MIE443 Contest 1:
Autonomous Robot Search of an Environment

Hui Di Wang, Si Xu, Hunter Richards, Yu-Chien Chen, Xu Bo Yang Huang

January 2020

1 Objective

The objective of this contest is to use the TurtleBot to autonomously navigate an unknown environment while simultaneously using the ROS Gmapping package to generate a map of the environment using sensory information from the on-board Kinect sensor. The robot will have a maximum of 8 minutes to perform the exploration and mapping task, and the goal is to map as much of the environment as possible within the time limit. The environment will be 4.87m x 4.87m and contain static objects within that area, though neither the layout nor the initial position and orientation of the robot will be known. The robot also should not exceed a maximum speed of 0.25m/s at all times, and when beside obstacles the robot should not exceed a speed of 0.1m/s to ensure mapping accuracy.

Since the mapping is done by the Gmapping package, the main challenge that our team needs to solve is creating the algorithm that the robot will use to navigate the environment. The algorithm must allow the robot to operate with complete autonomy with no human interventions, and also must utilize the information obtained by the on-board sensors. This algorithm must contain different sections responsible for both the high level and low level controls, performing tasks ranging from determining where the robot should go next to improve the generated map, to how the robot will react when running into a wall.

2 Strategy

The Turtlebot is a differential drive robot, and we can control its motion through giving it a linear velocity and an angular velocity value. To determine the velocity values, we will utilize the data obtained from the 2 on-board sensors, the Kinect laser sensor, and the bumper sensors. The robot also has a built-in odometry system that allows us to obtain the estimated position and orientation of the robot relative to it's starting location. The Gmapping package also provides a transform that allows the transforming of odometry coordinates to map coordinates, allowing the estimation of the robot's position on the generated map grid.

Our initial strategy was to have the robot circle each single obstacle it finds until it has circled every obstacle, and then go towards any unexplored area on the map. This would be done by using a wall following algorithm on the walls of an obstacle. The robot would also, while circling an obstacle, record other potential obstacles it detected with the laser sensor. Once the robot's pose matches it's pose when it first started circling this obstacle, it will move onto the next closest potential obstacle it detected and repeat the process. We quickly ran into issues with this approach due to two reasons. Firstly, it proved difficult, due to the narrow angle of the Kinect laser sensor and the irregular shapes of the obstacles, to have the robot turn a corner accurately to continue its wall

following. Secondly, the accuracy of Gmapping also proved troublesome, due to errors in the localization, the robot's current position on the map would be incorrectly estimated. The odometry position estimate was also inaccurate due to error build up over time as it had no self-correction mechanism to reduce said error. This meant path planning for the robot became unreliable as the robot's current position estimation was inaccurate.

During testing, we also found that basic wall following which randomly switched between right and left wall following was able to map most of the test environment. Since this wall following no longer requires the robot to fully circle an obstacle before it moves on to the next, the issue of the robot not being able to follow a wall around a corner was also resolved as it is acceptable for the robot to move on and start following another wall instead. Thus, we changed our strategy to first using random wall following to first map most of the environment, and then navigate to the few left over unexplored areas and spin the robot to map those areas. We will use frontier search to find locations where an obstacle-less area neighbors an unexplored area, and then use a bug algorithm to navigate to said location. We chose frontier search because it allows us to quickly find unexplored areas that are not surrounding by walls and inaccessible. We chose to use a bug algorithm because even though it is less efficient from a time perspective compared to intelligent search like A*, it is robust and can handle the potential high error in the robot pose estimate and the destination location estimate. We also implemented a tolerance to the destination location and a time out, so that if the destination is inaccessible to the robot, it will not waste all the time trying, and will instead move onto a different frontier location. Combining these, we believe our strategy will be able to map the environment reliably regardless of the errors in the pose estimations and generated map grid.

3 Detailed Robot Design and Implementation

3.1 Sensory Design

The robot contains an on-board Kinect laser sensor, and a bumper sensor. As discussed in our strategies section, we decided to mainly use the Kinect laser sensor for both wall following and the bug algorithm, and only use the bumper as an emergency backup. This is because we wanted to minimize our robot's contact with obstacles for a cleaner map. The bumper sensor was used only as an emergency backup when contact did occur.

3.1.1 Lasers

Our wall-following algorithm primarily depended on laser readings to determine how the robot should move. The laser readings are obtained through a laser callback function. In the laser callback, the entire array of laser readings obtained from the Kinect sensor is read out and stored in a vector ordered by the angle of the laser beam. This vector is then used in our control algorithms by comparing it with certain thresholds to determine

the robot's next course of action.

We tested the laser sensor to have roughly a 60 degrees and 5m range, with a dead zone between 0m and 0.5m range of the robot which would result in a NaN reading. We also noticed that the origin of the laser is offset to one side of the robot, which means even if the robot has similar laser readings on both the left and right side, this does not mean the left and right walls are of equal distance to the robot. We had to compensate for this in our wall following algorithm by tuning the robot's threshold parameters to be different values for left and right wall following.

During testing, we found that a laser reading of NaN occurs when the robot is too close or too far from a wall. When these values occurred, the robot behave unexpectedly as these NaN values would cause issues during the comparison with our threshold values as well as comparison to get the minimum value of a range of lasers. To resolve this issue, NaN values were replaced with 100 to facilitate better comparisons in the control algorithms. 100 was chosen because the range of the lasers reading values were within the range of [0.5, 5] during experimentation, so 100 would always be larger than valid readings.

To use this data for wall following, the laser readings were divided into three groups that represented the left, front, and right sides of the robot based on the angle of each laser beam. The minimum value of each group of laser readings were taken to represent the distance on that side of the robot. This is just to ensure the robot had a minimum chance of collision with an obstacle due to not detecting it.

The laser data is also used in the Bug algorithm used to navigate the robot to the destinations found with our frontier search algorithm. Specifically, we use the laser readings to determine if there is an obstacle between our robot and the destination, at which point the robot switches to wall following. This continues until it no longer detects an obstacle between it and the destination, at which point it returns to simply trying to go towards the destination in a straight line.

3.1.2 Bumpers

There are 3 bumper sensors on-board the robot, a front bumper, and a front-side bumper on either side of the robot. Together the 3 bumpers cover the front half of the robot. These bumpers are used to ensure that if the robot collides with an obstacle while moving forward due to errors in either laser readings, pose estimation, or mapping and motion planning, it can detect the collision and act in response without causing further damages. Specifically, we continuously sample the bumper status whenever the robot is in motion, and whenever any of the three bumpers detects a collision, the robot immediately stops what it was doing and moves away from the direction of the bumper collision before resuming wall following. Though this may run into potential issues in tight areas where

the robot might back into another obstacle, the TAs assured us that the environment will have enough space between the obstacles that this issue shouldn't come up as long as the robot only backs away a reasonable distance.

3.2 Controller Design

There are three major algorithms that we use to control the robot. We use wall following to control the robot's behaviour when navigating around walls; frontier search to locate unexplored areas; and Bug2 to navigate the robot to the desired frontier locations. The robot changes between the 3 algorithms in the fashion of a finite state machine.

3.2.1 Wall Following

The purpose of the wall following algorithm is to allow the robot to move parallel to the walls of an obstacle while keeping the distance between the wall and robot constant. The wall following algorithm uses the principles of a PI controller, but modified to work with the laser sensors we have. A sample of the wall following pseudo-code is presented below.

Algorithm 1 Left Wall Following

```
1:  procedure LeftWallFollow
2:      leftturncounter ← 0
3:      while state == leftWallFollow do
4:          if (min(anylaser) < mindist or min(anylaser) > maxdist) then
5:              turn right on spot;
6:              leftturncounter ← 0
7:          else if leftlaser > followingdist and leftlaser < maxdist  then
8:              leftturncounter ← leftturncounter + 1
9:              if leftturncounter > turntime then
10:                 turn left on the spot
11:             else
12:                 go forward with slight turn left
13:         else
14:             go forward
15:             leftturncounter ← 0
```

The left wall following algorithm, while active, will constantly sample the laser readings and bin them into three groups of laser readings: in front, to the left, and to the right of the robot. If any of the laser readings are less than the minimum distance, or larger than the max distance, indicating a value of NaN, then we turn right, as it means that either the robot is too close to the left wall it is following, or has a wall in front of it and thus needs to turn right. This represents the P part of a PI controller. An issue might arise if the robot is sandwiched rightly between two walls, but the TA assured us

that the environment will not contain such narrow areas.

Since the above can only cause the robot to turn away from the wall, it cannot cause the robot to get closer to the wall if it starts drifting away from it. Therefore we also implemented a section that causes the robot to turn more and more to the left over time if the robot is constantly too far away from the left wall. If the robot does not see the left wall for a while, it will turn left on the spot to try to find the wall again. This part corresponds to the I part of a PI controller, where the longer the robot spends away from the left wall, the more aggressively it turns to the left. The reason for this is because the robot cannot detect directly to it's left, so in order to make sure the robot does not immediately sharply turn left every time the robot is far from the left way, we have this counter that only turns the robot if over a time period, it cannot detect the left wall.

We implemented both left and right wall following with the right wall following being the same as structure as left wall following, just with the left and right sides flipped. Since the laser is off-centered, the parameters for the threshold values used are tuned independently for the 2 wall following algorithms. The bumper acts as an emergency interrupt to the wall following algorithm. If the bumper collides with the obstacle, the current motion is interrupted and the robot backs away from the wall. After which, it resume wall following.

To ensure the robot doesn't get stuck circling the same obstacle, the robot will store it's pose as it starts the wall following, and when it reaches the same location, the robot will switch to the other wall following and continue.

