ME3243 Project 2

Path Planning And Navigation In ROS

29 October 2020

Ng Wei Jie, Brandon A0184893L

Summary

The task is to simulate a Turtlebot3 robot, specifically the model burger, maneuvering in a maze. The maze is structured, and the map is divided into cells where each cell has four directions that the robot can move to in the next step. However, the robot does not have prior knowledge of the maze. Therefore, the maze is unknown to the robot. The robot starts from an initial cell (orange) and moves to a destination cell (green) as displayed in Figure 1. The project's learning objective is to understand and describe the path planning algorithm to make the Turtlebot3 robot move autonomously from initial position to the destination position.

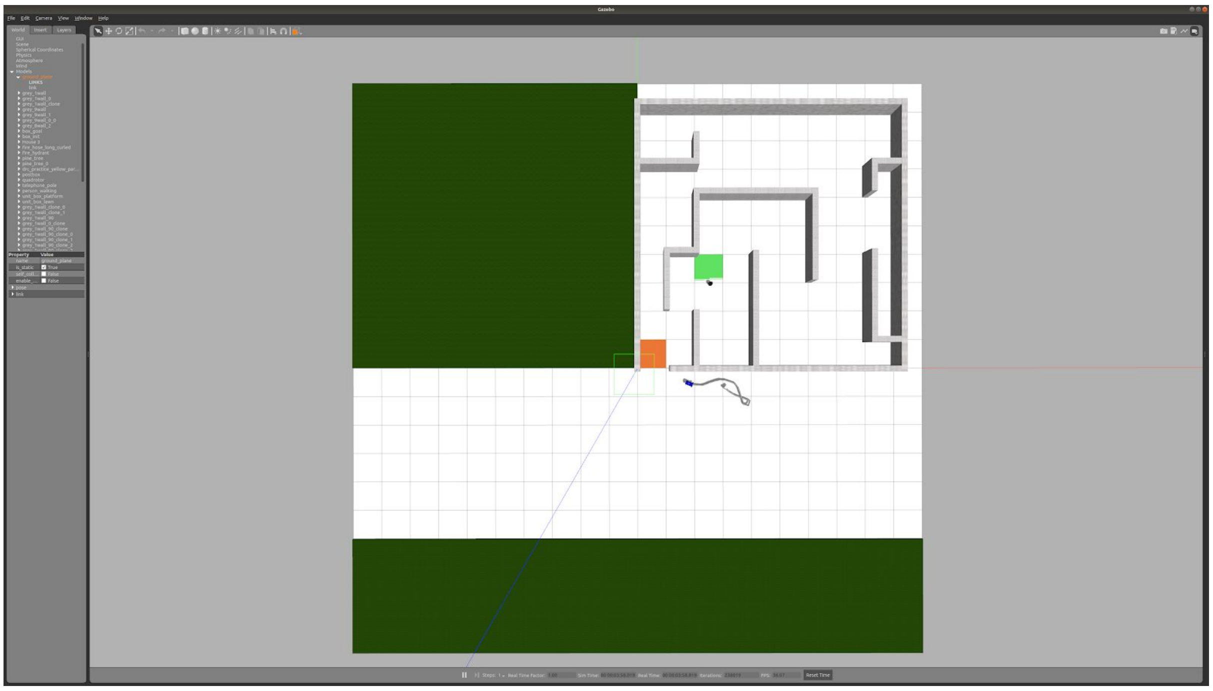


Figure 1: ROS simulation environment Gazebo

Initial Settings

The project uses ROS Melodic to simulate path planning and navigation for the Turtlebot3 robot in a maze. The ROS simulation environment is Gazebo. The maze is described in the test_world_1.world. The PID gains for steering and motor controls of the robot are specified in the config.yaml. The file PreDefine.hpp specifies reusable constants, such as the North, South, East and West directions, and the GOAL_X and GOAL_Y that specifies the final destination of the robot. In this simulation, it is safe to assume that the path cost from one cell to another is always one.

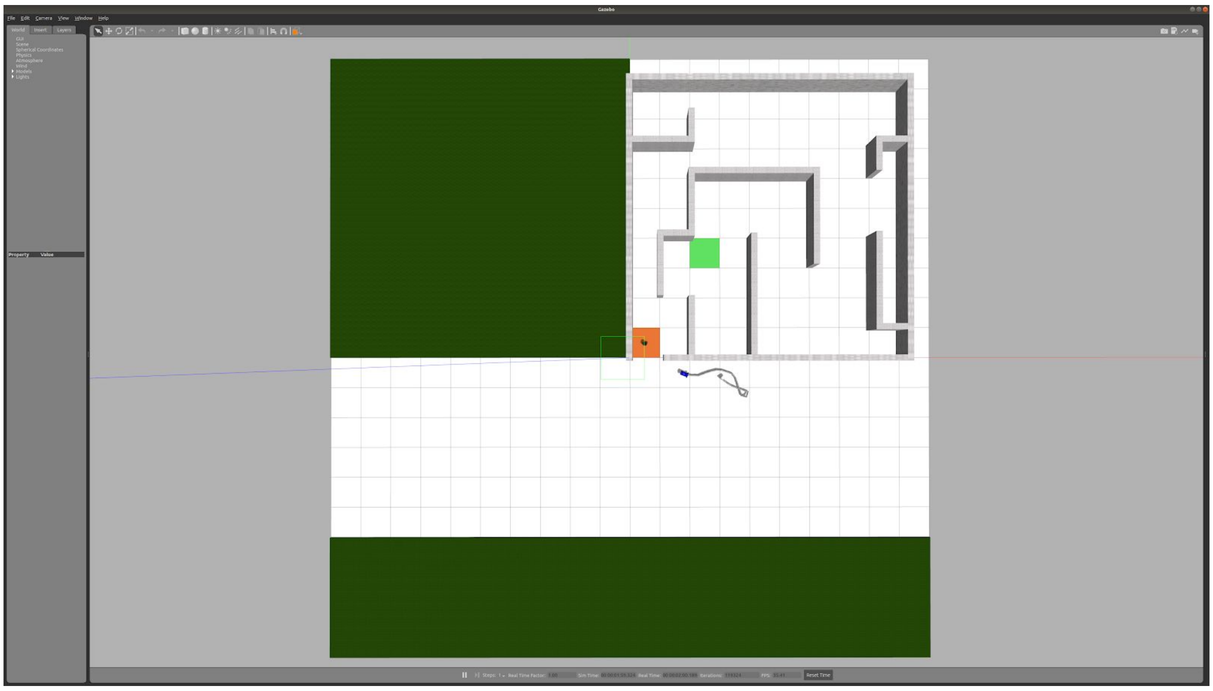


Figure 2: Initial settings of environment

The final destination is set according to the last letter of the matriculation number. As my last letter is L, the final destination's location is located at $(2,3)$ according to the given target_list.pdf. The location of the destination cell defined on line 413 in test_world_1.world is changed to 2.5 and 3.5 for the first two numbers. The macros GOAL_X and GOAL_Y in PreDefine.hpp are modified to 2 and 3 respectively.

