CONTEST 1 REPORT

UNIVERSITY OF TORONTO

MIE443 - MECHATRONICS SYSTEMS: DESIGN AND INTEGRATION

Where Am I?
Autonomous Robot Search of An Environment

Authors:

Project Group 18

Farhan Wadia — 1003012606

Henry Cueva Barnuevo — 1003585122

Yilin Huang — 1003145232

Professors:

Prof. Goldie Nejat

Dr. Meysam Effati

Head Course TA:

Pooya Poolad

February 24, 2021

Table of Contents

1.0 Problem Definition **2**

2.0 Functional Requirements **2**

2.1 Objectives 2

2.2 Constraints 2

3.0 Competition Strategy **3**

4.0 Sensor Design **4**

4.1 Microsoft Kinect RGBD Sensor 4

4.2 Odometry 5

4.3 Bumpers 5

4.4 IMU 5

5.0 Controller Design **6**

5.1 High-Level Control Architecture 6

5.1.1 No bumper pressed and finite minLaserDist 6

5.1.2 No bumper pressed and infinite minLaserDist 6

5.1.3 Bumper pressed 6

5.1.4 Potentially stuck 7

5.2 Low-Level Control Architecture 7

5.2.1 Choosing turn directions and/or angular velocities for the TurtleBot 7

5.2.2 Rotating a particular angle 8

5.2.3 Moving a particular distance 9

5.2.4 Bumper pressed 9

5.2.5 Moving after being stuck 9

6.0 Future Recommendations **10**

References **11**

Appendices **12**

Appendix A - Complete C++ Code 12

Appendix B - Attribution Table 21

1.0 Problem Definition

This problem is the first in a series of three contests related to the control and navigating capabilities of autonomous robots. For this first problem, a simulated TurtleBot will be required to autonomously drive around an unknown environment and dynamically map this environment as it traverses the area. This dynamic map will be generated using gmapping based on sensory information from the Kinect sensor, bumpers, and odometry, which are described in more detail in Section 4.0.

2.0 Functional Requirements

The objectives and constraints for this contest are discussed in the subsections below.

2.1 Objectives

In accordance with the contest requirements stipulated in Section 2.2, the following objectives have been identified:

1. The TurtleBot should maximize the amount of area it travels over and maps
2. The TurtleBot should be able to identify obstacles visually (ie. using the RGB sensor) as well as physically (ie. bumpers). Both should aid in the TurtleBot's ability to navigate the environment.
3. The TurtleBot should map the environment in the shortest possible time, minimizing the amount of time maneuvering around obstacles.
4. The TurtleBot should be able to use multiple sensors for estimating its state as a redundancy in case any single sensor gives faulty readings.

2.2 Constraints

The problem itself has several constraints including:

1. The environment that the TurtleBot is exploring and mapping is contained in a 6x6 m^2 2-dimensional simulated environment with static objects
2. The TurtleBot must both explore and map the environment in a maximum of 15 mins. It must also stop moving once it is done scanning, or when the 15 minute period is over (whichever occurs sooner)
3. It must be autonomous and use ROS gmapping libraries to create a map dynamically
4. The TurtleBot must move at a linear speed no greater than:
    a. 0.25 m/s when navigating the environment
    b. 0.1 m/s when close to obstacles. We define close to be < 0.6 m from an obstacle.

