise or counterclockwise away from the obstacle and continue to move forward and explore.

One issue the team noticed was that the laser scanning sensor seemed to be unreliable during testing of the code. In essence, the robot would move forward and begin to approach a wall/obstacle head on, meaning the object could be picked up by the laser scanner. It then proceeded to move forward into the object, resulting in a collision. Other instances, the laser scanner performed as expected and helped navigate a robot out of a space without any collisions. Moving forward, the team will look to implement more robust code to deal with this issue.

3.2.6 Exploration Code

This code is responsible for the robot's exploration behaviour, and is run for the majority of the robot's run. It moves the robot forward and then randomly decides whether to randomly pick a direction or conduct a scan.

The first step is to determine the available distance in front of the robot for motion, which is done using the laser scanner data, *minLaserDist*, and subtracting a set clearance the team wants the robot to maintain from any given obstacle before it. For example, if the *minLaserDist* is 0.9m, and the desired clearance is 0.1m, the clearance to move is 0.8m. This is compared to the team's maximum step (0.75m) whichever is the limiting value (in this example 0.75m) is selected. The robot then moves to the next substate responsible for moving forward this desired amount.

Once the forward motion has been completed the robot advances to the next substep of exploration, deciding what to do next. This is done by randomly deciding a number between zero and 100. If this number is smaller than our scanning threshold (*scanChance*, 30) then the robot's state is set to scanning. Otherwise the robot will randomly select a direction. This threshold between scanning and randomly turning was empirically chosen by the team after it was observed to provide a reasonable balance between speed, good mapping,

If the robot decides to head in a random direction, it first checks the presence of obstacles in front of it using the laser scanner data once again. If obstacles are present the rover sets the angle generator (*centeredAngle*) to turn to start past zero (*minDeflection*, 50 degrees) so it turns left or right. The maximum angle from the angle generator will always be *maxDeflection*, currently 135 degrees. The angle generator generates a value between the minimum and maximum value passed in, but will favour values closer to the lower end. This is so the robot generally takes random steps that would keep it going relatively forward so it does not waste time backtracking.

The robot then turns itself to this new angle and checks for obstacles present, using the laser scanner. If there is something present, the robot will turn to the other side by doubling the turn magnitude and switching the direction before continuing.

With that done, the code restarts the exploration state by setting its sub state to be zero.

3.2.7 Scanning Code

This code is responsible for handling the robot's occasional rotating scans of its environment. It has the robot spin a full rotation made up of several steps. At each step in the rotation the laser scanner readings are recorded as well as the robot's current absolute heading (yaw).

Once the robot completes its rotation it will go through all these recordings and select one randomly, weighed by the distance available to travel for each step, so it seeks open areas. This weighted direction is calculated by determining the sum of all distances found during the scan, picking a number between zero and this total, and seeing going around through the steps to see which value "buckets" this value.

For example if the robot scans north (1.5m), west (1.5m), south(2.0m), then east (1.0m) the total will be 6.0m. The robot will pick a number between 0 and 6.0, e.g. 3.5m. The rover will then go through the step until the accumulated distance exceeds this number, which will be south (1.5 + 1.5 + 2.0 = 5.0m). This is shown in Table 1.

Table 1: Weighted direction example

Step	Name	Distance	Accumulated Distance
1	North	1.5m	1.5m
2	West	1.5m	3.0m
3	South	2.0m	5.0m
4	East	1.0m	6.0m

A weighted random decision is used over an absolute maximum to avoid the robot getting stuck in a loop if the geometry of the obstacles around it generate a condition that would trap such behaviour. For example if the robot was travelling down a long corridor it would cycle between the two ends with no chance of escaping.

The final step in the scanning code is to have the robot align itself with the selected heading using the movement functions. Once aligned it switches the state back to exploration.

4.0 Future Recommendations

One item the team wished they had time to implement was extracting data from the map created by the ROS gmapping package. This would help inform decisions on where the robot should move next (frontier-based exploration). Currently, the robot experiences issues by becoming "trapped" in certain areas of the environment due to the layout of the practice world (i.e., areas that are heavily enclosed and only have one square of space to exit). Since the approach to the movement is a weighted random walk, the robot will not actively move towards empty locations in the map to actually "explore" those locations.

