MIE443 Mechatronics Systems: Design and Integration

Mechanical & Industrial Engineering
UNIVERSITY OF TORONTO

Contest 1 Report

Where am I? Autonomous Roourtle Search of an Environment

Qilong (Jerry) Cheng 1003834103
Maryam Hosseini 1004074814
Yuntao Cai 1006519581
Yicheng Wang 10039155551
Qingyun (Ada) Yang 1004404771

March 1, 2022

Contents

1 Problem Definition and Design Objectives

The primary objective of Contest 1, *Where am I? Autonomous Robot Search of an Environment*, is to develop a navigation algorithm that allows TurtleBot to autonomously explore an unknown simulated 6x6 m2 environment in Gazebo within a time limit of 15 minutes. Robot Operating System (ROS), a popular robotic framework, is used as a Inter-Process Communication (IPC) mechanism that enables data sharing for this contest. Then, the ROS GMapping package is used to receive information from sensors on the TurtleBot, and dynamically generate a map when roaming in the area.

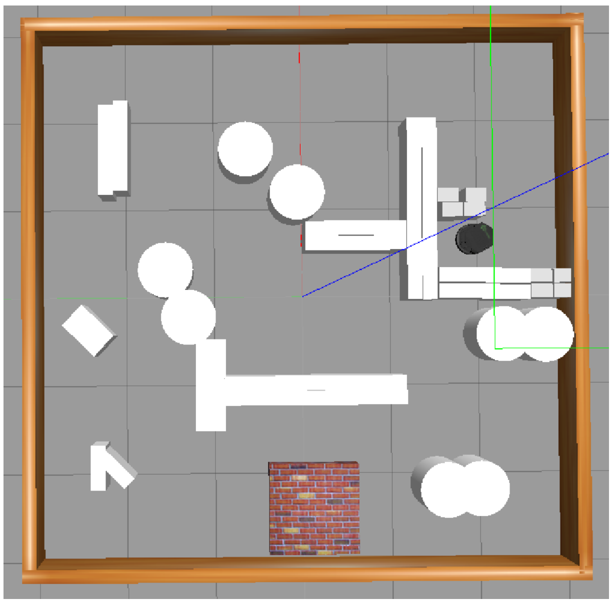


Figure 1: Practice Map

Based on the previous requirements, the team identified three main considerations for contest 1:

1. Time:
   (a) **Fixed time:** Map the world in a fixed time setting (15 minutes) as accurately as possible;
   (b) **Synchronized required time:** Avoid obstacles while navigating during the allocated time;
   (c) **Maximum time limit:** 15 minutes, though the TurtleBot must completely stop moving at 15 minutes
2. Subject of action:
   (a) **Robot-based:** Robot places itself in mapping.
3. Movement:
   (a) **Coverage:** Covers exploring as much unexplored area as possible;
   (b) **Movement-while:** Avoid obstacles while navigating.

The central objective is to maximize the efficiency and robustness at which the TurtleBot explores the environment, and to defend against unknown behaviours/situations that might cause failure or noisy map production. The grade of this contest is determined based on the percentage of the environment mapped and the detection of critical obstacles within the time limit.

1.1 Requirements and Constraints

There are some design requirements that must be considered when developing the navigation algorithm for Contest 1. Some of the design requirements are logistical in nature, that is, they are required of us by the course(ex. Errors in mapping when the robot moves any faster than the speed limit). The requirements and constraints are listed below:

1. The simulated TurtleBot should autonomously explore and map the 6x6 m2 3-dimensional environment within a maximum time limit of 15 minutes with zero human intervention.
2. The robot must completely stop moving when 15 minutes has passed upon start time, or when it has finished mapping the environment.
3. Sensory feedback should be should be provided by the TurtleBot to assist the robot in navigating the environment. The robot shall not use a fixed sequence of movements without the help of sensors.
4. Maximum speed limitation while approaching obstacles and walls is set to be 0.15m/s, with a maximum limitation of 0.3m/s in other cases to avoid errors in the mapping process. Note that this is to ensure consistency between teams' mapping algorithms.
5. The exploration algorithm should be robust to unknown environments with static obstacles.

2 Strategy

The following strategies are crafted to maximize the total mapped area based on the main objectives stated above. Four primary sensors are used for the approach: bumper, laser, odometer, and the occupancy grid (referred to as "map" in later context). The TurtleBot would switch between three different exploration modes based on their corresponding behavioural conditions that occur in the high level controller.

2.1 Controller Design

2.1.1 Reactive Control

The reactive control is directly derived from the bumper sensors. The bumper controller code is activated as soon as the sensor data returns the desired value.

There may be low obstacles, such as the red brick walls, in the unknown environment that are not detectable using the depth sensors. Upon the TurtleBot colliding with these low obstacles, the bumper controller is prioritized over other controllers. The bumper controller is designed as a simple reactive control mechanism: when either bumper is pressed, the bumper sensors will return a true boolean, 1, to activate the controller and stop other actions. Depending on which side the bumper is hit, it will move in a backwards direction and turn correspondingly to which bumper was hit. This mechanism helps avoiding low obstacles in the unknown map as well as assists the robot to vacate various erroneous situations.

2.1.2 Behaviour Control

The team mainly adapts the laser sensor as the secondary controller to prompt the behaviour of the TurtleBot. The algorithm can break down into several layers:

- **Behaviour mode 1:** The TurtleBot moves with a linear velocity of 0.2m/s when the minimum laser distance, taken from the center readings of the laser sensor, is between a safe range of 0.7m to 6m;
- **Behaviour mode 2:** The TurtleBot moves at a slower rate, linear velocity of 0.1 m/s, in order to avoid striking obstacles when the minimum laser distance is less than 0.7m;
- **Behaviour mode 3:** The TurtleBot stops completely when the distance is too close, being less than 0.5 or "infinity", in order for the robot to prevent any avoidable collisions and search for another angle with a finite value to turn.

In addition, the angular velocity of the TurtleBot is adjusted, by comparing the left and right laser data, to ensure it consistently moves in the center of the pathway.

Furthermore, the TurtleBot could possibly get stuck in one location without the bumper sensors being activated, resulting in the TurtleBot to get trapped while considering itself at a free zone trying to move at full speed. In such situations, the odometer's position and the linear velocity of the TurtleBot are constantly monitored over time. If the percentage change in Odom is within a certain range, while linear command velocity is also non-zero, it will return true and move backward to free the TurtleBot and returns to a forward motion behaviour.

2.1.3 Deliberate Control

The TurtleBot will initially be started under a biased random exploration mode, also known as the "Move Forward" mode. The robot will mainly be moving forward until its laser reading is too close or too far, then it will then determine the turning direction using a Bernoulli random distribution, where it has a 55% chance to be turning left to randomize the behaviour of the TurtleBot to ensure that it outputs contrasting results even starting at the same place.

One major problem the team encountered during testing phase occurred when the TurtleBot got stuck in one explored area of the map and was unable to find a new path or an unexplored area. To identify this situation, the total area of the mapped locations is compared over 100 second intervals. For instance, the areas recorded at 0s and 100s from the occupancy grid are compared at 100s. If the total area is changing above a certain threshold over the mentioned time interval, it continues with the on-going exploration mode. Otherwise, frontier exploration is activated to force the TurtleBot to explore unmapped areas. the exact algorithm for frontier detection, path planning, and path following is extensively discussed in Section 4.4.6. Upon reaching the target, the total mapped area is changed, and it will return to the random walk exploration mode as discussed above.

2.2 Exploration Algorithm

To summarize, we have three main strategies to maximize the area that the Turtlebot has explored. The bumper works very well in open/unknown areas. This is because since it does not need any high level control, it does not need a large amount of data, which are lacking in open/unknown areas. The Laser works best in close, straigh/curvilinear corridors, since it would have finite laser readings, and can adjust much better than the other algorithms. Lastly, the Frontier Exploration algorithm works well when the map has already been half way mapped, thereby giving it more data to compute a more optimal path. It does not work well as a beginning algorithm, and performs very poorly, similar to randomly selecting a direction, and going forward until the robot hits a wall. Therefore, we must design a carefully tuned high level controller, that determines when and where these algorithms should be used in what fashion, in order to maximize the amount of area, as well as the accuracy of the map explored by the Turtlebot.

Table 1: Exploration Algorithm

Strategy1	**Move Forward**
Descriptions	It is considered default state when the laser distance exceeds 0.7m and does not equal infinity. TurtleBot continues moving forward with a full linear speed of 0.3 m/s. A P-controller is used to adjust the angular speed and avoid obstacles based on the difference between the readings obtained from the left and right lasers.
Strategy2	**Turn and Find Exit**
Descriptions	It is utilized when the TurtleBot is stuck in a corridor, going back and forth in a straight line or a fixed pattern, and unable to find the exit. The TurtleBot exits the ”Move Forward” mode, stops when it finds left and right lasers' difference to be larger than 1m, and turns to another desired angle that differs from the current yaw. After determining the new desired turning angle, it resets back to the ”Move Forward” mode.
Strategy3	**Frontier Exploration**
Descriptions	It is applied when the TurtleBot is not exploring any new area and finds itself being stuck in the 50-80 percent of the total map area. The map area is calculated in each loop iteration and is compared to with the previous total area stored. If the total area remains the same for over the 100s interval, it exits the current mode and jumps into the ”Frontier Exploration” mode. Inside Frontier exploration, a targeted frontier point and a path are calculated each time the Frontier mode is activated. TurtleBot follows the calculated path until it reaches the target point. If during the path following stage, any bumper is activated or front minimum laser distance is less than 0.5m, it automatically exits the current path and calculates a new path using the same frontier point. If the current path conflicts with the bumper and the laser's reactive controller for more than 3 times, it resets to ”Move Forward” (False Frontier scenario); If it successfully reaches the frontier, it also exits out of the ”Frontier Exploration” and switch back to ”Move Forward”.

3 Sensory Design

3.1 Three Front Bumpers

The TurtleBot has three bumpers on its left, middle, and right sides, each covering 60 degrees of the frontal area of the robot's base level (180-degree area coverage in total). Upon robot colliding with any obstacles, the corresponding bumper will publish its state of “PRESSED”; otherwise its state will be “RELEASED”. Bumper states are stored in global variables and to be used by other control logic.

The motivation for bumper sensors is to assist the robot with avoiding obstacles. However, the /odom coordinate frame shifts every time the TurtleBot collides with an obstacle, resulting in inaccurate GMapping, consequently a more noisy map. Therefore, the team has decided to minimize the use of the bumper sensor, and to use it only in failure modes, or when there are obstacles which cannot be detected by other sensors utilized in the design.

Since bumper sensors are passive sensors, they use less energy since they receive energy present in the environment, at the cost of being more prone noise and disruptions. There are situations where cannot detect the obstacles sometimes when it is pressing on an undetected wall. For mitigation of this problem, see Section 4.4.1 and 4.4.7

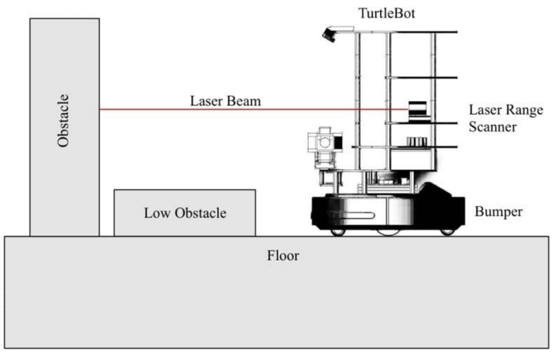


Figure 2: Illustration for the Bumper Sensor Data

3.2 Microsoft Kinect RGBD Camera

The laser data is obtained from a depth camera called Microsoft Kinect attached to the middle of the robot. In the GMapping algorithm, only one slice of the data is obtained, reducing the data from a 2d image array, to a 1D array of laser range data. The laser sensor provides data from 60 degrees of view, 30 degrees on each side where the robot is facing. The laser can detect obstacles as close as 0.45m and those as far as 5m, and readings outside of this range returns a nan (infinity) value. Nevertheless, the use of RGB camera proved very advantageous to the team, since many operations can be performed on the 1D array such as finding the minimum left, center, right distances, average distances, and regions of undefined/nan values.

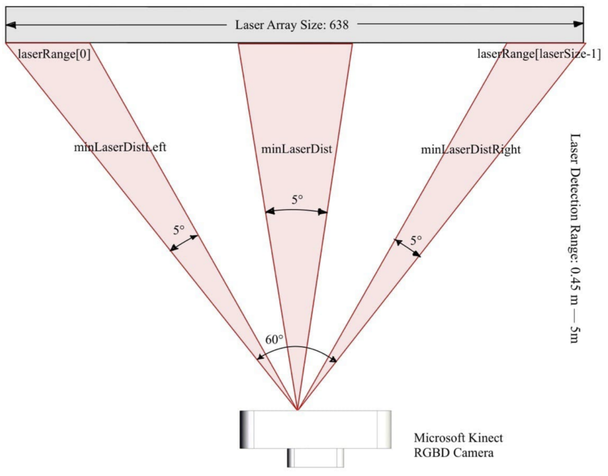


Figure 3: Schematics for the Laser Sensor Data

While experimenting, it was noticed that nan values would appear in the laserRange array, however the robot is within the allowed laser range, which implies this data is an outlier. Post-processing of the raw laser data is done by taking the average of the two adjacent non-"nan" values to avoid logic errors. Lastly, since the laserRange array gives data right to left, the array is reversed so that it is more intuitive.

The laser array enables the implementation of many other tasks. One of which is to navigate through a narrow corridor smoothly while avoiding obstacles. To achieve this, three primary locations from the laser data is taken: left, center and right, with each side comparing a 5-degree range and outputting a minimum distance returned by each location. The minimum front laser distance is used to check how far the TurtleBot is from the front obstacles. If the range is in the safe range, denoted as values between (0.7m - 5m), it will navigate the map at full linear full speed (0.3m/s) forward; if the range is between 0.7m and 0.5m, it will slow down to 0.15m/s to minimize the probability of hitting any obstacles; and if the range is less than 0.5m or infinity, the robot stops and turn to another angle that has a value no less than 0.5 or is infinity, and this behaviour is iterated throughout the robot's exploration. Based on those three behaviours, the difference between the minimum left and right laser data can be used as the error term of a p-controller, which slightly adjust the angular velocity allowing the robot to navigate through narrow corridors smoothly without stopping.

3.3 Odometry

The Odometry callback function returns the positions, orientations, and velocities of the TurtleBot's location with respect to the /Odom coordinate.

Without the Odometry values, one would have to hard code turning a specific angle, or moving a specific distance. With the Odometry data, one can establish a PID controller which can track any references, and disturb any rejections with minimal error. Additionally, in order to avoid the TurtleBot from going back to the place where it had just came from, the odometry data is stored in a list so that future values can be compared with elements of the list, thereby determining whether if the robot has been in the same location. If this occurs, it indicates that there is a high likely hood the robot is stuck in an explored area. When this happens, the robot is given instruction to turn the opposite angle that it had turned the first time it visited this area, in the hope that turning in a diffrent direction than last time would enable the robot to not get stuck. Should the size of the list becomes larger than 20, the oldest value will be disregarded, while the new value will be added in, maintaining the array size of 20 in order to prevent overflow.

Finally, the TurtleBot is unable to move when the back of the robot is in contact with the wall/obstacles or when the frontal area is in contact of the lower invisible red brick. When this behavior occurs, even with a non-zero angular velocity, it will remain stuck and thus GMapping will output inaccurate slam graph as well. To make the robot aware of this situation, time accumulated odom values are compared in order to command linear/angular velocity to evaluate whether the TurtleBot is stuck or not. When stuck, it outputs true to a global variable, where the high level controller can make changes according to the state of that variable, such as shifting to odometry controller and override the current commands.

4 Controller Design

4.1 ROS Architecture

- ROS Node: Contest1
- Subscribed ROS topics and its subscribing ROS messages
  - /odom → nav_msgs/Odometry
  - /scan → sensor_msgs/LaserScan
  - /mobile_base/events/bumper → kobuki_msgs/BumperEvent
  - /map → nav_msgs/OccupancyGrid and nav_msgs/MapMetaData

4.2 Callback Functions

Each topic that is subscribed by the team will return the raw data via ROS messages and the data is then processed and stored into global variables for later calculations and conditional checks. The linear and angular velocities are also stored in global variables and are published when necessary.

- /odom: provides the current posX, posY, yaw, linear and angular velocities.

  In addition, the current posX, posY are stored into two sized-20 vectors; The current logic in the code is that when the minimum middle laser distance returns a value less than 0.5m, we would try to find a new angle for the robot to turn to. Before this step, the current posX and posY values will push the data onto the vector. If the vector is filled up with 20 values, the new values will be pushed in and the last ones will be dropped. By storing the previous odometry readings, one can compare the current odometry readings to the previous, there by determining whether or not the robot has been in a similar location before. (less then 0.2m difference).

- /scan: provides the laser depth information as listed above.

  For the post-data-processing, three locations' minimum laser distances and average laser distances are calculated and stored in the global variable; left, center and right, each spanning a range of 5 degrees. The difference between the left and right average laser distance is also computed and stored in the global variable.

- /mobile_base/events/bumper: provides the states of each bumper.

  A global variable boolean called any_bumper_pressed will return true when either of the three bumpers is returning 1. Each bumper state and the boolean are all stored inside the global variable.s

- /map: provides the occupancy grid of the map created by Gmapping.

  Inside the nav_msgs/OccupancyGrid, the map resolution, dynamic map width and height, /map frame origin with respect to the map, and the mapped data are provided. The map data is given as -1, 0, and 100 representing unexplored, free space and obstacles, respectively, in a row-major 1D array fashion.

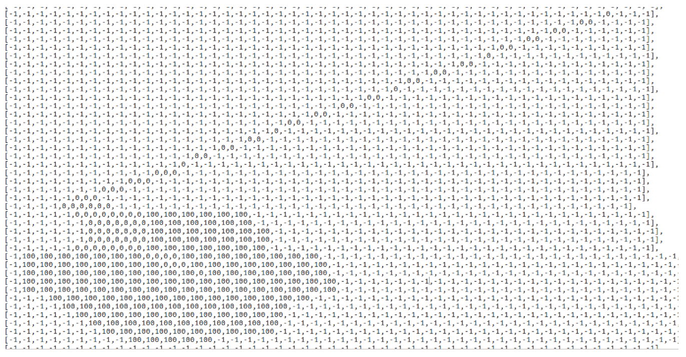


Figure 4: Raw Map Data

For purely visualization purposes, this data is transferred to Python where the matplotlib library provides many useful functions for the team to visualize the data using pixel values, and performing further debugs. The pictures can be seen in Figure 4.

It was discovered that the occupancy grid can reach up to hundred thousand points, and since many 2-d vector calculations are O(N*N), which could significantly decrease the amount of time for exploration, since the robot is "thinking hard, acting later". Therefore, the largest border was found that contains only the data points with values greater or equal to 0, unknown spaces are not considered. Furthermore, since a common 2-D array operation is to take all neighbors of a cell, and to perform tasks on them such as taking the average, taking the maximum, summations, etc. A padding of 3 cells is added to ensure that there are no segmentation faults due to attempts to access an illegal memory location, such as values smaller than 0, or larger than the rows/columns of the array, see Figures 5 and 6.

The odom of the TurtleBot is measured based on the /odom frame, which might shift overtime. A transformation is needed to obtain the precise location of the robot onto the Gmapping 2D array, by using the follwing /tf topic.