Path Planning

Path planning is done in the ros node path_plan_node. The path planning algorithm is specified in the PathPlan.cpp. Whenever the node receives odometry readings, the odomCallback method is executed. The main higher level functions in odomCallback are shown in Figure 3. The initial path cost of each cell is set to 1000 as infinite cost except for the destination cell that has a path cost of 0.

```
checkWall();
path_plan_alg();
if(!goal_reached_){
 setNextDestCell();
}
```

Figure 3: High level functions in path planning

The function checkWall() checks the distance in the North, South, East, and West directions with respect to the environment. If the distance in that direction is less than a threshold, a wall exists in the direction. Another function setWall is executed to update a variable wall_map which is a 9 by 9 by 4 matrix to hold information of whether the wall exists in a direction for a given cell in the maze. If the distance is above the threshold, the wall is removed. The algorithm to set the wall and remove in the function checkWall() for the North direction is shown in Figure 4. This function stores information about the walls in each cell to prevent the robot from crashing into the wall when moving to another cell if a wall is present.

```
if(dist_north_ < WALL_DETECT_DIST){
 setWall(pos_x_int, pos_y_int, NORTH);
}
if(dist_north_ > OPEN_DETECT_DIST){
 removeWall(pos_x_int, pos_y_int, NORTH);
}
```

Figure 4: Detecting walls at given cell

The next function path_plan_algo() calculates the shortest path from an arbitrary cell to the final destination. The function has three variables visited_map, is_queued, and reached_queue as shown in Figure 5. The variable visited_map stores information about whether the cells in the maze have been visited. This information determines the next unexplored node with the minimum cost to calculate the path cost. The variable reached_queue tracks nodes that are explored to calculate the path cost of neighbouring unexplored nodes in the next iteration of BFS. The variable is_queued, on the other hand, tracks whether the node is stored in the variable reached_queue to prevent storing of duplicates nodes.

```
mat visited_map(GRID_SIZE, GRID_SIZE, fill::zeros);
mat is_queued(GRID_SIZE, GRID_SIZE, fill::zeros);
std::vector<pair<int, int>> reached_queue;
reached_queue.push_back(pair<int, int>(GOAL_X, GOAL_Y));
```

Figure 5: Variables to calculate path cost

In every iteration of BFS, the unexplored neighbouring node with the minimum path cost is obtained in the iterative loop in Figure 6. The node is then marked as explored.

```
for(int i = 0; i<queue_length; i++){
 pair<int, int> tmp = reached_queue[i];
 if(visited_map.at(tmp.first, tmp.second) == 0){
  if(path_map_(tmp.first, tmp.second) < min_dist){
   min_dist = path_map_.at(tmp.first, tmp.second);
   node = tmp;
  }
 }
}
visited_map(node.first, node.second) = 1;
```

Figure 6: Getting unexplored cell with minimum path cost

The neighbours of the node obtained in Figure 6 is added to the reached_queue if there is no wall in that direction and if the neighbours are not in the queue. The current path cost of the node and the path cost to reach the neighbour is compared and the minimum one is set as shown in Figure 7. The path_map_ stores the path cost for each cell. It is updated whenever the robot moves to another cell.

```
if(!is_queued(x_queue, y_queue)){
 reached_queue.push_back(pair<int, int>(x_queue, y_queue));
 is_queued(x_queue, y_queue) = 1;
}
path_map_(x_queue, y_queue) = min(path_map_(x_queue, y_queue),
min_dist+1);
```

Figure 7: Calculating path cost for neighbouring cell

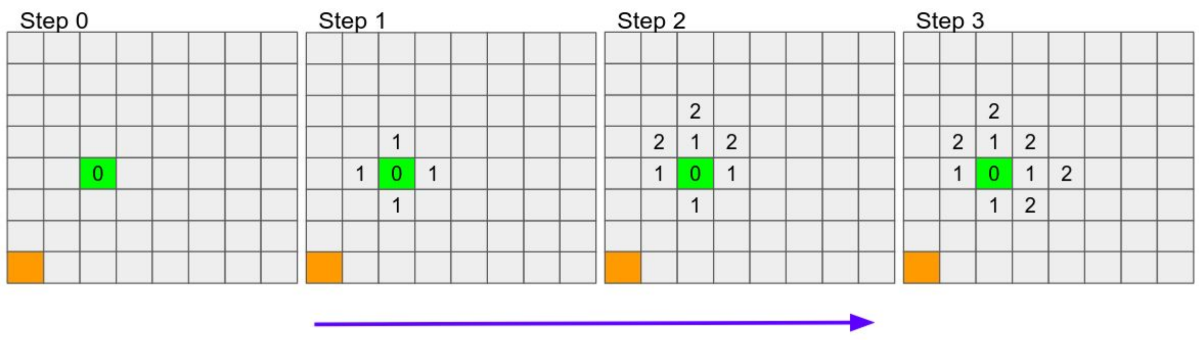
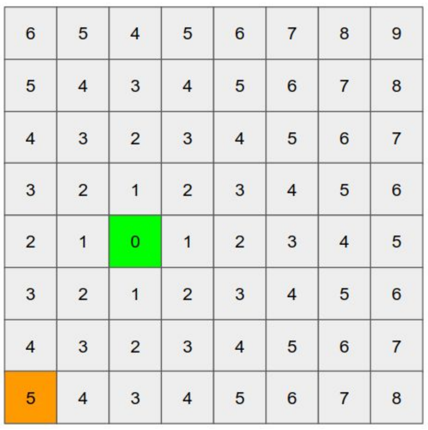


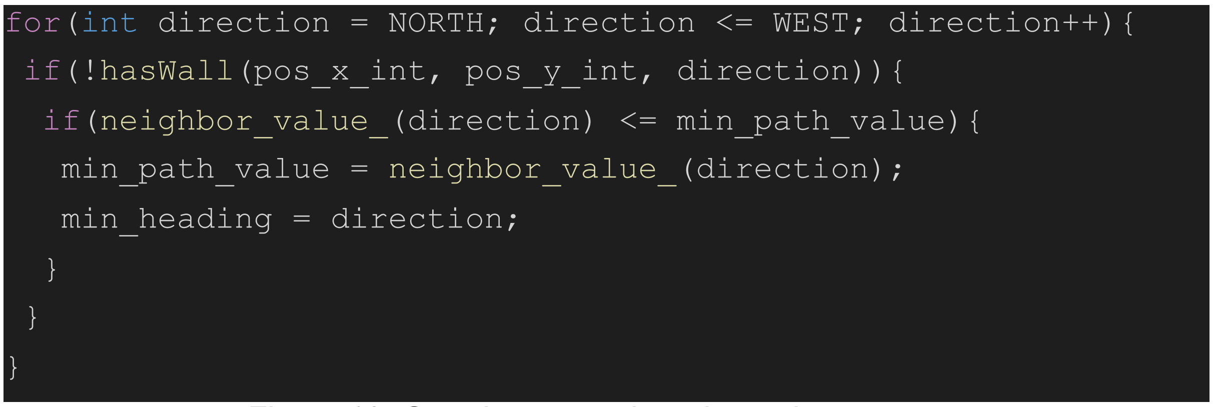
Figure 8: Initial steps in calculating path costs