Implementing a frontier-based exploration requires a more developed localization strategy (e.g., implementing some form of Monte Carlo localization) and a deeper understanding of the gmapping package. Combining the localization of the robot with the occupancy grid map created by the gmapping world would allow the team to develop a cost/reward function for points on the robot's frontier. It would then generate a path to the desired location using a pathfinding algorithm (e.g., A*). This would allow for a more focused exploration strategy than the weighted random walk the team used for contest 1.

Another major item the team would approach differently if more time was granted would be to tweak the parameters of the code, such as desired laser range or step size used to explore, and observe their influence on the robot's performance. This would allow the team to see what sets of parameters best handle different types of maps (e.g., open vs closed), as well as the quality of maps generated. In addition, this would allow for more in-depth troubleshooting of common issues, such as getting stuck in certain obstacle configurations.

Lastly, due to the issue with the laser scanner (mentioned previously in the laser code section), the team would have liked more time to properly assess the issues of the scanner and determine a robust algorithm to deal with them. During simulation, the scanner seemed to provide unreliable readings. At times, the robot would sense obstacles (even ones close up) and maneuver around them; however, in other instances, the robot collided with obstacles head on, even when it had adequate time and space to sense the obstacle and avoid it. Moving forward, the team will continue to test the laser scanning sensor to determine how best to code for these issues, or to determine the root of the issues and attempt to fix them if possible.

5.0 Contribution Table

Contributions noted by value: 1 - small amount, 3 - majority, blank for none.

Table 2: Contribution Table

Section	**Savo Bajic**	**Maximilian Glidden**	**Catherine Kucaba**
1.0			3
2.0	1	3	
3.1	1	3	
3.2	3	1	2
4.0	1	3	2
Robot Code	3	1	2

6.0 Appendices

Appendix A : contest1.cpp Code

```
#include "globals.h"
#include "bumper.h"
#include "laser.h"
#include "movement.h"
#include "scanning.h"
#include "explore.h"

/* State of robot
 0 - Initialization / start
 1 - Exploring
 2 - Bumper collision event
*/
u_int8_t state = SCAN_STATE;

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

   ros::Subscriber bumper_sub = nh.subscribe("mobile_base/events/bumper",
10, &bumperCallback);
   ros::Subscriber laser_sub = nh.subscribe("scan", 10, &laserCallback);
   ros::Subscriber odom = nh.subscribe("odom", 1, &odomCallback);
   ros::Publisher vel_pub =
nh.advertise<geometry_msgs::Twist>("cmd_vel_mux/input/teleop", 1);

   ros::Rate loop_rate(10);

   geometry_msgs::Twist vel;

   // contest count down timer
   std::chrono::time_point<std::chrono::system_clock> start;
   start = std::chrono::system_clock::now();
   uint64_t secondsElapsed = 0;
```

```
    u_int8_t lastState = 0;          // Used to check if state was changed
between loop iterations by the most recent callbacks
    u_int8_t subState = 0;           // Used for states within states
    u_int32_t iterationCount = 0;    // Used to keep track of iterations for
timed events like distance traversal

    // Process the callbacks once before entering the loop to kick things
off.
    ros::spinOnce();
    loop_rate.sleep();

    ROS_INFO("STARTING MAIN LOOP");

    while(ros::ok() && secondsElapsed <= 900) {
        ros::spinOnce(); // Check callback functions
        iterationCount++;
        bool doneMoving = false;

        ROS_DEBUG("Postion: (%f, %f) Orientation: %f degrees Range: %f",
posX, posY, RAD2DEG(yaw), minLaserDist);

        // Check if state is the same as the last loop
        if (lastState != state) {
            lastState = state;
            subState = 0;
            iterationCount = 0;
        }

        ///////////////////////////////////////////
        // Behaviour states
        // All in one 'if' block so only one is exectuted per loop cycle,
even when a change occurs

        if (state == BUMPER_STATE) {
            bumperMove(subState);
        }
        else if (state == EXPLORE_STATE) {
            explore(subState);
        }
        else if (state == SCAN_STATE) {
```

```
            scanAroundSpot(subState);
        }

        vel.angular.z = angular;
        vel.linear.x = linear;
        vel_pub.publish(vel);
        ROS_DEBUG("Current speeds linear: %.2f m/s angular: %.3f deg/s",
linear, RAD2DEG(angular));

        // 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 B : bumper.h Code

