



**EE4308 - Advances in Intelligent Systems & Robotics**

**Part 1 Project Report**

Turtlebot Autonomous Navigation in ROS Gazebo

Group 8

Franky Laurentis (A0144164L)

Tan RongDe Darius (A0146654Y)

Submitted on: 11/03/2019

AY 18/19 Semester 1

**Abstract**

The goal of this project is to design an autonomous navigation system for the Turtlebot in the Gazebo environment using ROS, which includes pathfinding as well as obstacle avoidance in a grid like maze world. For our project, we implemented the A\* search algorithm as our pathfinding tool to plan for the Turtlebot's path to goal depending on the obstacles detected by its on-board RGB-D Kinect sensor. Our system is robust enough to handle both known and unknown worlds as long as the start and end coordinates remain the same at (0, 0) and (4, 4) respectively.

Code repository can be found online at:

<https://github.com/dariustanrd/ROS-Turtlebot-Navigation-Project>

**TABLE OF CONTENTS**

1. Introduction	4
2. Path Planning Methods	4
2.1. Flood Fill Algorithm	5
2.2. A* Algorithm	6
2.3. Implemented Path Planning Algorithm	6
3. Navigation Strategies	7
3.1. Pose Determination	7
3.2. Coarse Movement	9
3.3. Fine Movement	9
3.4. Depth Determination	10
3.5. Basic Obstacle Avoidance	11
3.6. Preemptive Obstacle Avoidance	11
4. Results & Discussion	12
5. Conclusion	14
6. Individual Contributions	15
7. References & Appendices	16
7.1. References	16
7.2. Appendices	16

**LIST OF TABLES**

Table 1. Test results for world 1 and 2	13
Table 2. Test results for world 3	14
Table 3. Contribution breakdown	15

**LIST OF FIGURES**

Figure 1. Turtlebot robotics platform	4
Figure 2. Gazebo grid-like maze	4
Figure 3. Example of Gazebo world grid coordinates	5
Figure 4. Initial path planned by A*	7
Figure 5. Updated path upon wall detection	7
Figure 6. Error in <code>/tf</code> tree seen when running <code>roswtf</code>	8
Figure 7. Unconnected <code>/odom_combined</code> in <code>/tf</code> tree visualisation	9
Figure 8. Modified <code>/tf</code> tree visualisation to allow for usage of <code>/odom_combined</code>	9
Figure 9. Angular velocity applied to return to grid's centre	10
Figure 10. No need for adjustment since trajectory is already towards grid's centre	10
Figure 11. Graphical representation of preemptive obstacle avoidance	11
Figure 12. Commands used to start Gazebo environment in 1st terminal	12
Figure 13. Commands used to start Turtlebot navigation in 2nd terminal	12
Figure 14. World 1	12
Figure 15. World 2	12
Figure 16. World 3 (dead end)	12

## 1. Introduction

The purpose of this project is to design an autonomous navigation system for the Turtlebot robotics platform (Figure 1) using the Robot Operating System (ROS) which is then simulated in the Gazebo environment. These tools are open source robotics solutions developed by Open Robotics, also previously known as the Open Source Robotics Foundation. For this project, the Turtlebot2 is used with the Kinetic Kame release of ROS in the Ubuntu Xenial 16.04 operating system.

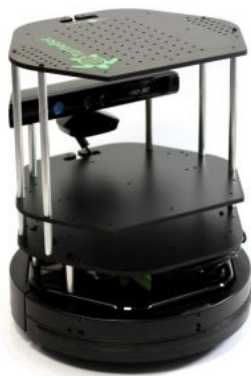


Figure 1. Turtlebot robotics platform

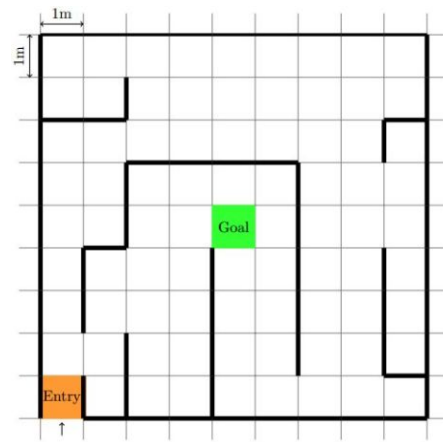


Figure 2. Gazebo grid-like maze

In this project, the Turtlebot is required to navigate within a grid-like maze, such as in Figure 2, from a designated entry point to the goal while avoiding the walls present within the maze. The world will be unknown to the Turtlebot other than the dimensions of the grid-like maze, as well as the entry and goal coordinates. All code written for this project can be found on our GitHub repository as mentioned in the abstract.