3.0 Competition Strategy

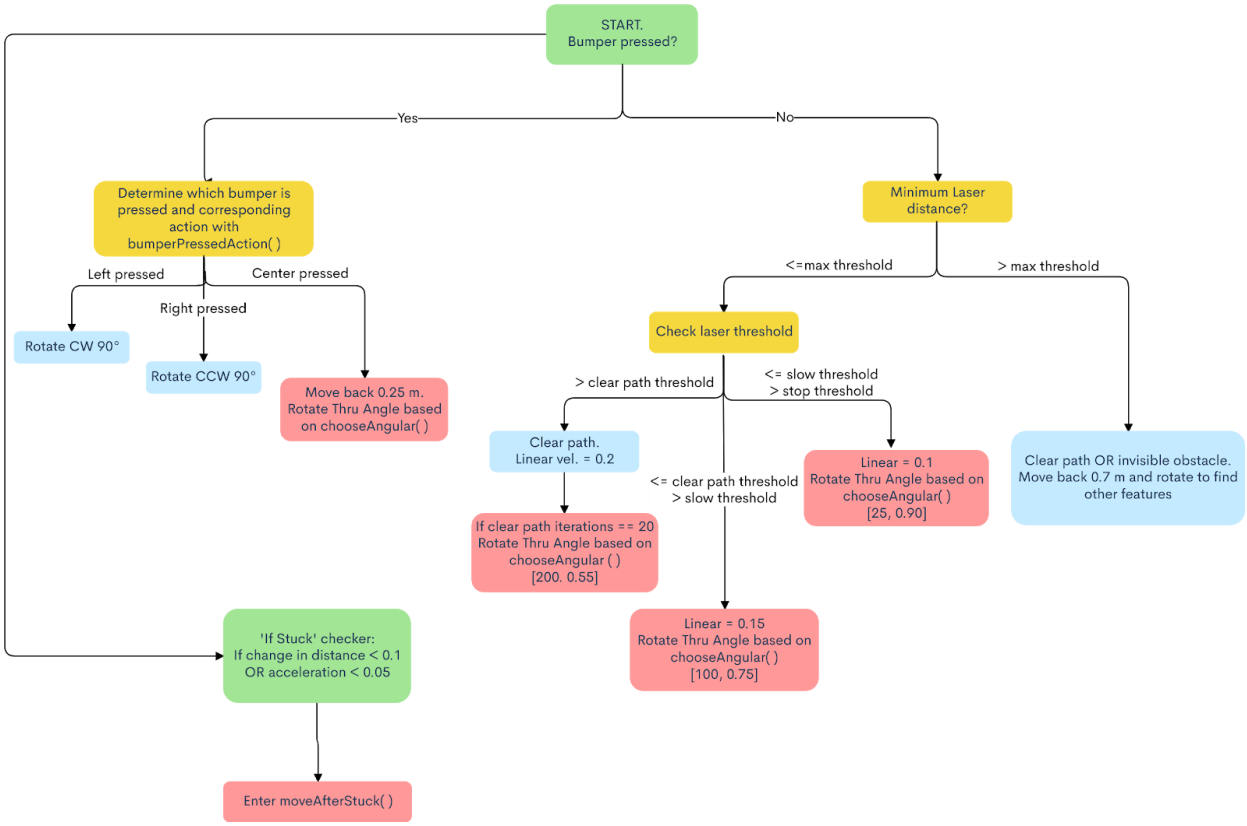

Figure 1: The high-level strategy for autonomous mapping of the environment. Green nodes represent initialization and continuous checks, yellow nodes represent decision points, blue nodes represent direct actions taken, and red nodes represent calls to handling functions and low-level controllers described in Section 5.2.

Based on material presented in the course lectures and discussions with the Lab TAs, the team chose to take a rule-based, behavioural approach to the strategy, known as 'reactive control', rather than a deliberate, plan-based strategy. A rule-based approach is the optimal choice for an unchanging environment with completely static elements that is of a moderate size (6 m x 6 m) [1], which is the environment for Contest 1 . Based on this, the main strategy consists of wall-following and random walk algorithms.

The team's initial approach was inspired by the behaviour-based control cockroaches use, and to therefore try and implement the RAMBLER algorithm discussed in [2], which was developed by studying cockroaches' behaviour within an arena. After trying to implement it, however, the team decided to simplify the behaviour to a more rule-based, reactive control strategy that was more applicable to the difficulty and size of this contest environment. This strategy is shown in Figure 1.

In essence, our team's strategy is a greedy strategy to seek out open spaces, but with biased random walks in the decision making so that the TurtleBot does not continuously seek out the same, most optimal open areas and rather tries to explore different areas. Simultaneously, the code checks for whether the TurtleBot has hit any of the bumpers in order to execute a fixed sequence of movements to move away from the obstacle that the bumper(s) hit. The program also actively monitors the angular velocity of the inertial measurement unit (IMU). The IMU in this case acts as a redundancy for faulty readings from the odometer that can occur when the TurtleBot gets stuck, and is described in further detail in Section 4.4.

4.0 Sensor Design

The simulated TurtleBot comes equipped with multiple sensors, including several of which are built into the Kobuki base. A layout of where these sensors are located is shown in Figure 2. For this contest, the bumpers, odometry, depth sensor (part of the RGBD sensor), and IMU are used. Each sensor and how it was used to meet the contest requirements are described in more detail in the following subsections.

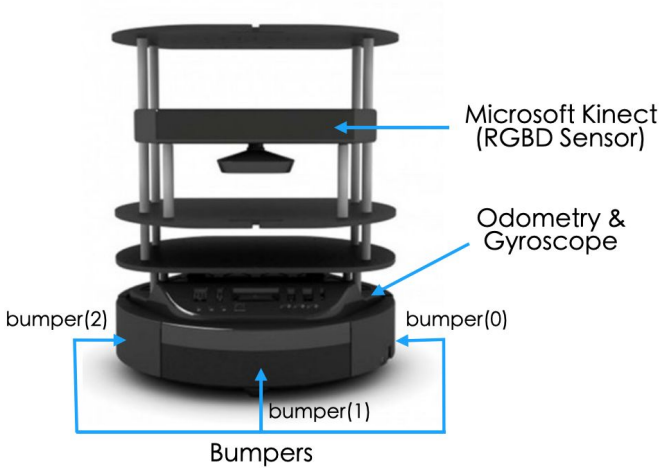

Figure 2: Placement of sensors on the simulated TurtleBot (Front view).

4.1 Microsoft Kinect RGBD Sensor

The Microsoft Kinect RGBD sensor is composed of two sensors, the RGB sensor and the depth sensor. For this first contest, solely the depth sensor is utilized, which we refer to as the 'lasers'. This sensor projects a speckle pattern of IR beams onto objects in the environment, which are then reflected and detected by the IR depth sensor. The distortion of the original pattern from this reflected pattern determines the depth information. This results in a 3D point cloud used to generate the map in RViz, but is not used in any of our algorithms.

During each iteration of our program's main loop, the laser sensor returns a one-dimensional array of laser distance measurements in a field of view 22.5° to the left and right of the TurtleBot's dead centre heading. We use this array in two ways:

1. We define `minLaserDist` as the minimum laser distance within the array, and use it as a metric to determine what the laser distance is at the current heading.
2. We define `LSLaserSum` and `RSLaserSum` as the sum of the last `n/2` and first `n/2` values respectively in the array, where `n` is the size of the array. Any array elements over 7 are not included in these sums, because any laser reading over 7 is larger than the size of the environment, and likely means the reading is infinity which, due to the lasers' limitations at close range, indicates the TurtleBot is actually very close to an obstacle. This threshold is referred to in the code as `maxLaserThreshold`. In essence, `LSLaserSum` and `RSLaserSum` represent which side of the TurtleBot is likely to have a clearer path. This is used extensively in other parts of the code to determine the direction of movement as part of our low-level controllers such as **`chooseAngular`**, all of which are discussed further in Section 5.2.