3.2.2 Frontier search

Although wall following is successful in mapping the majority of the unknown space, the lack of overall knowledge of the map means it cannot visit unseen regions intentionally, especially those which are not adjacent to a wall. Thus, there are cases where certain areas of the map will never be seen by the robot due to the obstacle layout.

To augment the wall following algorithm, a frontier-search based approach is employed to find unexplored areas of the map. Locations at the boundary of visited, obstacle-less areas and unexplored areas are identified using the Wavefront Frontier Detector (WFD) and the location is passed into the Bug2 algorithm for path planning and navigation to said location. The method was adapted from the pseudocode described in[1], included in Appendix A.

The Wavefront Frontier Detector was accepts the occupancy grid, in the form of a 2D array, and the initial position, in x,y map coordinates, outputted by the Gmapping package. The occupancy grid contains a value at each of the map grid point, which takes a value of either -1, or between 0 to 100. A value of -1 indicates the location is

unexplored, while a value between 0 to 100 is the likelihood of that point being occupied by am obstacle. Zero indicates the square is unoccupied and a value of 100 indicates it is occupied. The algorithm searches for points with a value of zero which are adjacent to a point with a value of -1, this is a frontier point. Frontier points boundaries to unseen areas, so the algorithm tries to group frontier points into clusters and inform the high-level controller of cluster locations by providing the a point that is at the center of the cluster. This is done instead of providing the coordinates to all the points in a cluster as the robot's laser sensor allows it to observe a large amount of frontier points within the same cluster without having to visit each point.

The algorithm itself employs a set of nested Breadth-First-Searches (BFS). The search is initialized at the robot's current pose in map coordinates and inserted into a queue. To obtain the robot's current position in the map grid, the current robot position in the odometry frame is obtained from the odometry package, and fed through a transform provided by Gmapping package to transform the point to the map frame which is then scaled to the map grid coordinates using the map origin and resolution. A point is then dequeued for processing. Points that are dequeued are marked with the first state: Map-Close-List, indicating it was visited by the outer BFS. Neighbouring points are found using an eight-directional adjacency list, and the neighbours which contain a value of near zero are then added to the queue and marked as Map-Open-List, indicating they have been enqueued. The next point is then dequeued and the process repeats until the queue is emptied. Map-Open-List and Map-Close-List are state tags used to ensure the same point will not be added multiple times, so the BFS will ignore points marked with these states during the enqueueing stage. This outer BFS will run until a frontier point, that is a point with value 0 that neighbors a point with value -1, is dequeued.

Once a frontier point dequeued from the first queue, the inner-BFS is initialized with a separate queue and the discovered frontier point is enqueued into this second queue. The goal of the Inner-BFS is to find all connected frontier points to this initial frontier point and group them into a frontier cluster. Points are then dequeued from this second queue, and those that are frontier points are marked as Frontier-Close-List and added to a cluster set. Neighbours of dequeued points are found using an eight-directional adjacency list similar to before, they are marked as Frontier-Open-List and added the queue for further exploration. Once the queue is exhausted, all frontier points belonging to the cluster are considered found and the points in the cluster set are marked as Map-Close-List so they do not get visited again by the outer BFS. Frontier-Open-List and Frontier-Close-List function like their Map counterparts, preventing the same point from being added multiple times in either the outer or inner BFS. With this step, the inner-BFS is complete, so the cluster is added to the master list of frontier clusters and the algorithm returns to the outer-BFS, adding neighboring points of the original points to the first queue etc.

Once all empty points are exhausted, the outer-BFS concludes and the master list of

frontier points is processed. For each cluster of points, we first check the size of the cluster and ignore clusters that are too small. during testing, we found that due to inaccuracies in the mapping, a few small clusters of unexplored areas may appear on the map. These small clusters are mostly caused by errors in the laser readings and are usually small enough to not warrant the time for the robot to go back and explore that area again. the points in each cluster of reasonable sizes are then sorted by distance from the origin (0,0) and the median point is found and stored. This was another innovation presented in the WFD paper, as conventional frontier search uses the centroid, which is argued to be unreliable since it may not be a real point on the map. Comparatively, the median point is guaranteed to have been explored through the BFS, so it is connected directly to explored areas and is unlikely to be inside an obstacle or otherwise unreachable areas. This list of frontier cluster medians are then sorted with a simple heuristic function that uses the difference between the exponential of the cluster size and the distance of the median with the robot's current location. This effectively causes the robot to move favor larger frontier clusters more than smaller ones, and closer frontier clusters over further ones, making the robot traverse to larger close frontiers first. This list of frontier medians are then converted from map grid coordinates back to odometry frame coordinates and passed to the Bug2 algorithm, which then explores each of the frontier medians in order.

The main advantage of WFD is that, unlike conventional frontier search which scans the entire occupancy grid, the wavefront frontier detector only scans known regions. The benefits of this are that only points connected to the current location through an unobstructed path will be explored, which can help avoid seeking objectives which are unreachable from the current location. This also greatly decreases runtime since the algorithm only explores areas that have been mapped and reachable. In our implementation, the median points are returned in under one second for a map of size 500x500. An issue that could potentially arise is if there are narrow pathways where the laser sensor can see through but are too narrow for the robot to traverse, but after consulting with the TAs, we were assured that this would not appear in the environment. Another issue would be if the frontier location seems accessible on the map even though it is not due to errors in the mapping. To account for this, we have the algorithm that directs the robot towards the frontier location time out and move on to the next frontier location if it takes too long reaching the current frontier location.

The choice of using eight adjacency directions over four was based on testing, we found that eight directions resulted in bigger clusters, which was desired as these clusters tend to be smaller than the maximum range of the laser sensor, which means the robot can detect and map all the points in these clusters by traversing to 1 frontier median. Four-directional search would have trouble clustering large frontiers together and often resulting in several different cluster medians that are diagonally next to one another. The downside of eight-directions was the search was more likely to find frontier points inside walls due to inaccuracies in the mapping causing some of the walls to have diagonal

"frontier points". But ignoring small frontier clusters mitigated this issue significantly.

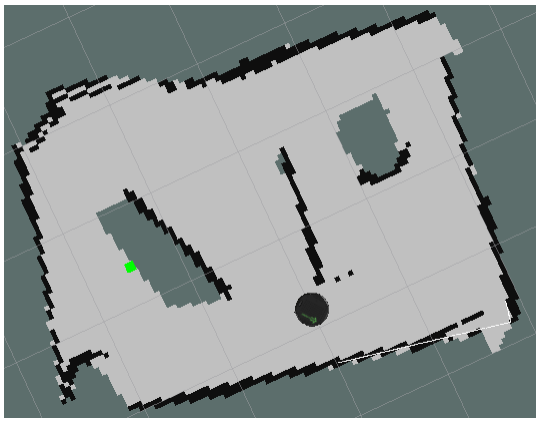

Figure 1: Frontier search finds the green median point

Figure 1 presents a visualization, using the markers from Rviz, of the frontier median found. For ordering of objectives, a priority was assigned to larger, and closer frontiers, hence why the large unseen area marked is preferred over the smaller potential frontier to the top-right.

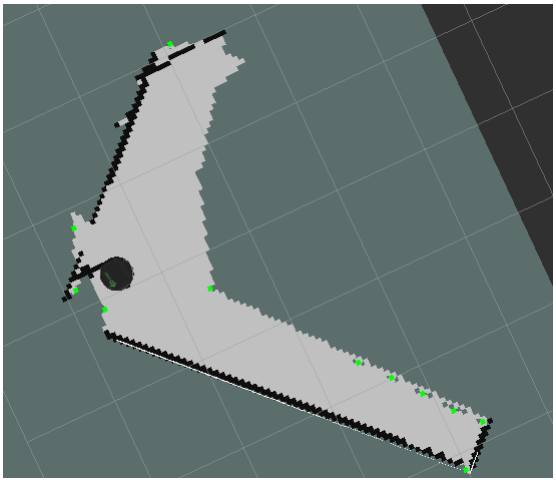

Figure 2: All detected frontier medians

In Figure 2, all frontier points detected are presented as green points on the map. This figure demonstrates two weaknesses of frontier search. First, unconnected regions will be detected as separate clusters even though they may be close together, and the robot will likely be able to map all of these frontier clusters from one of the frontier medians. On the bottom right, there were several empty points which were been missed by the laser scan, hence each of them are labeled as individual frontier medians as the algorithm believes them to be unexplored and separated. This was another reason for implementing a minimal cluster size requirement for the frontier clusters, as visiting each

of them would be a waste of time.

The second issue is that noise and inaccuracy in Gmapping can create holes in the walls which will then be considered as a frontier, as seen for example in the bottom right of Figure 2. The robot cannot reach that location but will continuously attempt to do so in the Bug2 Algorithm. A minimum cluster size helps in some cases but during testing we found sometimes the hole can be quite large, especially if the area was not well explored by the wall following algorithm. Thus, we required the Bug2 algorithm to have a set timeout, which would switch to a different frontier median once the robot spent too much time on a single frontier median. We also required it to have a set tolerance so that if the frontier median is beside, or even inside a wall in the physical environment, it will register as having reached the destination and move onto the next median as opposed to continuously ramming into the wall and backing away when the bumper triggers. .

3.2.3 Bug2 Algorithm

With a list of frontier medians provided by the WFD Frontier Search algorithm, a motion planning algorithm is needed so the robot can navigate through the environment to get to each frontier median while avoiding obstacles along the way. Given the time constraint for this project and the limits on the precision of the robot pose estimation and the accuracy of the map, we opted for a simple Bug2 algorithm which only requires the direction to goal, and local sensing abilities (i.e. wall/obstacle detection). More efficient algorithms like A* were considered, but due to the errors in the robot's current pose estimate, and the errors in the map, we opted for the more robust option.