## 2. Path Planning Methods

An important aspect of an autonomous navigation system is its ability to find the shortest and most optimal path between different points in the robot's deployed environment. Such paths could be implemented with the aim of optimising the the distance travelled, the number of turns, the robot power consumption or a myriad of other factors depending on each application's specifications.

In the planning for such paths, a representation of the robot's world environment has to be implemented and traditionally this representation can be done via two complementary

approaches: discrete or continuous approximations [1]. The discrete approximation applies for environments that can be segmented into different portions of either equal sizes, such as in a grid, or different sizes, also known as topological maps. The continuous approximation case involves a representation of polygonal boundaries with paths represented as a sequence of points [1].

In the context of this project, the Gazebo world is fixed in a grid / maze representation and we will hence be using a discrete approximation of the Turtlebot's world environment using a 2D array / Adjacency Matrix to represent the connected graph of individual grid coordinates in the world such as in Figure 3. The polarity of the axes follows standard X and Y cartesian coordinates.

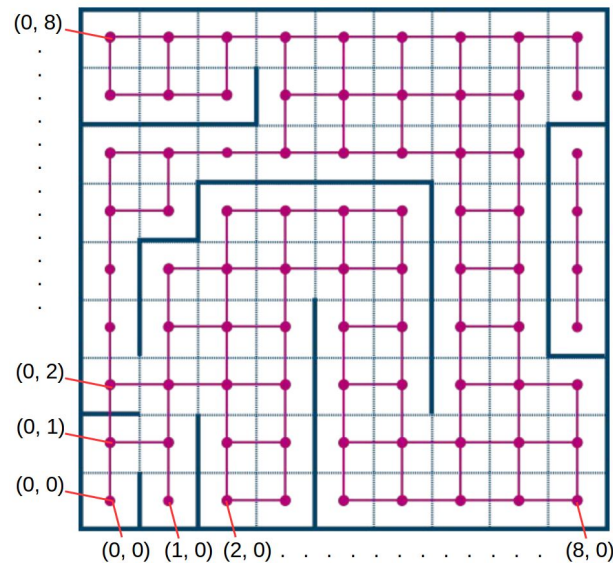


Figure 3. Example of Gazebo world grid coordinates

## 2.1. Flood Fill Algorithm

To implement the flood fill algorithm, we first set the goal cell at (4, 4) to have the value of 0. Then the values of the remaining cells were set depending on their distances to the centre. Essentially, the robot is then tasked to traverse from a higher numbered grid to a lower number grid if there is no obstacle between, limiting the direction to the 4 up down left right motions.

If the robot enters onto a grid and there is no neighbouring cell available for robot to travel towards, then the grid's value will be set +1 from the highest neighbour's value. This process is repeated until the robot reaches the goal.

Similar to our implementation of the A\* algorithm, for every grid the robot travelled, it will calculate for the entire trajectory with the assumption that the subsequent grids do not have obstacles (walls) in the direction of the goal position.

However, the flood fill algorithm has a long computational time and was very efficient at finding the most suitable path from start to the goal. Therefore, we decided to focus our efforts on the A\* algorithm instead.

## 2.2. A\* Algorithm

The A\* algorithm [2] works using the formula  $f(n) = g(n) + h(n)$  where  $n$  is the next neighbouring node,  $g(n)$  is the cost of the path from the previous node to  $n$ ,  $h(n)$  is the heuristic cost, the manhattan distance from  $n$  to the goal, and  $f(n)$  is the total cost to choose the node  $n$  as the next coordinate to move towards. In our case, as all coordinates are equally spaced and are identical, we set  $g(n)$  to be 1 for all nodes since we would only be moving between horizontally or vertically adjacent grids.  $h(n)$  is calculated for each coordinate using the manhattan distance, where  $h(n) = |x_1 - x_2| + |y_1 - y_2|$ . After the addition of both costs to  $f(n)$ , the algorithm will pick the coordinate with the lowest  $f(n)$  value as the next grid to travel to. This check would then occur for the next coordinate and the neighbours of that coordinate until the goal is found.