4.2 Odometry

The odometry sensor is used to estimate the TurtleBot's position and orientation relative to its starting location. From this sensor, we use `posX`, `posY`, and `yaw` to represent the horizontal position, vertical position, and angular orientation of the TurtleBot relative to its starting position.

We use these variables within the implementation of several of our lower-level controllers, such as to rotate a desired angle or move linearly for a desired distance.

4.3 Bumpers

There are three bumpers on the TurtleBot's Kobuki base located on the left, center and right sides of it, as shown in Figure 2. The bumpers are used when there are obstacles that the RGB sensor cannot see, such as the brick wall in the practice world simulations. The bumpers are initially set as released, with a state of 0, while a pressed bumper returns a state of 1.

Whenever a bumper is pressed, the program executes **`bumperPressedAction`**, a handling function which runs through the logic shown in Figure 1. This function is described further in Section 5.2.4.

4.4 IMU

The IMU sensor is used to obtain the angular velocity of the TurtleBot, `omega`. This measurement is primarily used to help determine when the TurtleBot may potentially be stuck. For example, when the **`rotateThruAngle`** (Section 5.2.2) function is called, there are sometimes cases where the `yaw` from odometry is changing, but the TurtleBot isn't actually moving since it has hit a wall on its backside, which is impossible to detect with the lasers or bumper. However, in such a situation, `omega` will have a low magnitude, and can therefore be used to check if the TurtleBot might be stuck and then execute **`moveAfterStuck`** (Section 5.2.5) to try and break out.

5.0 Controller Design

Section 5.1 discusses the high level control architecture that has been implemented in our program's **`main`** function, and Section 5.2 discusses the lower level control architectures that get called from **`main`**.

5.1 High-Level Control Architecture

Our high-level control architecture is essentially to seek out open spaces in conjunction with biased random walks. In each iteration of the program's main loop, we check for one of three states every time, and the potentially stuck state on only some iterations. These states are:

1. No bumper pressed and finite `minLaserDist`
2. No bumper pressed and infinite `minLaserDist`
3. Bumper pressed
4. Potentially stuck

5.1.1 No bumper pressed and finite `minLaserDist`

Within this state, there are another 3 sub-states: clear path, semi-clear path, and unclear path.

With a clear path (`minLaserDist > 0.7`), we set the linear speed of the TurtleBot to 0.2, and move straight. However, if the TurtleBot has been on a clear path for 20 consecutive iterations, it randomly rotates between ± 30° while continuing to move at 0.2 m/s. The magnitude of the turn is chosen completely at random, but the direction of the turn is chosen randomly but with a bias by the **`chooseAngular`** function. Further details for how this function implements a biased random turn direction can be found in Section 5.2.1.

With a semi-clear path (`0.6 < minLaserDist < 0.7`), the linear speed is reduced to 0.15 m/s, and a random turn with a higher bias as compared to the turns in the previous sub-state is used to guide the TurtleBot towards following the most clear path available.

For an unclear path (`minLaserDist < 0.6`), the speed is reduced to 0.1 m/s in order to satisfy the contest constraints, and the bias towards following the most clear path is increased to be even stronger.

5.1.2 No bumper pressed and infinite `minLaserDist`

In this state, the TurtleBot moves backwards 0.7m at 0.1 m/s to try and make a clear path by using the **`moveThruDistance`** function. Doing this is sufficient to handle this case because either the handling for checking if the TurtleBot is stuck will kick in if doing this action causes the TurtleBot to be stuck, **`moveThruDistance`** will exit after a finite number of iterations, or it will allow us to move back into the state with no bumper pressed and finite `minLaserDist`.

5.1.3 Bumper pressed

Execute the **`bumperPressedAction`** function described in Section 5.2.4.

5.1.4 Potentially stuck

Every 40 iterations of the main loop, we check if the TurtleBot is not on a clear path, how far it has moved since its position 40 iterations ago (defined by applying the Pythagorean theorem for the set of points), and the current magnitude of `omega`.

If the magnitude of the distance is less than 0.1, and the magnitude of `omega` is less than 0.05, the TurtleBot is considered to be stuck. The **`moveAfterStuck`** function, described in Section 5.2.5, is then executed to try and break out of being stuck.

5.2 Low-Level Control Architecture

This section describes the different functions called from the program's main loop that are used for implementing lower-level functionality like rotating a particular angle, moving a particular distance, and choosing turn directions.