- /tf topic: publishes the matrix transformation data with three translational values (x,y,z) and four rotational quaternion numbers (x,y,z,w) representing orientation. Initially, the TurtleBot's odometry is with respect to

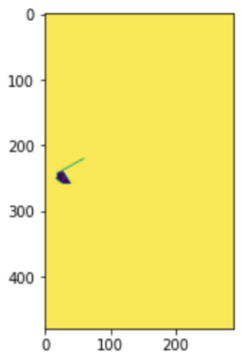


Figure 5: Colour Graded Post-processed Data Plotted in Python

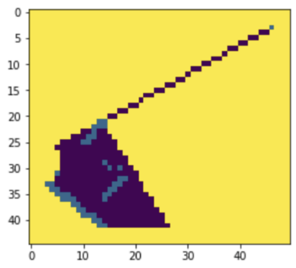
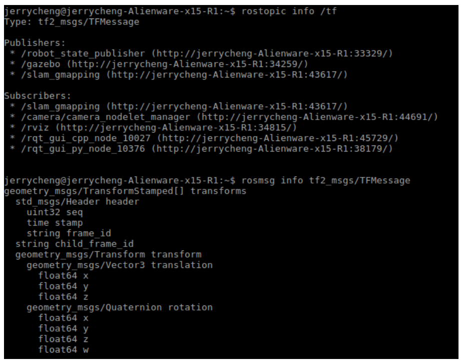


Figure 6: Cropped Map Data Plotted in Python

the odom frame, not the map frame. By obtaining the necessary transformation needed, one can obtain the values of the TurtleBot's odometry such as posX and yaw, with respect to the map frame rather than the odometry frame. This is because the values are much more accurate once the transformations needed have been applied. After some calculations, we can obtain a one to one correspondence of the posX and posY values, to the row and column of the Occupancy Grid. The new odom information is stored in the global variables as: posX_map and posY_map.



```
jerrycheng@jerrycheng-Alienware-x15-R1:~$ rostopic info /tf
Type: tf2_msgs/TFMessage

Publishers:
 * /robot_state_publisher (http://jerrycheng-Alienware-x15-R1:33329/)
 * /gazebo (http://jerrycheng-Alienware-x15-R1:34259/)
 * /slam_gmapping (http://jerrycheng-Alienware-x15-R1:43617/)

Subscribers:
 * /slam_gmapping (http://jerrycheng-Alienware-x15-R1:43617/)
 * /camera/camera_nodelet_manager (http://jerrycheng-Alienware-x15-R1:44691/)
 * /rviz (http://jerrycheng-Alienware-x15-R1:34815/)
 * /rqt_gui_cpp_node_10027 (http://jerrycheng-Alienware-x15-R1:45729/)
 * /rqt_gui_py_node_10376 (http://jerrycheng-Alienware-x15-R1:38179/)


jerrycheng@jerrycheng-Alienware-x15-R1:~$ rosmsg info tf2_msgs/TFMessage
geometry_msgs/TransformStamped[] transforms
  std_msgs/Header header
    uint32 seq
    time stamp
    string frame_id
  string child_frame_id
  geometry_msgs/Transform transform
    geometry_msgs/Vector3 translation
      float64 x
      float64 y
      float64 z
    geometry_msgs/Quaternion rotation
      float64 x
      float64 y
      float64 z
      float64 w
```

Figure 7: /tf Topic and its Published Messages

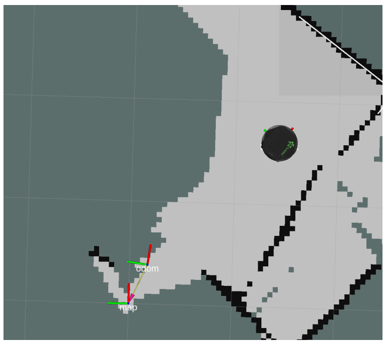


Figure 8: Relationship between the /map Frame and /odom & /Base_link Frames

4.3 High-level Controller Architecture

Two approaches have been considered for the high level controller:

1. The first method is to consider each behaviour as completely independent from the other, that is, one cannot have an effect on the other. Commands such as move forward 5m, turn counter-clockwise 90 degrees, and rotate 360 degrees are very common in this control architecture. Upon finishing one function, it loops through the main logical condition check and decides on the next action, then continues to the respective function.

2. Similar to calculus, one could view setting each behaviour as an incremental change for each iteration and switch its incremental change depending on the condition. For example: by setting the linear velocity as a certain value, whilst other functions can potentially override this particular linear velocity to something else, or in that because the velocity is above a certain threshold, change the some other part of the code. The linear and angular velocity values will then be published together only if it finishes all the condition check in the main loop.

During testing, it was found that the first architecture design is not be able to update the TurtleBot's conditional states in a timely manner. For example, when the previous command is to move forward 5m, since the TurtleBot does not stop until the odom values difference is around 5m, other sensory's data change may not be detected. If each of those behavioural functions are to be designed considering all the sensory state values, it can be repetitive and might cause conflict when two commands contradict each other: for instance, when the first function ask the TurtleBot to move forward 5m, but it hits the bumper during this action and needs to pull back -0.5m. This can cause it getting stuck in a while loop and not behave ideally.

Consequently, the architecture is modified based on the second strategy. Each action is broken down to small linear or angular changes depending on the conditions and shares the same while loop in the main() function. In this case the TurtleBot will never stop due to behavioral changes unless it is commanded to. The overall high level controller is shown below in the flow chart. All the conditional checks will be checked every time it loops through the main while loop. This can ensure the linear and angular velocities are only published once and avoid any conflicts possible.

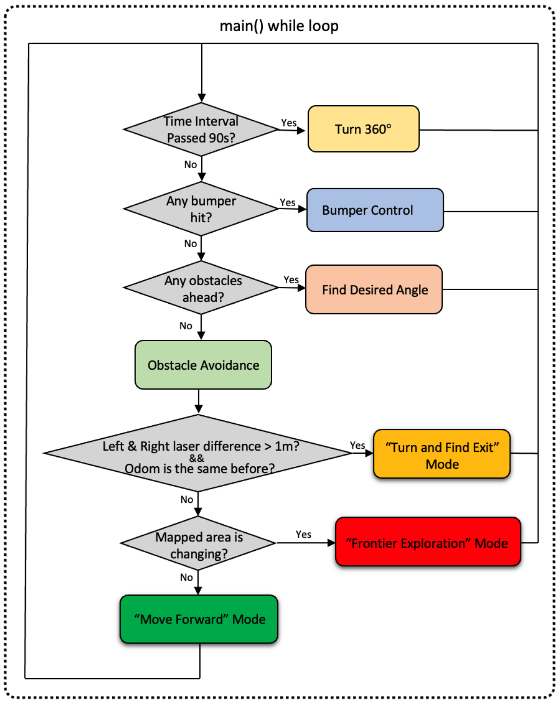


Figure 9: Flow Chart of the High Level Controller Design

4.4 Low-level Controller

A hybrid approach is chosen to control the TurtleBot. All reactive controller, deliberate controller and behaviour controllers are utilized in the algorithm, as mentioned in Section 2. Detailed algorithmic designs are explained in this section.

4.4.1 Bumper Controller

The bumper controller is used for the invisible red brick wall situations and any accidental obstacle collisions due to laser's data blind spot, see Figures 10 and 11.

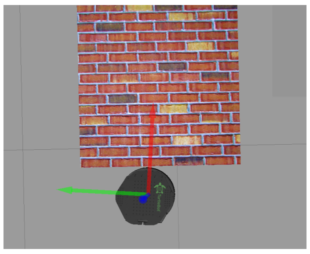


Figure 10: Low Obstacle - Red Brick

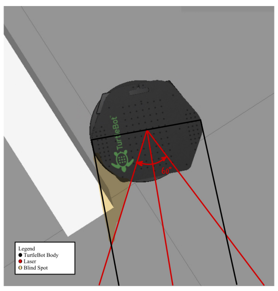


Figure 11: Laser Data Blind Spot

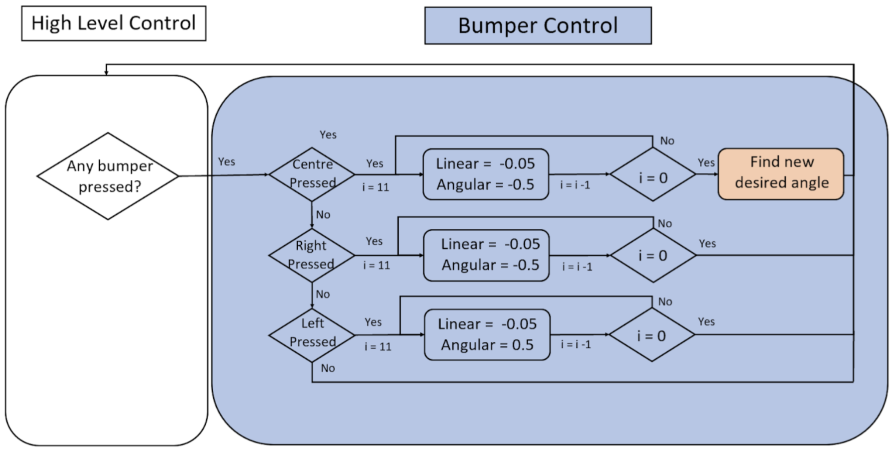


Figure 12: Flow Chart of the Bumper Controller Algorithm

The bumper state is checked every iteration when it looks through the main while loop. Once either of the bumper is activated, it will return a negative linear velocity and a respective angular velocity depending on which bumper is pressed. 10 loop times is set to move the TurtleBot at those reactive velocities before exiting out of the bumper controller and checking other conditions. This reactive behaviour can free the TurtleBot in both low red-brick situations and accidental obstacle collision situations.

In addition to the reactive control, during experiment it can be observed that when front bumper is hit, it is likely to get stuck in the same position due to the laser sensors limitation. The TurtleBot might get false signal, thinking the front orientation is clear and can be explored, but in reality keep hitting the invisible walls. To avoid such cases, an additional "Angle Selection" is added to find a path to explore. Since angle selection will eliminate the original orientation as its ideal angle, TurtleBot will choose a different path and free itself.

4.4.2 Obstacle Avoidance

Obstacle avoidance is mainly achieved by comparing the center minimum laser distance to control the linear velocity output of the TurtleBot.

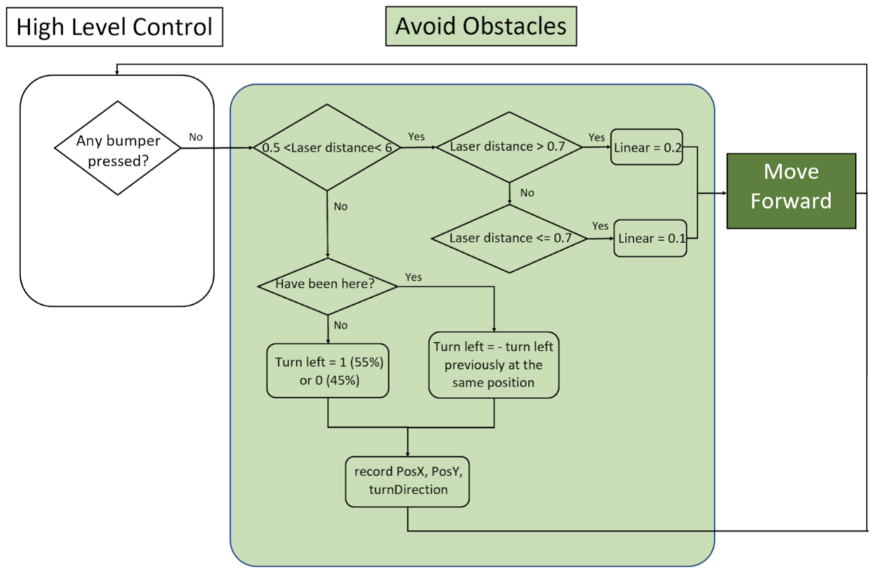


Figure 13: Flow chart of Obstacle Avoidance Algorithm

By checking the center minimum laser distance value, two Boolean values can be determined and stored in the global, while controlling TurtleBot's behaviours as follows:

- 0.7m - 5.0m: Moves forward at the maximum linear velocity at 0.2m/s
- 0.5m - 0.7m: Moves forward at a reduced linear velocity at 0.10m/s
- < 0.5m or infinity: Stops the TurtleBot and starts looking for a new desired angle to turn to via "Angle Selection"

By trial and error, the maximum velocity was determined. It can be noticed that GMapping has some update delays; for example, when moving too fast, it cannot localize the TurtleBot accurately. To minimize the inaccuracy of the map while reduce the time spent on exploration, the velocity is set to 0.2m/s.

As seen in Figure 14, when the maximum velocity is set as 0.3m/s, there are angular shift in the odom frame due to the high velocity, causing its phase shift.

In order to randomize the exploration each time running the TurtleBot, turning direction is determined by using Bernoulli distribution, which is implemented to output a random number with a pre-determined bias to one turning direction. A 0.55 chance of probability is given to turn left. This can ensure a different output even if the exploration starts at the same location.

$$f(k;p) = p^k(1-p)^{1-k} k \in \{0,1\} \tag{1}$$

If the Turtlebot is turning in a position where it has been, the turning direction would be the opposite direction as the last time to avoid being trapped in the same area. This is achieved by comparing the current position with the

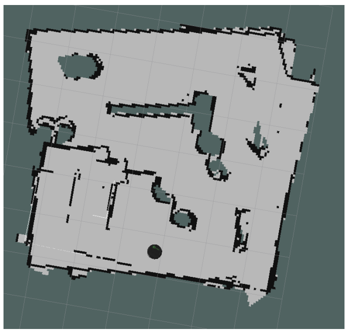


Figure 14: Generated Map with 0.3m/s Velocity throughout

previous positions recorded in the two sized-20 vectors, mentioned in 4.2, and reading another array that records previous turning directions. Then, the determined turning direction is passed to Find Desired Angle function to control the spin direction of TurtleBot.

4.4.3 Angle Selection

"Angle Selection" is activated when the TurtleBot is stopped if it's too close to the wall, or hit the front bumper. It is used to determine the most ideal open area to explore and avoid going back to the original orientation.

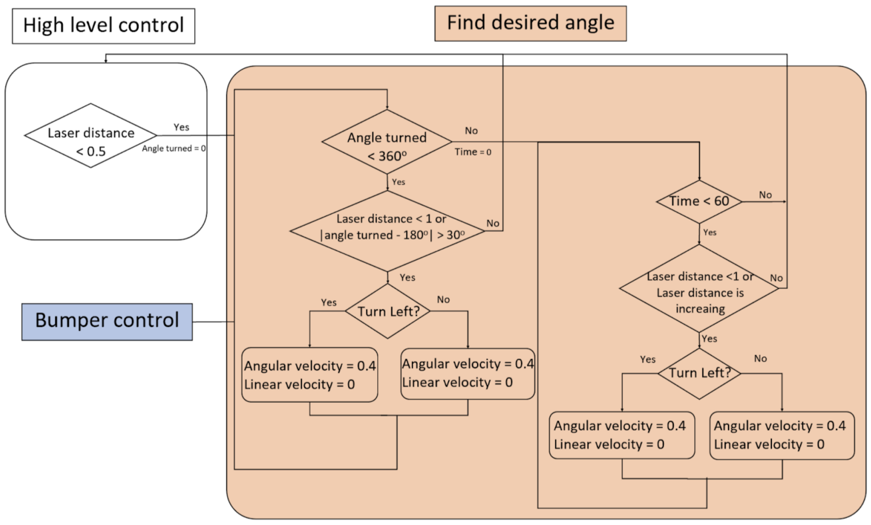


Figure 15: Flow Chart of the Find Desired Angle Algorithm

The turning direction is determined from previous exploration experience or the Bernoulli distribution which can ensure randomization and prevent the TurtleBot from trapping in the same area. The initial angle is recorded and then angular velocity is set to be 0.4 rad/s in the corresponding direction. The TurtleBot will keep turning while the laser distance is smaller than 1m or angle turned is equal to 180±30 degree. This enables the TurtleBot to turn to the direction of open area without traveling back to coming position. Considering the situation when TurtleBot travels into a narrow area with one opening, angle turned is always compared with 360 degree. If the angle turned is equal to 360 degree which means TurtleBot hasn't find any other suitable direction to go except the place it came from. In this case, angular velocity is set to be 0.4 rad/s and TurtleBot will stop turning when laser distance is not increasing and larger than 1 so that TurtleBot will travel in the direction of which the distance is local maximum.

4.4.4 Angular Velocity P-controller (Default Mode)

This mode is considered as TurtleBot's default mode. It is used to walk through narrow corridors without hitting the side walls, and precisely adjust its orientation even when there are uneven obstacles on two sides, see Figures 16. During the laser callback, the minimum distance on the left side and the right side of the robot is computed, and

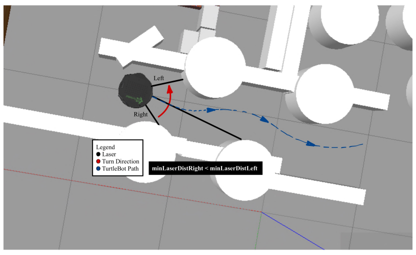


Figure 16: The TurtleBot Moving through a Corridor with Uneven Walls

the difference is taken. The angular velocity is set as a constant multiplied this difference, so that if the left wall is closer than the right wall in a tight corridor, which means the difference is negative, the robot will turn clock-wise, and therefore making the laser scan difference between the left and right wall smaller.

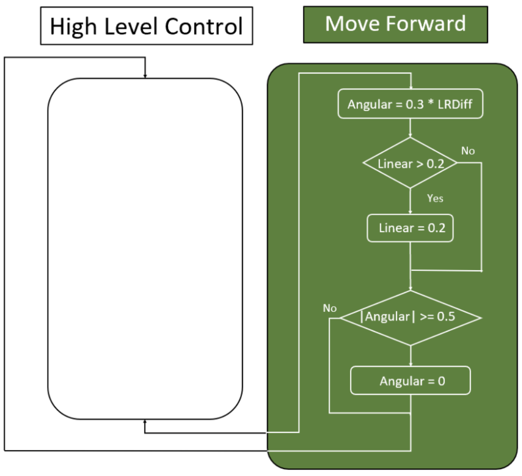


Figure 17: Flow Chart of the Move Forward Algorithm

4.4.5 Stop and Find Exit Mode

This mode is used specifically when the TurtleBot is stuck in a corridor and unable to find the exit, as seen in Figure 17.

In order to enter this mode, the TurtleBot's odom positon is first compared to the odom vector stored in the callback funciton. If it returns true, that TurtleBot has been to the same location before, it will start entering "Stop and Find Exit" mode when the difference between the left and right laser distance is greater than 1m. Once both condition are met, the turtlebot will stop and enter the "Angle Selection" mode to look for a new moving direction. This method can ensure the turtlebot not ignoring the exit in a long narrow corridor because of the P-controller's inability to stop when the left-right laser distance is too far apart.