```
#ifndef BUMPER_HEADER
#define BUMPER_HEADER

#include "globals.h"
#include "movement.h"


void bumperCallback(const kobuki_msgs::BumperEvent::ConstPtr& msg); //
Handles bumber events
void bumperName(u_int8_t index, char output[]); // Used to write which
side got hit given the index
void bumperMove(u_int8_t &subState);

#endif
```

Appendix C : bumper.cpp Code

```
#include "bumper.h"

// Bumper global variables
uint8_t bumper[3] = {kobuki_msgs::BumperEvent::RELEASED,
kobuki_msgs::BumperEvent::RELEASED, kobuki_msgs::BumperEvent::RELEASED};
bool anyBumperPressed = false;  // Stores overall bumper status
uint8_t lastBumper = 0;         // Stores last bumper pressed

// Retreat constants
const float BUMPER_RETREAT_SPEED = FAST_MOVE;       // Linear retreat
speed m/s (must be positive)
const float BUMPER_RETREAT_DIST = 0.1;              // Linear retreat
distance
const float BUMPER_RETREAT_ROT = M_PI / 4;          // Angle to twist if a
side bumper is struck (rad)
const float BUMPER_RETREAT_ROT_VEL = FAST_SPIN;     // Twist rate (rad/s)

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

```
    // Inform user if a bumper is released
    if (msg->state == kobuki_msgs::BumperEvent::RELEASED) {
        char sideName[10];
        bumperName(msg->bumper, sideName);
        ROS_DEBUG("%s bumper (%d) released.", sideName, msg->bumper);
    }

    // Example of single bumper monitoring
    // uint8_t leftState = bumper[kobuki_msgs::BumperEvent::LEFT];
    // kobuki_msgs::BumperEvent::PRESSED if bumper is pressed,
kobuki_msgs::BumperEvent::RELEASED otherwise

    // Record if any bumper is pressed and record that
    anyBumperPressed = false;
    for (u_int8_t i = 0; i < N_BUMPER; i++) {
        if (bumper[i] == kobuki_msgs::BumperEvent::PRESSED) {
            anyBumperPressed = true;

            // Inform user that bumpers are in contact
            char sideName[10];
            bumperName(i, sideName); // Writes which side got hit to
sideName
            ROS_WARN("%s bumper impact detected.", sideName);

            state = BUMPER_STATE; // Set robot to bumper state in main loop
            lastBumper = i; // Records which bumper was last struck
        }
    }
}

void bumperName(u_int8_t index, char sideName[])
{
switch (index) {
    case 0:
        strcpy(sideName, "Left");
        break;
    case 1:
        strcpy(sideName, "Center");
        break;
```

```
   case 2:
       strcpy(sideName, "Right");
       break;
}
}

void bumperMove(u_int8_t &subState)
{
   bool doneMoving = false;

   // Linear retreat from obstacle
   if (subState == 0) {
       doneMoving = travel(BUMPER_RETREAT_DIST, (-BUMPER_RETREAT_SPEED),
0, 0);

       if (doneMoving) {
           subState++;
           ROS_DEBUG("Done backing up from impact.");
       }
   }

   // Set direction based on which bumper was last hit
   if (subState == 1) {
       // Set robot to spin according to last bumper hit (0 is leftmost
bumper)
       switch (lastBumper) {
           case 0:
               doneMoving = travel(0,0, BUMPER_RETREAT_ROT,
(-BUMPER_RETREAT_ROT_VEL));
               break;
           case 1:
               doneMoving = travel(0, 0, M_PI / 2,
BUMPER_RETREAT_ROT_VEL);
               break;
           case 2:
               doneMoving = travel(0, 0, BUMPER_RETREAT_ROT,
BUMPER_RETREAT_ROT_VEL);
               break;
       }
   }
```

```
    // Wait until done spinning to resume operation
    if (doneMoving) {
        ROS_INFO("Bumper correction complete.");
        state = EXPLORE_STATE;
    }
}
```