5.2.1 Choosing turn directions and/or angular velocities for the TurtleBot

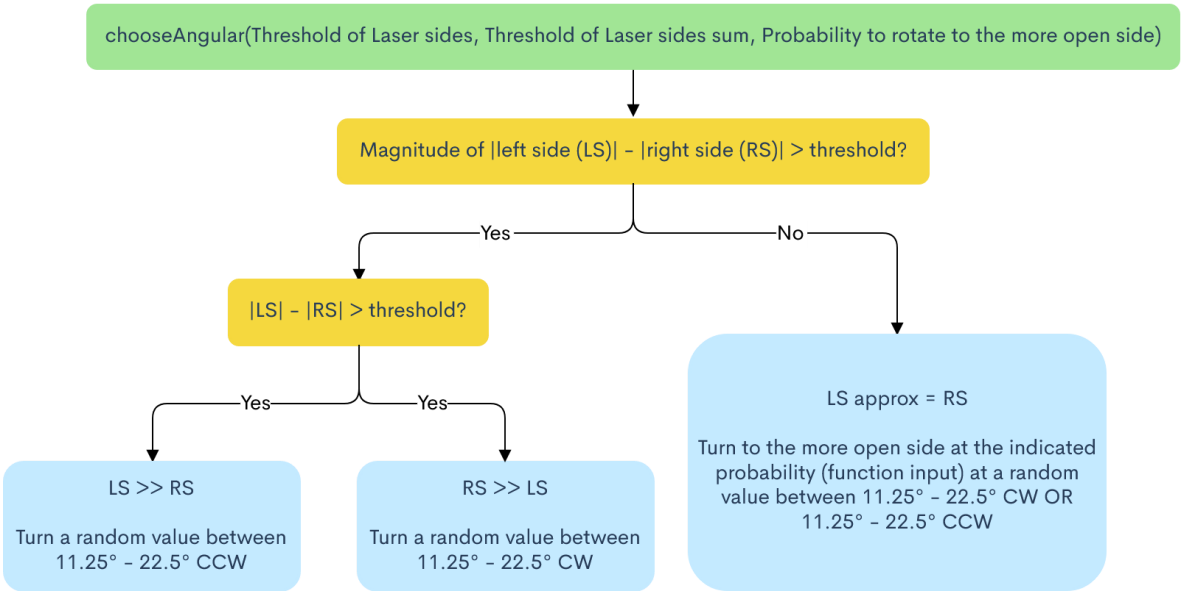

Figure 3: Logic of the chooseAngular function. Yellow are decision nodes and blue are actions taken.

The **`chooseAngular`** function, shown above in Figure 3 as a flow chart, uses a biased random decision making strategy in order to choose a direction and/or angular velocity for the TurtleBot to turn in.

The function takes in two parameters: `laserSideSumThreshold` and `probSpinToLarger`.

`laserSideSumThreshold` defines a threshold whereby if either `LSLaserSum` or `RSLaserSum` exceeds the other by this threshold, then the function always returns a random angular velocity between π/16 and π/8 in the more clear direction.

If neither `LSLaserSum` nor `RSLaserSum` exceed the other by `laserSideSumThreshold`, then the function returns a random angular velocity between π/16 and π/8 in the more clear direction, but only at probability `probSpinToLarger`.

Both parameters `laserSideSumThreshold` and `probSpinToLarger` work in tandem to bias the TurtleBot into choosing a particular turn direction. By setting a low `laserSideSumThreshold` and high `probSpinToLarger`, the TurtleBot can be biased to nearly always turn in the more clear direction, which is the approach that was taken when an unclear path was described in Section 5.1.1. Conversely, when the TurtleBot was on a clear path as described in Section 5.1.1, the `laserSideSumThreshold` was set exceedingly high and `probSpinToLarger` was set to a relatively low value in order to encourage exploration of the environment rather than trying to optimally follow the direction with the largest clear laser distances.

In order to only extract the turn direction of ± 1 with this function (positive defined as CCW), this function can be called in the second argument to `copysign()` in order to give the sign determined by this function to any parameter passed in the first argument of `copysign()`. This approach is typically used to define the `angleRAD` argument in calls to **`rotateThruAngle`**, which is described in the subsequent section.

5.2.2 Rotating a particular angle

The **`rotateThruAngle`** function is a simple controller that is used to turn the TurtleBot through a desired angle. The function takes in this desired angle (positive CCW), `angleRAD`; the initial yaw, `yawStart`; the initial laser distance, `laserDistStart`; a desired linear speed during the turn, `set_linear`; a boolean for whether or not to break out of the function before having turned to the desired angle if a clear path exists, `breakEarly`; and some other parameters to keep track of the elapsed time and for publishing the linear and angular velocities.

In order to turn the TurtleBot through a desired angle, the function continuously compares `yaw` and `yawStart`, and continuously sets `angular` to π/8 in the direction of the turn. Since this approach seemed to work relatively well, we did not bother trying to set `angular` with feedback from the difference in the yaw measurements (i.e. a proportional controller) or another more complicated approach like PID control.

To rotate in place, `set_linear` is passed as 0 rather than a non-zero value.

If `breakEarly` is set in the function call, then the loop will immediately break if `minLaserDist > 0.75`, even if the turn is not completed. This helps with being able to seek out clear paths and greedily follow them as discussed in Section 3.0. Other conditions for the loop to break are if loop iterations have exceeded 250, magnitude of `omega` is less than 0.05 and loop iterations are over 50 (this is a similar condition as the stuck check in Section 5.1.4), or if any of the bumpers are hit.

5.2.3 Moving a particular distance

The **`moveThruDistance`** function is used to move the TurtleBot a particular distance, and operates in a manner largely similar to **`rotateThruAngle`**.

As inputs, the function takes in the desired distance to travel (positive forward), `desired_dist`; the movement speed, `move_speed`; initial position (`startX`, `startY`), and some other parameters to keep track of the elapsed time and for publishing the linear and angular velocities.

By applying the Pythagorean Theorem as shown below, the function continuously calculates the distance between the starting and current position, and if this is less than `desired_dist`, then `linear` is continuously set to be of magnitude `move_speed` in the direction of `desired_dist` by using a while loop. Similar to **`rotateThruAngle`**, feedback control was not determined to be necessary to improve the control in setting `linear`.