6	5	4	5	6	7	8	9
5	4	3	4	5	6	7	8
4	3	2	3	4	5	6	7
3	2	1	2	3	4	5	6
2	1	0	1	2	3	4	5
3	2	1	2	3	4	5	6
4	3	2	3	4	5	6	7
5	4	3	4	5	6	7	8

Figure 9: Path costs at the end of calculation

Figure 8 visually illustrates how the BFS updates the path cost for the next neighbouring node that is unexplored starting with the goal next from left to right. In step 0, the path cost for the destination cell (2, 3) is set to 0 while the rest have the value of 1000. In step 1, the neighbours are given a path cost of 1 and appended to the vector reached_queued, and (2,3) is marked as visited. In the next iteration, the path costs for the neighbours for the nodes (2,4) and (3,3) are calculated in order for the next iterations. After all the iterations, the final path cost is shown in Figure 9. The path cost is updated whenever a new wall is detected at the cell.



```
for(int direction = NORTH; direction <= WEST; direction++){
 if(!hasWall(pos_x_int, pos_y_int, direction)){
  if(neighbor_value_(direction) <= min_path_value){
   min_path_value = neighbor_value_(direction);
   min_heading = direction;
  }
 }
}
```

Figure 10: Greedy approach to determine next cell

The next function setNextDestCell() uses the path costs from Figure 9 to determine the next cell to move to. It chooses the cell that has the lowest path cost as shown in Figure 10. The neighbouring cell of the selected cell with the lowest path cost becomes the next target the robot heads to. This greedy approach of always choosing the minimum path cost enables the robot to reach the destination through the shortest path.

Using the path cost from Figure 9, the Turtlebot3 robot achieves the shortest path from the starting point to the destination. When two neighbouring cells share the same path cost, the algorithm checks the neighboring cell clockwise from North to West and selects the one that is last computed because the comparison of minimum cost uses the <= operator. The path taken from (0,0) to (2,3) is shown in Figure 11. The only exception is for the first cell. This is due to the orientation of the robot that measures a distance from itself to the wall to the South at (1,0). At the start of the program, the walls are by default around (0, 0) and the measured distance is not below a threshold. Hence, a wall is marked to the East of (0, 0) and the robot heads North as the first step.

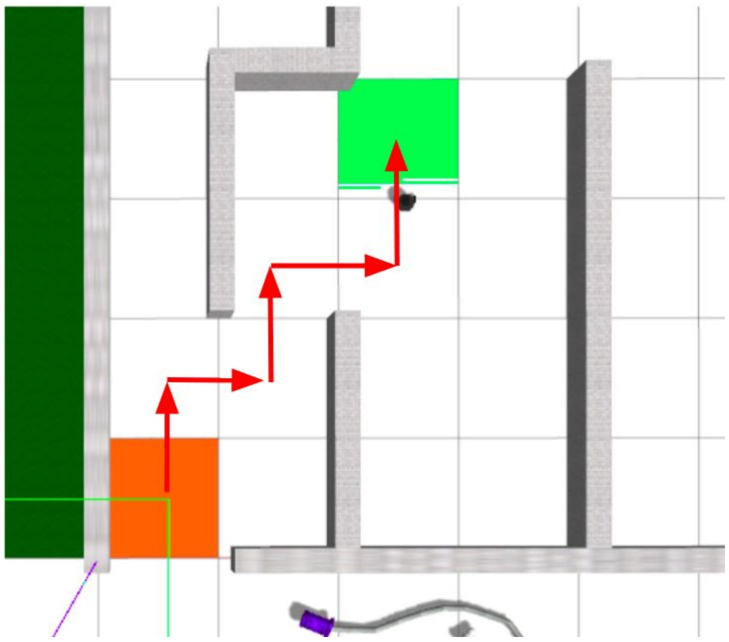


Figure 11: Minimum path to destination

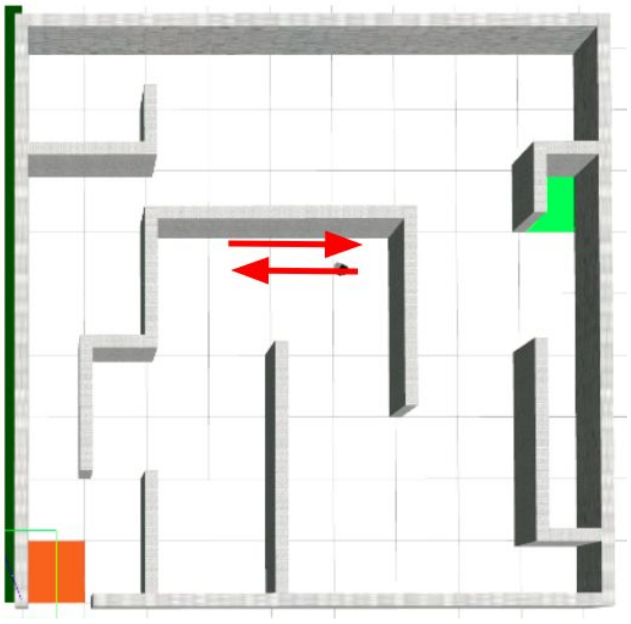


Figure 12: Robot stuck at a corner

This greedy approach does not always guarantee a solution as illustrated in Figure 12. When the destination is set at position (8,6), it is observed that the robot moves back and forth for cells (3,5), (4,5), and (5,5) at the corner of the obstacle in endless cycles. The robot did not manage to reach the final destination. This might be due to some bugs in updating the walls that affect how the path cost is calculated. As the robot also does not store previous cells, the robot revisits the old cell and ends up in an endless loop. The algorithm is modified to include backtracking with Depth First Search (DFS) and will be explained later.

Figure 13 shows a scenario where the robot is in cell (1,1) and the robot detects an obstacle in the North and East directions. As the robot is not aware of other obstacles in the maze, the path costs in other cells remain the same. However, the path cost for the cell where the robot is currently at is different. The path costs help to determine the next direction the robot heads to. It also helps the robot to achieve the shortest path to destination. In the ideal scenario where the walls are updated correctly, the updated path cost will guide the robot to the destination.