For our navigation system, since there is no way of knowing the map before the robot travels through it, the algorithm will continuously come up with a set of routes to reach the goal assuming that the grids after the next grid do not have obstacles at all. Thus our main strategy is for the algorithm to keep calculating for new routes as the map of the grid updates itself when the robot senses obstacle/walls.

Much the same for flood fill, the algorithm header file will only supply the coordinate for the next grid to `bot_control` node. Using the A\* algorithm, it is possible to not

only find the shortest route to the goal position but also to recover easily from dead ends.

### 2.3. Implemented Path Planning Algorithm

In this project, we decided to implement the A\* pathfinding algorithm as it is able to find the shortest path to the goal more efficiently as compared to the flood fill algorithm.

Due to the nature of this project being that the world map is initially unknown to the Turtlebot, upon initialisation, the path planning algorithm will plan a route from the start to goal coordinate assuming that there are no obstacles in the world as seen in Figure 4.

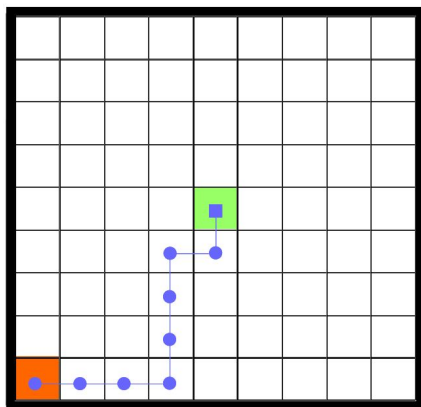


Figure 4. Initial path planned by A\*

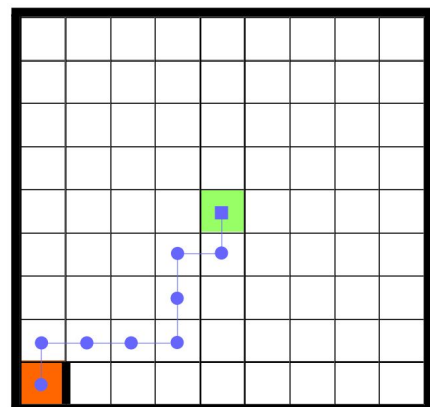


Figure 5. Updated path upon wall detection

Upon moving along the prescribed path, the Turtlebot then senses the presence of a wall and correspondingly updates the Turtlebot's internal map in the 2D array with the presence of a wall. The A\* algorithm will then execute again with the addition of the wall to continue navigating towards the goal while avoiding the wall as seen in Figure 5.

### 3. Navigation Strategies

In order to get the Turtlebot to move according to the planned path, a navigation / locomotion strategy is required. For this project, we limited the Turtlebot's movements to up, down, left, right as well as yawing in both clockwise and anticlockwise directions.



### 3.1. Pose Determination

Our navigation strategy is heavily reliant on the Turtlebot being able to localise its position on the grid as the Turtlebot is not provided with a map prior to the start of the mission. For this, we used odometry estimates to determine how far the Turtlebot has travelled with respect to its starting position.

However, as plain odometry values from `/odom` can be inaccurate due to wheel slip or encoder errors among other factors, we decided to get a more reliable odometry estimate using sensor fusion with the Turtlebot's onboard Inertial Measurement Unit (IMU). This sensor fusion was done via the Extended Kalman Filtering of raw odometry data and IMU sensor data using the `/robot_pose_ekf` node from the similarly named `/robot_pose_ekf` package [3]. When the node is added to our project launch file, the node subscribes to `/odom` and `/mobile_base/sensors/imu_data` using some param updates and remappings and publishes the combined output estimates in the `/robot_pose_ekf/odom_combined` topic which our `pos_info` node subscribes to for pose estimation. Upon further testing, we realised that this `/odom_combined` data was indeed more reliable and responsive to changes in the Turtlebot's true position as compared to `/odom`.