Appendix D : explore.h Code

```
#ifndef EXPLORE_HEADER
#define EXPLORE_HEADER

#include "globals.h"
#include "movement.h"

// Explore related functions
void explore(u_int8_t &subState);
float centeredAngle(float low, float high);
#endif
```

Appendix E : explore.cpp Code

```
#include "explore.h"

// Deflection allowance when randomly turning
const uint8_t minDeflection = 50;    // Used when we can't go forward
const uint8_t maxDeflection = 135;

const uint8_t scanChance = 30; // Chance to scan (of 100)

void explore(u_int8_t &subState)
{
    static float stepToTake, speedToExplore;
    bool doneMoving = false;

    // Bumble about
    if (subState == 0) {
        //Determine step to take
        ROS_INFO("Minimum laser distance is %.2fm.", minLaserDist);
```

```
        stepToTake = std::max(0.0f, minLaserDist - WALL_CLEARANCE); //Used 
to avoid negative is already too close
        stepToTake = std::min(EXPLORE_STEP, stepToTake);

        // See if we'll stay out of the slow zone or not
        if ((minLaserDist - stepToTake) < SLOW_DOWN_DIST) speedToExplore = 
SLOW_MOVE;
        else speedToExplore = FAST_MOVE;

        ROS_INFO("Exploring. Going forward %.2fm at a speed of %.2f m/s.", 
stepToTake, speedToExplore);
        subState++;
    }

    // Do the bumbling about
    if (subState == 1) {
        doneMoving = travel(stepToTake, speedToExplore, 0, 0);

        // If done bumbling about, randomly decide between a random turn or 
a scan
        if (doneMoving == true) {
            uint8_t diceRoll = rand() % 100; // "!rtd" at ctf_2fort

            if (diceRoll < scanChance) state = SCAN_STATE; // Do a proper 
scan
            else {
                // Determine how much to spin, up to a set max to not go 
backwards
                // If we're close to the wall, we want the spin to pass a 
minimum to go away
                if (minLaserDist < (SLOW_DOWN_DIST / 2.0)) {
                    stepToTake = DEG2RAD(centeredAngle(minDeflection, 
maxDeflection));
                }
                else stepToTake = DEG2RAD(centeredAngle(0, maxDeflection));

                // 50/50 to going left or right
                if ((rand() % 2) == 1) speedToExplore = FAST_SPIN;
                else speedToExplore = (-FAST_SPIN);
```

```
                ROS_INFO("Turning randomly. %.0f degrees, at a rate of %.0f
deg/s.", RAD2DEG(stepToTake), RAD2DEG(speedToExplore));
                subState++;
            }
        }
    }

    // Spin to new position and bumble again
    if (subState == 2) {
        doneMoving = travel(0, 0, stepToTake, speedToExplore);
        if (doneMoving) subState++;
    }

    if (subState == 3) {
        // Check if this direction is open
        if (minLaserDist < (SLOW_DOWN_DIST / 2.0)) {
            // Not open, swing around to opposite side
            subState++;
            stepToTake = 2 * stepToTake;
            speedToExplore = -speedToExplore;
        }
        else subState = 0;
    }

    // Swing around to other side
    if (subState == 4) {
        doneMoving = travel(0, 0, stepToTake, speedToExplore);
        if (doneMoving) subState = 0;
    }
}

float centeredAngle(float low, float high)
{
    // Find the range we're allowed
    float range = high - low;

    // Gets a random number in the range
    float angle = rand() % int(range);
```

```
    // Recalculate new angle, weighed towards 0 (to keep the robot going
centered)
    const float weight = 3.0;  // Weighing power, the higher this is the
tighter around 0 the results
    angle = pow(angle, weight) * (range / pow(range, weight));

    // Add offset for the start of the range
    angle = angle + low;

    return angle;
}
```