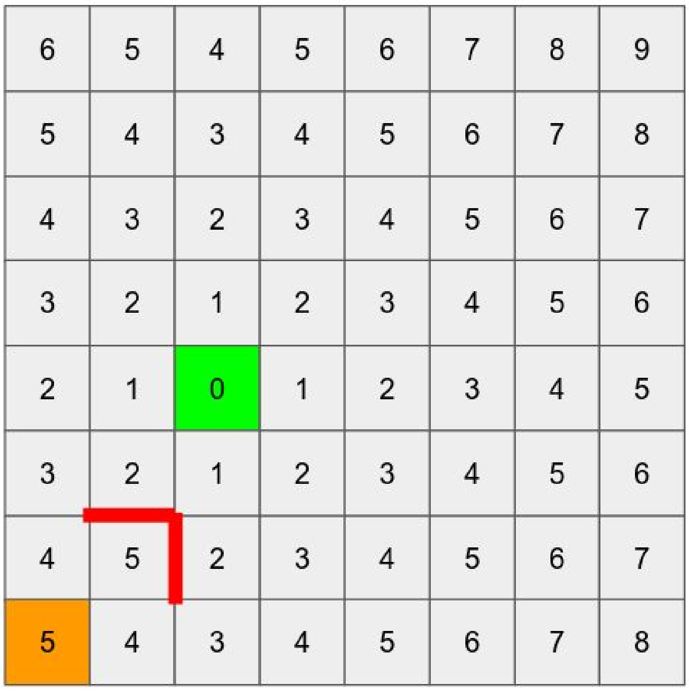


Figure 13: Path costs at an obstacle

Description Of ROS Simulation

The RQT graph displays the nodes and topics that are present in the ROS simulation with Gazebo as shown in Figure 14. Nodes are represented as ovals and topics are represented as rectangles.

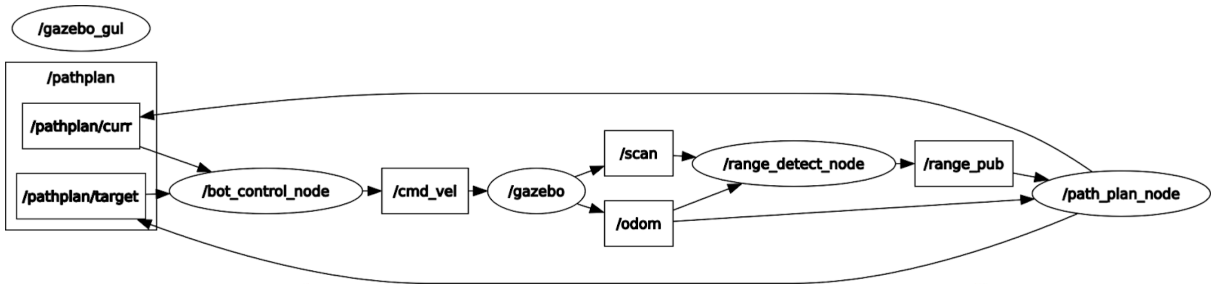


Figure 14: RQT graph of all nodes and topics in Gazebo simulation

Node: /gazebo

The node /gazebo is responsible for setting the Turtlebot3 robot and environment for simulation. Forces applied onto the robot, such as the translational and rotational velocity from the topic /cmd_vel, updates the robot position in the environment. The robot has a 2D LIDAR that continuously does a sweep around it and reads the distance at an angle. The information about the distance at different angles are published to the topic /scan. It also publishes the odometry readings to the topic /odom that contains information on the position and orientation of the robot in the environment.

Node: /range_detect_node

The node /range_detect_node is responsible for getting the distance reading in the North, South, East, and West directions as shown in Figure 15. It subscribes to the topic /scan that has the message type sensor_msgs::LaserScan. Figure 16 shows that the angle to the North, South, East, and West relative to the environment are computed and the distance from the payload is read through the index. When the distance is recorded as infinity, the sensor did not detect an obstacle at that angle and the max_dist_ of 4 in meters is assigned. The node publishes the distance reading to the topic /range_pub that has the message type of std_msgs::Float32MultiArray. The distance readings for North, East, South, West are stored in an array from 0 to 3 indexes and published.

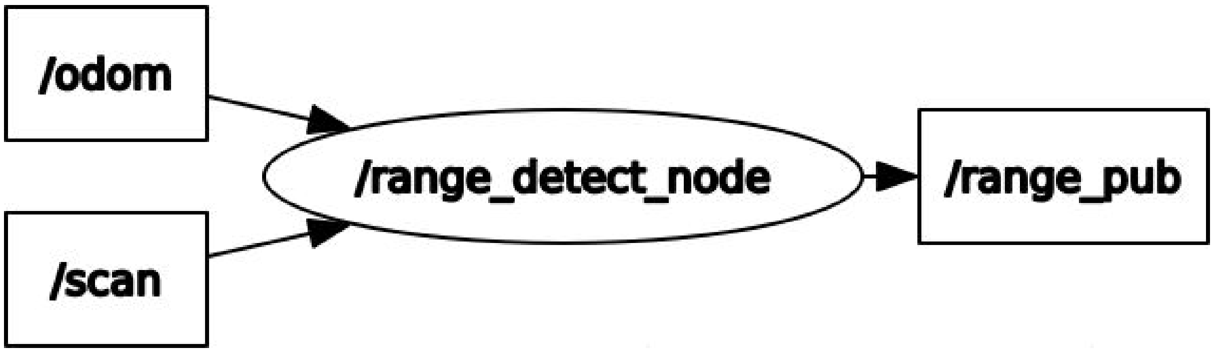


Figure 15: ROS node for range detection

```
scan_east_ = scan_data_[east_index];
scan_north_ = scan_data_[north_index];
scan_west_ = scan_data_[west_index];
scan_south_ = scan_data_[south_index];
if(isinf(scan_east_)) scan_east_ = max_dist_;
if(isinf(scan_north_)) scan_north_ = max_dist_;
if(isinf(scan_west_)) scan_west_ = max_dist_;
if(isinf(scan_south_)) scan_south_ = max_dist_;
```

Figure 16: Getting scan readings for NSEW directions

Node: /path_plan_node

The node /path_plan_node is responsible for updating the path costs and deciding the next target cell the robot should head to as shown in Figure 17. It subscribes to the topic /range_pub to get the distance in the North, South, East, and West directions as shown in Figure 18. The distance information is used to check for walls as described in Figure 4 in the function checkWall(). The node also subscribes to the topic /odom that has message type nav_msgs::Odometry to retrieve the current position and orientation of the robot. Everytime a new odometry reading is received, the robot calculates the path cost and determines the next target cell to head to reach the goal. When the target cell is determined, the coordinates are published to the topic /pathplan/target that has the message type geometry_msgs::Point. The current position of the robot is also published by the node /path_plan_node to the ros topic /pathplan/curr.