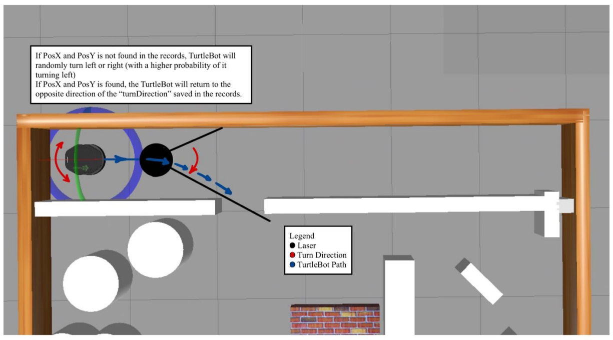

If PosX and PosY is not found in the records, TurtleBot will randomly turn left or right (with a higher probability of it turning left)
If PosX and PosY is found, the TurtleBot will return to the opposite direction of the "turnDirection" saved in the records.
Legend
● Laser
● Turn Direction
● TurtleBot Path

Figure 18: The TurtleBot's Behaviour in a Long Corridor

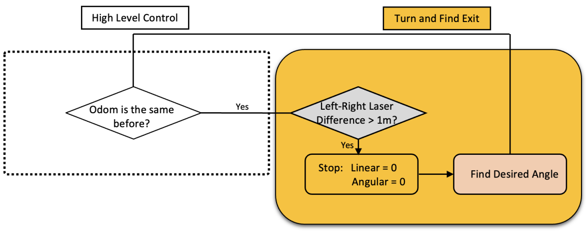


Figure 19: Flow Diagram of the Stop and Find Exit Mode

4.4.6 Frontier Exploration Mode

Finally, the last scenario occurs when the TurtleBot have explored only 60-80% of the map and is unable to explore the rest of new areas. To avoid this and to ensure the TurtleBot explore all the areas in the map, frontier exploration will be activated when the mapped total area is not changing, as mentioned in previous sections. The total mapped area is checked every 100s, and is compared with the previous values. If it is within a certain range and then "Frontier Exploration" mode will be started.

The Frontier Exploration algorithm consists of two components, path planning and path following. The first part finds a path to the optimal frontier point that is reachable by the robot. The latter makes specific changes to the robot's linear and angular velocities so that it can follow the path to the frontier point. This algorithm is then iterated until the high level controller decides otherwise.

The optimal frontier point is found by recognizing all frontier points, then using several criterions to determine the optimal point. This is done by using the /map callback stated in Section 4.2. A frontier point is defined as any cell which has the value 0, and has at least one neighboring cell with value -1. This indicates that the cell itself is a known point, and it is surrounded by at least one unknown point. The number of surrounding cells that are considered could be not just all cells that are adjacent to the cell, but also one layer further. For this assignment, it was found that aggregating cells whose distance from the cell is less than 0.25m (5 occupancy grids) is a decent option.

The number of neighboring cells with value -1 is recorded and denoted as I, for intensity, and the number of neighboring cells with value larger than 0 is also recorded, denoted as O, for obstacle. For example, the following cell would have an I value of 4, and an O value of 100.

For Frontier Point Selection, we use a function R(d,I,W), where d is the distance of the cell from the robot's current position, I and W are defined above. This function determines the "optimal" frontier point by taking a linear combination of these three criteria, and returning a single number that indicates how good of a frontier point it is. R is expressed as:

$$R = ad + bI - cW \qquad a, b, c \in R_{>=0} \tag{2}$$

-1	-1	-1
-1	0	100
0	0	0

Figure 20: Frontier Point Selection Method

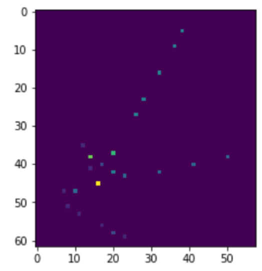


Figure 21: Frontier Points Selection Results

The function rewards a cell for having a greater distance from the robot's current position, because appealing to information theory, one should pick the cell selected gives the greatest information gain. It also rewards a cell with a greater intensity, since this means there are many unknown points near. It penalizes neighbouring obstacles, as this implies the frontier point is less likely to be accessible.
To find a path to the frontier points, Dijkstra's algorithm is used to find the shortest path between the robot's current position, to each frontier point. Although superior algorithms and simpler algorithms such as A* and Breadth first search are also a decent choice, A* is optimized to find the path to only one particular destination, not all, and BFS cannot compute diagonal paths, since the distance between each cell must have unit weight.

It is realized that the robot is a physical object, that is, it does not have a point mass. It would be better to carry the computation assuming that the robot has a point mass. This is done by increasing the thickness of the obstacles artificially, so that the robot does not hit any walls. The reason for the quotation marks around the word

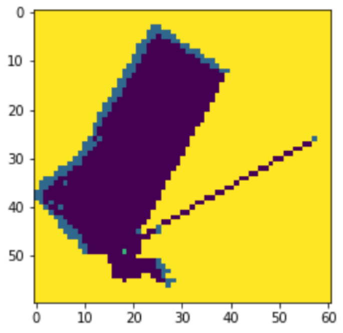


Figure 22: Without the Padded Boundaries around the Obstacles

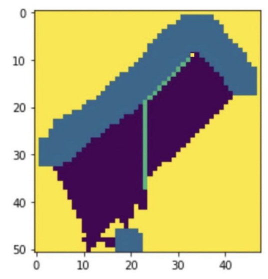


Figure 23: After Adding the 0.25m Padding around the Obstacles

optimal above, is because the cell with the highest value might not be the best point. This results from the scalar values not being tuned perfectly, and also that the best point might be unreachable, or an outlier. To mitigate this problem, the fitness proportionate selection operator inspired from genetic algorithms is used to give a probability to each frontier point, with the better frontier point having a higher likelihood to be picked. The probability of each frontier point being picked is computed by the following expression,

Fitness proportionate selection:

$$p_i = \frac{f_i}{\sum_{j=i}^{N} f_i} \tag{3}$$

Where pi is the probability of the frontier point being picked, Ri is the fitness of the individual frontier point, and N is the total number of frontier points.

Therefore, the optimal frontier is selected using this algorithm, provided that it is reachable by the robot's current location. If Djkstra's algorithm is performed correctly, this can be determined in O(1) time. The path can then be returned for path following.

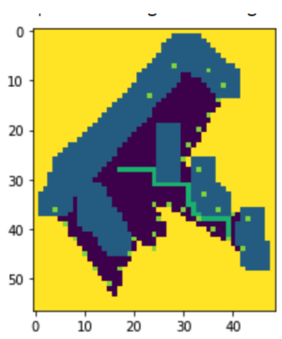


Figure 24: Path Planned Using Wavefront

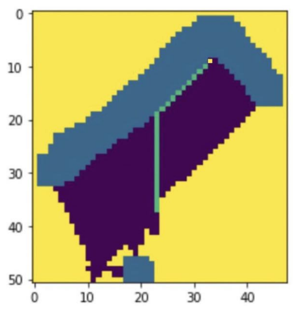


Figure 25: Path Planned using Dijkstra

After the path to the optimal frontier point has been determined, it remains to send a series of instructions to adjust the robot's linear and angular velocities so that the robot navigates to its final destination. There are three possible cases.

First, the array of the path is only size 1, then the robot remains in place and the function returns..

Second, if the array of the path is less than 8 cells long, then the robot first turns towards the cell, and then it moves forward until the euclidean distance between its own position and the position of the final cell is less than 0.1.

Lastly, if the array of the path is greater or equal to 8, a properly developed path-planning algorithm is needed. We first denote 4 variables, as shown below. The robot will first align itself with the toGo point, then it moves forward until the error between its position and the tolerance point is smaller than 1. Then we increment the current point, the tolerance point, and toGo point by 2. This process is iterated, along with a small p-controller to make minor adjustments to the angle, until the error between the robot's position and the final point is less than 1. See Figure 26 and 27.

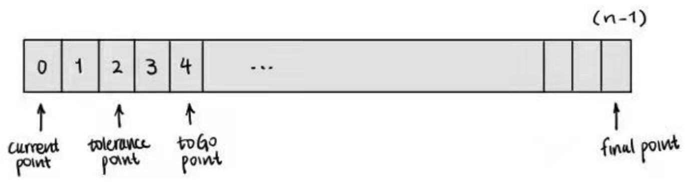


Figure 26: Initiate Path Planning Algorithm

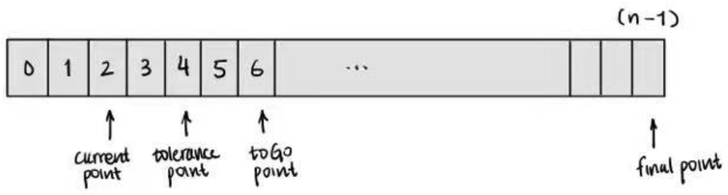


Figure 27: Iterate Until Final Point Is Reached

4.4.7 Turn 360°

Finally, through testing, it was seen that the bumper sensor will not return 0, when the TurtleBot is initialized directly facing and in contact with the wall, and the "Find desired angle" function will fail to rotate as well. Even if the angular velocity is non-zero, it will only rotate a small degree before stopping and get stuck. To mitigate this behaviour, a 360 degree turn is performed every 100s to free the TurtleBot from getting stuck. Sometimes this is also not enough and it was found that a combination of moving backwards and turning 360 degrees was enough to free the TurtleBot.

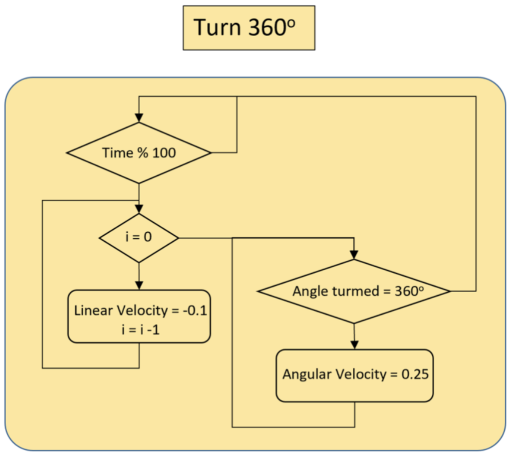


Figure 28: Frontier Point Selection

5 Final Map Results

Finally, a clean and well-shaped map is obtained using a clever combination of the above algorithms, as seen in Figure 29. Within the 15 minutes testing time, the TurtleBot is able to scan through the entire map and map out relatively precious locations of each obstacles. The initial /odom frame shift is the main source of the mapping accuracy. As can be seen on the graph below, mapped walls and obstacles are shifted outside the 6*6m environment. Improvements to be made are detailed in the following section.

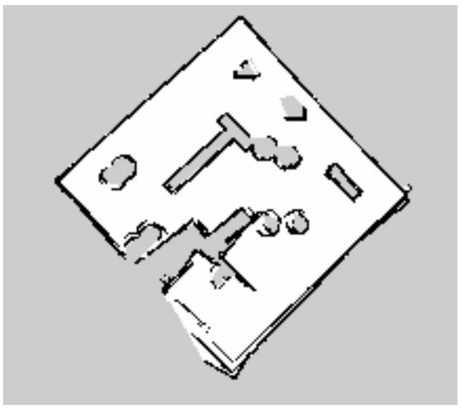


Figure 29: Final Mapping Result

6 Future Recommendations

This paper presents the proposed navigation algorithm of the TurtleBot in an unknown simulated environment. Throughout the course of this project, the team was able to develop a foundational exploration and mapping algorithm; however, there is ways to improve the current algorithm.

One of which is implementing a reinforcement learning (RL) in the algorithm which is to apply rewards immediately after the occurrence of a response to increase the probability of it happening. This assists with strengthening desirable outcomes and weakening undesirable outcomes, such as looping around and getting stuck in corners.

Another design complication the team would like to address first is utility problem in order to optimize the utility of stimulus by using or discarding them when required or not needed, respectively.

In addition, the code is currently written in one main file which makes it much more complicated to debug, add or modify the code within the team members. The team would integrate separate files, such as header files, to facilitate different nodes in the code, in which they can easily be distributed and assigned to the team members.

7 Contribution Table

The contribution table is denoted as follow:

Sections	Qilong Cheng	Maryam Hosseini	Yuntao Cai	Yicheng Wang	Qingyun Yang
Research	RS	RS	RS	RS	RS
Coding: High level controller	WD,MR,DE,FP		WD,MR,DE,FP	WR,MR,FP	
Coding: Low level and basic functions	WD,FP	WD,MR,DE, FP	WD,MR,DE,FP	WD,MR	WD
Coding: Random Walk			WD,MR,DE,FP	WD,MR,DE,FP	
Coding: Data Structures and overall architecture	WD,MR,DE,FP		WD,MR,DE,FP	WD,MR,DE	
Coding: Frontier Exploration			WD,MR,DE,FP		
Coding: Callback Functions	WD,MR,DE,FP		WD,MR,DE,FP	DE,FP	
Report : Problem Definition and Design Objectives	WD,MR,RS,FP	WD,MR,FP, RS,ET	FP	FP	WD,MR,RS,ET,FP
Report : Strategy	WD,MR,RS,FP	WD,MR,FP, RS,ET	FP	WD,MR,RS,FP	RS,WD,MR,ET,FP
Report : Sensory Design	WD,MR,RS,FP	WD,MR,FP, RS,ET	FP	FP	ET,FP
Report : Controller Design	WD,MR,RS,FP	WD,MR,FP, ET	FP	MR,RS,FP	ET,FP
Report : Strategy and Controller Design	WD,MR,RS,FP	WD,MR,FP, ET	WD,MR,FP	WD,MR,RS,FP	ET,FP
Future Recommendation, Objectives and Constraint		WD,MR,FP, RS,ET	FP		FP

RS - research
ET - edited for grammar and spelling
WD - Wrote Draft
FP - final proofread of assignment
MR - Major revision
DE - Major debugging

References

[1] Poolad, P., 2022. 8th ed. [ebook] Toronto, ON: University of Toronto. Available at: https://q.utoronto.ca/courses/255899/files?preview=18668457.