In this algorithm[2], the robot attempts to move in a straight line. directly towards the goal. If an obstacle is encountered, the robot will start following the the perimeter of the obstacle using the wall following algorithm until it intersects with the straight line between the starting location and the goal location. It will then leave the wall following state and continue towards the goal along the straight line. This repeats until the robot is within a set tolerance of the goal location, or until a set timeout is reached, at which point the next frontier median is made the new goal. Once all the frontier medians has been reached, the WFD Frontier Search algorithm is called again and a new set of frontier medians is generated and the process repeats.

This algorithm is not the most efficient algorithm since there are cases where the robot may unnecessarily traverse through a large area of the environment circling a large obstacle before getting to its destination. However, during testing, we found that our wall-following algorithm is able to map most of the environment by itself, so the amount of frontiers that needs to be explored will be few, and thus we can afford to use a more robust and less efficient algorithm.

Some issues that were encountered during testing arose from sensor inaccuracies: the robot may deadlock and constantly switch between following the straight line to the destination and wall following when it is close to a wall and near the straight line. To resolve this, we added a time buffer where if the robot recently exited wall following, it cannot re-enter the wall following state until the time buffer has passed. This however causes another issue, which is that if the destination is inside an obstacle or beyond a wall, then the robot will simply run into the obstacle/wall continuously as it cannot switch back to wall following mode due to the time buffer. To resolve this, we added a bumper override that ignores the time buffer and switches to wall following state if any of the bumpers is pressed. Additionally, a timeout is added to prevent the robot from wasting its time trying to get to a point that is impossible to get to. The robot will then try to go to the next frontier on the list instead. This solution will not cause the robot to skip most of the valid frontier medians because for most frontier medians in accessible locations, the robot won't encounter a scenario where it exits wall following and immediately hits the wall unless there is a complex shaped obstacle. The robot will most likely only continuously run into walls when the destinations in inaccessible locations as it is inaccessible.

3.2.4 Finite State Machine

The low level controller for our robot is a finite state machine as shown in Figure 3. In code, the controller is written as a C++ class object with various public and private members. All of the ROS topic callbacks, algorithm functions, and tuning parameters are private. Several high-level, essential methods are public including start, stop, reset, and update which returns velocity commands based on the current state. The velocity updates are then published at a rate of 20Hz allowing the robot to react quickly to sensory input.

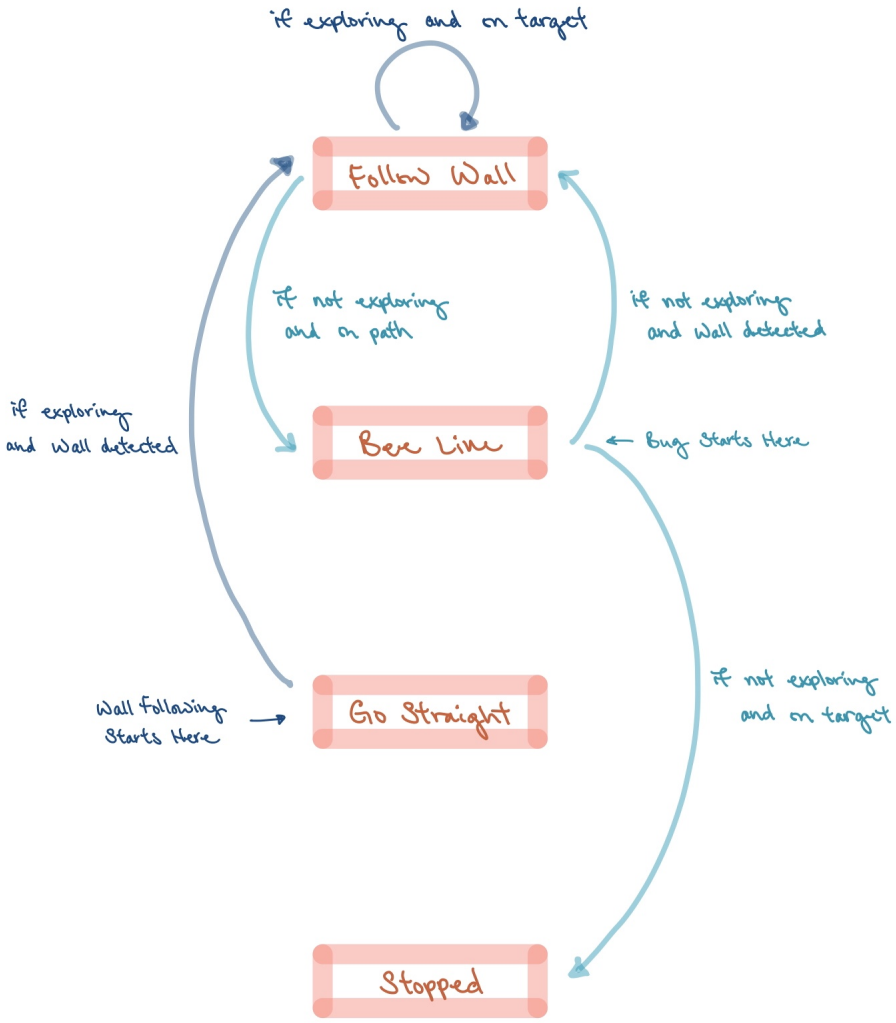

Figure 3: Finite State Machine

The robot starts off in the exploration phase goes straight until it detects a wall. It then performs wall following until a certain amount of time has passed at which point it exits the exploration phase and finds a set of destinations use WFD frontier search. A destination point is chosen and a straight line path is set from the current location to the destinations. While it is on this path, the robot stays in the Bee Line state as it makes a Bee line towards the destination. If it encounters a wall, it enters the Follow Wall state and uses the wall following algorithm to circle the obstacle until it is back on the path, at which point it goes back to the Bee Line and heads for the destination once more. This continues until the robot travels to all the target destinations or until the time limit of 8 minutes is reached, at which point the robot enters the Stopped state and stops.

4 Results

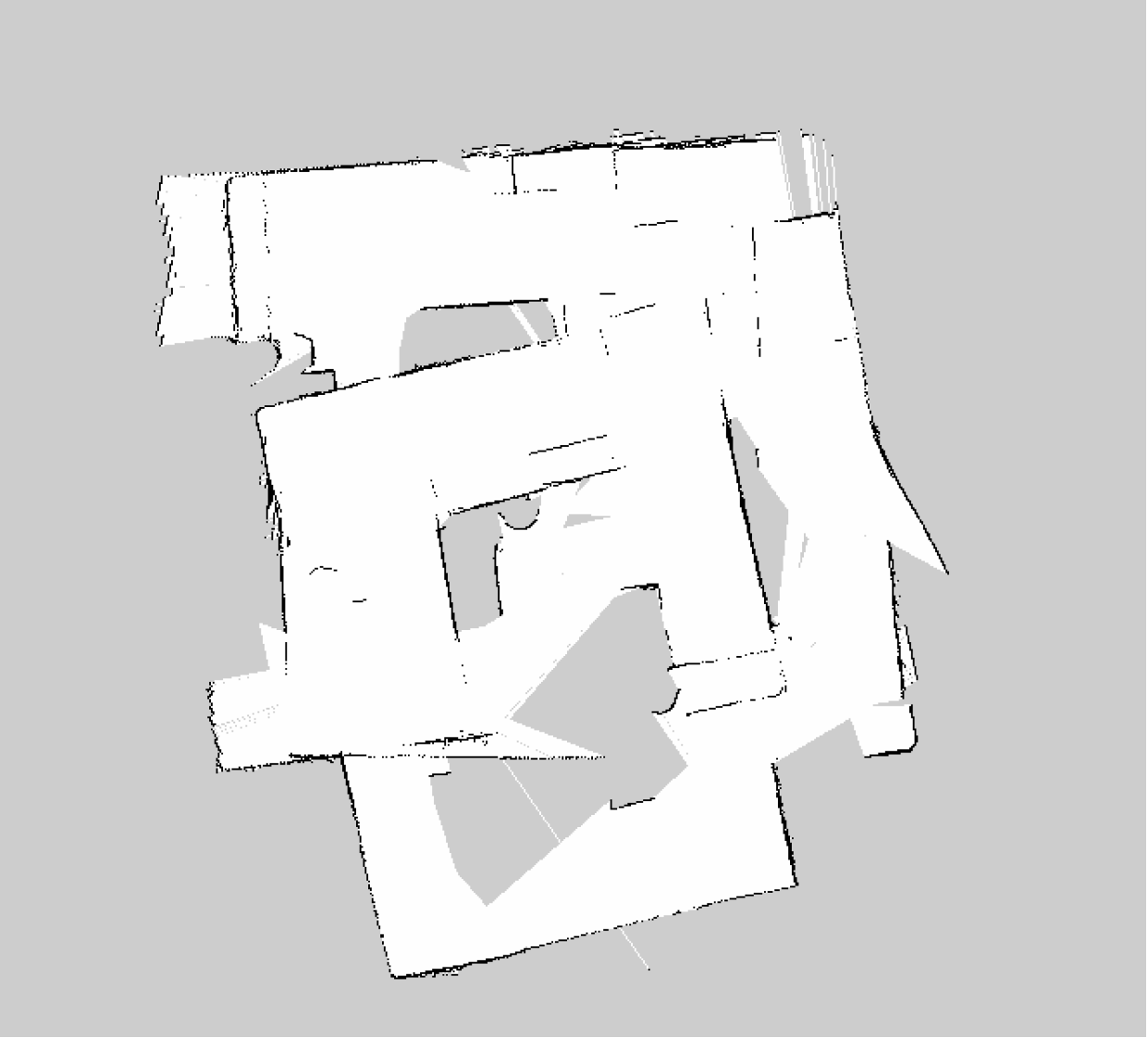

Figure 4: Map of official maze: first attempt

The robot was able to traversed the entire environment completely and successfully. The robot, using the wall following algorithm, was able to follow the outer wall and traverse the entire exterior. Once the switch between the left and right wall following happened, the robot was able to traverse the interior area. Additionally, frontier search and Bug2 performed well, navigating around the obstacles to frontiers and spinning to detect the areas surrounding the frontier points. However the mapping was not very precise and reflective of the environment. The issue seems to be that the mapping algorithm does not successfully close the loop, resulting in the the same wall showin.