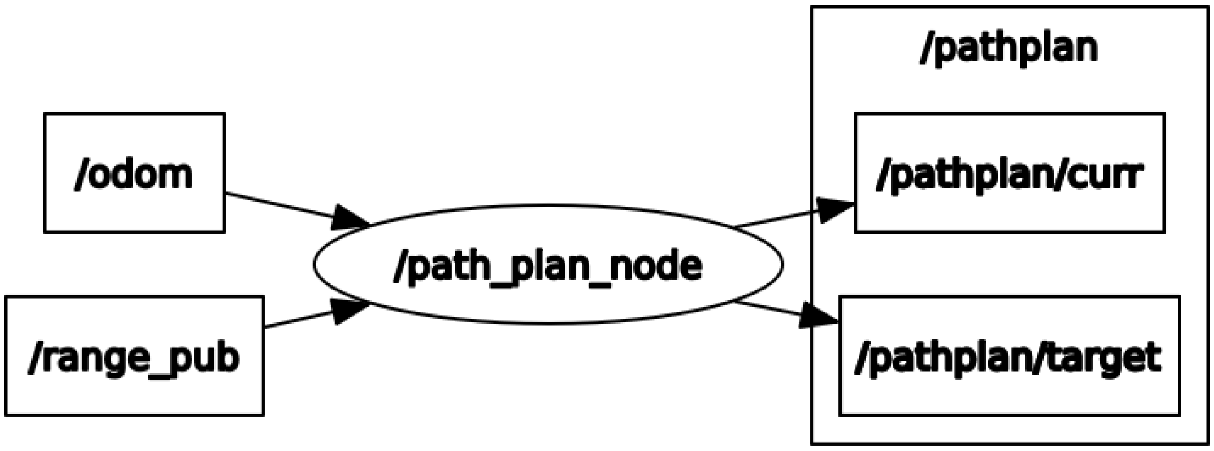


Figure 17: ROS node for path planning

```
dist_north_ = rangeMsg.data[0];
dist_east_ = rangeMsg.data[1];
dist_south_ = rangeMsg.data[2];
dist_west_ = rangeMsg.data[3];
```

Figure 18: Reading the distance for NSEW directions from subscribed topic

Node: /bot_control_node

The node bot_control_node is responsible for the steering and motor control as shown in Figure 19. It subscribes to the topic /pathplan/target and /pathplan/curr which provides information on the position of target and the robot current position. Everytime the target is received, the function controlPub() is executed. The controlPub controls the movements using PIs by correcting the errors between the position of the target and robot current position. However, the implementation of the PI controls is different from that of Project 1 where the D gain is accounted for as shown in Figure 20. There is a control logic that checks for the heading error. If it is above a threshold, the robot only corrects its heading but there is no linear motion.

Otherwise, the heading and linear motion is calculated and published to the topic /cmd_vel that has a message type of geometry_msgs::Twist.

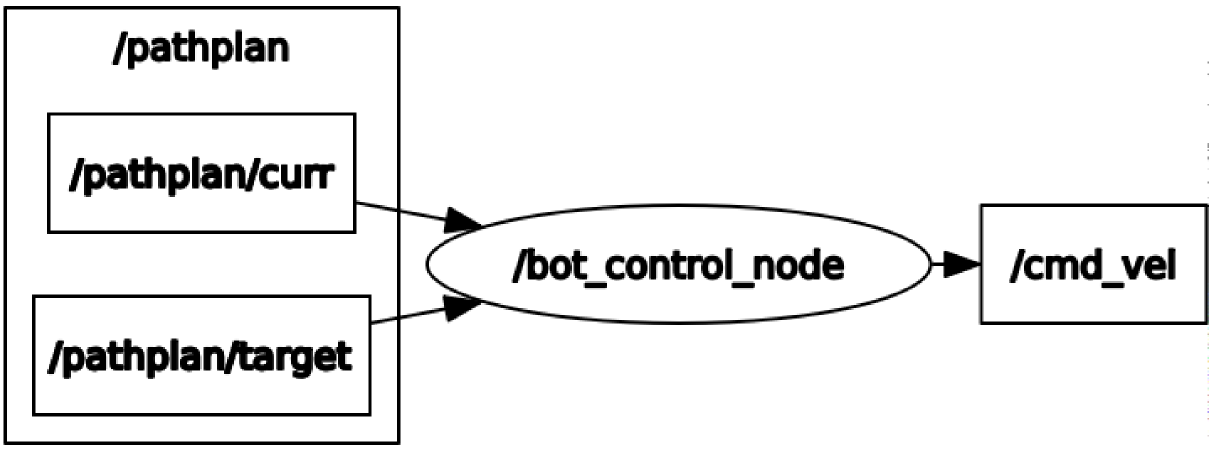
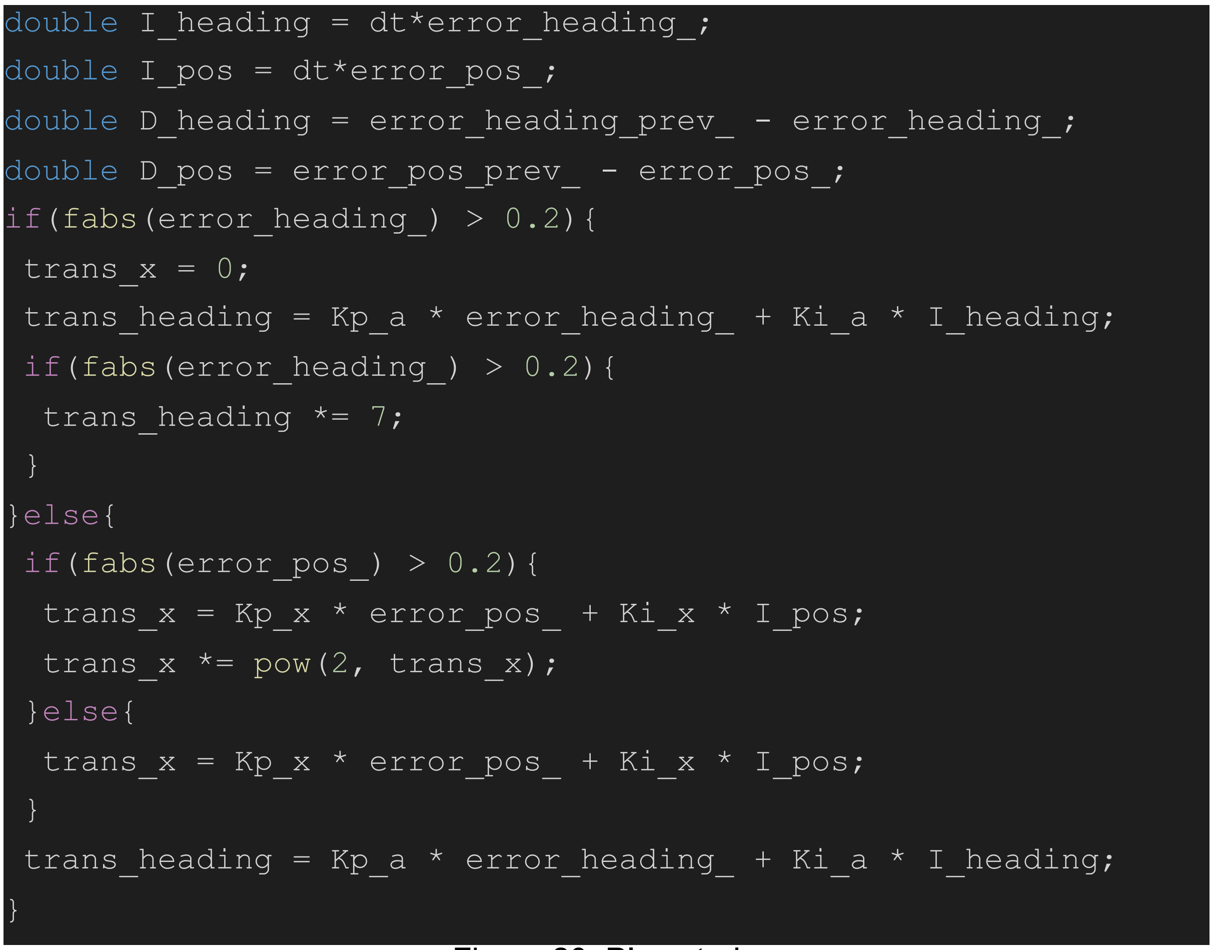