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Similar to **`rotateThruAngle`**, the function will break out of the while loop if more than 100 iterations have passed, or if any of the bumpers get pressed.

5.2.4 Bumper pressed

The **`bumperPressedAction`** function gets called any time one of the TurtleBot's bumpers is hit. Regardless of which bumper has been hit, the TurtleBot will first move back 0.25 m at 0.2 m/s by calling the **`moveThruDistance`** function.

If the left bumper is pressed, the TurtleBot then rotates in place 90° CW by calling **`rotateThruAngle`** with `breakEarly=true` so that the rotation exits early if a clear path is detected (i.e. `minLaserDist > 0.75`). Similarly, if the right bumper is pressed, the only difference is that the rotation is set to be 90° CCW.

The centre bumper is checked last, since if the centre bumper is pressed in conjunction with a side bumper, executing the side handling cases is typically sufficient to get the TurtleBot back to facing a clear path. If only the centre bumper is hit, the turn direction of CW or CCW after moving backwards is determined by calling **`chooseAngular`** with `laserSideSumThreshold=150` and `probSpinToLarger=0.75`. The turn is set to be a full 90° with no ability to break out early. This is because with the brick wall, the laser sensor is always clear, so allowing an early break out triggers nearly immediately, and causes an infinite loop between the bumper being hit and the TurtleBot trying to break out early. In this case, the ability to determine if the TurtleBot is stuck as described in Section 5.1.4 is limited, since the lasers will think that the path is clear, meaning the check for being stuck will not get triggered.

5.2.5 Moving after being stuck

In this case, the **`moveAfterStuck`** function is called.

This function checks if `minLaserDist < 0.45 || minLaserDist > 7` (i.e. if the TurtleBot is close to a wall since the laser distance is small or infinite), and then uses **`moveThruDistance`** to move back 0.15 m at 0.1 m/s. In the other case when the condition above is false, it is likely that the back of the TurtleBot is hit, which cannot be detected by lasers or by bumpers, so we set the TurtleBot to move forwards 0.15 m at 0.1 m/s. This speed is set to satisfy constraints of likely being near a wall in either case.

Next, **`rotateThruAngle`** in conjunction with a call to **`chooseAngular`** in order to choose a turn direction is used to turn the TurtleBot 90°, with the ability to break out early if a clear path is detected.

6.0 Future Recommendations

As described in the Competition Strategy in Section 3.0, the program is primarily reactive and reacts based on sensory input. The TurtleBot therefore chooses its paths and speeds based on thresholds that the lasers and bumpers meet. This works decently for a smaller environment with static obstacles like the one for Contest 1, but would be less efficient as the environment to explore gets larger. In a situation like this, it would be advantageous to use the generated map to determine where the robot has been and where there are still unexplored spaces. This is known as Frontier Exploration, and is where the robot is able to localize itself on the map that it generates in RViz; using this map, the robot is able to determine which locations it has yet to visit, a key feature of Simultaneous Localization and Mapping (SLAM) [3]. SLAM algorithms could be implemented, but work best when the robot is able to do 360° spins in place relatively quickly (e.g. drones) to perform on-the-spot loop closure. This is more difficult to do with the TurtleBot as it is not ideal for spinning in place. Ultimately, for a small environment, SLAM was not necessary, but for a larger unknown environment, it would be very beneficial, and likely crucial.

Figure 4: Example of Frontier Exploration on a map generated with RViz, where the green represents the robot localizing itself on the map it has generated[1].

[1] Figure adapted from 2021 lecture material from MIE443

References

[1] A. Imhof, M. Oetiker and B. Jensen, "Wall following for autonomous robot navigation", *2012 2nd International Conference on Applied Robotics for the Power Industry (CARPI)*, 2012.

[2] K. Daltorio, B. Tietz, J. Bender, V. Webster, N. Szczecinski, M. Branicky, R. Ritzmann and R. Quinn, "A model of exploration and goal-searching in the cockroach, Blaberus discoidalis", *Adaptive Behavior*, vol. 21, no. 5, pp. 404-420, 2013.

[3] A.. Topiwala, P. Inani J. Bender, and A. Kathpal, "Frontier Based Exploration for Autonomous Robot", University of Maryland Robotics Center , 2018.

Appendices

Appendix A - Complete C++ Code

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

#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 posX = 0.0, posY = 0.0, yaw = 0.0, yaw_imu = 0.0, yawStart = 0.0, vel_odom = 0.0,
omega = 0.0, accX = 0.0, accY = 0.0;
uint8_t bumper[3] = {kobuki_msgs::BumperEvent::RELEASED,
kobuki_msgs::BumperEvent::RELEASED, kobuki_msgs::BumperEvent::RELEASED};
const uint8_t LEFT = 0, CENTER = 1, RIGHT = 2;
bool LSLaserClearer = true;

float minLaserDist = std::numeric_limits<float>::infinity(), minLSLaserDist =
std::numeric_limits<float>::infinity(), minRSLaserDist =
std::numeric_limits<float>::infinity();
float LSLaserSum = 0, RSLaserSum = 0;
int32_t nLasers=0, desiredNLasers=0, desiredAngle=22.5;