Appendices

Appendix A - Source Code on the Controller

```
#include <ros/console.h>
#include "ros/ros.h"
#include <geometry_msgs/Twist.h>
#include <kobuki_msgs/BumperEvent.h>
#include <sensor_msgs/LaserScan.h>
#include <stdio.h>
#include <cmath>
#include <vector>
#include <chrono>
#include <algorithm>

#include <nav_msgs/Odometry.h>
#include <tf/transform_datatypes.h>
#include <random>
#include <iostream>
#include <bits/stdc++.h>
#include "std_msgs/String.h"
#include <sstream>
#include <fstream>
#include <float.h>


#include <tf2_ros/transform_listener.h>

#include <nav_msgs/OccupancyGrid.h>
#include <nav_msgs/MapMetaData.h>
#include <math.h>

#include <geometry_msgs/TransformStamped.h>
#include <turtlesim/Spawn.h>
#include "dataStructure3.h"






#define N_BUMPER (3)
#define RAD2DEG(rad) ((rad)*180. / M_PI)
#define DEG2RAD(deg) ((deg)*M_PI / 180.)
#define CCW (1)
#define CW (0)


GlobalVar turtlebot;

enum BumperEnum
{
    Left = 0,
    Center = 1,
    Right = 2,
};

enum bfsVals
{
    bfsLeft = 1,
    bfsRight = 2,
    bfsUp = 3,
    bfsDown = 4,
    bfsUpLeft = 5,
    bfsUpRight = 6,
    bfsDownLeft = 7,
    bfsDownRight = 8
};



float minLaserDist = std::numeric_limits<float>::infinity();

float lasterror;
float middelLaserDist = 0;

int32_t nLasers = 0, desiredNLasers = 0, desiredAngle = 12;

std::chrono::time_point<std::chrono::system_clock> start;
uint64_t secondsElapsed = 0;
int backStreetBous = 1;




bool stopThree = false;
bool isTurning = false;
bool zhuanSLL = false;

float initialDistance = 0.0;
float previousDistance = 0.0;
float initialYaw = 0, diffrenceYaw = 0, previousYaw = 0;
int zhuan360Avoid = 100;
float presentSecond = 0;

std::random_device aRandomNum;
int seed = 123;
//std::mersenneTwister19937 mersenneTwister(seed);
std::mt19937 mersenneTwister(aRandomNum());
bool goL = false;
std::bernoulli_distribution dist(0.55);



void pause3seconds();
void turn360();
inline void publishVelocity();
std::pair<int, int> xyTOij(float x, float y);
void turn2Angle(float target);
void setup();
void move2Distance(float targetDistance);
std::pair<bool, float> angleDifference(float angle1, float angle2);
bool areaNotChanging();
std::vector<std::vector<int>> findBFS(std::vector<std::vector<int>> mat, int source_i, int source_j);














int verifySimi(float xPos, float yPos)
{
    for (int i = 0; i < turtlebot.odom.posXAssim.size(); i++)
    {
        float xDifs = fabs(xPos - turtlebot.odom.posXAssim[i]);
        float yDifs = fabs(yPos - turtlebot.odom.posYAssim[i]);

        if (xDifs < 0.19 && yDifs < 0.19)
        {
            return i;
        }
    }
    return -1;
}

void bumperCallback(const kobuki_msgs::BumperEvent::ConstPtr &msg)
{

    turtlebot.bumper.bumperState[msg->bumper] = msg->state;
    turtlebot.bumper.any_bumper_pressed = false;

    for (uint32_t b_idx = 0; b_idx < N_BUMPER; ++b_idx)
    {
        turtlebot.bumper.any_bumper_pressed |= (turtlebot.bumper.bumperState[b_idx] == kobuki_msgs::BumperEvent::PR
    }

    if (turtlebot.bumper.any_bumper_pressed == 1)
    {

        turtlebot.bumper.fronBumpPresed |= (turtlebot.bumper.bumperState[1] == kobuki_msgs::BumperEvent::PRESSED);

        turtlebot.bumper.lefttBumpPresed |= (turtlebot.bumper.bumperState[0] == kobuki_msgs::BumperEvent::PRESSED);

        turtlebot.bumper.rightBumpPresed |= (turtlebot.bumper.bumperState[2] == kobuki_msgs::BumperEvent::PRESSED);
    }
}

void laserCallback(const sensor_msgs::LaserScan::ConstPtr &msg)
{

    minLaserDist = std::numeric_limits<float>::infinity();

    nLasers = (msg->angle_max - msg->angle_min) / msg->angle_increment;
    desiredNLasers = M_PI/30 / msg->angle_increment;


    for (uint32_t laser_idx = nLasers / 2 - desiredNLasers; laser_idx < nLasers / 2 + desiredNLasers; ++laser_idx)
    {

        minLaserDist = std::min(minLaserDist, msg->ranges[laser_idx]);
    }





    turtlebot.laser.laserRange.clear();
    float tempLaserPlaceholder = std::numeric_limits<float>::infinity();
    for (int i = 0; i < msg->ranges.size(); i++)
    {
        if (msg->ranges[i] < 0.45)
        {
             turtlebot.laser.laserRange.push_back(tempLaserPlaceholder);
        }
        else
        {
            turtlebot.laser.laserRange.push_back(msg->ranges[i]);
        }
    }
    turtlebot.laser.minLaserDistLeft = 100;
    turtlebot.laser.minLaserDistRight = 100;

    std::reverse(turtlebot.laser.laserRange.begin(), turtlebot.laser.laserRange.end());



    for(int laserCounter=20;laserCounter<int(turtlebot.laser.laserRange.size()/2);laserCounter++)
    {
        turtlebot.laser.minLaserDistLeft=std::min(turtlebot.laser.minLaserDistLeft,turtlebot.laser.laserRange[laserC

    }
    for(int laserCounter=int(turtlebot.laser.laserRange.size()/2);laserCounter<int(turtlebot.laser.laserRange.size(
    {
        turtlebot.laser.minLaserDistRight=std::min(turtlebot.laser.minLaserDistRight,turtlebot.laser.laserRange[lase
    }


    turtlebot.laser.LRDiff = turtlebot.laser.minLaserDistLeft - turtlebot.laser.minLaserDistRight;


    middelLaserDist = turtlebot.laser.laserRange[320];

}

void odomCallback(const nav_msgs::Odometry::ConstPtr &msg)
{

    turtlebot.odom.posX = msg->pose.pose.position.x;
    turtlebot.odom.posY = msg->pose.pose.position.y;
    turtlebot.odom.yaw = tf::getYaw(msg->pose.pose.orientation);
    turtlebot.odom.goodPosX=turtlebot.odom.posX+ turtlebot.odom.tfX;

    turtlebot.odom.goodPosY=turtlebot.odom.posX+ turtlebot.odom.tfY;
    turtlebot.odom.goodYaw=turtlebot.odom.posX+ turtlebot.odom.tfYaw;



    std::pair<float, float> tempFLoat123(turtlebot.odom.posX, turtlebot.odom.posY);
    turtlebot.map.allMapPositions.push_back(tempFLoat123);
    if (turtlebot.map.allMapPositions.size() > 12000)
    {
        turtlebot.map.allMapPositions.erase(turtlebot.map.allMapPositions.begin());
    }


    float changeBetweenPos = sqrt(pow((turtlebot.odom.posX - turtlebot.map.allMapPositionAggr.back().first), 2) + po
    if (changeBetweenPos > 0.2)
    {
        std::pair<float, float> tempFLoat1(turtlebot.odom.posX, turtlebot.odom.posY);
        tempFLoat1.first = tempFLoat1.first;
        tempFLoat1.second = tempFLoat1.second;

        turtlebot.map.allMapPositionAggr.push_back(tempFLoat1);
    }

    turtlebot.map.allYawOrientations.push_back(turtlebot.odom.yaw);

    if (turtlebot.map.allYawOrientations.size() > 1000)
    {
        turtlebot.map.allYawOrientations.erase(turtlebot.map.allYawOrientations.begin());
    }
    if (turtlebot.map.allYawOrientations.size() > 2000)
    {

    }
    bool noSpinButShouldSpin = false;
    int numOPointsTOConsider =300;
    if (turtlebot.velocity.angular != 0)
    {
        float changeElectricShock = 0;
        for (int i = turtlebot.map.allYawOrientations.size() - 1; i < turtlebot.map.allYawOrientations.size() - num
        {
            changeElectricShock += (turtlebot.map.allYawOrientations[i] - turtlebot.map.allYawOrientations[i - 1]);
        }

        if (changeElectricShock < 0.3)
        {
            noSpinButShouldSpin = true;
        }

    }

    if (turtlebot.velocity.linear != 0)
    {
        float changeElectricShock = 0;
        for (int i = turtlebot.map.allMapPositions.size() - 1; i < turtlebot.map.allMapPositions.size() - numOPoint
        {
            changeElectricShock += sqrt(pow((turtlebot.map.allMapPositions[i].first - turtlebot.map.allMapPositions
        }

        if (changeElectricShock < 0.3)
        {

            noSpinButShouldSpin = true;
        }



    }

    if(noSpinButShouldSpin==true&&secondsElapsed>10)
    {
         //zhuanSLL=1;

    }

}

void OccupancyCallBack(const nav_msgs::OccupancyGrid::ConstPtr &msg)
{
 turtlebot.map.mapData.clear();
    turtlebot.map.usefulMapData.clear();
    int tempHeight = msg->info.height;
    int tempWidth = msg->info.width;

    turtlebot.map.mapHeight = tempHeight;
    turtlebot.map.mapWidth = tempWidth;
    turtlebot.map.resolution = msg->info.resolution;

    int dataArraySize = msg->data.size();

    int tempCounter = 0;


    for (int i = 0; i < tempHeight; i++)
    {

        turtlebot.map.mapData.push_back(std::vector<int>());
        for (int j = 0; j < tempWidth; j++)
        {

            int tempStorageOfIthElement = msg->data[tempCounter];
            turtlebot.map.mapData[i].push_back(tempStorageOfIthElement);

            // if(j<tempWidth-1)
            {
               // myfileMap << tempStorageOfIthElement << ",";
            }
            // else
            {
                // myfile2 << tempStorageOfIthElement  << "],";
            }
            tempCounter += 1;
        }

    }
    // myfile2 << "])\n\n\n";
    int tempTop = tempHeight;
    int tempBottom = 0;
    int tempLeft = tempWidth;
    int tempRight = 0;

    for (int i = 0; i < tempHeight; i++)
    {
        for (int j = 0; j < tempWidth; j++)
        {

            int tempVariable = turtlebot.map.mapData[i][j];

            if (tempVariable != -1)
            {
                if (i < tempTop)
                {
                    tempTop = i;
                }
                if (i > tempBottom)
                {
                    tempBottom = i;
                }
                if (j < tempLeft)
                {
                    tempLeft = j;
                }
                if (j > tempRight)
                {
                    tempRight = j;
                }
            }
        }
    }

    if (tempTop - 3 > 0)
    {
        tempTop -= 3;
    }
    if (tempBottom + 3 < tempHeight)
    {
        tempBottom += 3;
    }
    if (tempLeft - 3 > 0)
    {
        tempLeft -= 3;
    }
    if (tempRight + 3 < tempWidth)
```

```
{
    tempRight += 3;
}


turtlebot.map.originX = ((msg->info.origin.position.x / msg->info.resolution * -1));
turtlebot.map.originY = ((msg->info.origin.position.y / msg->info.resolution * -1));

turtlebot.map.usefulOriginRow = round(turtlebot.map.originY - tempTop) + 1;
turtlebot.map.usefulOriginCol = round(turtlebot.map.originX - tempLeft) + 1;


turtlebot.map.usefulMapRow = tempBottom - tempTop + 1;
turtlebot.map.usefulMapCol = tempRight - tempLeft + 1;
int tempRowCounte2 = 0;
int tempColCounte2 = 0;

for (int i = tempTop; i <= tempBottom; i++)
{
    turtlebot.map.usefulMapData.push_back(std::vector<int>());
    tempColCounte2 = 0;
    for (int j = tempLeft; j <= tempRight; j++)
    {
        turtlebot.map.usefulMapData[tempRowCounte2].push_back(turtlebot.map.mapData[i][j]);
        tempColCounte2 += 1;
    }
    tempRowCounte2 += 1;
}




turtlebot.map.increaseWallGrid.clear();

for (int i = 0; i < turtlebot.map.usefulMapRow; i++)
{
    turtlebot.map.increaseWallGrid.push_back(std::vector<int>());

    for (int j = 0; j < turtlebot.map.usefulMapCol; j++)
    {
        turtlebot.map.increaseWallGrid[i].push_back(turtlebot.map.usefulMapData[i][j]);
    }
}

for (int i = 0; i < turtlebot.map.usefulMapRow; i++)
{
    for (int j = 0; j < turtlebot.map.usefulMapCol; j++)
    {
        if (turtlebot.map.increaseWallGrid[i][j] == 100)
        {

            int hundaoCounter = 0;
            for (int i2 = -turtlebot.map.increaseWallPadding; i2 < turtlebot.map.increaseWallPadding + 1; i2++)
            {
                for (int j2 = -turtlebot.map.increaseWallPadding; j2 < turtlebot.map.increaseWallPadding + 1; j2
                {
                    if (i2 + i < 0 || i2 + i >= turtlebot.map.usefulMapRow || j2 + j < 0 || j2 + j >= turtlebot
                    {
                        int youDummy = 0;
                    }
                    else
                    {
                        if (turtlebot.map.increaseWallGrid[i2 + i][j2 + j] == 100)
                        {

                            hundaoCounter += 1;
                        }
                    }
                }
            }
            if (hundaoCounter > turtlebot.map.increaseWallPadding - 2)
            {
                for (int i2 = -turtlebot.map.increaseWallPadding; i2 < turtlebot.map.increaseWallPadding + 1; i2
                {
                    for (int j2 = -turtlebot.map.increaseWallPadding; j2 < turtlebot.map.increaseWallPadding +
                    {
                        if (i2 + i < 0 || i2 + i >= turtlebot.map.usefulMapRow || j2 + j < 0 || j2 + j >= turtle
                        {
                            int youDummy = 0;
                        }
                        else
                        {
                            if (turtlebot.map.increaseWallGrid[i2 + i][j2 + j] != 100)
                            {

                                turtlebot.map.increaseWallGrid[i2 + i][j2 + j] = 101;
                            }
                        }
                    }
                }
            }
            else
            {
                turtlebot.map.increaseWallGrid[i][j] = 0;
            }
        }
    }
}
for (int i = 0; i < turtlebot.map.usefulMapRow; i++)
{
    for (int j = 0; j < turtlebot.map.usefulMapCol; j++)
    {
        if (turtlebot.map.increaseWallGrid[i][j] == 101)
        {
            turtlebot.map.increaseWallGrid[i][j] = 100;
        }
    }
}

for (int i2 = -turtlebot.map.increaseWallPadding; i2 < turtlebot.map.increaseWallPadding + 1; i2++)
{
    for (int j2 = -turtlebot.map.increaseWallPadding; j2 < turtlebot.map.increaseWallPadding + 1; j2++)
    {
        if (i2 + turtlebot.map.usefulOriginRow < 0 || i2 + turtlebot.map.usefulOriginRow >= turtlebot.map.useful
        {
            int youDummy = 0;
        }
        else
        {

            turtlebot.map.increaseWallGrid[i2 + turtlebot.map.usefulOriginRow][j2 + turtlebot.map.usefulOriginC
        }
    }
}


turtlebot.map.clusterBombMatrix.clear();

for (int i = 0; i < turtlebot.map.usefulMapRow; i++)
{
    turtlebot.map.clusterBombMatrix.push_back(std::vector<int>());
    for (int j = 0; j < turtlebot.map.usefulMapCol; j++)
    {
        turtlebot.map.clusterBombMatrix[i].push_back(turtlebot.map.usefulMapData[i][j]);
    }
}


for (int clusterBombCounter = 0; clusterBombCounter < turtlebot.map.allMapPositionAggr.size(); clusterBombCount
{

    std::pair<int, int> currentBombPair = xyTOij(turtlebot.map.allMapPositionAggr[clusterBombCounter].first, tu
    int i = currentBombPair.first;
    int j = currentBombPair.second;

    for (int i2 = -turtlebot.map.bombRadius; i2 < turtlebot.map.bombRadius + 1; i2++)
    {
        for (int j2 = -turtlebot.map.bombRadius; j2 < turtlebot.map.bombRadius + 1; j2++)
        {
            if (i2 + i < 0 || i2 + i >= turtlebot.map.usefulMapRow || j2 + j < 0 || j2 + j >= turtlebot.map.usef
            {
                int youDummy = 0;
            }
            else
            {
                turtlebot.map.clusterBombMatrix[i2 + i][j2 + j] = 200;
            }
        }
    }
}
```

```
    int numOtwoHundo = 0;
    for (int i = 0; i < turtlebot.map.usefulMapRow; i++)
    {
        for (int j = 0; j < turtlebot.map.usefulMapCol; j++)
        {
            if (turtlebot.map.clusterBombMatrix[i][j] == 200)
            {
                numOtwoHundo += 1;
            }
        }
    }
    turtlebot.map.clusterBombAreaArray.push_back(numOtwoHundo);

    if (turtlebot.map.clusterBombAreaArray.size() > 5)
    {
        turtlebot.map.clusterBombAreaArray.erase(turtlebot.map.clusterBombAreaArray.begin());
    }


    bool tempBool = areaNotChanging();
}



void transforomCallback(const geometry_msgs::TransformStamped::ConstPtr &msg)
{
    float transfomredX = msg->transform.translation.x;
    float transfomredY = msg->transform.translation.y;
    float transfomredYaw = msg->transform.rotation.z;

    turtlebot.odom.tfX = transfomredX;
    turtlebot.odom.tfY = transfomredY;
    turtlebot.odom.tfYaw = transfomredYaw;
}



std::pair<bool, float> angleDifference(float angle1, float angle2)
{

    bool compare; // 0 if angle1 > angle2; 1 if angle1 < angle2;
    float deltaAngle = 0;
    bool direction = 0; // 0 as ccw; 1 as cw
    float path1, path2;

    if (angle1 > angle2)
    {
        compare = 0;
    }
    if (angle1 <= angle2)
    {
        compare = 1;
    }
    if (compare == 0)
    {
        path1 = angle1 - angle2;
        path2 = 2 * M_PI - path1;
        if (path1 < path2 && compare == 0)
        {
            direction = 1;
            deltaAngle = path1;
        }
        if (path1 < path2 && compare == 1)
        {
            direction = 0;
            deltaAngle = path1;
        }
        if (path1 > path2 && compare == 0)
        {
            direction = 0;
            deltaAngle = path2;
        }
        if (path1 > path2 && compare == 1)
        {
            direction = 1;
            deltaAngle = path2;
        }
    }
    else
    {
        path1 = angle2 - angle1;
        path2 = 2 * M_PI - path1;
        if (path1 < path2 && compare == 0)
        {
            direction = 1;
            deltaAngle = path1;
        }

        if (path1 < path2 && compare == 1)
        {
            direction = 0;
            deltaAngle = path1;
        }

        if (path1 > path2 && compare == 0)
        {
            direction = 0;
            deltaAngle = path2;
        }
        if (path1 > path2 && compare == 1)
        {
            direction = 1;
            deltaAngle = path2;
        }
    }
    ros::spinOnce();


    return std::make_pair(direction, deltaAngle);
}
void move2Distance(float targetDistance)
{
    turtlebot.velocity.linear = turtlebot.velocity.linearMax;
    turtlebot.velocity.angular = 0;
    float kp_distance = 1;
    // float ki_distance = 0.01;
    float kd_distance = 0.3;
    // float total_distance = 0;
    float previous_distance = 0;
    float distance_walked = 0;
    float distance = targetDistance;
    float deltaTime = 0;
    float previous_time = 0;
    float previous_velocity = 0;
    float average_velocity = 0;
    float control_signal_distance = 0;

    std::chrono::time_point<std::chrono::system_clock> start;
    start = std::chrono::system_clock::now();
    if (targetDistance > 0)
    {
        while (distance > 0.0001)
        {
            float diff_distance = distance - previous_distance;
            distance = targetDistance - distance_walked;
            control_signal_distance = kp_distance * distance + kd_distance * diff_distance; //+ ki_distance*total_di
            previous_distance = distance;
            // total_distance = total_distance + distance;
            turtlebot.velocity.linear = std::min(control_signal_distance, turtlebot.velocity.linearMax);
            publishVelocity();

            float tsecondsElapsed = std::chrono::duration_cast<std::chrono::seconds>(std::chrono::system_clock::now
            deltaTime = tsecondsElapsed - previous_time;
            previous_time = tsecondsElapsed;
            // average_velocity = (turtlebot.velocity.linearTarget+previous_velocity)/2;
            distance_walked = distance_walked + deltaTime * turtlebot.velocity.linear;
            previous_velocity = turtlebot.velocity.linear;

            ros::spinOnce();
        }
    }
    else if (targetDistance < 0)
    {
        while (distance < -0.0001)
        {
            float diff_distance = distance - previous_distance;
            distance = targetDistance - distance_walked;
            control_signal_distance = kp_distance * distance + kd_distance * diff_distance; //+ ki_distance*total_di
            previous_distance = distance;
            // total_distance = total_distance + distance;
            turtlebot.velocity.linear = std::max(control_signal_distance, -turtlebot.velocity.linearMax);
            publishVelocity();
            float tsecondsElapsed = std::chrono::duration_cast<std::chrono::seconds>(std::chrono::system_clock::now
            deltaTime = tsecondsElapsed - previous_time;
            previous_time = tsecondsElapsed;
            // average_velocity = (turtlebot.velocity.linearTarget+previous_velocity)/2;
            distance_walked = distance_walked + deltaTime * turtlebot.velocity.linear;
            previous_velocity = turtlebot.velocity.linear;
            ros::spinOnce();
        }
    }

    turtlebot.velocity.linear = 0;
    publishVelocity();
    ROS_INFO("Moved to the targeted distance!");
}

void turn2Angle(float target)
{
    std::pair<bool, float> anglePair = angleDifference(turtlebot.odom.goodYaw, target);
    float deltaAngleTotal = anglePair.second;
    float duration = deltaAngleTotal / turtlebot.velocity.angular;
    float deltaTime = 0;
    turtlebot.velocity.linear = 0;
    std::chrono::time_point<std::chrono::system_clock> start;
    start = std::chrono::system_clock::now();
    float kp = 1;
    std::pair<bool, float> deltaAngle = angleDifference(turtlebot.odom.goodYaw, target);

    while (float(deltaAngle.second) > 0.05)
    {

        deltaAngle = angleDifference(turtlebot.odom.goodYaw, target);

        turtlebot.velocity.angular = deltaAngle.second * turtlebot.velocity.angularMax * kp;
        if (deltaAngle.first == 0)
        {
            turtlebot.velocity.angular = turtlebot.velocity.angular;
        }
        else
        {
            turtlebot.velocity.angular = -turtlebot.velocity.angular;
        }
        publishVelocity();
        // ROS_INFO ("Current yaw position is at: %f", yaw);
        deltaTime = std::chrono::duration_cast<std::chrono::seconds>(std::chrono::system_clock::now() - start).coun
        // ROS_INFO("Angular: %f, Linear: %f", turtlebot.velocity.angularTarget, turtlebot.velocity.linearTarget);
        // ROS_INFO ("Current Yaw: %f", turtlebot.odom.yaw);
        // ROS_INFO("Angle Difference: %f", deltaAngle.second);
        ros::spinOnce();
    }
    ROS_INFO("Turned to the targeted angle! ");
    turtlebot.velocity.linear = 0;
    turtlebot.velocity.angular = 0;
    publishVelocity();
}

void turnRadius(float target_angle)
{
    turtlebot.velocity.linear = 0;
    float kp_angle = 1;
    float kd_angle = 0.5;
    float ki_angle = pow(target_angle, 2) * 0.02;
    float angle_turned = 0;
    float angle_remain = 0;
    float previous_angle = 0;
    float delta_time = 0;
    float previous_time = 0;
    float previous_angular_speed = 0;
    float average_angular_speed = 0;
    float total_angle = 0;

    std::chrono::time_point<std::chrono::system_clock> start;
    start = std::chrono::system_clock::now();
    if (target_angle > 0)
    {
        while (angle_turned < target_angle)
        {
            float diff_angle = angle_remain - previous_angle;
            angle_remain = target_angle - angle_turned;
            float control_signal_angle = kp_angle * angle_remain + kd_angle * diff_angle + ki_angle * total_angle;
            previous_angle = angle_remain;
            total_angle = total_angle + angle_remain;
            turtlebot.velocity.angular = std::min(control_signal_angle, turtlebot.velocity.angularMax);
            publishVelocity();
            float tsecondsElapsed = std::chrono::duration_cast<std::chrono::seconds>(std::chrono::system_clock::now
            delta_time = tsecondsElapsed - previous_time;
            previous_time = tsecondsElapsed;
            // average_angular_speed = (previous_angular_speed + turtlebot.velocity.angular)/2;
            angle_turned = angle_turned + delta_time * turtlebot.odom.angluarZ;
            // previous_angular_speed = turtlebot.velocity.angular;
            ros::spinOnce();
        }
    }
    else if (target_angle < 0)
    {
        while (angle_turned > target_angle)
        {
            float diff_angle = angle_remain - previous_angle;
            angle_remain = target_angle - angle_turned;
            float control_signal_angle = kp_angle * angle_remain + kd_angle * diff_angle + ki_angle * total_angle;
            previous_angle = angle_remain;
            total_angle = total_angle + angle_remain;
            turtlebot.velocity.angular = std::max(control_signal_angle, turtlebot.velocity.angularMax * -1);
            publishVelocity();
            float tsecondsElapsed = std::chrono::duration_cast<std::chrono::seconds>(std::chrono::system_clock::now
            delta_time = tsecondsElapsed - previous_time;
            previous_time = tsecondsElapsed;
            // average_angular_speed = (previous_angular_speed + turtlebot.velocity.angular)/2;
            angle_turned = angle_turned + delta_time * turtlebot.odom.angluarZ;
            // previous_angular_speed = turtlebot.velocity.angular;
            ros::spinOnce();
        }
    }
    else
    {
        turtlebot.velocity.angular = 0;
        publishVelocity();
    }
    turtlebot.velocity.linear = 0;
    turtlebot.velocity.angular = 0;
    publishVelocity();
}

bool areaNotChanging()
{
    float totalChange = 0;