Appendix F : globals.h Code

```
/* Global declaration file

All libraries are #include'd here so we only have to call
#include "globals.h" once at the start of every file. It also
serves the more important purpose of allowing the global
variables to be shared across multiple files without issue
(as long as it is used correctly! (as with most things)).

This allows callback functions to be defined seperately of
the main file for easier collaboration.

To add global variables they need to be defined once in the
conventional sense in the .cpp file. Then it has to be once again
defined in this file, prefaced with "extern" so the compiler
knows that the global variable is defined in another place.

For example with the bumper pressed variable:

    // Conventional declaration in "bumper.cpp"
    bool any_bumper_pressed = false;

    // Extern declaration in this file
    extern bool any_bumper_pressed;
    // Note the lack of a value assignment here

*/
```

```
#ifndef GLOBAL_HEADER
#define GLOBAL_HEADER

// Libraries we need
#include <ros/console.h>
#include "ros/ros.h"
#include <geometry_msgs/Twist.h>
#include <kobuki_msgs/BumperEvent.h>
#include <sensor_msgs/LaserScan.h>
#include <nav_msgs/Odometry.h>
#include <tf/transform_datatypes.h>
#include <stdio.h>
#include <cmath>
#include <chrono>

// Some misc. stuff
const uint8_t N_BUMPER = 3;
#define RAD2DEG(rad) ((rad) * 180. / M_PI)
#define DEG2RAD(deg) ((deg) * M_PI / 180.)

// Motion variables
extern float angular;
extern float linear;
const float SLOW_SPIN = (M_PI/12);
const float FAST_SPIN = (M_PI/6);
const float SLOW_MOVE = 0.1;
const float FAST_MOVE = 0.25;
const float MAX_LIN = 0.25;
const float MAX_ROT = (M_PI/6);

// Robot state
extern u_int8_t state;
const u_int8_t SCAN_STATE = 0;
const u_int8_t EXPLORE_STATE = 1;
const u_int8_t BUMPER_STATE = 2;

// Odometery values
extern float posX, posY, yaw;

// Bumper values
```

```
extern uint8_t bumper[3];
extern bool anyBumperPressed;
extern uint8_t lastBumper;

// Laser related values
extern float minLaserDist;
extern int32_t nLasers, desiredNLasers; // Number of laser beams
available, how many we will store given our desired range
extern int32_t desiredAngle;    // Desired view range aropund center axis,
in degrees
extern float aveLaserDist;

// Motion related
const float WALL_CLEARANCE = 0.1;    // Distance to keep from obstacles
const float SLOW_DOWN_DIST = 0.3;      // Proximity to start slowing slow
down at
const float EXPLORE_STEP = 0.75;      // Maximum exploration step to take
#endif
```

Appendix G : laser.h Code

```
#ifndef LASER_HEADER
#define LASER_HEADER

#include "globals.h"

// Laser related functions
void laserCallback(const sensor_msgs::LaserScan::ConstPtr& msg);

#endif
```

Appendix H : laser.cpp Code

```
#include "laser.h"

// Laser related values
float minLaserDist = 0;          // Minimum distance picked up but most
recent laser scan
int32_t nLasers = 0;             // Number of laser beams available
int32_t desiredNLasers = 0;      // Number of lasers we will store given
our desired range
```

```
int32_t desiredAngle = 10;      // Desired view range around center axis,
in degrees
float aveLaserDist = 0;         // Average laser distance on last scan

void laserCallback(const sensor_msgs::LaserScan::ConstPtr& msg)
{
   // Set globals to defaults
   minLaserDist = std::numeric_limits<float>::infinity();
   aveLaserDist = 0;

   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);
   //ROS_INFO("Width of laser scan array %.3f to %.3f",
RAD2DEG(msg->angle_min), RAD2DEG(msg->angle_max));

   // Find minimum distance in selected FOV range
   // Set up loop limits based on our desired range
   uint32_t startIndex = 0, endIndex = nLasers; // Default to max range
   if (desiredAngle * M_PI / 180 < msg->angle_max && -desiredAngle * M_PI
/ 180 > msg->angle_min) {
       // If narrorer than allowed range
       startIndex =  nLasers / 2 - desiredNLasers;
       endIndex = nLasers / 2 + desiredNLasers;
   }

   // Loop through finding minimum and adding non-infinite values to get
an average
   for (uint32_t laser_idx = startIndex; laser_idx < endIndex;
++laser_idx) {
       float reading = msg->ranges[laser_idx];
       minLaserDist = std::min(minLaserDist, reading);
       if ((reading > 0) && (reading !=
std::numeric_limits<float>::infinity())) {
           aveLaserDist = aveLaserDist + reading;
       }
   }
```

```
    aveLaserDist = aveLaserDist / float(nLasers); // Divide sum to get
average

    // The laser scanner we use has a lower limit of 0.45m, anthing closer
is registered as infinity.
    // So if the minimum recording is 0.45m or inf, we treat it as a
contact
    if (minLaserDist < 0.46) minLaserDist = 0;
    if (minLaserDist == std::numeric_limits<float>::infinity())
minLaserDist = 0;

    ROS_DEBUG("Laser scan complete. Min: %.2fm, Average: %.2fm.",
minLaserDist, aveLaserDist);
}
```