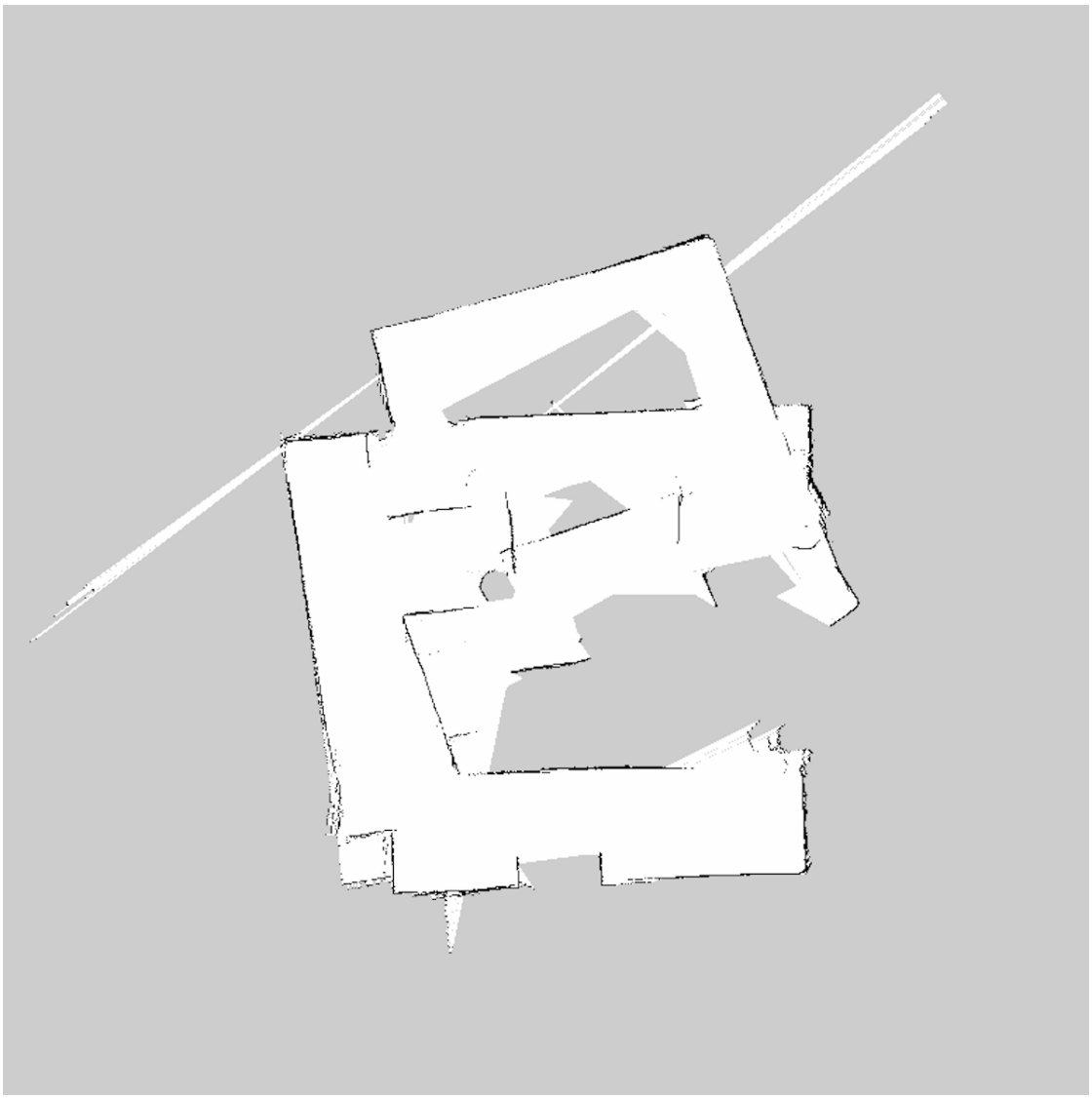

Figure 5: Map of official maze: second attempt

The wall following successfully navigated the outer part of the maze by following the perimeter wall. The robot failed to map the interior area of the maze. The frontier search ran into an issue where it kept running into wall. Since this occurred near the end of the run, the buffer timeout was not enabled.

5 Future Recommendations

Due to time constraints and sensory limitations, the algorithm provided here is limited in its efficiency and reliability. A number of improvements can be made given enough time and better sensory reliability. The first is to have a more efficient path planning algorithm that utilizes a generated cost map from the occupancy grid and an algorithm

similar to a potential field path planning algorithm which will ensure the robot maintains maximum distance away from all obstacles during it's motion. This will ensure the robot does not take unnecessarily long paths and also maximize the robustness as the robot will try to be as far away from all obstacles as possible so small errors in obstacle position estimates will not cause the robot to collide with them. But this requires a relatively accurate estimate of the robot's position which we currently do not have.

Related to sensor reliability, the robot sensors can be calibrated to achieve better accuracy. The wheels of the robot are not exactly the same diameter and sensor readings are not perfectly symmetric. These should be calibrated to make the odometry and laser scan estimations more accurate. We can then modify the Gmapping launch file to rely more heavily on odometry. The result will be a more accurate generated map.

Another improvement to improve wall following is to have the robot be able to detect how far away from the wall it is at all times. This can be done either by having another set of sensors that points to either sides of the robot, or with a more lax time constraint, we can have the robot periodically turn towards the wall that it is following to measure the distance between it and the wall. Either way, this will allow for a more accuracy wall following algorithm and reduce the issue the robot has with following the edges of thin obstacles beside the robot that the laser cannot see.

A forth, simple improvement is to have the robot periodically spin around in place before continuing it's motion. We found during test that due to the narrow field of view of the laser sensor, the robot has trouble mapping the center area of the environment if it ends up only following the bounding wall of the environment. If, however, the robot spun periodically, then this issue would be resolved. While exploring, the robot, when switching between left and right wall following, take a few seconds to go in a straight line in an arbitrary direction away from the current wall. This ensures more randomness, and thus more likely for the robot to perform wall following on all the obstacles even if all the obstacles are far away from each other.

Lastly, if we had a more relaxed time constraint, we would decrease the speed at which the robot moves at, implement buffer states to make the transition between moving forward and turning more smoothly, and also increase the resolution of the generated map. All three of these would help generate a more smooth and accurate map, but all three will cause the robot to require more time to do the mapping. Moving more slowly would of course increase the amount of time needed to traverse the whole environment. Adding buffer states would make transitions between moving forward and turning more smoothly, but also cause the transitions to be slower. This will add up over time and contribute to a significant amount of time increase when traversing through the environment. Lastly, by increasing the resolution of the generated map, we found the map to be more accurate, but also found our frontier search algorithm takes much longer to computer the frontier medians.This makes sense as the WFD Frontier Search algorithm

performs BFS over the whole explored region of the map using the occupancy grid. By increasing the resolution, the number of grid points also increases, which would result in longer time required for the BFS.

6 Appendix

6.1 Appendix A:Wavefront Frontier Detector Pseudocode

```
Require: queue_m // queue, used for detecting frontier points from
    a given map
Require: queue_f // queue, used for extracting a frontier from a
    given frontier cell
Require: pose // current global position of the robot

 1: queue_m ← ∅
 2: ENQUEUE(queue_m, pose)
 3: mark pose as "Map-Open-List"

 4: while queue_m is not empty do
 5:    p ← DEQUEUE(queue_m)

 6:    if p is marked as "Map-Close-List" then
 7:       continue
 8:    if p is a frontier point then
 9:       queue_f ← ∅
10:       NewFrontier ← ∅
11:       ENQUEUE(queue_f, p)
12:       mark p as "Frontier-Open-List"

13:       while queue_f is not empty do
14:          q ← DEQUEUE(queue_f)
15:          if q is marked as {"Map-Close-List","Frontier-Close-
             List"} then
16:             continue
17:          if q is a frontier point then
18:             add q to NewFrontier
19:             for all w ∈ adj(q) do
20:                if w not marked as {"Frontier-Open-
                   List","Frontier-Close-List", "Map-Close-List"}
                   then
21:                   ENQUEUE(queue_f,w)
22:                   mark w as "Frontier-Open-List"
23:          mark q as "Frontier-Close-List"
24:       save data of NewFrontier
25:       mark all points of NewFrontier as "Map-Close-List"
26:    for all v ∈ adj(p) do
27:       if v not marked as {"Map-Open-List","Map-Close-List"}
          and v has at least one "Map-Open-Space" neighbor then
28:          ENQUEUE(queue_m,v)
29:          mark v as "Map-Open-List"
30:    mark p as "Map-Close-List"
```

Figure 6: Pseudo-code for WFD algorithm from [1]

6.2 Participation

Hui Di Wang: She took up the responsibility of developing, testing and tuning the Wall Following algorithm. This includes designing and implementing the structure of the algorithm, characterising and obtaining the relevant parameters of the different sensors, and testing and tuning the threshold values of the algorithm to control the robot in such a way that the robot will be able to explore the unknown environment efficiently and

with minimal collisions. She also helped greatly in the integration of the wall following algorithm with the Bug2 algorithm, and writing and editing the report. She is an integral part of the team, and without her contributions the robot would not be able to navigate the environment.