    for (int i = 1; i < turtlebot.map.clusterBombAreaArray.size(); i++)
    {
        totalChange += turtlebot.map.clusterBombAreaArray[i] - turtlebot.map.clusterBombAreaArray[i - 1];
    }

    if (totalChange < 10)
    {
        return true;
    }
    else
    {
        return false;
    }
}

std::pair<int, int> xyTOij(float x, float y)
{
    float x2 = x / turtlebot.map.resolution;
    float y2 = y / turtlebot.map.resolution;
    // ROS_INFO("x2 %f  and y2 %f",x2,y2);
    int x3 = round(x2);
    int y3 = round(y2);


    std::pair<int, int> xyIJReturn(turtlebot.map.usefulOriginRow + y3, turtlebot.map.usefulOriginCol + x3);
    return xyIJReturn;
}
std::pair<float, float> ijTOxy(int i, int j)
{
}


float findEuclideanDistance(float i1, float j1, float i2, float j2)
{
    float tempStroasf = pow((i1 - i2), 2);
    float tempsfsec2 = pow((j1 - j2), 2);
    return sqrt(tempStroasf + tempsfsec2);
```

```
}

void pause3seconds()
{

    if (secondsElapsed - presentSecond < 3)
    {
        turtlebot.velocity.linear = 0;
        turtlebot.velocity.angular = 0;
    }
    else
    {
        stopThree = false;
    }
}


void turn360()
{

    while (turtlebot.odom.yaw < previousYaw)
    {
        turtlebot.odom.yaw  += 2 * M_PI;
    }
    diffrenceYaw = fabs(turtlebot.odom.yaw  - initialYaw);

    if ((2 * M_PI - diffrenceYaw) > 0.1)
    {

        turtlebot.velocity.linear = 0;
        turtlebot.velocity.angular = M_PI / 12;
        previousYaw = turtlebot.odom.yaw ;
    }
    else
    {

        zhuan360Avoid=100;
        previousYaw = 0;
        zhuanSLL = false;
        stopThree = true;
        presentSecond = secondsElapsed;
        turtlebot.velocity.angular = 0;
        turtlebot.velocity.linear = 0;

    }
}



inline void publishVelocity()
{
    ros::NodeHandle nh;
    ros::Publisher vel_pub = nh.advertise<geometry_msgs::Twist>("cmd_vel_mux/input/teleop", 1);
    geometry_msgs::Twist vel_msgs2;
    vel_msgs2.linear.x = turtlebot.velocity.linear;
    vel_msgs2.angular.z = turtlebot.velocity.angular; // if angular is positive --> counter-clockwise; if angular i
    vel_pub.publish(vel_msgs2);
    ros::spinOnce();
}

void turndesired()
{
    float starting_time = std::chrono::duration_cast<std::chrono::seconds>(std::chrono::system_clock::now() - start
    float time_run = 0;
    while (minLaserDist - previousDistance > 0 || minLaserDist < 1 || minLaserDist > 6)
    {
        float time_now =std::chrono::duration_cast<std::chrono::seconds>(std::chrono::system_clock::now() - start).
        time_run = time_now-starting_time;
        if (time_run > 60)
        {
            break;
        }
        previousDistance = minLaserDist;

        turtlebot.velocity.linear = 0;
        float tempSettter=M_PI/6;
        if (goL==1)
        {
            turtlebot.velocity.angular = tempSettter;
        }
        else if (goL==0)
        {
            turtlebot.velocity.angular = -tempSettter;
        }
        publishVelocity();

    }


    turtlebot.velocity.angular = 0;
    turtlebot.velocity.linear = 0;
    publishVelocity();
    pause3seconds();
}


void no_backward_turndesired()
{
    diffrenceYaw = 0;
    initialYaw = turtlebot.odom.yaw;


    while ((2 * M_PI - diffrenceYaw) > 0.1)
    {
       int dummy=0;
        if (goL==1)
        {
            while (turtlebot.odom.yaw < previousYaw && (fabs(turtlebot.odom.yaw - initialYaw) > 0.01))
            {
                int oissef=0;
                turtlebot.odom.yaw += 2 * M_PI;
                int secsecsecsc=0;
            }
        }
        else
        {
            while (turtlebot.odom.yaw > previousYaw && (fabs(turtlebot.odom.yaw - initialYaw) > 0.01))
            {
                turtlebot.odom.yaw -= 2 * M_PI;
                int asefesfesfss=0;
            }
        }
        diffrenceYaw = fabs(turtlebot.odom.yaw - initialYaw);

        if (fabs(M_PI - diffrenceYaw) < M_PI / 6 || minLaserDist < 1 || minLaserDist > 6)
        {
            previousDistance = minLaserDist;
            turtlebot.velocity.linear = 0;
            if (goL==1)
            {
                turtlebot.velocity.angular = M_PI / 8;
            }
            else if (goL==0)
            {
                turtlebot.velocity.angular = -M_PI / 8;
            }
            previousYaw = turtlebot.odom.yaw;
            publishVelocity();

            presentSecond = std::chrono::duration_cast<std::chrono::seconds>(std::chrono::system_clock::now() - sta

        }
        else
        {
            turtlebot.velocity.angular = 0;
            turtlebot.velocity.linear = 0;
            publishVelocity();
            pause3seconds();
            previousYaw = turtlebot.odom.yaw;
            break;
        }
    }
    if ((2 * M_PI - diffrenceYaw) <= 0.1)
    {

        turtlebot.velocity.angular = 0;
        turtlebot.velocity.linear = 0;
        publishVelocity();
        previousYaw = turtlebot.odom.yaw;
        initialDistance = minLaserDist;
        previousDistance = initialDistance;
        turndesired();
    }
}

void frontierExplorationMarkNegative3()
{
    int startingTimeHere=secondsElapsed+30;
    turtlebot.map.backEndGrid.clear();
    for (int i = 0; i < turtlebot.map.usefulMapRow; i++)
    {
        turtlebot.map.backEndGrid.push_back(std::vector<int>());
        for (int j = 0; j < turtlebot.map.usefulMapCol; j++)
        {
            turtlebot.map.backEndGrid[i].push_back(-1);
        }
    }

    int tempadjacencyCounter = 0;
    for (int i = 0; i < turtlebot.map.usefulMapRow; i++)
    {
        for (int j = 0; j < turtlebot.map.usefulMapCol; j++)
        {
            if (i == 0 || j == 0 || i == turtlebot.map.usefulMapRow - 1 || j == turtlebot.map.usefulMapCol - 1)
            {
                int lo3 = 32;
            }
            else
            {

                tempadjacencyCounter = 0;
                if (turtlebot.map.increaseWallGrid[i][j] == 0)
                {
                    if (turtlebot.map.increaseWallGrid[i - 1][j - 1] == -1)
                    {
                        tempadjacencyCounter += 1;
                    }
                    if (turtlebot.map.increaseWallGrid[i - 1][j] == -1)
                    {
                        tempadjacencyCounter += 1;
                    }
                    if (turtlebot.map.increaseWallGrid[i - 1][j + 1] == -1)
                    {
                        tempadjacencyCounter += 1;
                    }
                    if (turtlebot.map.increaseWallGrid[i][j - 1] == -1)
                    {
                        tempadjacencyCounter += 1;
                    }
                    if (turtlebot.map.increaseWallGrid[i][j + 1] == -1)
                    {
                        tempadjacencyCounter += 1;
                    }
                    if (turtlebot.map.increaseWallGrid[i + 1][j - 1] == -1)
                    {
                        tempadjacencyCounter += 1;
                    }
                    if (turtlebot.map.increaseWallGrid[i + 1][j] == -1)
                    {
                        tempadjacencyCounter += 1;
                    }
                    if (turtlebot.map.increaseWallGrid[i + 1][j + 1] == -1)
                    {
                        tempadjacencyCounter += 1;
                    }
                    turtlebot.map.backEndGrid[i][j] = tempadjacencyCounter * 2;
                }
                else if (turtlebot.map.increaseWallGrid[i][j] > 0)
                {

                    turtlebot.map.backEndGrid[i][j] = -turtlebot.map.increaseWallGrid[i][j];
                }
            }
        }
    }

    std::vector<std::vector<float>> frontierList;
    int aggregatePixelsSize = 2;
    std::vector<std::pair<int, int>> frontierListTotal;

    for (int i = 0; i < turtlebot.map.usefulMapRow; i++)
    {
        for (int j = 0; j < turtlebot.map.usefulMapCol; j++)
        {
            if (turtlebot.map.backEndGrid[i][j] > 0)
            {
                std::pair<int, int> tempfronefsLList(i, j);
                frontierListTotal.push_back(tempfronefsLList);
            }
        }
    }

    for (int i = 0; i < turtlebot.map.usefulMapRow; i++)
    {

        for (int j = 0; j < turtlebot.map.usefulMapCol; j++)
        {

            if (turtlebot.map.backEndGrid[i][j] > 0)
            {

                float aggregationCounter = 0;
                for (int i2 = -aggregatePixelsSize; i2 < aggregatePixelsSize + 1; i2++)
                {
                    for (int j2 = -aggregatePixelsSize; j2 < aggregatePixelsSize + 1; j2++)
                    {
                        if (i2 + i < 0 || i2 + i >= turtlebot.map.usefulMapRow || j2 + j < 0 || j2 + j >= turtlebot
                        {

                            int YOUSUCK = 3;
                        }
                        else
                        {
                            if (turtlebot.map.backEndGrid[i + i2][j + j2] > 0)
                            {
                                aggregationCounter += turtlebot.map.backEndGrid[i + i2][j + j2];
                                turtlebot.map.backEndGrid[i + i2][j + j2] = -420;
                            }
                            if (turtlebot.map.backEndGrid[i + i2][j + j2] < 0 && turtlebot.map.backEndGrid[i + i2][
                            {
                                aggregationCounter += turtlebot.map.backEndGrid[i + i2][j + j2] % 20;
                            }
                        }
                    }

                    turtlebot.map.backEndGrid[i][j] = int(aggregationCounter) + 1;

                    std::vector<float> tempfronLList = {float(i), float(j), std::round(aggregationCounter)};
                    frontierList.push_back(tempfronLList);
                }
            }
        }
    }



    for (int i = 0; i < turtlebot.map.usefulMapRow; i++)
    {
        for (int j = 0; j < turtlebot.map.usefulMapCol; j++)
        {
            if (turtlebot.map.backEndGrid[i][j] == -420)
            {
                turtlebot.map.backEndGrid[i][j] = 0;
            }
        }
    }

    turtlebot.map.frontierListe.clear();

    float currentLocationX = turtlebot.odom.goodPosX;
    float currentLocationY = turtlebot.odom.goodPosY;
    std::pair<int, int> currentLocationPair = xyTOij(currentLocationX, currentLocationY);
    turtlebot.map.currentLocationX = currentLocationPair.first;
    turtlebot.map.currentLocationY = currentLocationPair.second;
    std::pair<float, float> totalFrontierVector(0.0, 0.0);

    int scalra = 1;
    int sclrab2;
    int sclarc = -0.1;

    for (int i = 0; i < frontierList.size(); i++)
    {
        std::pair<int, int> tempFrontierPushPair(frontierList[i][0], frontierList[i][1]);
```

```
        turtlebot.map.frontierListe.push_back(tempFrontierPushPair);

        float divideUnitVector = sqrt(pow(currentLocationPair.first - frontierList[i][0], 2) + pow(currentLocationP
        totalFrontierVector.first += (frontierList[i][0] - currentLocationPair.first) / divideUnitVector * frontierI
        totalFrontierVector.second += (frontierList[i][1] - currentLocationPair.second) / divideUnitVector * fronti

        frontierList[i][2] += scalra * divideUnitVector;
    }

    std::pair<int, int> maximumViablePair;

    float maximumDistance = -10000;



    std::vector<std::vector<int>> bfsArray = findBFS(turtlebot.map.increaseWallGrid, turtlebot.map.usefulOriginRow,

    if (bfsArray.size() > 0)
    {
        for (int findViableFroCounter = 0; findViableFroCounter < frontierList.size(); findViableFroCounter++)
        {
            if (bfsArray[int(frontierList[findViableFroCounter][0])][int(frontierList[findViableFroCounter][1])] !=
            {
                float tempDistance = frontierList[findViableFroCounter][2];

                if (tempDistance > maximumDistance)
                {
                    maximumDistance = tempDistance;
                    maximumViablePair = turtlebot.map.frontierListe[findViableFroCounter];
                }
            }
        }
    }



    std::vector<std::pair<int, int>> testsa;
    int i2 = maximumViablePair.first;
    int j2 = maximumViablePair.second;



    if (bfsArray.size() > 0 && bfsArray[i2][j2] != 0)
    {
        while (!(i2 == turtlebot.map.usefulOriginRow && j2 == turtlebot.map.usefulOriginCol))
        {

            int parente = bfsArray[i2][j2];
            int nextI;
            int nextJ;
            if (parente == bfsLeft)
            {
                nextI = i2;
                nextJ = j2 - 1;
            }
            else if (parente == bfsRight)
            {
                nextI = i2;
                nextJ = j2 + 1;
            }
            else if (parente == bfsUp)
            {
                nextI = i2 - 1;
                nextJ = j2;
            }
            else if (parente == bfsDown)
            {
                nextI = i2 + 1;
                nextJ = j2;
            }
            else if (parente == bfsUpLeft)
            {
                nextI = i2 - 1;
                nextJ = j2 - 1;
            }
            else if (parente == bfsUpRight)
            {
                nextI = i2 - 1;
                nextJ = j2 + 1;
            }
            else if (parente == bfsDownLeft)
            {
                nextI = i2 + 1;
                nextJ = j2 - 1;
            }
            else if (parente == bfsDownRight)
            {
                nextI = i2 + 1;
                nextJ = j2 + 1;
            }
            std::pair<int, int> tempParis(i2, j2);
            testsa.push_back(tempParis);

            i2 = nextI;
            j2 = nextJ;
        }
    }

    std::pair<int, int> tempParis(i2, j2);
    testsa.push_back(tempParis);

    if (testsa.size() <= 1)
    {
        int doNothing = 0;
    }
    else if (testsa.size() < 6)
    {
        std::reverse(testsa.begin(), testsa.end());
        std::pair<int, int> currentPosPair = xyTOij(turtlebot.odom.goodPosX, turtlebot.odom.goodPosY);
        float tempIHolder = testsa.back().first - currentPosPair.first;
        float tempJHolder = testsa.back().second - currentPosPair.second;
        float newX = tempJHolder;
        float newY = tempIHolder;

        float targetYaw;
        if (newY > 0)
        {
            targetYaw = atan2(newY, newX);
        }
        else
        {
            targetYaw = M_PI - atan2(newY, newX);
        }

        turn2Angle(targetYaw);

        turtlebot.velocity.linear = turtlebot.velocity.linearMax * 0.3;
        turtlebot.velocity.angular = 0;
        publishVelocity();
        int dummyCar = 0;
        float tolerance;
        while (tolerance > 2)
        {
            currentPosPair = xyTOij(turtlebot.odom.goodPosX, turtlebot.odom.goodPosY);
            tolerance = sqrt(pow(testsa.back().first - currentPosPair.first, 2) + pow(testsa.back().second - current
        }
    }
    else
    {

        std::reverse(testsa.begin(), testsa.end());

        int currrentPoint = 0;
        int finalPoint = testsa.size() - 1;
        int tolerancePoint = 2;
        int toGoPoint = 4;

        std::pair<int, int> currentPosPair = xyTOij(turtlebot.odom.goodPosX, turtlebot.odom.goodPosY);
        float tempIHolder = testsa[toGoPoint].first - currentPosPair.first;
        float tempJHolder = testsa[toGoPoint].second - currentPosPair.second;
        float newX = tempJHolder;
        float newY = tempIHolder;

        float targetYaw;
        if (newX >= 0)
        {
            targetYaw = atan2(newY, newX);
        }
        else if (newY >= 0)
        {
            targetYaw = M_PI + atan2(newY, newX);
        }
        else
        {
            targetYaw = -M_PI + atan2(newY, newX);
        }

        turn2Angle(targetYaw);

        turtlebot.velocity.linear = turtlebot.velocity.linearMax * 0.3;
        turtlebot.velocity.angular = 0;
        publishVelocity();
        float tolerance;

        while ((toGoPoint < finalPoint - 3)&&startingTimeHere>secondsElapsed&&turtlebot.bumper.any_bumper_pressed==
        {
            secondsElapsed = std::chrono::duration_cast<std::chrono::seconds>(std::chrono::system_clock::now() - st
            currentPosPair = xyTOij(turtlebot.odom.goodPosX, turtlebot.odom.goodPosY);
            tolerance = sqrt(pow(testsa[tolerancePoint].first - currentPosPair.first, 2) + pow(testsa[tolerancePoint



            if (tolerance < 4)
            {
                currrentPoint += 2;
                tolerancePoint += 2;
                toGoPoint += 2;


                currentPosPair = xyTOij(turtlebot.odom.goodPosX, turtlebot.odom.goodPosY);
                tempIHolder = testsa[toGoPoint].first - currentPosPair.first;
                tempJHolder = testsa[toGoPoint].second - currentPosPair.second;
                newX = tempJHolder;
                newY = tempIHolder;

                if (newX >= 0)
                {
                    targetYaw = atan2(newY, newX);
                }
                else if (newY >= 0)
                {
                    targetYaw = M_PI + atan2(newY, newX);
                }
                else
                {
                    targetYaw = -M_PI + atan2(newY, newX);
                }
            }
            std::pair<bool, float> howmuchToTUrn = angleDifference(turtlebot.odom.goodYaw, targetYaw);
            if (howmuchToTUrn.first == CCW)
            {
                turtlebot.velocity.angular = std::min(0.37 * fabs(howmuchToTUrn.second), turtlebot.velocity.angularM
            }
            else
            {
                turtlebot.velocity.angular = std::max(-0.37 * fabs(howmuchToTUrn.second), -turtlebot.velocity.angula
            }

            publishVelocity();
        }
        turtlebot.velocity.linear = 0;
        turtlebot.velocity.angular = 0;
        publishVelocity();


    }

    // float newX = totalFrontierVector.second;
    // float newY = totalFrontierVector.first;

    // float deterimineTurnAngle;
    // if (totalFrontierVector.first > 0)
    // {
    //     deterimineTurnAngle = atan2(newY, newX);
    // }
    // else
    // {
    //     deterimineTurnAngle = M_PI - atan2(newY, newX);
    // }
    turtlebot.velocity.linear = 0;
    turtlebot.velocity.angular = 0;
    publishVelocity();
```

```
}

void caiInitSpin()
{
    turtlebot.velocity.linear = 0;
    turtlebot.velocity.angular = 0;
    publishVelocity();
    int dummyVar = 0;
    while (dummyVar < 10000000)
    {
        dummyVar += 1;
    }
    turtlebot.velocity.linear = 0;
    turtlebot.velocity.angular = turtlebot.velocity.angularMax * 0.3;
    publishVelocity();
    while (turtlebot.odom.yaw > DEG2RAD(-5))
    {
        publishVelocity();
    }
    while (turtlebot.odom.yaw < DEG2RAD(-10))
    {
        publishVelocity();
    }
    turtlebot.velocity.linear = 0;
    turtlebot.velocity.angular = 0;
    publishVelocity();
}

int seenAngleBefore(float wantYaw)
{

    if (turtlebot.map.allYawOrientations.size() < 100)
    {
        return -1;
    }
    float wantYawInDegres = wantYaw;
    int lastFewYaws = int(turtlebot.map.allYawOrientations.size() * 0.1);
    int numOfOneEightys = 0;
    float averageFloatCounter = 0;

    int countNumberOfTimesLazu = 0;
    std::vector<float> tempSortArray;

    // ROS_INFO("first: %f second: %f",turtlebot.map.allYawOrientations.size() - 1, turtlebot.map.allYawOrientations
    for (int i = turtlebot.map.allYawOrientations.size() - 1; i > turtlebot.map.allYawOrientations.size() - lastFewY
    {
        tempSortArray.push_back(turtlebot.map.allYawOrientations[i]);
        countNumberOfTimesLazu += 1;
        if (turtlebot.map.allYawOrientations[i] > DEG2RAD(175) || turtlebot.map.allYawOrientations[i] < DEG2RAD(-17
        {
            numOfOneEightys += 1;
        }
        averageFloatCounter += turtlebot.map.allYawOrientations[i];
    }



    std::sort(tempSortArray.begin(), tempSortArray.end());
    float minYaw = tempSortArray.front();
    float maxYaw = tempSortArray.back();
    averageFloatCounter = averageFloatCounter / countNumberOfTimesLazu;

    float biggestDiffrenceBetwenYaw = maxYaw - minYaw;

    float dontWantTurnBackAngle;

    if (numOfOneEightys < int(countNumberOfTimesLazu * 0.6))
    {
        if (biggestDiffrenceBetwenYaw > DEG2RAD(30))
        {
            return -1;
        }
        if (fabs(wantYawInDegres - averageFloatCounter) > DEG2RAD(5))
        {
            return 0;
        }
        else
        {
            return 1;
        }
    }
    else
    {

        if (wantYawInDegres < DEG2RAD(174) && wantYawInDegres > DEG2RAD(-174))
        {
            return 0;
        }
        else
        {
            return 1;
        }
    }

    return -1;
}

std::vector<std::vector<int>> findBFS(std::vector<std::vector<int>> mat, int source_i, int source_j)
{

    int row = mat.size();
    std::vector<std::vector<int>> directionArray;
    if (row == 0)
    {
        return directionArray;
    }
    int col = mat[0].size();
    if (col == 0)
    {
        return directionArray;
    }

    float dist[row][col];

    for (int i = 0; i < row; i++)
    {
        directionArray.push_back(std::vector<int>());
        for (int j = 0; j < col; j++)
        {
            directionArray[i].push_back(0);
            dist[i][j] = INT_MAX;
        }
    }

    std::queue<std::pair<int, int>> q;
    q.push(std::make_pair(source_i, source_j));

    dist[source_i][source_j] = 0;


    while (!q.empty())
    {

        int x = q.front().first;

        int y = q.front().second;

        q.pop();

        if (y - 1 >= 0 && mat[x][y - 1] == 0)
        {

            if (dist[x][y] + 1 < dist[x][y - 1])
            {
                dist[x][y - 1] = dist[x][y] + 1;
                q.push(std::make_pair(x, y - 1));
                directionArray[x][y - 1] = bfsRight;
            }
        }


        if (y + 1 < col && mat[x][y + 1] == 0)
        {

            if (dist[x][y] + 1 < dist[x][y + 1])
            {
                dist[x][y + 1] = dist[x][y] + 1;
                q.push(std::make_pair(x, y + 1));
                directionArray[x][y + 1] = bfsLeft;
            }
        }
        if (x - 1 >= 0 && mat[x - 1][y] == 0)
        {
            if (dist[x][y] + 1 < dist[x - 1][y])
            {
                dist[x - 1][y] = dist[x][y] + 1;
                q.push(std::make_pair(x - 1, y));
                directionArray[x - 1][y] = bfsDown;
            }
        }

        if (x + 1 < row && mat[x + 1][y] == 0)
        {

            if (dist[x][y] + 1 < dist[x + 1][y])
            {
                dist[x + 1][y] = dist[x][y] + 1;
                q.push(std::make_pair(x + 1, y));
                directionArray[x + 1][y] = bfsUp;
            }
        }
        if (y - 1 >= 0 && x - 1 >= 0 && mat[x - 1][y - 1] == 0)
        {

            if (dist[x][y] + sqrt(2) < dist[x - 1][y - 1])
            {
                dist[x - 1][y - 1] = dist[x][y] + sqrt(2);
                q.push(std::make_pair(x - 1, y - 1));
                directionArray[x - 1][y - 1] = bfsDownRight;
            }
        }
        if (y - 1 >= 0 && x + 1 < row && mat[x + 1][y - 1] == 0)
        {

            if (dist[x][y] + sqrt(2) < dist[x + 1][y - 1])
            {
                dist[x + 1][y - 1] = dist[x][y] + sqrt(2);
                q.push(std::make_pair(x + 1, y - 1));
                directionArray[x + 1][y - 1] = bfsUpRight;
            }
        }
        if (y + 1 < col && x - 1 >= 0 && mat[x - 1][y + 1] == 0)
        {

            if (dist[x][y] + sqrt(2) < dist[x - 1][y + 1])
            {
                dist[x - 1][y + 1] = dist[x][y] + sqrt(2);
                q.push(std::make_pair(x - 1, y + 1));
                directionArray[x - 1][y + 1] = bfsDownLeft;
            }
        }
        if (y + 1 < col && x + 1 < row && mat[x + 1][y + 1] == 0)
        {

            if (dist[x][y] + sqrt(2) < dist[x + 1][y + 1])
            {
                dist[x + 1][y + 1] = dist[x][y] + sqrt(2);
                q.push(std::make_pair(x + 1, y + 1));
                directionArray[x + 1][y + 1] = bfsUpLeft;
            }
        }
    }

    return directionArray;
}


int main(int argc, char **argv)
{

    start = std::chrono::system_clock::now();

    ros::init(argc, argv, "maze_explorer2");
    ros::NodeHandle nh;
    ros::Subscriber bumper_sub = nh.subscribe("mobile_base/events/bumper", 10, &bumperCallback);
    ros::Subscriber laser_sub = nh.subscribe("scan", 10, &laserCallback);
    ros::Subscriber odom = nh.subscribe("odom", 1, &odomCallback);
    ros::Subscriber occupancy_grid = nh.subscribe("map", 1, &OccupancyCallBack);

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