Figure 19: ROS node for controller



```
double I_heading = dt*error_heading_;
double I_pos = dt*error_pos_;
double D_heading = error_heading_prev_ - error_heading_;
double D_pos = error_pos_prev_ - error_pos_;
if(fabs(error_heading_) > 0.2){
 trans_x = 0;
 trans_heading = Kp_a * error_heading_ + Ki_a * I_heading;
 if(fabs(error_heading_) > 0.2){
  trans_heading *= 7;
 }
}else{
 if(fabs(error_pos_) > 0.2){
  trans_x = Kp_x * error_pos_ + Ki_x * I_pos;
  trans_x *= pow(2, trans_x);
 }else{
  trans_x = Kp_x * error_pos_ + Ki_x * I_pos;
 }
 trans_heading = Kp_a * error_heading_ + Ki_a * I_heading;
}
```

Figure 20: PI controls

Topic: /scan

The topic /scan provides 2D LIDAR measurements of the simulated environment. It has the message type sensor_msgs::LaserScan. The distance information in this payload is used to determine whether there is a wall in the North, South, East, or West direction of the robot.

Topic: /range_pub

The topic /range_pub provides the distance information in the North, South, East, and West directions of the robot in an array format and is accessible through its indexes. It has the message type std_msgs::Float32MultiArray. A wall exists in a direction if the distance in the direction falls below a certain threshold.

Topic: /odom

The topic /odom provides odometry readings of the robot. The message type is of nav_msgs::Odometry. It provides information on the current position and orientation of the robot. This information is used to determine which cell the robot is at, its angle relative to the world, whether the robot reached the next target cell, or whether the robot reached the destination cell.

Topic: /pathplan/target

The topic /pathplan/target provides the position and orientation of the target cell the robot heads to next. The message type is of geometry_msgs::Point. It is used in the PI controller to help the robot to navigate and drive the robot to the target.

Topic: /pathplan/curr

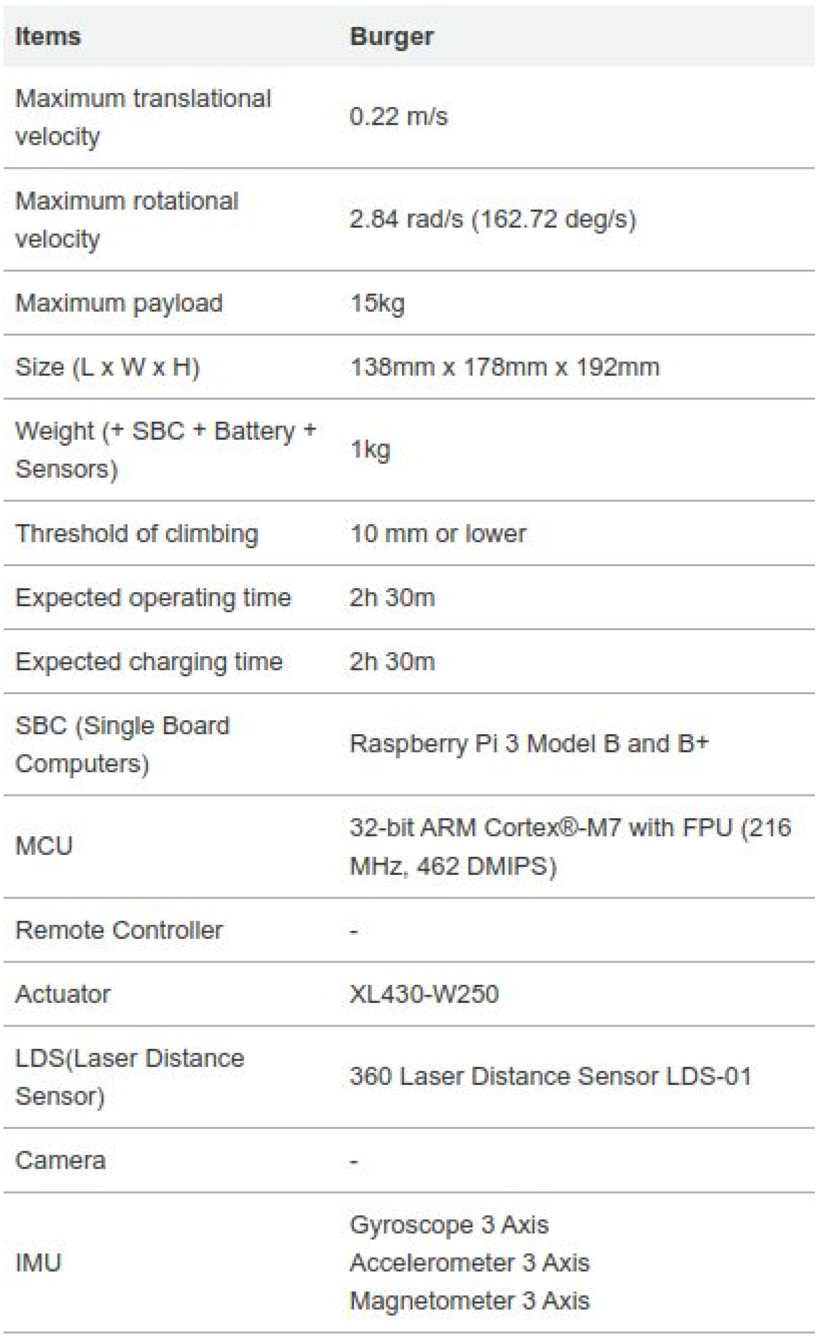
The topic /pathplan/target provides the position and orientation of the robot in the environment. The message type is of geometry_msgs::Point. It is used in the PI controller to help the robot to navigate and drive the robot to the target.

Topic: /cmd_vel

The topic /cmd_vel provides the position and orientation movements of the robot. The message type is of geomtry_msgs::Twist. The PIs calculates the position and orientation movements from error in the position of the target and position of the robot.

Real Implementation Of ROS

The ROS simulation in Gazebo for path planning and navigation uses Turtlebot3 Burger. The specifications of the robot in the real world are shown in Figure 21. By default, the values in ROS are in SI units.