Si Xu: He took up the responsibility of designing a suitable algorithm to find locations that have not been explored. This includes figuring out how to use relevant packages, finding reference to and implementing WFD Frontier Search algorithm, and setting up a testing environment to test the algorithm. He also helped greatly in the integration of the WFD Frontier Search algorithm with the Bug2 algorithm, and writing and editing the report. He is an integral part of the team, and without his contributions, the robot would not be able to locate frontiers of unexplored regions to complete the mapping of the environment.

Hunter Richards: He took up the responsibility of integrating all the different parts into a coherent, readable package, and assisted in the development of the motion planning algorithm. This includes helping implement and test the Bug2 algorithm, and taking charge in integration of the three major algorithms together. He spent considerable effort setting up a simulation environment and tuning Gmapping launch file parameters with the goal to test, debug, and improve our overall project. He also helped greatly in writing and editing the report. He is an integral part of the team, and without his contributions, the robot would not be able to utilize the different algorithms together to generate a complete map of the environment.

Yu-Chien Chen: She took up the responsibility of developing and implementing the motion planner Bug2 algorithm, and assisted in the overall integration of the 3 algorithms. This included implementing and testing different motion planning algorithms and finding the most suitable one for our robot, testing, tuning, and improving the algorithm, and also aiding to ensure the different algorithms are interfaced appropriately without any potential issues. She also helped greatly in writing and editing the report. She is an integral part of the team, and without her contributions, the robot would not be able to navigate to the generated frontiers to complete the mapping of the environment.

Xu Bo Yang Huang: He took up the responsibility of drafting up sections of the final report, and assisted in the implementation and improvement of the WFD Frontier Search algorithm. This includes designing, implementing, and improving the heuristic used to sort the medians, and certain sub-functions and parameters used within them. He also helped greatly in the testing and integration of the algorithms. He is an integral part of the team, and without his contributions, the final report would not be in the state that it is right now.

6.3 C++ ROS Code

Contest1.cpp

```cpp
#include <cstdio>
#include <chrono>
#include <cmath>
#include <ros/ros.h>
#include <ros/console.h>
#include <nav_msgs/Odometry.h>
#include <nav_msgs/OccupancyGrid.h>
#include <geometry_msgs/Twist.h>
#include <sensor_msgs/LaserScan.h>
#include <kobuki_msgs/BumperEvent.h>
#include <tf/transform_listener.h>
#include <tf/transform_datatypes.h>

#include "wall_follow.h"
#include "frontier_search.h"

#include "globals.h"

/*DISABLE RVIZ MARKERS */
//#define DISABLE_VISUALIZATIONS
/*
#ifndef DISABLE_VISUALIZATIONS
#include "make_marker.h"
#endif*/

/* Occupancy grid callback globals */
int occ_width = 0;  //Occupancy grid meta data
int occ_height = 0;
std::vector<std::vector<int>> occ_grid;  //Occupancy grid map
float pose_pos [3] = {-1, -1, -1}; //xyz
float pose_origin [3] = {-1, -1, -1}; //xyz
float pose_orientation [4] = {-1, -1, -1, -1}; //Quaternion xyzw
float res;

template <typename T> int sgn(T val)
{
        return (T(0) < val) - (val < T(0));
}

```

```
40  class Controller
41  {
42          enum class Direction {
43                  RIGHT,
44                  LEFT,
45          };
46
47          enum class State {
48                  NONE,
49                  FOLLOW_WALL,
50                  BEE_LINE,
51                  GO_STRAIGHT,
52                  STOPPED,
53          };
54
55          struct Point2D {
56                  double x;
57                  double y;
58          };
59
60          struct Pose2D {
61                  double x;
62                  double y;
63                  double theta;
64          };
65
66          struct Twist2D {
67                  double linear;
68                  double angular;
69          };
70
71          struct Line2D{
72                  double m;
73                  double b;
74          };
75
```

```
76  public:
77          Controller(ros::NodeHandle &nh);
78
79          void start(pair<double,double> point);
80          void start(void);
81          void reset(void);
82
83          geometry_msgs::Twist update(void);
84
85          bool stopped() { return state == State::STOPPED; }
86
87  private:
88          const double tol = 0.25;
89          const double vlmax = 0.25;
90          const double vamax = 0.4;
91
92          int state_timer;
93          bool exploring;
94          Controller::State state;
95          Controller::Direction dir;
96
97          std::vector<double> lasers;
98
99          Controller::Point2D dest;
100         Controller::Twist2D vel;
101         Controller::Pose2D pose;
102         Controller::Line2D path;
103
104         ros::Subscriber sub_odom;
105         ros::Subscriber sub_laser;
106         ros::Subscriber sub_bump;
107         ros::Subscriber sub_map;
108
109         void odomCallback(const nav_msgs::Odometry::ConstPtr &msg);
110         void laserCallback(const sensor_msgs::LaserScan::ConstPtr &msg);
111         void bumperCallback(const kobuki_msgs::BumperEvent::ConstPtr &msg);
112         void mapCallback(const nav_msgs::OccupancyGrid::ConstPtr &msg);
```

```
114         bool bumperDetected();
115         bool wallDetected();
116         bool onPath();
117         bool onTarget();
118         geometry_msgs::Twist gotoPoint(void);
119         geometry_msgs::Twist followWall(void);
120 };
121
122 Controller::Controller(ros::NodeHandle &nh)
123 {
124         using nav_msgs::Odometry;
125         using sensor_msgs::LaserScan;
126         using kobuki_msgs::BumperEvent;
127         using nav_msgs::OccupancyGrid;
128
129         reset();
130
131         sub_odom = nh.subscribe<Odometry>("/odom", 10,
132                 &Controller::odomCallback, this);
133
134         sub_laser = nh.subscribe<LaserScan>("/scan", 10,
135                 &Controller::laserCallback, this);
136
137         sub_bump = nh.subscribe<BumperEvent>("/mobile_base/events/bumper", 10,
138                 &Controller::bumperCallback, this);
139
140         sub_map = nh.subscribe<OccupancyGrid>("/map", 10,
141                 &Controller::mapCallback, this);
142 }
143
144 void Controller::odomCallback(const nav_msgs::Odometry::ConstPtr &msg)
145 {
146         pose.x = msg->pose.pose.position.x;
147         pose.y = msg->pose.pose.position.y;
148         pose.theta = tf::getYaw(msg->pose.pose.orientation);
149 }
```

```
151 void Controller::laserCallback(const sensor_msgs::LaserScan::ConstPtr &msg)
152 {
153         _laserCallback(msg);
154         lasers = _lasers;
155 }
156
157 void Controller::bumperCallback(const kobuki_msgs::BumperEvent::ConstPtr &msg)
158 {
159         _bumperCallback(msg);
160 }
161
162 void Controller::mapCallback(const nav_msgs::OccupancyGrid::ConstPtr &msg)
163 {
164         res = msg->info.resolution;
165         occ_height = msg->info.height;
166         occ_width = msg->info.width;
167
168         pose_origin[0] = msg->info.origin.position.x;
169         pose_origin[1] = msg->info.origin.position.y;
170         pose_origin[2] = msg->info.origin.position.z;
171
172         pose_orientation[0] = msg->info.origin.orientation.x;
173         pose_orientation[1] = msg->info.origin.orientation.y;
174         pose_orientation[2] = msg->info.origin.orientation.z;
175         pose_orientation[3] = msg->info.origin.orientation.w;
176
177         /* y, x form (y rows of x length).
178          */
179         occ_grid = vector<vector<int>>(occ_height, vector<int>(occ_width, -1));
180
181         for (int i = 0; i < occ_width*occ_height; i++) {
182                 int prob = msg->data[i];
183                 occ_grid[i/occ_width][i%occ_width] = msg->data[i];
184         }
185 }
186
```

```
187 void Controller::start(pair<double,double> point)
188 {
189         state = State::BEE_LINE;
190
191         dest.x = point.first;
192         dest.y = point.second;
193
194         path.m = (dest.y-pose.y) / (dest.x-pose.x);
195         path.b = pose.y - path.m*pose.x;
196 }
197
198 void Controller::start()
199 {
200         exploring = true;
201         state = State::GO_STRAIGHT;
202 }
203
204 void Controller::reset()
205 {
206         state_timer = 0;
207         left_unseen_count = 0;
208         right_unseen_count = 0;
209         exploring = false;
210         state = State::STOPPED;
211         dir = Direction::LEFT;
212 }
213
214 bool Controller::bumperDetected()
215 {
216         using kobuki_msgs::BumperEvent;
217
218         double min_dist;
219         double center_dist;
220
221         for (int i = 0; i < NUM_BUMPER; i++) {
222                 if (bumper[i] == BumperEvent::PRESSED)
223                         return true;
224         }
225 }
```

```
227 bool Controller::wallDetected()
228 {
229         using kobuki_msgs::BumperEvent;
230
231         double min_dist;
232         double center_dist;
233
234         for (int i = 0; i < NUM_BUMPER; i++) {
235                 if (bumper[i] == BumperEvent::PRESSED)
236                         return true;
237         }
238
239         if (lasers.empty())
240                 return false;
241
242         min_dist = *std::min_element(lasers.begin(), lasers.end());
243
244         center_dist = *std::min_element(lasers.begin()+300, lasers.begin()+400);
245
246         if ((min_dist > 50) || (min_dist < 0.5) || (center_dist > 50) ||
            (center_dist < 0.5))
247                 return true;
248         else
249                 return false;
250 }
251
252 bool Controller::onPath()
253 {
254         double y = path.m*pose.x + path.b;
255
256         return (fabs(pose.y-y) < tol);
257 }
258
259 bool Controller::onTarget()
260 {
261         double d = hypot(dest.x-pose.x, dest.y-pose.y);
262
263         return (fabs(d) < tol);
264 }
```

```
geometry_msgs::Twist Controller::gotoPoint()
{
        double dx, dy, da;
        geometry_msgs::Twist ret;

        dx = dest.x - pose.x;
        dy = dest.y - pose.y;
        da = remainder(atan2(dy, dx)-pose.theta, 2*M_PI);

        if (fabs(da) > tol) {
                vel.linear = 0;
                vel.angular = sgn(da)*vamax;
        }
        else {
                vel.linear = vlmax;
                vel.angular = 0;
        }

        ret.linear.x = vel.linear;
        ret.angular.z = vel.angular;

        return ret;
}

geometry_msgs::Twist Controller::followWall()
{
        return _followWall((int) dir);
}

geometry_msgs::Twist Controller::update()
{
        geometry_msgs::Twist ret;

         if (bumperDetected() && state != State::FOLLOW_WALL && state !=
             State::GO_STRAIGHT)
        {
```

```
301                 ROS_INFO("BUMP");
302                 state = State::FOLLOW_WALL;
303                 state_timer = 50;
304                 ret.linear.x = 0;
305                 ret.angular.z = 0;
306                 return ret;
307         }
308         /* Decide velocity control signal from state.
309          */
310         switch (state)
311         {
312         case State::FOLLOW_WALL:
313                 ROS_INFO("wall follow");
314                 ret = followWall();
315                 break;
316         case State::BEE_LINE:
317                 ROS_INFO("beeline");
318                 ret = gotoPoint();
319                 break;
320         case State::GO_STRAIGHT:
321                 ROS_INFO("go straight");
322                 ret.linear.x = vlmax;
323                 ret.angular.z = 0;
324                 break;
325         default:
326                 ret.linear.x = 0;
327                 ret.angular.z = 0;
328                 break;
329         };
330
331         /* Do not proceed if the state_timer is set.
332          */
333
334         if (state_timer > 0) {
335                 state_timer--;
336                 return ret;
337         }
338
```

```
339         /* Perform a state change if required.
340          */
341         switch (state)
342         {
343         case State::FOLLOW_WALL:
344 
345                 if (exploring && onTarget()) {
346                         if (dir == Direction::LEFT)
347                                 dir = Direction::RIGHT;
348                         else
349                                 dir = Direction::LEFT;
350                         ROS_INFO("Position: (%.3f, %.3f)", pose.x, pose.y);
351                         state_timer = 400;
352                 }
353 
354                 if (!exploring && onPath()) {
355                         if (dir == Direction::LEFT)
356                                 dir = Direction::RIGHT;
357                         else
358                                 dir = Direction::LEFT;
359                         left_unseen_count  = 0;
360                         right_unseen_count = 0;
361                         state = State::BEE_LINE;
362                         state_timer = 40;
363                 }
364                 break;
365         case State::BEE_LINE:
366 
367                 if (!exploring && wallDetected()) {
368                         state = State::FOLLOW_WALL;
369                         state_timer = 400;
370                 }
371 
372                 if (!exploring && onTarget()) {
373                         state = State::STOPPED;
374                         state_timer = 400;
375                 }
376                 break;
377         case State::GO_STRAIGHT:
```

```
                if (exploring && wallDetected()) {
                        dest.x = pose.x;
                        dest.y = pose.y;
                        state = State::FOLLOW_WALL;
                        state_timer = 400;
                }
                break;
        default:
                break;
        };

        return ret;
}

std::vector<pair<double,double>> frontier_medians(tf::TransformListener
    &tf_listener)
{
        /*Returns a vector of pairs of x,y coordinates as destination targets */
        std::vector<pair<double,double>> list_of_medians = wfd(tf_listener);

        /*
        //Debugging code to plot the medians on occ_grid
        std::cout << "Obtained list of medians: There were " <<
            list_of_medians.size() << std::endl;
        for(int i = 0; i<list_of_medians.size(); i++){
                std::cout << "x/y = (" << list_of_medians[i].first << ", " <<
                    list_of_medians[i].second << endl;
                occ_grid[list_of_medians[i].first][list_of_medians[i].second] = 50;
        } */

        return list_of_medians;
}