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

```
void laserCallback(const sensor_msgs::LaserScan::ConstPtr& msg){
       // Calculates minLaserDist overall, per side as minLSLaserDist and
minRSLaserDist,
    // and the sum of laser measurements on each side LSLaserSum and RSLaserSum

    minLaserDist = std::numeric_limits<float>::infinity();
    minLSLaserDist = std::numeric_limits<float>::infinity();
    minRSLaserDist = std::numeric_limits<float>::infinity();
    LSLaserSum = 0, RSLaserSum = 0;
    float maxLaserThreshold = 7;
    nLasers = (msg->angle_max - msg->angle_min) / msg->angle_increment;
    desiredNLasers = DEG2RAD(desiredAngle)/msg->angle_increment;

    int start = 0, end = nLasers;
    if (DEG2RAD(desiredAngle) < msg->angle_max && DEG2RAD(-desiredAngle) >
msg->angle_min){
        start = nLasers / 2 - desiredNLasers;
        end = nLasers / 2 + desiredNLasers;
    }
    for (uint32_t laser_idx = start; laser_idx < end; ++laser_idx){
        minLaserDist = std::min(minLaserDist, msg->ranges[laser_idx]);
        if (laser_idx <= nLasers / 2){
            minRSLaserDist = std::min(minRSLaserDist, msg->ranges[laser_idx]);
            if (msg->ranges[laser_idx] < maxLaserThreshold){
                RSLaserSum += msg->ranges[laser_idx];
            }
        }
        else{
            minLSLaserDist = std::min(minLSLaserDist, msg->ranges[laser_idx]);
            if (msg->ranges[laser_idx] < maxLaserThreshold){
                LSLaserSum += msg->ranges[laser_idx];
            }
        }
    }
    LSLaserClearer = LSLaserSum > RSLaserSum;
    ROS_INFO("Min Laser Distance: %f \n Left: %f \n Right: %f \n LSum: %f \n RSum: %f",
minLaserDist, minLSLaserDist, minRSLaserDist, LSLaserSum, RSLaserSum);
}

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);
    vel_odom = msg->twist.twist.linear.x;
    ROS_INFO("Position: (%f, %f) \n Orientation: %f deg. \n Velocity: %f", posX, posY,
RAD2DEG(yaw), vel_odom);
}
```

```
void imuCallback(const sensor_msgs::Imu::ConstPtr& msg){
    omega = msg->angular_velocity.z;
    accX = msg->linear_acceleration.x;
    accY = msg->linear_acceleration.y;
    yaw_imu = tf::getYaw(msg->orientation);
    ROS_INFO("Acceleration: (%f, %f) \n IMU Yaw: %f deg Omega %f", accX, accY,
RAD2DEG(yaw_imu), omega);
}

template <class T>
T randBetween(T a, T b){
    // Returns a random number between a and b inclusive. Assumes a<b.
    if (std::is_same<T, int>::value){
        float x = float(rand())/float(RAND_MAX);
        return int(round(float(b-a)*x + float(a)));
    }
    else{
        T x = T(float(rand())/float(RAND_MAX));
        return (b-a)*x + a;
    }
}

void update(geometry_msgs::Twist* pVel, ros::Publisher* pVel_pub, uint64_t*
pSecondsElapsed,
            const std::chrono::time_point<std::chrono::system_clock> start, ros::Rate*
pLoop_rate){
    // Sets linear and angular velocities, updates main loop timer
    (*pVel).angular.z = angular;
    (*pVel).linear.x = linear;
    (*pVel_pub).publish(*pVel);

    // Update the timer.
    *pSecondsElapsed =
std::chrono::duration_cast<std::chrono::seconds>(std::chrono::system_clock::now()-start
).count();
    (*pLoop_rate).sleep();
}

bool anyBumperPressed(){
    // Returns true if any bumper is pressed, false otherwise
    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);
    }
    return any_bumper_pressed;
}
```

```
void moveThruDistance(float desired_dist, float move_speed, float startX, float startY,
geometry_msgs::Twist* pVel, ros::Publisher* pVel_pub,
                      uint64_t* pSecondsElapsed, const
std::chrono::time_point<std::chrono::system_clock> start, ros::Rate* pLoop_rate){
    // Moves turtlebot desired_dist at move_speed. Negative desired_dist moves
backwards. Only magnitude of move_speed is used.
    int i = 0;
    float current_dist = sqrt(pow(posX-startX, 2) + pow(posY-startY, 2));
    while (current_dist < fabs(desired_dist) && i < 100 && *pSecondsElapsed < 900){
        ros::spinOnce();
        angular = 0;
        linear = copysign(fabs(move_speed), desired_dist); //move move_speed m/s in
direction of desired_dist
        update(pVel, pVel_pub, pSecondsElapsed, start, pLoop_rate); // publish linear
and angular
        current_dist = sqrt(pow(posX-startX, 2) + pow(posY-startY, 2));

        if (anyBumperPressed()){
            break;
        }
        i+=1;
    }
}
```

```
void rotateThruAngle(float angleRAD, float yawStart, float laserDistStart, float
set_linear, bool breakEarly, geometry_msgs::Twist* pVel,
                     ros::Publisher* pVel_pub, uint64_t* pSecondsElapsed, const
std::chrono::time_point<std::chrono::system_clock> start, ros::Rate* pLoop_rate){