In order to use `/robot_pose_ekf/odom_combined`, the `/tf` tree had to be edited as both `/odom` and `/odom_combined` are attempting to be the parent frame of `base_footprint` as seen in the error seen in Figure 6 when running `roswtf`.

```
ERROR TF re-parenting contention:
* reparenting of [base_footprint] to [odom_combined] by [/robot_pose_ekf]
* reparenting of [base_footprint] to [odom] by [/gazebo]

ERROR TF multiple authority contention:
* node [/robot_pose_ekf] publishing transform [base_footprint] with parent [odom_combined] already
published by node [/gazebo]
* node [/gazebo] publishing transform [base_footprint] with parent [odom] already published by node
[/robot_pose_ekf]
```

Figure 6. Error in `/tf` tree seen when running `roswtf`

This would hence cause `/odom_combined` to be unable to work as per intended as seen from the `/tf` tree visualisation in Figure 7 as it becomes an unconnected frame. As `/tf` does not allow for multiple parents, the `/tf` publishing from `/odom` needs to be removed to allow for the usage of `/odom_combined` instead as seen from the new `/tf`

tree visualisation in Figure 8. This was done by modifying the world launch files in the `turtlebot_gazebo` and `gazebo_ros` packages to those found in our `bot` package.

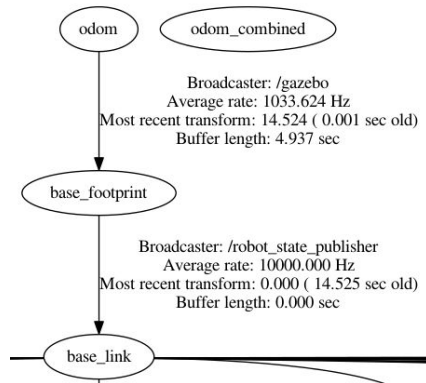


Figure 7. Unconnected `/odom_combined` in `/tf` tree visualisation

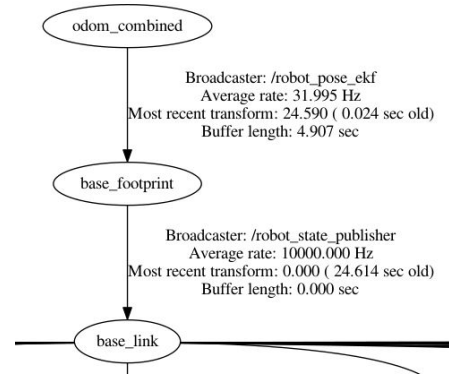


Figure 8. Modified `/tf` tree visualisation to allow for usage of `/odom_combined`

The `/robot_pose_ekf/odom_combined` topic is subscribed to by our `pos_info` node and the pose estimate for localisation is then published by the node to `/auto_ctrl/pos_info`.

### 3.2. Coarse Movement

Movement is simplified into 4 directions, up, down, left, right and is controlled in our `bot_control` node. For example, when the robot is commanded to move straight up, it will first confirm that it is indeed facing up through the usage of the yaw value obtained from the `/auto_ctrl/pos_info` topic. If it's not facing upwards, the robot will then execute turning until the desired odometer yaw value is reached. This is to compliment a grid based navigation system since now the robot only need to care about moving up, down, left, right to a neighbouring grid.

### 3.3. Fine Movement

In order to keep the Turtlebot in the middle of the path that it is on, fine adjustments to its movement needs to be implemented to prevent it from drifting.

In our `bot_control` node, while executing a move straight function, the robot will use odometry data to maintain a straight line and also ensure that the robot continues to face towards one of the 4 directions.

Essentially, the robot will have the pose estimate data from `/robot_pose_ekf/odom_combined` and the rounded coordinate value after the robot entered into the next grid. The code measure the difference of the robot's odom position and the grid's coordinate values, then adjust the angular velocity of the robot such that the turn will reduce the difference of the values. For example, going up at  $y = 2$ , and the robot is currently at  $\text{odom}_y = 2.1$ . If the robot's yaw is turned away from  $y = 2$ , the robot will apply the angular velocity along with the linear velocity such that it will smoothly turns towards the  $y = 2$  line. But if the robot is already slightly facing towards  $y = 2$ , the angular velocity won't be applied since we already assumed that the robot will slowly turn towards the line. An example of this code can be seen in Appendix 1 for the UP direction, where `YAW_VEL_CORRECTION` was defined to be 0.1.