```

```
int main(int argc, char **argv)
{
        using namespace std::chrono;
        using geometry_msgs::Twist;

        int timer = 0;
        int timer2 = 0;
        time_point<system_clock> start;
        time_point<system_clock> start2;
        time_point<system_clock> now;

        int index = 0;
        int max_index = 0;
        pair<double,double> dest;
        std::vector<pair<double,double>> list_of_frontiers;

        /* Initialize ros environment.
         */
        ros::init(argc, argv, "terrorizing_turtle");

        ros::NodeHandle nh;
        ros::Publisher pub;

        Controller controller(nh);
        tf::TransformListener tf_listener(nh);

        pub = nh.advertise<Twist>("cmd_vel_mux/input/teleop", 1);

        /* For visualization, disable during real run */
/*
#ifndef DISABLE_VISUALIZATION
        ros::Publisher marker_pub =
            n.advertise<visualization_msgs::Marker>("visualization_marker", 1);
#endif*/

        /* Start the clock, 8 minutes.
         */
        start = system_clock::now();
        ros::Rate rate(20);
```

```
448         /* Start by wall following an obstacle in front for 3 minutes.
449          */
450         controller.start();
451 
452         while (ros::ok() && (timer < 300)) {
453                 ros::spinOnce();
454                 pub.publish(controller.update());
455 
456                 rate.sleep();
457 
458                 now = system_clock::now();
459                 timer = duration_cast<seconds>(now-start).count();
460 
461                 // ROS_INFO("Time: %d", timer);
462         }
463 
464         controller.reset();
465 
466 /*
467 #ifndef DISABLE_VISUALIZATION
468                 generate_markers(marker_pub, list_of_frontiers);
469 #endif*/
470 
471         /* Use frontier search to finish unexplored areas of the map.
472          * Get list of frontiers from frontier search.
473          */
474         while (ros::ok() && (timer < 480)) {
475                 ros::spinOnce();
476                 try{
477                         if (index == max_index) {
478                                 list_of_frontiers = frontier_medians(tf_listener);
479                                 index = 0;
480                                 max_index = list_of_frontiers.size();
481                         }
482 
483                         if (controller.stopped() || (timer2 > 60)) {
484                                 dest = list_of_frontiers.at(index++);
485                                 controller.start(dest);
486                                 start2 = system_clock::now();
```

```
                        }

                        pub.publish(controller.update());
                } catch(...){
                        rate.sleep();
                }
                rate.sleep();

                now = system_clock::now();
                timer = duration_cast<seconds>(now-start).count();
                timer2 = duration_cast<seconds>(now-start2).count();

                ROS_INFO("Time: %d", timer);
        }

        return 0;
}

```

Wall_follow.h

```
#ifndef WALL_FOLLOW_H
#define WALL_FOLLOW_H

#include <cstdio>
#include <geometry_msgs/Twist.h>
#include <sensor_msgs/LaserScan.h>
#include <kobuki_msgs/BumperEvent.h>

#define NUM_BUMPER    3

extern int bumper[];
extern std::vector<double> _lasers;
extern int left_unseen_count;
extern int right_unseen_count;

void _laserCallback(const sensor_msgs::LaserScan::ConstPtr &msg);
void _bumperCallback(const kobuki_msgs::BumperEvent::ConstPtr &msg);

geometry_msgs::Twist _followWall(int direction);

#endif
```