Appendix I : movement.h Code

```
#ifndef TEAM_MOVEMENT_HEADER
#define TEAM_MOVEMENT_HEADER

#include "globals.h"

void odomCallback (const nav_msgs::Odometry::ConstPtr& msg);
bool monitorMotion ();
void setMotion(double dist, double linSpeed, double rot, double rotSpeed);

/** @name travel
  *  @brief Use this for controlled travel by repeatedly calling it.
  *  @param  dist Linear distance to travel (magnitude)
  *  @param  linSpeed Linear velocity (determines forwards or back)
  *  @param  rot Rotational displacement in radians (magnitude)
  *  @param  rotSpeed Rotational velocity rad/s (+ive is left)
  *  @return  If the movement is complete, true.
  *
  *  @note It knows when you call it the first time or change course, so
long as the parameters change.
  *  (Calling travel(1,1,0,0) and then travel(1,1,0,0) again after that is
completed will not register
  *  as a new "move", these need to be seperated by a unique call such as
travel(0,0,0,0) to repeat)
  */
```

```
bool travel(double dist, double linSpeed, double rot, double rotSpeed);

/** @name setHeading
* @brief Will align the robot optimally with some absolute heading.
* @param heading The absolute heading (rad) to aim for
* @param speed Magnitude of rotation (rad)
* @return If the movement is complete, true.
* */
bool setHeading(float heading, float speed);
#endif
```