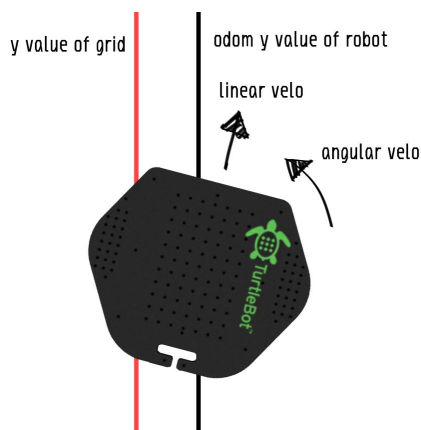


Figure 9. Angular velocity applied to return to grid's centre

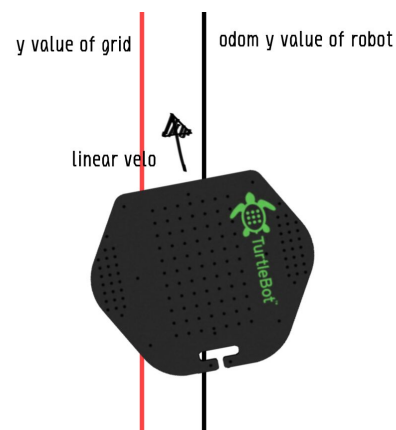


Figure 10. No need for adjustment since trajectory is already towards grid's centre

If the bot is going up and the bot is drifting outwards and further away from the centre of the grid as seen in Figure 9, it will then adjust with angular velocity of 0.1 to the opposite direction and turn back towards the grid's axis. If the robot is already moving towards the grid's axis, there is no need to add the angular velocity anymore as seen in Figure 10.

### 3.4. Depth Determination

#### Using RGB-D Kinect images in `depth_info` node

The `depth_info` node subscribes to the raw RGB-D kinect camera images of `/camera/rgb/image_raw` and `/camera/depth/image_raw` and publishes the depth

information for the point directly in front of the Turtlebot to `/auto_ctrl/depth_info`. However, as this method only allows us to detect the distance at the middle of the camera image, we used the `/scan` topic directly for extreme left and right readings as well.

#### Using laserscan data from `/scan` in the `scan_info` node

The `scan_info` node subscribes to the `/scan` topic by the `laserscan_nodelet_manager` that obtains 'fake' laserscan values from the depth image using `depthimage_to_laserscan` [4]. This laserscan values are then used to detect the depths at the extreme left, right and middle of the scanning range of the RGB-D sensor.

### 3.5. Basic Obstacle Avoidance

The basic obstacle avoidance allows us to detect obstacles directly in front of the Turtlebot using the `/auto_ctrl/depth_info` topic. If this form of obstacle avoidance were to be implemented on its own, it would cause the Turtlebot to constantly have to turn to check if there is the presence of a wall in its intended path of motion at each coordinate, which reduces the speed of clearing the maze.

Therefore, as the basic `depth_info` node was not able to detect the depths of the extreme left and right sides of the laser scanner, we used the `scan_info` node to subscribe to the `/scan` topic directly instead, allowing for preemptive obstacle detection and avoidance.

### 3.6. Preemptive Obstacle Avoidance

Using Laserscan, it is possible to measure the robot's distance to anything in front of it as long as it is within range. We decided to just use the extreme left, right and the middle measurements for simplicity. Using the `angle_min` and `angle_max` values in the `sensor_msgs::LaserScan` message, we were able to gather that the extreme left and right of the

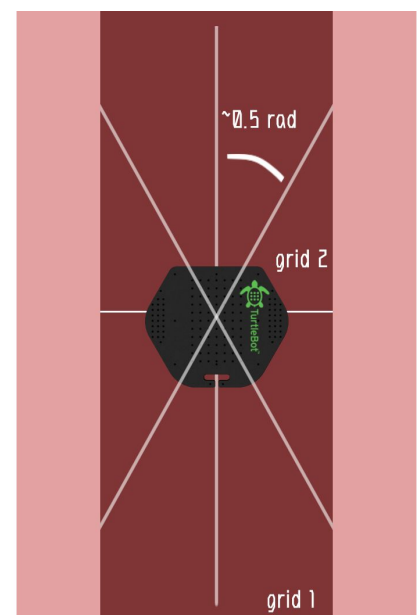


Figure 11. Graphical representation of preemptive obstacle avoidance

sensor deviates from the middle point at an angle of 30 degrees, or approximately 0.5 rad. This would allow us to detect the presence of side walls at the next coordinate when the Turtlebot has left its previous coordinate and is moving to the next coordinate.