Wall_follow.cpp

```
50 static double get_min_laser_dist(int min_laser_idx, int max_laser_idx)
51 {
52         return *std::min_element(_lasers.begin() + min_laser_idx,
53                                  _lasers.begin() + max_laser_idx);
54 }
55
56 geometry_msgs::Twist _followWall(int direction)
57 {
58         using kobuki_msgs::BumperEvent;
59
60         double linear;
61         double angular;
62
63         bool any_bumper_pressed = false;
64
65         bool left_bumper_pressed  = (bumper[0] == BumperEvent::PRESSED);
66         bool front_bumper_pressed = (bumper[1] == BumperEvent::PRESSED);
67         bool right_bumper_pressed = (bumper[2] == BumperEvent::PRESSED);
68
69         // ROS_INFO("Bumpers: %d, %d, %d", left_bumper_pressed,
70         //         front_bumper_pressed, right_bumper_pressed);
71
72         for (int i = 0; i < NUM_BUMPER; i++)
73                 any_bumper_pressed |= (bumper[i] == BumperEvent::PRESSED);
74
75         if (any_bumper_pressed)
76         {
77                 ROS_INFO("Backing up!");
78
79                 backup_count = 25;
80
81                 if (front_bumper_pressed)
82                 {
83                         linear = -0.1;
84                         angular = 0;
85                 }
86                 else if (left_bumper_pressed)
87                 {
88                         linear = -0.1;
89                         angular = -0.3;
90                 }
91                 else if (right_bumper_pressed)
92                 {
93                         linear = -0.1;
94                         angular = 0.3;
95                 }
96         }
```

```cpp
#include <cstdio>
#include <cmath>
#include <ros/ros.h>
#include <ros/console.h>
#include <geometry_msgs/Twist.h>
#include <sensor_msgs/LaserScan.h>
#include <kobuki_msgs/BumperEvent.h>

#include "wall_follow.h"

int bumper[] = {
        kobuki_msgs::BumperEvent::RELEASED,
        kobuki_msgs::BumperEvent::RELEASED,
        kobuki_msgs::BumperEvent::RELEASED
};

int nLasers;
std::vector<double> _lasers;

double left_laser_dist;
double front_laser_dist;
double right_laser_dist;

int backup_count = 0;
int left_unseen_count = 0;
int right_unseen_count = 0;

geometry_msgs::Twist vel;

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

void _laserCallback(const sensor_msgs::LaserScan::ConstPtr &msg)
{
        _lasers.clear();

        nLasers = (msg->angle_max - msg->angle_min) / msg->angle_increment;

        for (int i = 0; i < nLasers; i++)
        {
                if (std::isnan(msg->ranges[i]))
                        _lasers.push_back(100);
                else
                        _lasers.push_back(msg->ranges[i]);
        }
}
```

```
97          else if (backup_count > 0)
98          {
99                  backup_count--;
100
101                 linear = vel.linear.x;
102                 angular = vel.angular.z;
103         }
104         else if (!_lasers.empty())
105         {
106                 left_laser_dist  = get_min_laser_dist(nLasers-10, nLasers-1);
107                 front_laser_dist = get_min_laser_dist(nLasers/2-150,
108                                                       nLasers/2+150);
109
110                 if (direction == 0)
111                         right_laser_dist = get_min_laser_dist(0, 10);
112                 else
113                         right_laser_dist = get_min_laser_dist(0, 30);
114
115                 // ROS_INFO("Lasers: %f, %f, %f", left_laser_dist,
116                 //         front_laser_dist, right_laser_dist);
117
118                 linear = 0.25;
119                 angular = 0;
120
121                 if ((front_laser_dist < 0.5)
122                         || (left_laser_dist < 0.5)
123                         || (right_laser_dist < 0.5)
124                         || (direction == 1 && right_laser_dist > 50)
125                         || (direction == 0 && left_laser_dist > 50)
126                         ||(front_laser_dist >50)
127                         ||(left_laser_dist > 50 && right_laser_dist > 50))
128                 {
129                         linear = 0;
130                         left_unseen_count = 0;
131                         right_unseen_count = 0;
132
133                         if (direction == 0)
134                                 angular = -0.3;
135                         else
136                                 angular = 0.3;
137                 }
138
139                 else if ((left_laser_dist > 1)
140                         && (left_laser_dist < 50)
141                         && (direction == 0))
142                 {
```

```
143                         linear = 0.2;
144                         angular = 0.2;
145                         left_unseen_count++;
146
147                         if ((left_unseen_count > 50)
148                                 && (left_unseen_count < 1000))
149                         {
150                                 ROS_INFO("Sharp left!");
151
152                                 linear = 0.05;
153                                 angular = 0.3;
154                         }
155                 }
156                 else if ((right_laser_dist > 1)
157                         && (right_laser_dist < 50)
158                         && (direction == 1))
159                 {
160                         linear = 0.2;
161                         angular = -0.2;
162                         right_unseen_count++;
163
164                         if ((right_unseen_count > 60)
165                                 && (right_unseen_count < 1000))
166                         {
167                                 ROS_INFO("Sharp right!");
168
169                                 linear = 0.01;
170                                 angular = -0.3;
171                         }
172                 }
173         }
174         else
175         {
176                 linear = 0;
177                 angular = 0;
178         }
179
180         vel.linear.x = linear;
181         vel.angular.z = angular;
182
183         return vel;
184 }
185
```

globals.h

```
/* This file lists globals for use in other files
   Define your globals in contest1.cpp and include this where needed */
#ifndef GLOBALS_H
#define GLOBALS_H

// Occupancy grid globals for storing callback data
extern std::vector<std::vector<int>> occ_grid;
extern int occ_width;
extern int occ_height;
extern float res;
extern float pose_pos[3];
extern float pose_orientation[4];
extern float pose_origin [3];
//extern ros::NodeHandle nh;

#endif

```

frontier_search.h

```
#ifndef FRONTIER_SEARCH_H
#define FRONTIER_SEARCH_H

using namespace std;

#include <ros/console.h>
#include "ros/ros.h"
#include <geometry_msgs/Twist.h>
#include <kobuki_msgs/BumperEvent.h>
#include <sensor_msgs/LaserScan.h>

#include <nav_msgs/Odometry.h>
#include <tf/transform_datatypes.h>
#include <tf/transform_listener.h>
#include <nav_msgs/OccupancyGrid.h>
#include <nav_msgs/MapMetaData.h>
#include <geometry_msgs/PointStamped.h>
#include <time.h>

#include <stdio.h>
#include <cmath>

#include <chrono>

#include <vector>
#include <queue>

//Constants for frontier search labels
enum Marker {MAP_OPEN_LIST=0, MAP_CLOSE_LIST, FRONTIER_OPEN_LIST, FRONTIER_CLOSE_LIST};

bool is_a_frontier_point(const pair<int,int> &p);
vector<pair<int,int>> get_medians(vector<vector<pair<int,int>>> list_of_frontiers);
bool has_open_neighbour(const vector<vector<int>> &marker_list, const pair<int,int> &p);
vector<pair<int,int>> findNeighbours(const pair<int,int> &p);
vector<pair<double,double>> wfd(tf::TransformListener &tf_listener);

#endif
```