    // Rotates turtlebot angleRAD rad CW(-) or CCW(+) depending on angleRAD's sign at
pi/8 rad/s.
    // Make sure angleRAD is between +/- pi
    // Use set_linear = 0 to rotate in place
    int i = 0;
    float clearPathThreshold = 0.75, maxLaserThreshold = 7;
    ROS_INFO("In rotating thru. \n Start yaw: %f \n Current yaw: %f \n minLaserDistance
%f \n Desired angle: %f", yawStart, yaw, minLaserDist, angleRAD);
    ROS_INFO("Condition check: %i", fabs(yaw - yawStart) <= fabs(angleRAD));

    while (fabs(yaw - yawStart) <= fabs(angleRAD) && i < 250 && *pSecondsElapsed <
900){
        ros::spinOnce();
        ROS_INFO("ROTATING %f \n Start yaw: %f \n Current yaw: %f \n minLaserDistance
%f \n IMU: %f \n Omega: %f \n Iter: %i", angleRAD, yawStart, yaw, minLaserDist,
yaw_imu, omega, i);
        ROS_INFO("Condition check %i \n LS: %f \n RS %f \n", fabs(yaw - yawStart) <=
fabs(angleRAD), fabs(yaw - yawStart), fabs(angleRAD));
        angular = copysign(M_PI/8, angleRAD); //turn pi/8 rad/s in direction of
angleRAD
        linear = set_linear;
        update(pVel, pVel_pub, pSecondsElapsed, start, pLoop_rate); // publish linear
and angular

        if (anyBumperPressed()){
            ROS_INFO("Breaking out of rotate due to bumper press \n Left: %d \n Center:
%d \n Right: %d \n", bumper[LEFT], bumper[CENTER], bumper[RIGHT]);
            break;
        }
        if(fabs(omega) < 0.035 && i > 50){
            ROS_INFO("Moving forward and braking out. Likely stuck.");
            moveThruDistance(0.1, 0.1, posX, posY, pVel, pVel_pub, pSecondsElapsed,
start, pLoop_rate);
            break;
        }

        if (minLaserDist > clearPathThreshold && minLaserDist < maxLaserThreshold &&
breakEarly){
            ROS_INFO("Breaking out of rotate due to large distance");
            break;
        }
        i +=1;
    }
}
```

```
float chooseAngular(float laserSideSumThreshold, float probSpinToLarger){
    // Chooses the angular velocity and direction
    // Can also call this in second argument of copysign() to only extract a direction
(e.g. for a rotation)
    float prob = randBetween(0.0, 1.0), angular_vel = M_PI/8, maxLaserThreshold = 7;
    ros::spinOnce();
    if(fabs(fabs(LSLaserSum) - fabs(RSLaserSum)) > laserSideSumThreshold){
        //If one side's laser distance > other side by more than laserSideSumThreshold,
go to that side
        if(fabs(LSLaserSum) - fabs(RSLaserSum) > laserSideSumThreshold){
            ROS_INFO("LS >> RS. Spin CCW");
            angular_vel = randBetween(M_PI/16, M_PI/8); //improves gmapping resolution
compared to always using constant value
        }
        else{
            ROS_INFO("RS >> LS. Spin CW");
            angular_vel = -randBetween(M_PI/16, M_PI/8);
        }
    }
    else{
        // Laser distances approx. equal. Go to larger at probSpinToLarger probability
        ROS_INFO("LS ~ RS. Spinning to larger at %.2f chance", probSpinToLarger);
        if ((fabs(LSLaserSum) > fabs(RSLaserSum) || fabs(minLSLaserDist) >
fabs(minRSLaserDist)) && prob < probSpinToLarger){
            angular_vel = randBetween(M_PI/16, M_PI/8);
        }
        else{
            angular_vel = -randBetween(M_PI/16, M_PI/8);
        }
    }
    return angular_vel;
}
```

```
void bumperPressedAction(geometry_msgs::Twist* pVel, ros::Publisher* pVel_pub,
uint64_t* pSecondsElapsed,
                         const std::chrono::time_point<std::chrono::system_clock>
start, ros::Rate* pLoop_rate){
    bool any_bumper_pressed = true;
    any_bumper_pressed = anyBumperPressed();

    if (any_bumper_pressed && *pSecondsElapsed < 900){
        ROS_INFO("Bumper pressed \n Left: %d \n Center: %d \n Right: %d \n",
bumper[LEFT], bumper[CENTER], bumper[RIGHT]);

        if (bumper[LEFT]){
            ROS_INFO("Left hit. Move back and spin 90 CW");
            moveThruDistance(-0.25, 0.2, posX, posY, pVel, pVel_pub, pSecondsElapsed,
start, pLoop_rate);
            rotateThruAngle(DEG2RAD(-90), yaw, minLaserDist, 0, true, pVel, pVel_pub,
pSecondsElapsed, start, pLoop_rate);
        }
        else if (bumper[RIGHT]){
            ROS_INFO("Right hit. Move back and spin 90 CCW");
            moveThruDistance(-0.25, 0.2, posX, posY, pVel, pVel_pub, pSecondsElapsed,
start, pLoop_rate);
            rotateThruAngle(DEG2RAD(90), yaw, minLaserDist, 0, true, pVel, pVel_pub,
pSecondsElapsed, start, pLoop_rate);
        }
        else if (bumper[CENTER]){
            ROS_INFO("Center hit. Move back and spin");
            moveThruDistance(-0.25, 0.2, posX, posY, pVel, pVel_pub, pSecondsElapsed,
start, pLoop_rate);
            rotateThruAngle(copysign(M_PI/2, chooseAngular(150, 0.75)), yaw,
minLaserDist, 0, false, pVel, pVel_pub, pSecondsElapsed, start, pLoop_rate);
        }