Appendix J : movement.cpp Code

```
#include "movement.h"

// Movement values
float angular = 0.0; // Global
float linear = 0.0;  // Global

// File scoped globals
float distanceRemaining, prevX, prevY;
float rotationRemaining, prevYaw;
float rotMaintain, linMaintain;

// Odometery values
float posX = 0.0, posY = 0.0, yaw = 0.0;

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));
}

bool monitorMotion ()
{
   //Maintain speeds
   angular = rotMaintain;
```

```
    linear = linMaintain;

    // See how mauch distance has been traversed
    float dx = posX - prevX;
    float dy = posY - prevY;
    float displacement = sqrt((dx * dx) + (dy * dy));
    prevX = posX;
    prevY = posY;

    // Find remaining distance to travel
    distanceRemaining = distanceRemaining - displacement;
    if (distanceRemaining <= 0.0) linear = 0; // Stop

    // Monitor rotational displacement
    displacement = std::abs(yaw - prevYaw);
    // Check if displacement is too big (crossing from -pi to +pi)
    // Heading is kept in the range [-pi,+pi] so when turning over pi
    // e.g. -0.98PI to 0.98PI, we need this case
    if (displacement > M_PI) {
        displacement = (M_PI - abs(yaw)) + (M_PI - abs(prevYaw));
    }
    rotationRemaining = rotationRemaining - displacement;
    if (rotationRemaining <= 0.0) angular = 0; // Stop when we've rotated
as required
    prevYaw = yaw;

    rotMaintain = angular;
    linMaintain = linear;

    // Return true if motion completed (no speed)
    bool doneMotion = (angular == 0) && (linear == 0);

    if (doneMotion) ROS_DEBUG("Reached destination.");
    else {
        ROS_DEBUG("In motion: D:%.2f S:%.2f | A:%.0f S:%.0f",
distanceRemaining,
        linear, RAD2DEG(rotationRemaining), RAD2DEG(angular));
    }
    return doneMotion;
}
```

```
void setMotion(double dist, double linSpeed, double rot, double rotSpeed)
{
   // Record new travel requirements
   distanceRemaining = std::abs(dist);
   rotationRemaining = std::abs(rot);

   // Set start point
   prevX = posX;
   prevY = posY;
   prevYaw = yaw;

   // Need to change the global variable
   linear = linSpeed;
   angular = rotSpeed;

   // Records speeds to maintain
   rotMaintain = angular;
   linMaintain = linear;

   ROS_DEBUG("Set up motion: D:%.2f S:%.2f | A:%.0f S:%.0f",
distanceRemaining, linear,
       RAD2DEG(rotationRemaining), RAD2DEG(angular));
}

bool travel(double dist, double linSpeed, double rot, double rotSpeed)
{
   static double pd = 0, pls = 0, pr = 0, prs = 0; // Used to store
previous state of inputs

   // Check if this is a repeated call
   bool repeatedCall = false;
   if ((dist == pd) && (linSpeed == pls) && (rot == pr) && (rotSpeed ==
prs)) repeatedCall = true;
   pd = dist;
   pls = linSpeed;
   pr = rot;
   prs = rotSpeed;

   bool doneMotion = false;
```

```
    if (repeatedCall == true) {
        // Repeated call, so we're maintaining course
        doneMotion = monitorMotion();
    }
    else {
        // New call, set new course
        setMotion(dist, linSpeed, rot, rotSpeed);
    }

    return doneMotion;
}

bool setHeading(float heading, float speed)
{
    static float lastHeading = 0, lastSpeed = 0; // Used to monitor
parameters
    static float change = 0, rotVelocity = 0; // Internal parameters to
describe the change
    bool doneAlignment = false;

    if ((lastHeading == heading) && (lastSpeed == speed)) {
        // Parameters didn't change, repeat travel() call
        doneAlignment = travel(0, 0, change, rotVelocity);
    }
    else {
        // New parameters, therefore an initial call
        // Record parameters for future reference
        lastHeading = heading;
        lastSpeed = speed;

        // Determine change needed
        float changeForward, changeBackward, tempHeading;

        // Find forward displacement
        if (heading < yaw) tempHeading = heading + 2 * M_PI; // Loop around
        else tempHeading = heading;
        changeForward = tempHeading - yaw;

        // Find backwards displacement
```

```
        if (heading > yaw) tempHeading = heading - 2 * M_PI; // Loop back
        else tempHeading = heading;
        changeBackward = yaw - tempHeading;

        ROS_DEBUG("Going forward %.0f deg, backwards %.0f.",
RAD2DEG(changeForward), RAD2DEG(changeBackward));

        // Compare to see which direction is optimal
        if (changeBackward < changeForward) {
            // Going backwards
            rotVelocity = -speed;
            change = changeBackward;
            ROS_INFO("Aligning to heading %.0f deg, currently at %.0f.
Change of %.0f right.",
                RAD2DEG(heading), RAD2DEG(yaw), RAD2DEG(change));
        }
        else {
            // Going forwards
            rotVelocity = speed;
            change = changeForward;

            ROS_INFO("Aligning to heading %.0f deg, currently at %.0f.
Change of %.0f left.",
                RAD2DEG(heading), RAD2DEG(yaw), RAD2DEG(change));
        }



        // Start moving as needed
        doneAlignment = travel(0, 0, change, rotVelocity);
    }

    if (doneAlignment) ROS_INFO("Aligned with new heading.");
    return doneAlignment;
}
```

Appendix K : scanning.h Code

```
#ifndef SCANNING_HEADER
#define SCANNING_HEADER
```

```
#include "globals.h"
/** @name scanAroundSpot
* @brief Scans around robot current position and select a new heading.
* @param subState Substate variable from main loop
**/
void scanAroundSpot(u_int8_t &subState);

/** @name headingDecider
* @brief Scores and selects headings based on readings taken as robot
rotated
* @param yawVals Yaw values at each step of scan
* @param minDists Minimum distance recorded by laser scanner at each step
* @return Absolute heading (rad) that was selected.
*/
float headingDecider(const float yawVals[], const float minDists[]);

#endif
```