```
ros::Subscriber subscribeToTechnoblade = nh.subscribe("/odom2map_transform", 1, &transforomCallback);
ros::Rate loop_rate(10);

turtlebot.velocity.linear = 0.0;
turtlebot.velocity.angular = 0.0;


int controlFlag = 0;
int goBackwardSpin = 100;
std::chrono::time_point<std::chrono::system_clock> startCai;
startCai = std::chrono::system_clock::now();
uint64_t secondsElapsedCai = 0;

int tempFlagsef = 0;
std::pair<float, float> tempFLoat1(0.0, 0.0);
turtlebot.map.allMapPositionAggr.push_back(tempFLoat1);


int tempCounterfortest = 0;
int initialSpinTwiceFlag = 0;

int laserFlag = 0;

int currentYawFlag = 0;
float currentYawNeed1;

while (ros::ok() && secondsElapsed <= 900)
{
    ros::spinOnce();

    if (turtlebot.odom.posXAssim.size() > 21)
    {
        turtlebot.odom.posXAssim.erase(turtlebot.odom.posXAssim.begin());
        turtlebot.odom.posYAssim.erase(turtlebot.odom.posYAssim.begin());
        turtlebot.odom.rotateZuo.erase(turtlebot.odom.rotateZuo.begin());
    }

    if (stopThree)
    {
        pause3seconds();
        goto markPond;
    }
    if (zhuanSLL)
    {

        turn360();
        if(zhuan360Avoid>0)
        {
            zhuan360Avoid = zhuan360Avoid - 10;

            turtlebot.velocity.linear=-0.05;
        }

        goto markPond;
    }


    if (isTurning == 0)
    {
        initialYaw = turtlebot.odom.yaw;
    }

    if (secondsElapsed % 101 == 1)
    {
        if (secondsElapsed > 2)
        {
            zhuanSLL = true;

            goto markPond;
        }
    }


    if (turtlebot.bumper.fronBumpPresed == 1)
    {

        turtlebot.velocity.linear = -0.05;
        turtlebot.velocity.angular = backStreetBous * M_PI / 6;

        goBackwardSpin -= 8;
```

```
            if (goBackwardSpin < 0)
            {
                goBackwardSpin = 100;
                turtlebot.bumper.fronBumpPresed = 0;

                initialDistance = minLaserDist;
                previousDistance = initialDistance;
                backStreetBous = backStreetBous * 1;
                no_backward_turndesired();
            }

            goto markPond;
        }

        else if (turtlebot.bumper.lefttBumpPresed == 1)
        {

            turtlebot.velocity.linear = -0.05;
            turtlebot.velocity.angular = -M_PI / 6 ;
            goBackwardSpin -= 10;


            if (goBackwardSpin < 0)
            {
                goBackwardSpin = 100;
                turtlebot.bumper.lefttBumpPresed = 0;
            }
        }

        else if (turtlebot.bumper.rightBumpPresed == 1)
        {

            turtlebot.velocity.linear = -0.05;
            turtlebot.velocity.angular = M_PI / 6;
            goBackwardSpin -= 10;



            if (goBackwardSpin < 0)
            {
                goBackwardSpin = 100;
                turtlebot.bumper.rightBumpPresed = 0;
            }
        }

        else if (turtlebot.bumper.any_bumper_pressed == 0)
        {

            if (minLaserDist > 0.5 && minLaserDist < 6)
            {
                if (middelLaserDist > 0.69)
                {

                    turtlebot.velocity.linear = 0.2;
                    turtlebot.velocity.angular = 0.3 * turtlebot.laser.LRDiff;
                }
                else
                {
                    turtlebot.velocity.linear = 0.1;
                    turtlebot.velocity.angular = 0.3 * turtlebot.laser.LRDiff;
                }

                if (turtlebot.velocity.linear > 0.2)
                {
                    turtlebot.velocity.linear = 0.2;
                    if (initialSpinTwiceFlag == 0&&secondsElapsed>850&&minLaserDist==50)
                    {
                        initialSpinTwiceFlag = 1;


                        caiInitSpin();
                        frontierExplorationMarkNegative3();
                        frontierExplorationMarkNegative3();
                        frontierExplorationMarkNegative3();
                    }
                }
                if (turtlebot.velocity.angular < turtlebot.velocity.angularMax / 1.5 && turtlebot.velocity.angular >
                {
                    int youscjsdd = 0;
                }
                else
                {
                    turtlebot.velocity.angular = 0;
                }
            }

            else if (minLaserDist <= 0.5 || minLaserDist >= 6)
            {
                if (secondsElapsed > 1)
                {

                    int chess = 0;int initialSpinTwiceFlag = 0;
                    if (turtlebot.odom.posXAssim.size() > 0)
                    {
                        chess = verifySimi(turtlebot.odom.posX , turtlebot.odom.posY);

                        if (chess != -1)
                        {
                            goL = (goL == turtlebot.odom.rotateZuo[chess] ? !goL : goL);
                        }
                    }

                    turtlebot.odom.posXAssim.push_back(turtlebot.odom.posX);
                    turtlebot.odom.posYAssim.push_back(turtlebot.odom.posY);
                    turtlebot.odom.rotateZuo.push_back(goL);

                    initialDistance = minLaserDist;
                    previousDistance = initialDistance;
                    lasterror = 0;
                    initialYaw = turtlebot.odom.yaw;
                    previousYaw = turtlebot.odom.yaw;
                    no_backward_turndesired();
                    goto markPond;
                }
            }
        }

    markPond:

        publishVelocity();

        isTurning = true;
        if (turtlebot.velocity.angular == 0)
        {
            isTurning = false;
        }
        goL = dist(mersenneTwister);



        secondsElapsed = std::chrono::duration_cast<std::chrono::seconds>(std::chrono::system_clock::now() - start)
        secondsElapsedCai = std::chrono::duration_cast<std::chrono::seconds>(std::chrono::system_clock::now() - sta
        loop_rate.sleep();
    }