Based on the geometry of the map grid and the size of the walls, we are able to reliably detect the presence of sidewalls when the Turtlebot has moved halfway between coordinates as seen in Figure 11.

This is a vast improvement over the basic obstacle detection system since the robot will now be able to see the grid in front of it for walls and there will be no need to take the additional time needed to turn and then see what is directly in front of it.

#### 4. Results & Discussion

Assuming that the steps taken to setup and configure the catkin workspace as listed in our github repository (<https://github.com/dariustanrd/ROS-Turtlebot-Navigation-Project>) README.md file has been followed, the commands below are used to start the Gazebo environment as well as to start the Turtlebot's navigation to goal.

```
$ source devel/setup.bash  
$ chmod +x project_init.sh  
$ ./project_init.sh
```

Figure 12. Commands used to start Gazebo environment in 1st terminal

```
$ source devel/setup.bash  
$ roslaunch bot bot.launch
```

Figure 13. Commands used to start Turtlebot navigation in 2nd terminal

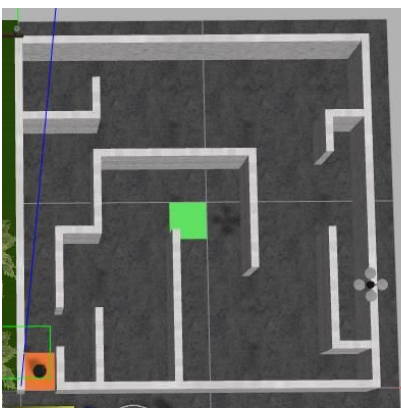


Figure 14. World 1

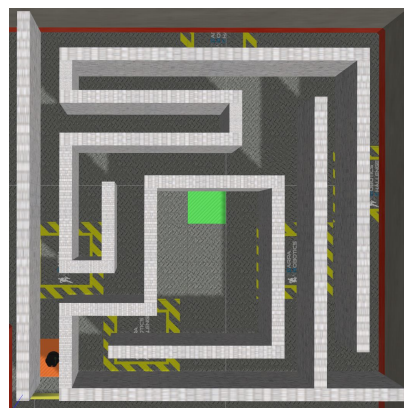


Figure 15. World 2

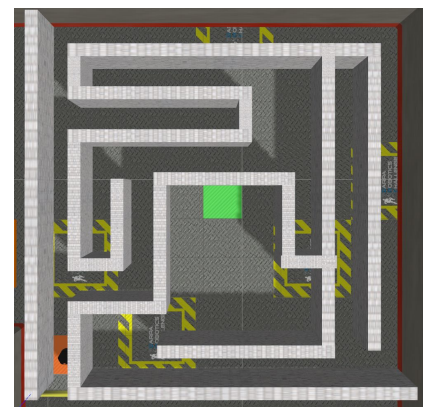


Figure 16. World 3 (dead end)

From our testing of the given worlds 1 & 2, we decided to implement a 3rd world as seen in Figure 14 to test the ability of our system in navigating around and recover from the presence of dead ends. As seen in Figure 14, world 3 is essentially a modified version of world 2 with the presence of multiple dead ends.


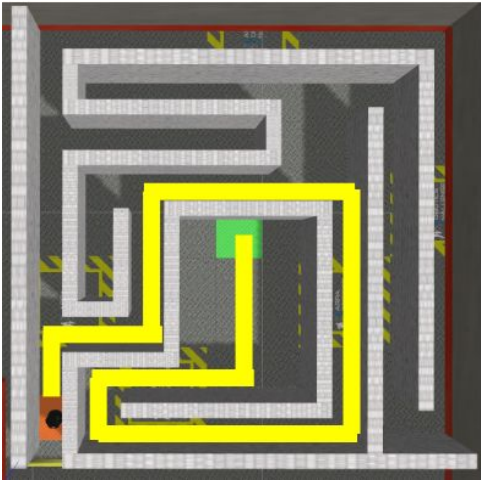
World	Path Taken	Shortest Path?	Average Time	Test Failure Rate
1		Yes	Approx 50s	0%
2		Yes	Approx 100s	0%

Table 1. Test results for world 1 and 2

From our testing, the Turtlebot managed to complete both world 1 and world 2 with the shortest possible path and had no failures in all our tests, reaching the goals in approximately 50 and 100 seconds respectively as seen in Table 1.