Items	Burger
Maximum translational velocity	0.22 m/s
Maximum rotational velocity	2.84 rad/s (162.72 deg/s)
Maximum payload	15kg
Size (L x W x H)	138mm x 178mm x 192mm
Weight (+ SBC + Battery + Sensors)	1kg
Threshold of climbing	10 mm or lower
Expected operating time	2h 30m
Expected charging time	2h 30m
SBC (Single Board Computers)	Raspberry Pi 3 Model B and B+
MCU	32-bit ARM Cortex®-M7 with FPU (216 MHz, 462 DMIPS)
Remote Controller	-
Actuator	XL430-W250
LDS(Laser Distance Sensor)	360 Laser Distance Sensor LDS-01
Camera	-
IMU	Gyroscope 3 Axis Accelerometer 3 Axis Magnetometer 3 Axis

Figure 21: Hardware specifications of Turtlebot3 Burger

Maximum translational velocity

In the real Turtlebot3 burger robot, the maximum translational velocity is 0.22m/s. In simulation, the Turtlebot3 burger is given a maximum translational velocity of 0.7m/s. This is specified in the variable max_vel inside the file BotControl.hpp in the ros node /bot_control_node. When the PI calculates a forward velocity greater than the max_vel, the forward velocity is changed and set to max_vel and not anything higher. This is shown in Figure 22.

```
if(trans_x > max_vel) trans_x = max_vel;
if(trans_x < -max_vel) trans_x = -max_vel;
```

Figure 22: Maximum translation velocity

Maximum rotational velocity

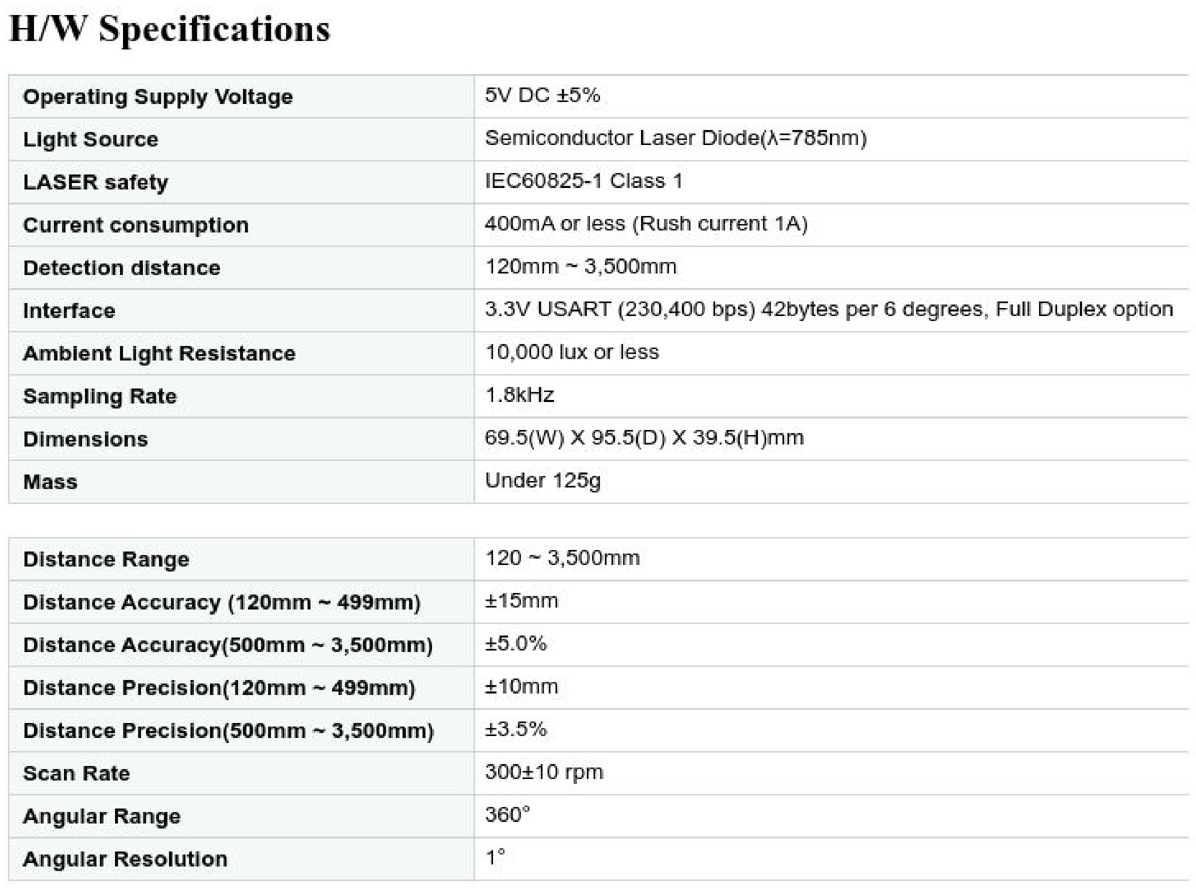
In the real Turtlebot3 burger robot, the maximum rotational velocity is 2.84rad/s. In simulation, the Turtlebot3 burger has a maximum rotational velocity of 3.1415926rad/s. This is specified in the variable max_ang inside the file BotControl.hpp in the ros node /bot_control_node. When the PI calculates a rotational velocity greater than the max_ang, the rotational velocity is changed and set to max_ang and not anything higher. This is shown in Figure 23.

```
if(trans_heading > max_ang) trans_heading = max_ang;
if(trans_heading < -max_ang) trans_heading = -max_ang;
```

Figure 23: Maximum rotational velocity

Laser Distance Sensor (LDS)

In the real Turtlebot3 burger robot, the LDS is 360 LDS-01. The LDS has the specification shown in Figure 24. It has a detection distance of 120mm to 2500mm and angle resolution of 1°. It is an active sensor that emits light energy and calculates the distance based on the difference in the reflected intensity and the output intensity. The LDS sensor is specified in the turtlebot3_burger.urdf.xacro. In Gazebo simulation, this information can be obtained from the ros topic /scan that has the message type sensor_msgs::LaserScan. The minimum and maximum detection distance is specified as the variables range_min (0.12m) and range_max (3.5m) of message sensor_msgs::LaserScan respectively. The angle resolution is specified as the variable angle_increment (0.0175rad). The light intensities and distance readings can be obtained from the variables intensities and ranges in the message payload.



H/W Specifications

Operating Supply Voltage	5V DC ±5%
Light Source	Semiconductor Laser Diode(λ=785nm)
LASER safety	IEC60825-1 Class 1
Current consumption	400mA or less (Rush current 1A)
Detection distance	120mm ~ 3,500mm
Interface	3.3V USART (230,400 bps) 42bytes per 6 degrees, Full Duplex option
Ambient Light Resistance	10,000 lux or less
Sampling Rate	1.8kHz
Dimensions	69.5(W) X 95.5(D) X 39.5(H)mm
Mass	Under 125g

Distance Range	120 ~ 3,500mm
Distance Accuracy (120mm ~ 499mm)	±15mm
Distance Accuracy(500mm ~ 3,500mm)	±5.0%
Distance Precision(120mm ~ 499mm)	±10mm
Distance Precision(500mm ~ 3,500mm)	±3.5%
Scan Rate	300±10 rpm
Angular Range	360°
Angular Resolution	1°

Figure 24: 360 LDS-01 specifications

Inertial Measurement Unit (IMU)