frontier_search.cpp

```cpp
#include "frontier_search.h"
#include "globals.h"

const int MEDIAN_SIZE_LIMITER_upper = 50;
const int MEDIAN_SIZE_LIMITER_lower = 3;

const bool print_times = false;
pair<int, int> map_grid;


bool is_a_frontier_point(const pair<int,int> &p){
    if(occ_grid[p.first][p.second] == -1){
        return false;
    }
    vector<pair<int,int>> neighbours = findNeighbours(p);
    if(occ_grid[p.first][p.second] <= 10){
        for(auto nei: neighbours){
            if(occ_grid[nei.first][nei.second] == -1){
                return true;
            }
        }
    }
    return false;


}
```

```
vector<pair<int,int>> get_medians(vector<vector<pair<int,int>>> list_of_frontiers){
    //pick out the median frontier point from each set
    vector<pair<int, int>> destinations;
    vector<pair<pair<int, int>, double>> des_sort;
    int split;
    //vector<double> priority;
    int median_position;
    double heu_val;

    for (auto cur_front: list_of_frontiers){
        std::cout << "Size of this median was: " << cur_front.size() << std::endl;
        if(cur_front.size() < MEDIAN_SIZE_LIMITER_lower){
            continue;
        }
        split = cur_front.size()/MEDIAN_SIZE_LIMITER_upper + 1;

        sort(cur_front.begin(), cur_front.end());
        for (int i = 1; i <= split; i++){

            median_position = ((cur_front.size()/split)*i)/2;


            //double size_exp = cur_front.size()/(double)split;
            //double dist_x = (double)(cur_front[median_position].first - map_grid.first);
            //double dist_y = (double)(cur_front[median_position].second - map_grid.second);
            //heu_val = exp(size_exp) - exp(sqrt(pow(dist_x, 2) + pow(dist_y, 2)));
            heu_val = exp(cur_front.size()/(double)split) -
                sqrt(pow(res*(double)(cur_front[median_position].first - map_grid.first), 2) +
                pow(res*(double)(cur_front[median_position].second - map_grid.second), 2));
            des_sort.push_back(make_pair(cur_front[median_position], heu_val));
            //priority.push_back()
        }
        //destinations.push_back(cur_front[cur_front.size()/2]);
    }

    sort(des_sort.begin(), des_sort.end(), [](pair<pair<int, int>, double> a, pair<pair<int, int>, double> b) {
        return a.second > b.second; });

    for (auto des: des_sort){
        destinations.push_back(des.first);
    }


    return destinations;
}
```

```
bool has_open_neighbour(const vector<vector<int>> &marker_list, const pair<int,int> &p){
    //this point has a neighbour that is map_open_list

    vector<pair<int,int>> neighbours = findNeighbours(p);
    for(pair<int,int> point: neighbours){
        if(marker_list[point.first][point.second]==MAP_OPEN_LIST)
        {
            return true;
        }

    }
    return false;
}
vector<pair<int,int>> findNeighbours(const pair<int,int> &p){
    //BFS find adj neighbours
    static vector<pair<int,int>> directions = {{-1,-1},{-1,0},{-1,1},{0,-1},{0,1},{1,-1},{1,0}, {1,1}};
    //static vector<pair<int,int>> directions = {{-1,0},{0,-1},{0,1},{1,0}};
    vector<pair<int,int>> neighbours;

    for(auto d: directions){
        pair<int, int> temp = {p.first + d.first, p.second + d.second};
        if(temp.first < 0 || temp.second < 0 ||
            temp.first >= occ_height || temp.second >= occ_width ||
            occ_grid[temp.first][temp.second] >=10)
        {
            continue;
        }
        neighbours.push_back(temp);
    }
    return neighbours;
}

vector<pair<double,double>> wfd(tf::TransformListener &listener)
{
    //Store all new frontiers here

    //Get initial point


    ROS_INFO("Start frontier search: Initializing transform");

    //std::cout << "Start timing" << endl;
    //auto start = std::chrono::system_clock::now();


    listener.waitForTransform("/map", "/odom", ros::Time::now(), ros::Duration(1.0));
    ros::Time cur_time = ros::Time::now();
    listener.waitForTransform("/map", "/odom", cur_time, ros::Duration(10.0));
```

```
    //auto end = std::chrono::system_clock::now();
    //auto elapsed =     std::chrono::duration_cast<std::chrono::seconds>(end - start);
    //std::cout << "Elapsed time" << elapsed.count() << '\n';

    //std::cout << "Start BFS timing" << endl;
    //start = std::chrono::system_clock::now();


    geometry_msgs::PointStamped cur_pose_stamped;
    cur_pose_stamped.header.frame_id = "/odom";
    cur_pose_stamped.header.stamp = cur_time;

    cur_pose_stamped.point.x = pose_pos[0];
    cur_pose_stamped.point.y = pose_pos[1];
    cur_pose_stamped.point.z = 0;

    //listener.waitForTransform("/map", "/odom", cur_time, ros::Duration(10.0));
    listener.transformPoint("/map", cur_pose_stamped, cur_pose_stamped);

    //ROS_INFO("%s", dest.header.frame_id.c_str());
    //ROS_INFO("CUR_POSE is (%.3f, %.3f, %.3f)", pose_pos[0], pose_pos[1], pose_pos[2]);

    map_grid.second = int(round((cur_pose_stamped.point.x - pose_origin[0])/res));
    map_grid.first = int(round((cur_pose_stamped.point.y - pose_origin[1])/res));

    //ROS_INFO("MAP_POSE is (%.3f, %.3f, %.3f)", cur_pose_stamped.point.x, cur_pose_stamped.point.y,
        cur_pose_stamped.point.z);
    //ROS_INFO("Grid_Point is (%.3f, %.3f)", map_grid.first, map_grid.second);

    //ROS_INFO("ORIGIN_POSE is (%.3f, %.3f, %.3f)", pose_origin[0], pose_origin[1], pose_origin[2]);

    vector<vector<pair<int, int>>> list_of_frontiers;
    std::cout<<"INITIALIZING WFD: OCC_GRID/HEIGHT" << occ_width <<"/" << occ_height << std::endl;

    //For keeping track of visited
    vector<vector<int>> marker_list(
        occ_height,
        vector<int>(occ_width, -1)
    ); //y,x form (y rows of x length)

    //BFS queue 1
    queue<pair<int, int>> queue_m;
    int value_at_initial = occ_grid[map_grid.first][map_grid.second];
    //std::cout << "Value at initial" << value_at_initial << std::endl;
    //std::cout << "x/y: " <<map_grid.first << "/" << map_grid.second << endl;
    //pair<int, int> pose(int(round(cur_pose_stamped.point.x)), int(round(cur_pose_stamped.point.y)));
    //get_current_pose();
    queue_m.push(map_grid);
    marker_list[map_grid.first][map_grid.second] = MAP_OPEN_LIST;
    //std::cout << "PUSHING MAP GRID" << std::endl;
```

```
    while(!queue_m.empty()){
        pair<int, int> p = queue_m.front();
        queue_m.pop();

        //std::cout<<"MARKER LIST" << occ_width <<"/" << occ_height << std::endl;

        if(marker_list[p.first][p.second]==MAP_CLOSE_LIST){
            continue;
        } else if(is_a_frontier_point(p)){
            queue<pair<int, int>> queue_f;
            vector<pair<int, int>> new_frontier;
            queue_f.push(p);
            marker_list[p.first][p.second] = FRONTIER_OPEN_LIST;

            while(!queue_f.empty()){
                pair<int, int> q = queue_f.front();
                queue_f.pop();
                if(marker_list[q.first][q.second] == MAP_CLOSE_LIST ||
                    marker_list[q.first][q.second] == FRONTIER_CLOSE_LIST){
                    continue;
                }
                else if(is_a_frontier_point(q)){
                    new_frontier.push_back(q);
                    for(auto adj : findNeighbours(q)){
                        if( marker_list[adj.first][adj.second] == FRONTIER_OPEN_LIST ||
                            marker_list[adj.first][adj.second] == FRONTIER_CLOSE_LIST ||
                            marker_list[adj.first][adj.second] == MAP_CLOSE_LIST){
                            continue;
                        }
                        queue_f.push(adj);
                        marker_list[adj.first][adj.second] = FRONTIER_OPEN_LIST;
                    }
                    marker_list[q.first][q.second] = FRONTIER_CLOSE_LIST;
                }
            }
            list_of_frontiers.push_back(new_frontier); //Optimize this

            for(pair<int,int> mem : new_frontier){
                marker_list[mem.first][mem.second] = MAP_CLOSE_LIST;
            }
        }

        for(pair<int,int> v : findNeighbours(p)){
            if(marker_list[v.first][v.second]==MAP_CLOSE_LIST || marker_list[v.first][v.second]==MAP_OPEN_LIST)
            {
                continue;
            }
            if(has_open_neighbour(marker_list,v)){
                queue_m.push(v);
                marker_list[v.first][v.second] = MAP_OPEN_LIST;
            }
```

```
        }
        marker_list[p.first][p.second] = MAP_CLOSE_LIST;
    }

    vector<pair<int,int>> list_of_medians = get_medians(list_of_frontiers);

    vector<pair<double, double>> list_of_medians_odom;
    pair<double, double> median_odom;

    //end = std::chrono::system_clock::now();
    //elapsed =     std::chrono::duration_cast<std::chrono::seconds>(end - start);
    //std::cout << "Elapsed time for BFS: " << elapsed.count() << '\n';

    //start = std::chrono::system_clock::now();
    //listener.waitForTransform("/odom", "/map", cur_time, ros::Duration(10.0));

    /* Convert back to map frame */
    geometry_msgs::PointStamped cyc_pose_stamped;
    cyc_pose_stamped.header.frame_id = "/map";
    cyc_pose_stamped.header.stamp = cur_time;

    for(pair<int, int> median: list_of_medians){
        cyc_pose_stamped.point.x = median.second*res + pose_origin[0];
        cyc_pose_stamped.point.y = median.first*res + pose_origin[1];
        cyc_pose_stamped.point.z = 0;
        listener.transformPoint("/odom", cyc_pose_stamped, cyc_pose_stamped);
        median_odom.first = cyc_pose_stamped.point.x;
        median_odom.second = cyc_pose_stamped.point.y;
        list_of_medians_odom.push_back(median_odom);
    }
    /*
    end = std::chrono::system_clock::now();
    elapsed =     std::chrono::duration_cast<std::chrono::seconds>(end - start);
    std::cout << "Elapsed time to return medians: " << elapsed.count() << '\n';
    */
    return list_of_medians_odom;
}
```

make_marker.h

```h
#include <visualization_msgs/Marker.h>

void generateMarkers(ros::Publisher marker_pub,
    std::vector<std::pair<double, double>> list_of_medians);

```

make_marker.cpp

```cpp
#include <ros/ros.h>
#include "make_marker.h"
const int shape = visualization_msgs::Marker::CUBE;
//vector<pair<double,double>>
//ros::Publisher marker_pub = n.advertise<visualization_msgs::Marker>("visualization_marker", 1);
void generateMarkers(ros::Publisher marker_pub,
    std::vector<std::pair<double, double>> list_of_medians){
    visualization_msgs::Marker marker;
    marker.header.frame_id = "/map";
    marker.header.stamp = ros::Time::now();

    // Set the namespace and id for this marker.  This serves to create a unique ID
    // Any marker sent with the same namespace and id will overwrite the old one
    marker.ns = "basic_shapes";
    marker.id = 0;

    // Set the marker type.  Initially this is CUBE, and cycles between that and SPHERE, ARROW, and CYLINDER
    marker.type = shape;

    // Set the marker action.  Options are ADD, DELETE, and new in ROS Indigo: 3 (DELETEALL)
    marker.action = visualization_msgs::Marker::ADD;

    // Set the pose of the marker.  This is a full 6DOF pose relative to the frame/time specified in the header
    marker.pose.position.x = list_of_medians[0].first;
    marker.pose.position.y = list_of_medians[0].second;
    marker.pose.position.z = 0;
    marker.pose.orientation.x = 0.0;
    marker.pose.orientation.y = 0.0;
    marker.pose.orientation.z = 0.0;
    marker.pose.orientation.w = 1.0;

    // Set the scale of the marker -- 1x1x1 here means 1m on a side
    marker.scale.x = 0.1;
    marker.scale.y = 0.1;
    marker.scale.z = 0.1;

    // Set the color -- be sure to set alpha to something non-zero!
    marker.color.r = 0.0f;
    marker.color.g = 1.0f;
    marker.color.b = 0.0f;
    marker.color.a = 1.0;

    marker.lifetime = ros::Duration();

    // Publish the marker
    while (marker_pub.getNumSubscribers() < 1)
    {
      if (!ros::ok())
      {
        return;
      }
      ROS_WARN_ONCE("Please create a subscriber to the marker");
      sleep(1);
    }
    marker_pub.publish(marker);
}
```

References

[1] A. Topiwala, P. Inani, and A. Kathpal, "Frontier Based Exploration for Autonomous Robot," CoRR, 2018.

[2] LaValle, S. (2001). BUG Algorithms. [online] Msl.cs.uiuc.edu. Available at: http://msl.cs.uiuc.edu/ lavalle/cs497_2001/book/uncertain/node3.html.