Appendix L : scanning.cpp Code

```
#include "scanning.h"
#include "movement.h"

// File globals
const uint8_t NUMBER_OF_STEPS = 12; // Number of stops taken around the
circle
const float STEP_INCREMENT = 2 * M_PI / NUMBER_OF_STEPS;

const float SCAN_SPEED = FAST_SPIN;
const float ALIGN_SPEED = FAST_SPIN;

void scanAroundSpot(u_int8_t &subState) {
   static float yawValues[NUMBER_OF_STEPS];
   static float minDistances[NUMBER_OF_STEPS];
   static float targetHeading;

   static uint8_t stepsCompleted;
   bool movementDone = false;

   // Reset for each spin
   if (subState == 0) {
```

```
        stepsCompleted = 0;
        subState++;
        ROS_INFO("Performing 360-degree scan.");
    }

    // Rotating
    if (subState == 1) {
        movementDone = travel(0, 0, STEP_INCREMENT, SCAN_SPEED);

        if (movementDone) {
            // Record parameters for this step
            yawValues[stepsCompleted] = yaw;
            minDistances[stepsCompleted] = minLaserDist;

            ROS_INFO("Step %d. Distance: %.2f, Yaw: %.0f.", stepsCompleted,
minLaserDist, RAD2DEG(yaw));

            stepsCompleted++;

            // Check if the robot needs to do more steps
            if (stepsCompleted < NUMBER_OF_STEPS) travel(0, 0, 0, 0); //
Clears old travel instruction (so we can repeat the same step one)
            else {
                // Find ideal heading and head that way for the next
substep
                targetHeading = headingDecider(yawValues, minDistances);
                subState++;
                ROS_INFO("Scan completed, heading selected %.0f.",
RAD2DEG(targetHeading));
            }
        }
    }

    // Align to heading to take
    if (subState == 2) {
        movementDone = setHeading(targetHeading, ALIGN_SPEED);

        if (movementDone) state = EXPLORE_STATE;
    }
}
```

```
float headingDecider(const float yawVals[], const float minDists[]) {
   float decidedHeading;
   // Calculate a score for each step
   float scores[NUMBER_OF_STEPS];
   float totalScore = 0;
   uint8_t selectedIndex = 0;

   for (uint8_t i = 0; i < NUMBER_OF_STEPS; i++) {
       scores[i] = minDists[i];

       totalScore = scores[i] + totalScore;
   }

   if (totalScore == 0) {
       ROS_ERROR("Laser scan came back with only 0's! Going straight.");
       return yaw;
   }

   // Weighted random for openness
   // Selects a random number from 0 to the cumulative scores for a scan.
   // Whichever scan step "buckets" this value is selected
   float selectedValue = 0;
   float accumulatedScore = 0;

   // Gets the selected value
   int32_t maxValue = int(floor(totalScore * 100.0)); // Get upper limit
of random value

   selectedValue = rand() % maxValue;
   selectedValue = selectedValue / 100.0;

   for (uint8_t i = 0; i < NUMBER_OF_STEPS; i++) {
       accumulatedScore = scores[i] + accumulatedScore;

       // Once the value is "bucketed" exit
       if (accumulatedScore > selectedValue) {
           selectedIndex = i;
           break;
       }
```

```
    }
    ROS_INFO("Total score %.2f, selected value %.2f, index %d.",
totalScore, selectedValue, selectedIndex);

    /*
    // Just going for max
    for (uint8_t i = 1; i < NUMBER_OF_STEPS; i++) {
        if (scores[i] > scores[selectedIndex]) selectedIndex = i; // Update
to new max
    }

    */
    decidedHeading = yawVals[selectedIndex];
    return decidedHeading; // Return ideal yaw
}
```

7.0 References

[1] *MIE443H1S: Contest 1: Where am I? Autonomous Robot Search of an Environment*, University of Toronto, Toronto, ON, 2021, pp. 1-2.