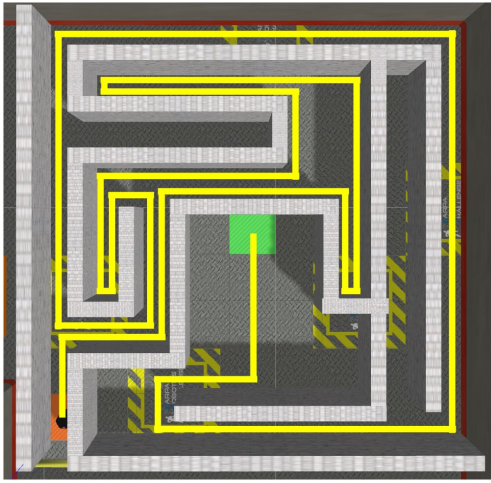
World	Path Taken	Shortest Path?	Average Time	Test Failure Rate
3		No	Approx 2200s	0%

Table 2. Test results for world 3

For world 3, it is the worst case scenario for the Turtlebot with multiple dead ends and the Turtlebot managed to complete the maze and managed to recover from the dead ends, eventually arriving at the goal destination.

## 5. Conclusion

A potential improvement to our implementation of this project could be to allow for the smoothing of turns around the corners instead of stopping and rotating. This would reduce the overall time taken to reach the goal as the Turtlebot would be able to continue moving around the corner. Diagonal grid motion could also have been implemented to reduce the number of moves needed to reach the goal.

Such a system has potential applications in delivery robots or drones as such a system of pathfinding between a start and end coordinate while avoiding obstacles could provide the basic skeleton for the delivery systems.

**6. Individual Contributions**

Team Member	Tasks Performed
Franky Laurentis	Code: <ul style="list-style-type: none"> <li>- Implemented flood fill search algorithm (incomplete)</li> <li>- Implemented depth determination using laserscan</li> <li>- Implemented fine movement controls</li> <li>- Made new world for further testing with more dead ends</li> </ul>
	Report: <ul style="list-style-type: none"> <li>- Wrote 60% of the report</li> </ul>
	Presentation: <ul style="list-style-type: none"> <li>- Prepared the slides</li> <li>- Presented 50% of the slides</li> </ul>
Tan RongDe Darius	Code: <ul style="list-style-type: none"> <li>- Implemented A* search algorithm</li> <li>- Implemented pose determination using odometry (given by GA)</li> <li>- Implemented pose determination using extended kalman filter</li> <li>- Implemented depth determination using RGB-D camera (given by GA)</li> <li>- Implemented coarse movement controls</li> <li>- Implemented basic obstacle avoidance</li> <li>- Implemented preemptive obstacle avoidance</li> <li>- Implemented GitHub repository</li> </ul>
	Report: <ul style="list-style-type: none"> <li>- Wrote 40% of the report</li> </ul>
	Presentation: <ul style="list-style-type: none"> <li>- Prepared the slides</li> <li>- Presented 50% of the slides</li> </ul>

Table 3. Contribution breakdown



## 7. References & Appendices

### 7.1. References

- [1] N. Correll, *Introduction to Autonomous Robots, 2nd Ed.* Magellan Scientific, 2016
- [2] P. E. Hart, N. J. Nilsson and B. Raphael, "A Formal Basis for the Heuristic Determination of Minimum Cost Paths," in *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100-107, July 1968.
- [3] W. Meeussen, "robot\_pose\_ekf wiki," *ros.org*. [Online]. Available: [http://wiki.ros.org/robot\\_pose\\_ekf](http://wiki.ros.org/robot_pose_ekf).
- [4] C. Rockey, "depthimage\_to\_laserscan wiki," *ros.org*. [Online]. Available: [http://wiki.ros.org/depthimage\\_to\\_laserscan](http://wiki.ros.org/depthimage_to_laserscan).