    turtlebot.velocity.linear=0;
    turtlebot.velocity.angular=0;
    publishVelocity();
    return 0;
}
```

MIE443 Mechatronics Systems: Design and Integration

Contest 2 Report

Finding Objects of Interest in an Environment

Qilong (Jerry) Cheng 1003834103
Maryam Hosseini 1004074814
Yuntao Cai 1006519581
Yicheng Wang 10039155551

March 22, 2022

Contents

1 Problem Definition and Design Objectives

The primary objective of Contest 2, *Finding Objects of Interest in an Environment*, is to develop an algorithm in which the simulated TurtleBot navigates a 6x6 m2 environment in order to find and identify ten objects that are placed at different locations in the environment. These boxes are identifiable by an image on the side, corresponding to one of the provided tags, The coordinates of each object as well as the tag images are provided in advance. The TurtleBot is to return to its starting position upon successful identification of the ten objects.

1.1 Requirements and Constraints

There are some design requirements that must be considered when developing the algorithm for Contest 2. Some of these design requirements are logistical in nature, that is, they are required of the teams by the course [1]. The requirements and constraints are listed below:

1. The contest map is a flat 6x6 m2 environment with no obstacles other than the ten objects that are to be examined by the TurtleBot as shown in Figure 1. Each object is represented by a 50x32x40 cm3 box with a tag on one side. Object coordinates with respect to the TurtleBot coordinate frame will be provided, as seen in the practice world below.

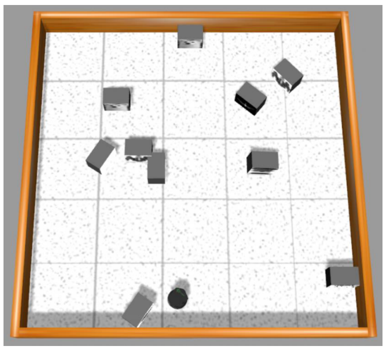


Figure 1: Practice World Layout with 10 Objects

2. Team is provided with three sets of data: (1) an empty 2D map of the contest without any objects that is generated with gmapping, (2) object coordinates within the environment, and (3) a dataset of image tags that used in the contest.
3. The location of objects in the environment are identified by the coordinates of their centres as well as their orientations, (x,y,φ), where φ is measured about the z-axis relative to the origin of the TurtleBot. Locations are obtained from the objects' local frames relative to the TurtleBot coordinate frame at its origin, which is determined by the TurtleBot's starting location when the map was created, as seen in Figure 2 and 3.
4. Objects are distinguishable using a dataset containing 15 potential high contrast images with unique features that are provided to teams.

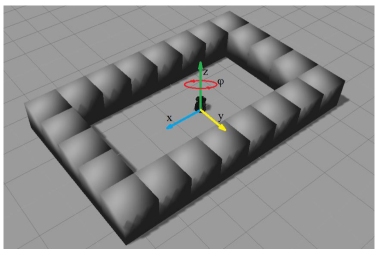


Figure 2: Practice World Coordinate Frame

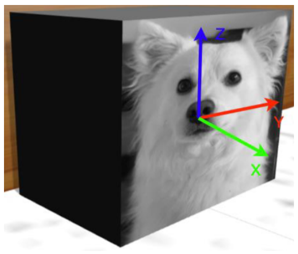


Figure 3: Object Local Frame with an Image Tag

5. A new set of object locations are provided for contest 2. Teams must ensure their algorithms are robust to be able to handle such dynamic changes to the locations of the objects.
6. The starting location for the simulated TurtleBot is randomly selected by the course instructors.
7. The TurtleBot is to complete traversing and identifying the ten objects on scene before returning to its starting location and indicating its completion status within a maximum time limit of 8 minutes.
8. The simulated TurtleBot must utilize the provided navigation library to drive the robot base safely. It is also to utilize the RGB camera on the Kinect sensor to perform SURF feature detection in order to find and identify the image tag on objects.
9. Out of the ten objects placed in this contest, one or two will have duplicated tags, and one or two will not have a tag (i.e. blank tag), and the remaining are taken from the set of images provided to the teams.
10. The algorithm must output a file with all the tags it has found and their respective location in the exact order they are found, once the simulated TurtleBot has traversed to all objects and has returned to its starting location.
11. The team is to optimize the robot's path to find objects in order to reduce run time as much as possible.

2 Strategy

Overall, there are two main tasks for the TurtleBot to complete during Contest 2: First is to navigate through the given mapped environment with an optimized path; Second is to classify the images the TurtleBot found on the boxes in the environment and store the tag of the identified image into a text file.

2.1 Navigation Design

The main goal for navigation design is to move the TurtleBot in front of each box with the best field of view for the RGB camera to capture the images. The navigation algorithm allows the TurtleBot to first locate a reachable point for each box by searching the availability of the points around the box. Then it uses a path planning algorithm called nearest neighbour algorithm to find an optimal path that connects all the points. Since the environment is already known for the contest, a navigation package's called Adaptive Monto Carlo Localization (AMCL), which is a package optimized for the tasks of travelling to points in a known environment, will be used to plan a path for each target point and follow the path till it reaches the goals.

A few edge case conditions have also been considered before proceeding locating the points.

1. Tight spaces: where the TurtleBot is able to reach but likely to get stuck
2. Nearby boxes: where there are boxes near the box that the robot is trying to reach. This could cause the robot to get stuck, or could cause the image recognition algorithm later on to produce poor results due to the closeness of the boxes.

If any of the listed edge case returns true, the path planning algorithm will find a point that will better suit the above conditions.

2.2 Path Planning Algorithm

The goal for path planning is to find an optimized path between the located points found from navigation design to reduce the total time travelled. Since the TurtleBot's initial position is unknown in the final contest, navigating to the boxes randomly based on the current box index can be inefficient and time-consuming. The path planning algorithm should output a sorted box index and their targeted points' positions for the TurtleBot to navigate to. Nearest Neighbour Algorithm is chosen for its fast computing speed and relatively well optimized path. In comparison with the other methods such as brute force , it saves up over 4s' computing time with very similar travelling times for the computed path.

2.3 Image Pipeline Design

The image pipeline algorithm computes a best match to the given image database whenever it reaches the targeted point next to the box. In the contest, a total of 15 images are given for us to compare in the database. We are required to classify each image we capture from the ten boxes in the environment. Based on the matched features and matched area for each image, the image with the best overall probability will be outputted onto the text file with the associative box index.

To begin, raw images captured from the camera are pre-processed each time before running the classification algorithm.

Based on the results from the image pipeline function, the algorithm will first use relatively low thresholds to eliminate the most improbable images. This normally will reduce the solution space to only 2-3 potential matches to choose from. In addition, since the TurtleBot will not be able to capture the perfect frontal image for each box, a projective transformation will be used to rectify the captured images. Due to time constraints, the algorithm will only be activated when the TurtleBot is reached to the target point, to minimize the total exploration time.

Finally, for each boxes, three diverse locations with a different field of view will be chosen for three seperate trials. Results will be compared and output the most probable result to the text file.

3 Sensory Design

The main focus of this contest is on using computer vision to classify the images. In addition, navigation is done by the open package move_base to automatically avoid obstacles and move to the targeted locations. Hence only the odometry and the Kinect camera have been strongly focused at and utilized during this contest.

3.1 Odometry Sensory

Odometry is used to locate the TurtleBot's current location and the targeted boxes' locations. It returns the transformed coordinates to us with the TurtleBot's current x, y and phi(yaw) based on the /map coordinate via the AMCL algorithm subscribing to the ROS topic /tf – instead of directly from /odom topic from the previous contest. As seen in the figure below, based on the current position and the boxes' coordinates, it can also be implemented to calculate the desired target points near the boxes using the equations below.

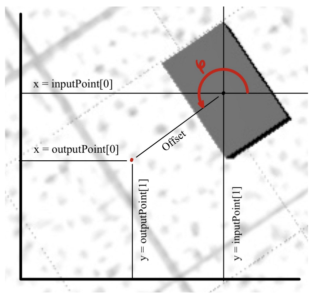


Figure 4: Odometry in Path Planning Algorithm

$$x_{target} = x_{boxes\_coord} + offset \cdot cos(\varphi_{box}) \tag{1}$$

$$y_{target} = y_{boxes\_coord} + offset \cdot sin(\varphi_{box}) \tag{2}$$

$$\varphi_{target} = \varphi_{box} + \pi \tag{3}$$

During testing, it is worth noting that the coordinates of each boxes are not relative to the /map frame but relative to the original TurtleBot's position. By plotting the x and y coordinates onto an xy plane and comparing the plot with gazebo world, the team is able to identify the index of each box and the origin which is near the TurtleBot's initial position.

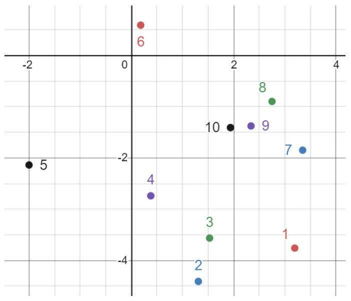


Figure 5: Odometry in Path Planning Algorithm

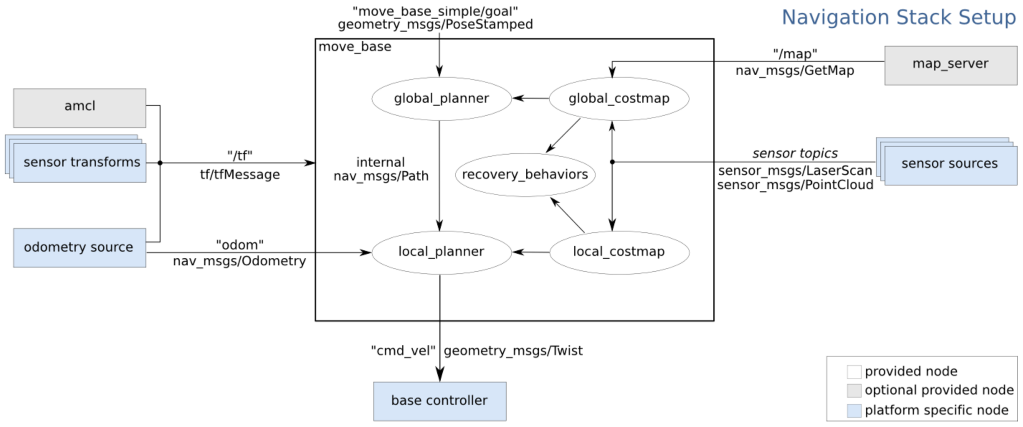


Figure 6: Navigation Stack from "move_base" Package [2]

3.2 Microsoft Kinect Laser Scanner

Laser scanner is a critical sensor in obstacle avoidance and setup initialization. It provides distance information to surroundings and assists with identification of key features in the environment. For AMCL algorithm to work, an initial pose estimation is needed to help the TurtleBot locate itself. It uses the depth information to match with the existing map provided by the contest via a particle filter to localize itself by computing the probability of its location in the provided map with insignificant errors, as seen in Figure 6. This process continues simultaneously while the TurtleBot moves around the environment. The estimated pose location will get more accurate as the TurtleBot scans more obstacles inside the map, as seen in Figures 8 and 9.

Moreover, the laser scanner provides depth information which helps the navigation and obstacle avoidance algorithms by identifying surrounding obstacles. Hence, it assists with path planning and guiding the TurtleBot through the planned path, moving to the required location while avoiding obstacles.

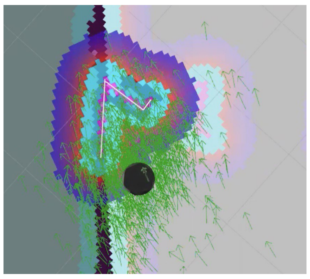


Figure 7: Imprecise Initial Localization

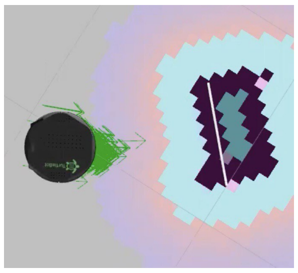


Figure 8: More Accurate Localization over Time

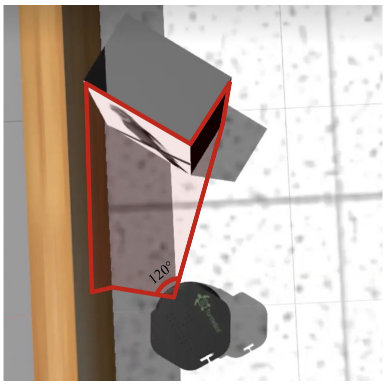


Figure 9: Laser Scanner's Utilization for Localization and Obstacle Avoidance Purposes

Lastly, the laser scanner significantly influences the image identification accuracy via the implementation of the targetPoint algorithm in which the TurtleBot's view of objects is controllable. Various positions and their respective orientations around each object are developed in the navigation algorithm that assists with the image recognition algorithm in order to acquire the most ideal image. TurtleBot is capable to complete this task upon arrival at the object by using the laser scanner to measure the offset distance away from the object, explained more in detail in the corresponding section. This sensor is essential in the sensory design due to the critical depth information that is used by the TurtleBot to discover its surroundings, localize itself, reach the target points accurately, and avoid obstacles.

3.3 Microsoft Kinect RGB Camera

The most essential sensor utilized in this contest's openCV is Microsoft Kinect's RBG camera, which assists by capturing images and storing them into a high dimensional array for future processing and image classifications. The raw camera data from ROS topic:/camera/rgb/image_raw provides the TurtleBot with a size of 640 pixels wide by 480 pixels tall RGB image at a rate of 30 frames per second. By performing a series of image processing techniques (Features2D, homography, etc.) the TurtleBot is able to successfully perform the image recognition process, as seen in Figure 7. Feature indicator and matchmaking techniques are applied to the information acquired from the RGB camera in order to perform image identification and classification on image tags in the database. Homography on scene images acquired by the RGB camera allows for discovering image tags in the scene by converting key points from image tags in the database. In addition, the RGB camera allows for the tuning of navigation parameters in order for the TurtleBot to move to the ideal position in front of boxes with the best field of views.

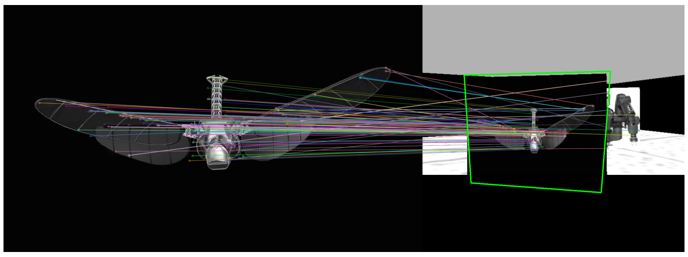


Figure 10: Image Recognition on the Scene Image Taken by the RGB Camera

4 Detailed Design

4.1 ROS Control Architecture

4.1.1 ROS Packages

In total, two additional packages are used in this contest:

1. "TurtleBot_navigation" package: for navigation, which is based on the AMCL algorithm (Adaptive Monte Carlo Localization).
2. OpenCV's ROS package: for image recognition and classifications.

4.1.2 ROS Node – contest2

This is the only node that is needed to run the program, and it consists of five dependent cpp program files and the respective header files, as listed below:

1. contest2.cpp:

   It consists of the high level controller, point locating algorithm, localization algorithm and path-planning algorithm. More detailed descriptions of each algorithm will be explained in the later sections.

2. navigation.cpp

   Used to move the TurtleBot from the current position to the input targeted position. To assist the program, a function overload is added to moveToGoal which allows it takes a std::vector¡float¿ as the target position input.

3. boxes.cpp

   This program provides us with a vector of boxes' coordinates and their indexes. as well as the image data to be used for classifications. In addition to the provided variables, sortedBoxIndex and sortedBoxCoords are created to store the sorted target points which the TurtleBot will move to sequentially.

4. image_pipeline.cpp

   Provides the image classification algorithm. It is used to extract the matching features, descriptors and the total area bounded by the four corner vertices used during comparison.

5. robot_pose.cpp

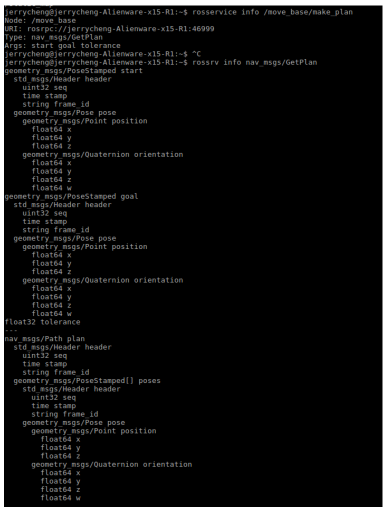
   It contains the posecallback function that return the current odom of the TurtleBot. Unlike the previous contest, this odoom is derived from geometry_msgs/PoseWithCovarianceStamped ROS message, which is a transformed /odom message in respect to the /map coordinate frame. This gives us a more accurate coordinate for localization and navigation.

4.1.3 ROS Services

In total, two ROS services are used for the contest.

1. /move_base/NavfnROS/make_plan:

   This service takes in the current pose, the target pose and a tolerance (how accurate the TurtleBot should move to the targeted point), and outputs a planned path in the form of a pose array. The original purpose of this service is to plan out a path which the TurtleBot will follow.
   However, since it is not clear if the target point inputted is a viable point to output a path, we modified the output of this service into a function called checkAvailability() as a conditional check to see if the target point is viable. The function takes in a target location vector and returns a boolean. If it is not a viable point, the output pose array will be empty. In this case, it returns false to inform the program to input another target point for checking.