    }
    update(pVel, pVel_pub, pSecondsElapsed, start, pLoop_rate);
}

void moveAfterStuck(float startX, float startY, geometry_msgs::Twist* pVel,
ros::Publisher* pVel_pub, uint64_t* pSecondsElapsed,
                    const std::chrono::time_point<std::chrono::system_clock> start,
ros::Rate* pLoop_rate){
    // Handles trying to break the turtlebot out of being stuck
    int direction = 1;
    ROS_INFO("Moving out of stuck area");
    if (minLaserDist < 0.45 || minLaserDist > 7){
        direction = -1;
    }
    moveThruDistance(direction*0.15, 0.1, posX, posY, pVel, pVel_pub, pSecondsElapsed,
start, pLoop_rate);
```

```
    rotateThruAngle(copysign(M_PI, chooseAngular(50, 0.9)), yaw, minLaserDist, 0, true,
pVel, pVel_pub, pSecondsElapsed, start, pLoop_rate);
}

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::Subscriber imu_sub = nh.subscribe("mobile_base/sensors/imu_data", 1,
&imuCallback);

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

    ros::Rate loop_rate(10);
    float maxLaserThreshold = 7, clearPathThreshold = 0.75, slowThreshold = 0.6,
stopThreshold = 0;
    float bestAngle = 0, prob = 1, stuckStartX = posX, stuckStartY = posY;
    int clearPathIters = 0, checkStuckIters = 0;

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

    while(ros::ok() && secondsElapsed <= 900) {
        ros::spinOnce();
        ROS_INFO("Position: (%f, %f) Yaw: %f deg IMU %f minLaserDist: %f", posX, posY,
RAD2DEG(yaw), RAD2DEG(yaw_imu), minLaserDist);

        bool any_bumper_pressed = anyBumperPressed();

        prob = randBetween(0.0, 1.0);

        if (!any_bumper_pressed && minLaserDist < maxLaserThreshold){
            if (minLaserDist > clearPathThreshold){
                ROS_INFO("Clear path. Iter: %d", clearPathIters);
                linear = 0.2;
                angular = 0;
                clearPathIters ++;
                if (clearPathIters > 20){
                    rotateThruAngle(copysign(randBetween(M_PI/9, M_PI/6),
chooseAngular(200, 0.55)), yaw, minLaserDist, 0.2, false, &vel, &vel_pub,
```

```
&secondsElapsed, start, &loop_rate);
                    clearPathIters = 0;
                }
            }
            else if(minLaserDist > slowThreshold && minLaserDist <=
clearPathThreshold){
                ROS_INFO("Slowing and following clearer path");
                linear = 0.15;
                rotateThruAngle(copysign(randBetween(M_PI/12, M_PI/8),
chooseAngular(100, 0.75)), yaw, minLaserDist, 0.15, true, &vel, &vel_pub,
&secondsElapsed, start, &loop_rate);
            }
            else if (minLaserDist > stopThreshold && minLaserDist <= slowThreshold){
                ROS_INFO("Slowing down");
                linear = 0.1;
                rotateThruAngle(copysign(M_PI, chooseAngular(25, 0.9)), yaw,
minLaserDist, 0.1, true, &vel, &vel_pub, &secondsElapsed, start, &loop_rate);
                clearPathIters = 0;
            }
        }
        else if (minLaserDist > maxLaserThreshold && !any_bumper_pressed){
            ROS_INFO("Laser inf, bumper free. Moving back");
            moveThruDistance(-0.7, 0.1, posX, posY, &vel, &vel_pub, &secondsElapsed,
start, &loop_rate);
            clearPathIters = 0;
        }
        else if (any_bumper_pressed){
            ROS_INFO("Bumper pressed");
            bumperPressedAction(&vel, &vel_pub, &secondsElapsed, start, &loop_rate);
            clearPathIters = 0;
        }

        checkStuckIters ++;
        if(checkStuckIters == 40 && clearPathIters == 0 && (sqrt(pow(posX -
stuckStartX, 2) + pow(posY - stuckStartY, 2)) < 0.1 || fabs(omega) < 0.05)){
            ROS_INFO("STUCK");
            moveAfterStuck(posX, posY, &vel, &vel_pub, &secondsElapsed, start,
&loop_rate);
            checkStuckIters = 0;
        }
        if(checkStuckIters == 40){
            checkStuckIters = 0;
        }
        update(&vel, &vel_pub, &secondsElapsed, start, &loop_rate);
    }
    ROS_INFO("15 minutes elapsed");
    return 0;
}
```

Appendix B - Attribution Table

Category		Farhan	Henry	Yilin
Code	Strategy	RS	RS	RS
	Handler Functions	RD	RD	RD
	High-level Strategy Functions	RD, MR	RD	
	Integration	MR, ET		
	Clean-up	MR, ET		
	Testing	MR, ET, FP	FP	FP
Report	FOCs			RD
	Strategy			RD
	Sensors	MR, ET	ET	RS, RD, MR
	Controller	MR, ET		RD
	Complete Report	FP	FP, ET	CM, RD, ET, FP

RS	Research
RD	Wrote first draft
MR	[CODE] Major revision to strategy, functions used [REPORT] Major revision to organization
ET	[CODE] Edited for errors, de-bugging [REPORT] Edited for grammar and spelling
FP	[CODE] Final check for applicability to practice worlds [REPORT] Final check for flow and consistency
CM	Responsible for compiling the elements into the complete document