### 7.2. Appendices

#### Appendix 1: Code snippet for fine motion control in UP direction in bot\_control node

```
case UP:
    if (posY - initY < MOVE_DIST) {
        linear_cmd = LIN_VEL;
        std::cout << "Straight UP" << std::endl;
    }
    if (posX >= curCoord.first){
        if (yaw <= 0){
            yaw_cmd = YAW_VEL_CORRECTION;
        }
    }
    else if (posX <= curCoord.first){
        if (yaw >= 0){
            yaw_cmd = -YAW_VEL_CORRECTION;
        }
    }
    break;
```

#### Appendix 2: Code snippet from depth\_info node

```
void imageCb(const sensor_msgs::ImageConstPtr &rgbmsg,
             const sensor_msgs::ImageConstPtr &depthmsg){
```

```

cv_bridge::CvImageConstPtr cv_ptr;
try {cv_ptr = cv_bridge::toCvShare(depthmsg, enc::TYPE_32FC1);}
catch (cv_bridge::Exception &e){
    ROS_ERROR("cv_bridge exception: %s", e.what());
    return;
}
std_msgs::Float64 msg;
msg.data = cv_ptr->image.at<float>(320, 240);
depth_pub.publish(msg);
std::cout<<"depth at centre"<<cv_ptr->image.at<float>(320,
240)<<std::endl;
cv::imshow(dWINDOW, cv_ptr->image);
}

```

### Appendix 3: Code snippet from scan\_info node

```

void scanCallback(const sensor_msgs::LaserScan::ConstPtr& scan) {
    scan_vect = scan->ranges;
    minRan = scan->range_min;
    maxRan = scan->range_max;
    minAng = scan->angle_min;
    maxAng = scan->angle_max;

    int midIdx = (int)round(scan_vect.size() / 2.0);
    scan_right = scan_vect.front();
    scan_mid = scan_vect[midIdx];
    scan_left = scan_vect.back();
    if (startup) {
        if (std::isnan(scan_left)) send_left = MAX_DEPTH;
        if (std::isnan(scan_mid)) send_mid = MAX_DEPTH;
        if (std::isnan(scan_right)) send_right = MAX_DEPTH;
        scanHistLeft.push_back(send_left); //testing
        scanHistMid.push_back(send_mid); //testing
        scanHistRight.push_back(send_right); //testing
    }else {
        if (std::isnan(scan_left)) {
            if (*scanHistLeft.begin() < *scanHistLeft.rbegin())
                send_left = MAX_DEPTH;
            else if (*scanHistLeft.begin() == *scanHistLeft.rbegin())
                send_left = send_left;
            else send_left = MIN_DEPTH;
        } else send_left = scan_left;
        if (std::isnan(scan_mid)) {
            if (*scanHistMid.begin() < *scanHistMid.rbegin())
                send_mid = MAX_DEPTH;

```

```

        else if (*scanHistMid.begin() == *scanHistMid.rbegin())
            send_mid = send_mid;
        else send_mid = MIN_DEPTH;
    } else send_mid = scan_mid;
    if (std::isnan(scan_right)) {
        if (*scanHistRight.begin() < *scanHistRight.rbegin())
            send_right = MAX_DEPTH;
        else if (*scanHistRight.begin() == scanHistRight.rbegin())
            send_right = send_right;
        else send_right = MIN_DEPTH;
    } else send_right = scan_right;
    scanHistLeft.pop_front(); //testing
    scanHistMid.pop_front(); //testing
    scanHistRight.pop_front(); //testing
    scanHistLeft.push_back(send_left); //testing
    scanHistMid.push_back(send_mid); //testing
    scanHistRight.push_back(send_right); //testing
}
startup = false;
std::cout << "Send Left: " << send_left << " Send Mid: " << send_mid << " Send
    Right: " << send_right << std::endl;
scan_msg.data.resize(3);
scan_msg.data[0] = send_left;
scan_msg.data[1] = send_mid;
scan_msg.data[2] = send_right;
scan_info.publish(scan_msg);
}

```

### Appendix 3: Code snippet from bot\_control node

```

if (!checkFace(nextStep)) {
    std::cout << "Move to Correct Face" << std::endl;
    moveStr(STOP);
    moveFace(nextStep);
} else if (!checkObs()) {
    //clear path
    std::cout << "Path Clear" << std::endl;
    moveStr(nextStep);
    if (!preemptedWall && movedDistance(nextStep, 0.5)) {
        preemptWall(nextStep); //update wall pre-emptively while moving
        between coordinates.
        preemptedWall = true; //only run once
    }
}
}

```