In the real Turtlebot3 burger robot, the IMU has a gyroscope, accelerometer, and magnetometer, each giving readings for the x, y, and z axis. The IMU sensor is specified in the turtlebot3_burger.urdf.xacro. In Gazebo simulation, the information can be obtained from the ros topic /imu that has a message type of sensor_msgs::Imu. The variable orientation inside the message stores the orientation of the robot and has data type of geometry_msgs::Quaternion. The orientation in the form quaternion can compute the role, pitch and yaw of the robot. The linear acceleration is obtained from the variable linear_acceleration while the angular_velocity is obtained from the variable angular_velocity. The IMU also provides the orientation of the robot in quaternion in the odometry measurements in the ros node /range_detect_node. The yaw angle or the steering angle of the robot can be obtained as shown in Figure 25. Leveraging existing ROS packages,[1] the IMU measurements are combined with the data from rotary encoder sensors to get the best value of quaternion orientation.[2]

```
double qx = odomMsg.pose.pose.orientation.x;
double qy = odomMsg.pose.pose.orientation.y;
double qz = odomMsg.pose.pose.orientation.z;
double qw = odomMsg.pose.pose.orientation.w;
ang_z_ = atan2(2*(qw*qz+qx*qy), 1-2*(qz*qz+qy*qy));
```

Figure 25: Steering angle from quaternion

Size

In the real Turtlebo3 burger robot, the size (L by W by H) of the robot is 138mm by 178mm by 192mm and the mass is 1kg. The size of the robot in Gazebo simulation is specified in the turtlebot3_burger.urdf.xacro. It is possible to adjust the size and mass of individual components on the Turtlebot, such as modifying the contents in the xml tags for scale, mass, box, cylinder etc.

[1] http://wiki.ros.org/imu_filter_madgwick
[2] https://answers.ros.org/question/289720/orientation-from-wheel-based-odometry-vs-imu/

Software Improvements

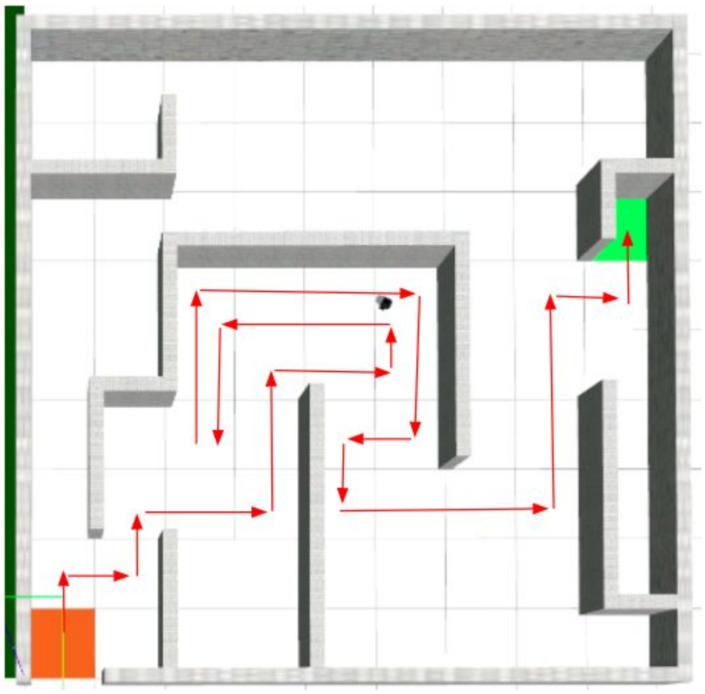


Figure 26: Path taken for DFS

Improvements to the path planning algorithm were made to help the achieve reach the destination cell. Together with the path cost, Depth First Search (DFS) with backtracking was implemented to help the robot achieve the goal with a relatively good path cost as shown in Figure 26. Note that some of the turns might be unexpected because of false detection of walls when the robot is orientated at an angle. The main difference with this implementation is that the history of cells that the robot visited are recorded. In the event the robot gets stuck, it will move to the previous cell it came from. Two additional variables history of type stack to store a pair of cell coordinates and explored of type set to store a pair of cell coordinates that was explored as shown in Figure 27.

```
std::stack<std::pair<int, int>> history;
std::set<std::pair<int, int>> explored;
```

Figure 27: Variables for DFS solution

Figure 28 shows how the robot decides the next direction to head to next. There are two modifications to this block of code. The first is the boolean is_all_blocked to handle the case where the robot is at the starting position. This is to check when all directions of the cell have walls and the robot cannot move. The second is a boolean is_in to check if the next cell to explore has been explored. This is to prevent the robot from exploring a cell that it was stuck in and did not reach the goal .

```
bool is_all_blocked = true;
for(int direction = NORTH; direction <= WEST; direction++){
  if(!hasWall(pos_x_int, pos_y_int, direction)){
    is_all_blocked = false;
    if(neighbor_value_(direction) <= min_path_value){
      int x; int y;
      if(direction == NORTH){x = pos_x_int; y = pos_y_int+1;}
      if(direction == EAST){x = pos_x_int+1; y = pos_y_int;}
      if(direction == SOUTH){x = pos_x_int; y = pos_y_int-1;}
      if(direction == WEST){x = pos_x_int-1; y = pos_y_int;}
        const bool is_in = explored.find(std::pair<int,int>(x,
y)) != explored.end();
      if (is_in) {
        continue;
      }
      min_path_value = neighbor_value_(direction);
      min_heading = direction;
    }
  }
}
```

Figure 28: Algorithm to visit the next neighbouring cell

Figure 29 shows the logic for backtracking. If all the directions in the cell have walls, the robot does not change targets. If the robot has not reached the next target cell, the robot will continue moving towards the center of the target cell. If the previous conditions satisfy, the algorithm checks the heading. If the heading is undefined, this means that the robot either visited neighbouring cells or is blocked due to the walls, the robot is stuck and backtracks to a previous cell and continues exploring. Otherwise, the robot will set the cell as the next target and append to the history of cells explored. Refer to the video ME3243: Project 2 Path Planning And Navigation With BackTracking on Youtube to see how the backtracking works.

```
if (is_all_blocked) {
  return;
}

std::pair<int, int> node = history.top();
if (!(abs(pos_x_ - (node.first + 0.5)) <= 0.1 && abs(pos_y_ -
(node.second + 0.5)) <= 0.1)) {
  return;
}

if(min_heading < 0){
  history.pop();
  std::pair<int, int> prev = history.top();
  target_x_ = prev.first;
  target_y_ = prev.second;
 }else{
  target_x_prev_ = target_x_;
  target_y_prev_ = target_y_;
  if(min_heading == NORTH){target_x_ = pos_x_int; target_y_ =
pos_y_int+1;}
  if(min_heading == EAST){target_x_ = pos_x_int+1; target_y_ =
pos_y_int;}
  if(min_heading == SOUTH){target_x_ = pos_x_int; target_y_ =
pos_y_int-1;}
  if(min_heading == WEST){target_x_ = pos_x_int-1; target_y_ =
pos_y_int;}

  history.push(std::pair<int,int>(target_x_, target_y_));
  explored.insert(std::pair<int, int>(target_x_, target_y_));
}
```

Figure 29: Algorithm to backtrack