```
jerrycheng@jerrycheng-Alienware-x15-R1:~$ rosservice info /move_base/make_plan
Node: /move_base
URI: rosrpc://jerrycheng-Alienware-x15-R1:46999
Type: nav_msgs/GetPlan
Args: start goal tolerance
jerrycheng@jerrycheng-Alienware-x15-R1:~$ ^C
jerrycheng@jerrycheng-Alienware-x15-R1:~$ rossrv info nav_msgs/GetPlan
geometry_msgs/PoseStamped start
  std_msgs/Header header
    uint32 seq
    time stamp
    string frame_id
  geometry_msgs/Pose pose
    geometry_msgs/Point position
      float64 x
      float64 y
      float64 z
    geometry_msgs/Quaternion orientation
      float64 x
      float64 y
      float64 z
      float64 w
geometry_msgs/PoseStamped goal
  std_msgs/Header header
    uint32 seq
    time stamp
    string frame_id
  geometry_msgs/Pose pose
    geometry_msgs/Point position
      float64 x
      float64 y
      float64 z
    geometry_msgs/Quaternion orientation
      float64 x
      float64 y
      float64 z
      float64 w
float32 tolerance
---
nav_msgs/Path plan
  std_msgs/Header header
    uint32 seq
    time stamp
    string frame_id
  geometry_msgs/PoseStamped[] poses
    std_msgs/Header header
      uint32 seq
      time stamp
      string frame_id
    geometry_msgs/Pose pose
      geometry_msgs/Point position
        float64 x
        float64 y
        float64 z
      geometry_msgs/Quaternion orientation
        float64 x
        float64 y
        float64 z
        float64 w
```

Figure 11: getplan ROS Service info and srv Message Type

2. /move_base/clear_costmaps [2]:

   Allows an external user to tell move_base to clear obstacles in the costmaps used by move_base. This allows the TurtleBot to drop the existing costmap and remap a new one.
   During experiments, there are scenarios when the robot got stuck by the narrow space between boxes or through a corridor. This can be extremely catastrophic because it will not only halt the moveToPoint function and abandon the target point, also it may not ever find the way out of the trapped situation due to the odom inaccuracy. Therefore, by clearing up the existing map and redo the path planning, the robot will have a higher chance of finding the right path to follow again.

4.2 High Level Controller Design

A hybrid-controller design is used as the main architecture. The high level controller in this contest mainly focuses on the modularity of the algorithm by breaking down each action into functions and different cpp files, since there is not much crossover between the image classification and navigation parts. Three modules includes the target point locating algorithm, localization algorithm and the image pipeline algorithm. Their relationship is shown below:

1. First, the program will start with an initialization algorithm and rotate 360° for two times upon the user finishes the initial pose estimation. In the meantime, the initial position is recorded for later access to return to.

2. Next, based on the given coordinates of the boxes, the program outputs all the targeted points where the robot will move to for image capturing through the findTargetPoints() function. The output target points and their respective index will be stored inside the global variable for path planning algorithm to access.

3. sortTargetPoints() function will use the "Nearest Neighbour Algorithm" to compute an optimal path for the TurtleBot to follow through and store the sorted points and their sorted index inside two global variables: sortedBoxesPoints and sortedBoxesIndex

4. A modified moveToPoint() function which incorporates the AMCL algorithm will takes in each sorted target point and start proceeding to the goal location.

5. Once reached the goal location, imagePipeline() will be activated to capture 7 images in total at one location for feature extraction and image classification.

6. Upon reaching the targeted point, changePerspective() function will shift the TurtleBot's position for a slight bit for three times to get three different viewing perspectives. Each time imagePipeline() will be activated again to get a total of 7*3=21 results.

7. Finally through the comparison algorithm written in the main() loop, the program compares the indicators calculated from step 5-6 and appends the best result for each box onto the text file.

Overall, the controller implemented a sequential deliberate sense-plan-act control architecture is implemented as the high level controller, where the robot is to implement a series of steps in order to accomplish a task. This allows the robot to a simple, direct and predictable reaction, while also requiring serial dependency for functioning. In this contest, counter solutions to most acknowledged failure cases are considered in the algorithm to avoid such cases.

1. Sense: the sensing is mainly derived from the sensors listed previously
   (a) Odometry: Used for the initial localization and target point calculation.
   (b) Laser Scanner: Used for obstacle avoidance when moving to the targeted points.
   (c) RGB camera: For capturing the images on each box for image pipeline identifications.
2. Plan: Before executing any movement, two plannings are needed
   (a) Point Locating: This will help locate the ideal target point to for each box in the environment. At the same time, it helps to check if the target points are viable to go to using the getPlan service mentioned above.
   (b) Path Planning: Pre-planned optimal path that connects all the targeted points to save time.
   (c) Image Classification: Save the pre-classified results from three different perspectives and compare them to get a better result with higher accuracy before pushing to the text file.
3. Act: Rapid action regardless of the complexity once localization and path planning are finalized
   (a) Move to Point: Move the TurtleBot to the targeted point following the path planned.
   (b) Store Results: Save image classified results to the text file.

4.3 Navigation Algorithm Design

4.3.1 AMCL move_base Algorithm

The move_base package is provided to allow the TurtleBot move to the targeted points and navigate through the environment with obstacles avoidance enabled. Function MoveToGoal is also written for us in the contest which takes in the target point's x, y and yaw and moves the TurtleBot to the correct location. When the target point is not a viable point, the function will return false and will not perform any movement to the TurtleBot. This is avoided by implementing the Target Point Locating Algorithm to first pre-find the available points that are inside the map and not intersecting with any of the boxes. A failed target point will cause the algorithm to drop out the current target point and proceed with the next target point.

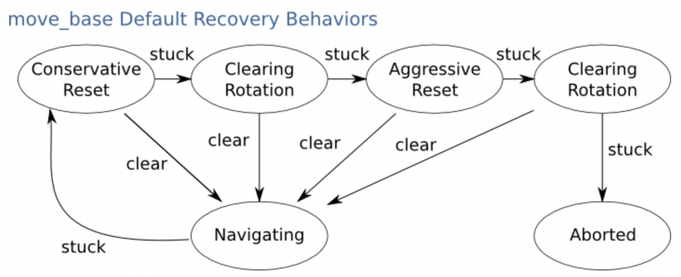


Figure 12: Potential Target Points Checking at Default 0.5m Distance

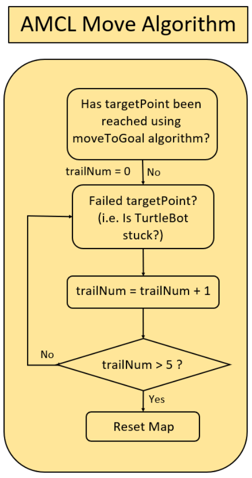

AMCL Move Algorithm
Has targetPoint been reached using moveToGoal algorithm?
trailNum = 0
No
Failed targetPoint? (i.e. Is TurtleBot stuck?)
trailNum = trailNum + 1
No
trailNum > 5 ?
Yes
Reset Map

Figure 13: AMCL Move Strategy in Failure Cases Flow Diagram

4.3.2 Target Point Locating Algorithm

The goal for Target Point Locating is to locate the best target point for each box to move to. The searching algorithm prioritize the points that are right in front of the box for its best field of view. The most optimal point for each box is located by first testing the reachablility of a default target point, which offsets 0.5m away in front of the box, as shown in Figure 14.

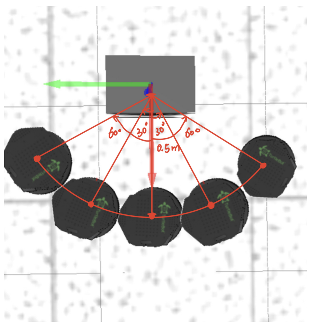


Figure 14: Potential Target Points Checking at Default 0.5m Distance

If the point is not viable, then the algorithm will proceed checking the points offsetting with different angles, namely $30°$, $-30°$,$60°$, $-60°$ in the sequential manner. If 0.5m offset is not ideal for all checking points, it will increase it to 0.6m, 0.7m, 0.8m, and then decrease the offset to 0.4m, 0.3m, 0.2m until it find a viable point. By checking each point's availability before executing the MoveToPoint function helps avoid situations when move_base package is not able to return a valid path.

An offset of 0.5m is chosen as the default distance for its best field of view when capturing the image on the box. This value was obtained through extensive testing, and 0.5m was chosen to be the best result since the image will have the most of its area is occupied with the correct image. Moreover, the minimum distance between the box and TurtleBot is chosen as 0.2m, while the maximum is chosen as 0.8m. If the TurtleBot is too close, then the entire box might not be visible, and also the Turtlebot's physical body could obscure the bottom of the image, and some parts of the image will be out of the field of view. On the contrary, by moving too far, the camera will be more likely to capture the images around the target box and confuse the image pipeline algorithm. The distances and their respective images captured are shown below:

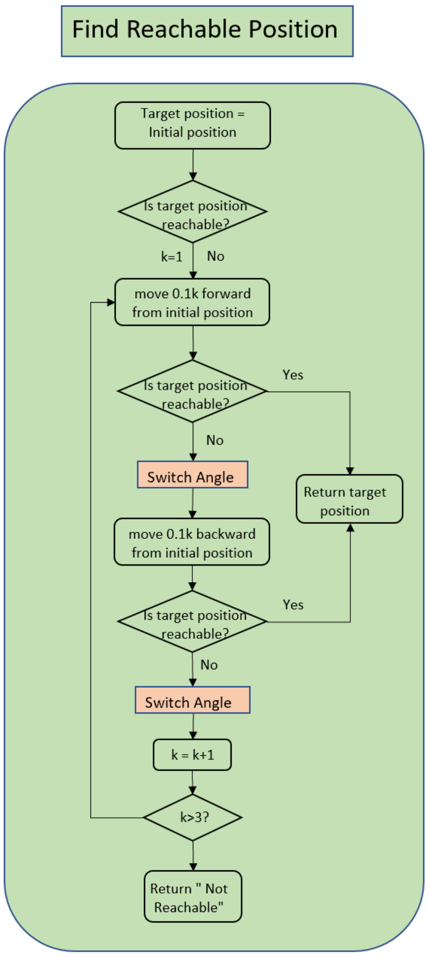

Find Reachable Position
Target position = Initial position
Is target position reachable?
k=1
No
move 0.1k forward from initial position
Is target position reachable?
Yes
No
Switch Angle
Return target position
move 0.1k backward from initial position
Is target position reachable?
Yes
No
Switch Angle
k = k+1
k>3?
Return " Not Reachable"

Figure 15: Point Reachable Check Algorithm Flow Diagram

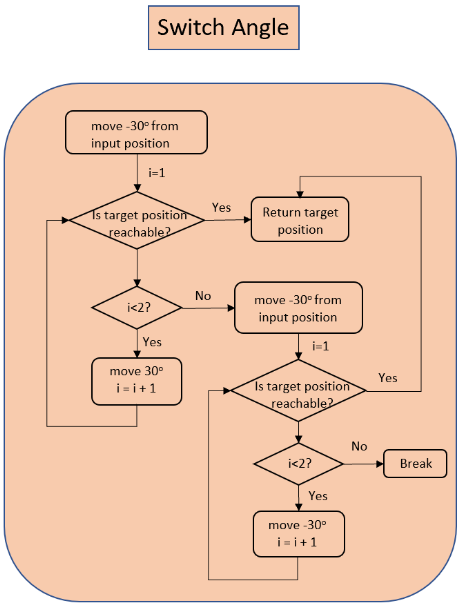

Switch Angle
move -30° from input position
i=1
Is target position reachable?
Yes
Return target position
i<2?
No
move -30° from input position
Yes
move 30° i = i + 1
i=1
Is target position reachable?
Yes
i<2?
No
Break
Yes
move -30° i = i + 1

Figure 16: Shift Angle Algorithm Flow Diagram

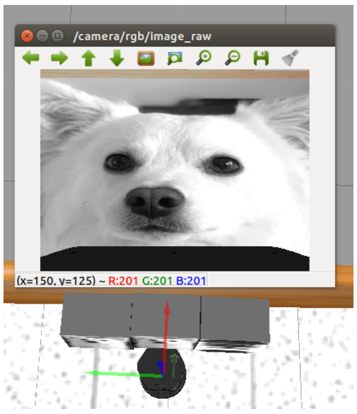


Figure 17: Image Captured At the 0.2m Distance

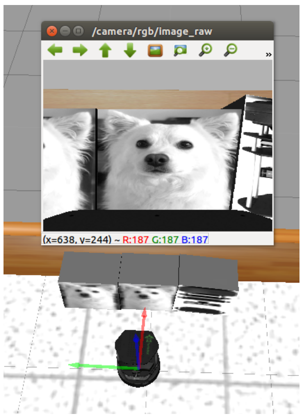


Figure 18: Image Captured At the 0.5m Distance

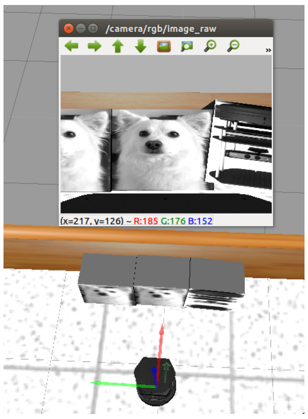


Figure 19: Image Captured At the 0.8m Distance

Similarly, the angle offsets are tested and the most extreme angle offset is set as 60°. Angles higher than that will only capture a slim projection of the image which will lose most of the features. Moreover, if there are other nearby boxes, large angular offset is likely to include other undesired boxes' images.

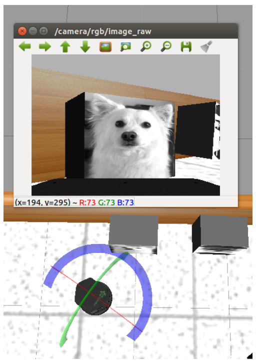


Figure 20: Image Captured At 30° angular offset

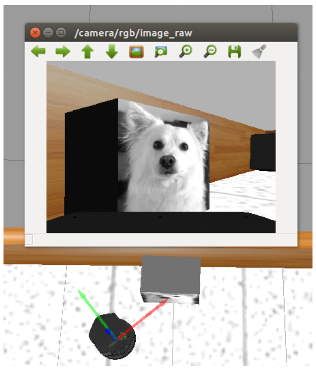


Figure 21: Image Captured At 60° angular offset

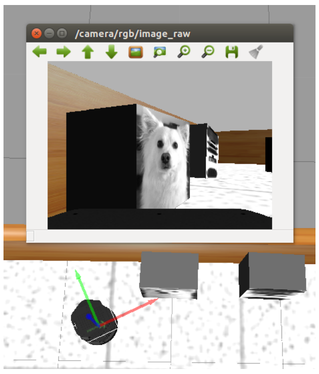


Figure 22: Image Captured At 80° angular offset

Given the above algorithm for point finding the TurtleBot is able to locate the most ideal point for image capturing even when there are obstacles. However, during testing, there are edge cases where the algorithm cannot output the desired point we need or will cause the TurtleBot to stuck in between a tight area.

1. **Nearby Boxes:**

   For the first case, an example is given below in the image , even though the algorithm is able to output a viable point, but due to the tight placement of the boxes, the target point is blocked by the closely placed boxes and thus unable to get the correct image on the box. A box proximity function is used to check the closeness between each box. If the boxes are two close, the function output false and computes which side is the nearby box. To identify the close by box's relative location to the target box, LEFT, RIGHT and FRONT enumerates are used to classify them.

   (a) LEFT means the nearby box is on the left hand side of which the target box is facing, hence right angle locations opposite to nearby box will be checked first.

   (b) RIGHT means the nearby box is on the right side of the target box, and similar to LEFT, target point would be chosen at the opposite angle.

   (c) FRONT means the nearby box is in front of the target box, and TurtleBot will check either side of the target points and omit the frontal candidate points.

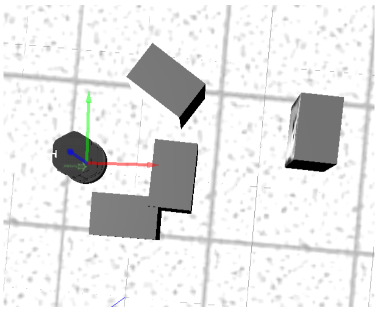


Figure 23: Edge Case When There are Other Boxes Nearby

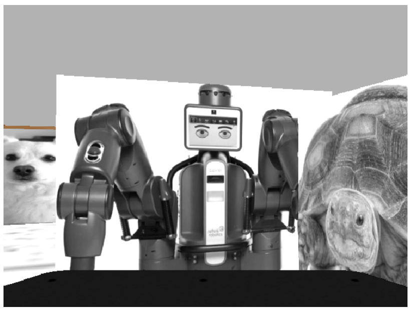


Figure 24: Overlapped Images Captured When There are Other Boxes Nearby

2. **Tight Spaces:**

   The second scenario is when two boxes are close by and form a tight space. This is different from the first case because the space is separative enough that there are still viable points in front of the targeted box. However, during testing the team found that the actual scene's obstacle is slightly smaller than what it predicts in RVIZ. This creates a huge problem when there is a narrow corridor or tight spaces where the TurtleBot is able to pass through. Due to the prediction offsets, TurtleBot will predict the path is too narrow to go through and it is unable to plan out a path out. In addition, because the initial pose estimation for the TurtleBot is not perfectly aligned with the ground truth, and the TurtleBot will localize itself consistently as it moves around the environment. The original viable path planned might fail too, causing it to stuck in the tight space. Therefore, it is best to avoid the TurtleBot getting stuck in tight spaces, since failed path planning will cause AMCL algorithm to drop the current target location and proceeds with the next one. Worst case, it will even halt the entire program.

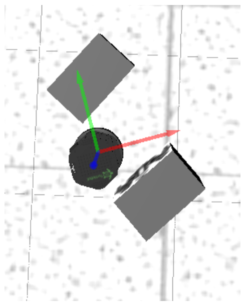


Figure 25: Edge Case When Stuck Between Tight Spaces

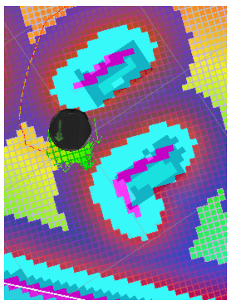


Figure 26: RVIZ Simulation on Stuck in Tight Spaces

   A condition check is created to verify if the path in front of the targeted point is too narrow. If it returns true, it will abandon the front target points even if they are viable, and, as seen in the flow diagram, start looking for angular offset points instead. This greatly reduces risk of getting stuck without sacrificing the quality of the captured images.

4.3.3 Localization Algorithm

Localization algorithm is used at the beginning of the contest, as well as when the TurtleBot failed to follow the planned path due to tight spaces.

Despite the fact that there will be an initial pose estimation done at the beginning of the contest, it is inevitable to have estimation errors and cause the TurtleBot to plan a faulty path and easily get stuck somewhere. Based on numerous testings, by rotating the TurtleBot 360° two times after the initial pose estimation, AMCL localization algorithm is able to assist the TurtleBot to get a more accurate pose estimation in the environment. In fact, the estimation correction is so good that if the initial pose position is somewhere nearby the actual pose location, the manual pose estimation process can be omitted all together.

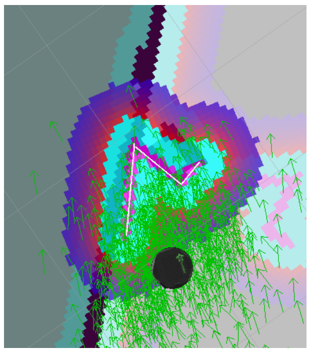


Figure 27: Initial Pose Estimation

Additionally, localization algorithm is able to assist remapping the environment when the TurtleBot is stuck at tight spaces and unable to plan a path to get out. In this situation, clearCostmap ROS service is firstly called to clean up the existing mapped environment. Then followed by the localization algorithm which rotates the TurtleBot to remap and nearby environment and locate itself. After remapping, it will proceeds a new planned path to follow and free itself from the tight spaces.

4.4 Path Planning Algorithm

The path planning algorithm sorts the reachable target coordinates and their corresponding index for an optimized path to minimize the overall time the TurtleBot spends on traveling. This step is crucial due to the 8 minutes time constraints. Two algorithms were tested using Euclidean distance as weight during the contest – the Repetitive Nearest-Neighbour Algorithm and the Brute-Force Algorithm.

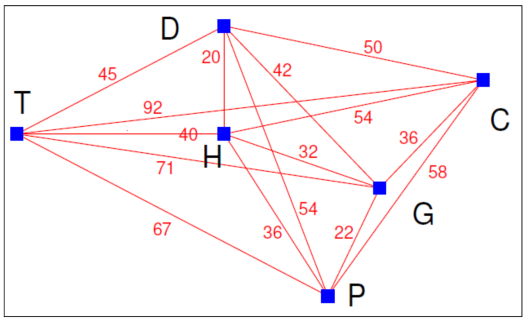


Figure 28: Complete Graph

1. **The Brute-Force Algorithm:**
   (a) List all Hamilton circuits. A Hamilton circuit of an undirected graph G(V,E) is defined as a simple cycle that contains each vertex in V.
   (b) Calculate the total weight of each Hamilton circuit by adding up weights of the edges.
   (c) Choose the one with the smallest total weight.

2. **The Repetitive Nearest-Neighbour Flow Diagram:** The explanation for the Repetitive Nearest-Neighbour is listed below.
   (a) Define a variable called currentPoint, and initialize this variable with the initial location of the TurtleBot upon spawn.

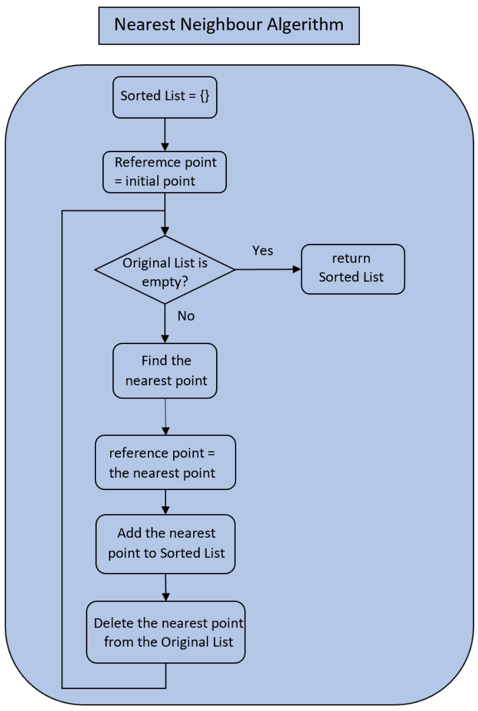

Nearest Neighbour Algerithm
Sorted List = {}
Referemce point = initial point
Original List is empty?
Yes
return Sorted List
No
Find the nearest point
reference point = the nearest point
Add the nearest point to Sorted List
Delete the nearest point from the Original List

Figure 29: Nearest-Neighbour Algorithm Flow Diagram

(b) Compute the distance between currentPoint and all of the coordinates of the boxes that have not been visited. Select the nearest coordinate, and denote this coordinate as nextPoint.

(c) Update currentPoint with the values of nextPoint, that is, currentPoint will now contain the values of nextPoint.

(d) Repeat steps b-c until all boxes has been visited

In the practice map, the Brute-Force algorithm and the Nearest-Neighbour algorithm result in the same path. Comparing the time, the Brute-Force algorithm takes 4 seconds to return the optimal solution, while the Nearest-Neighbour Algorithm is able to finish calculating in 0.001s, but its solution is near-optimal, not optimal. However, since the hamiltonian circuit is an np-complete problem, its time complexity varies factorially with respect to the input. Therefore, considering that the Brute-Force algorithm may take minutes or even hours to finish if one were to increase the input slightly, as well as the relatively insignificant difference between the results of these two algorithms, the teams decided to implement the Repetitive Nearest-Neighbour Algorithm for path planning.

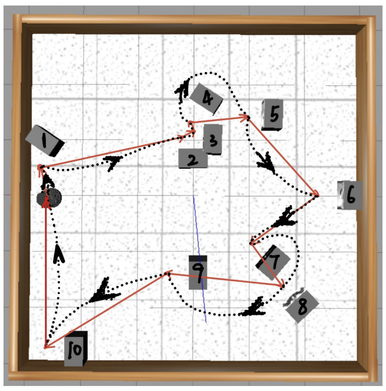


Figure 30: Path Planning Using Repetitive Nearest Neighbour

4.5 Image Classification

Image Classification is used to capture images and classify them into one of the 15 images in the given database. The classification needs to accurately predict the correct tags and reject the mismatched ones. The images that are to be predicted are provided through the Microsoft Kinect RGB camera. Those images are pre-processed, and are fed through an image classifier function that returns a list of several indicators, all contributing to the probability of the image being each tag. Post-processing is later done on these probabilities to return the most likely tag the image corresponds to.

4.5.1 Image Pre-processing and Transformation Algorithm

The transformation algorithm is mainly used to crop the unnecessary areas from the raw images captured and add additional filters for later feature extraction algorithm's easy processing. The following pipeline is done each time TurtleBot reaches a new target pose for image captures.

1. Cropping the Bottom Part:

   Since the front of the TurtleBot is within the camera's field of view, all the useful image data is blocked by the black base. By removing the bottom part, nearly 15% of the image can be removed to save processing time.

2. Cropping the Top Part:

   Initially, this was done only in critical times, since there are chances when the image will occupy the entire camera's field of view. By removing the top part, it means it can lead to feature lost. However, after testing, cropping the 8% of the top of the image is found to drastically reduce the processing time while maintaining the accuracy of feature detection.

3. Cropping the Sides and Resize the image:

   Similar to "Cropping the top part", the percentage cropped is determined via trial and error. When executed correctly, this step can help the image classification avoid faulty identification when two images are tightly clustered together. Additionally, because the rough dimension of each box all have a 5:4 aspect ratio, the sides are cropped so that the outputs also feature the same 5:4 aspect ratio. Finally, despite the potential of feature lost, the accuracy at the clustered boxes increased from around 40% to over 90%.

4. Apply Grey Scale Filter

   Since RGB images' data are in an array of:

   $$[rediIntensity[0 \sim 255], greenIntensity[0 \sim 255], blueIntensity[0 \sim 255]] \times image\_size \tag{4}$$

   while grey scales are stored in the form of:

   $$blackIntensity[0 \sim 255] \times image\_size \tag{5}$$

   In addition, all the images provided by the image database are in grey scale, by converting the captured images to grey scale, the processing time could be reduced by 2 folds.

Moreover, during testing, it is worth to mention that when the cropped images are too small in terms of the number of pixels per image, OpenCV will return false. Therefore, a minimum pixel count is required.

This step is crucial, because it not only reduces the total time spent on processing the large quantity of images, but it also could potentially assist image classification algorithm greatly by removing/eliminating the non-targeted images from the image that is to be processed. Overall, the time saved from Image Augmentation and Transformation Algorithm helped the team improve from a total run-time of over 13 minutes to under 8 minutes.

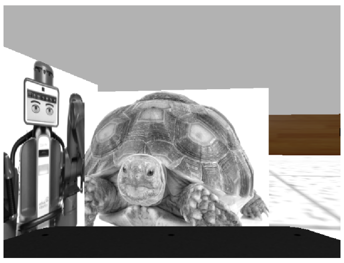


Figure 31: Original Captured Image Raw

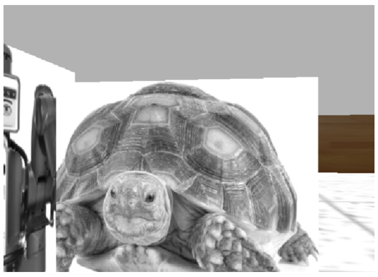


Figure 32: Output Image After all the Pre-processing Steps

4.5.2 Tag Pre-processing Algorithm

This part is used to pre-process the provided image database and extract the useful information before checking the match points using OpenCV.

Before executing to "SURF Feature Detection Algorithm and Homography Algorithm" and start comparing the features between the tags and captured images. The tags in the database is initially pre-processed to acquire the extracted features and the features found will be stored in the global variable.

4.5.3 SURF Feature Detection Algorithm and Homography

SURF Feature Detection
Once the robot reaches the optimal position to capture an image, OpenCV ROS library is used to perform classification on that image. SURF Feature Detector algorithm is then used, which takes an image as an input, and it outputs an array of features/keypoints within that image, as seen in Figure 33.

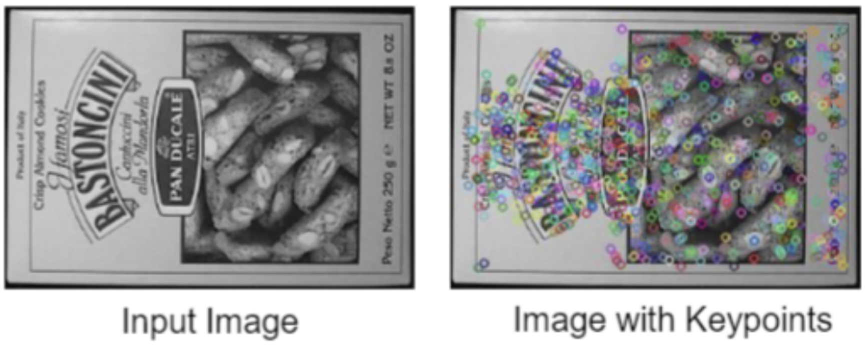


Figure 33: SURF Algorithm Feature Extraction Illustration

This algorithm is performed once on the image taken from the camera, and once on each of the tags. A total of 7 iterations will be carried out at each target point, and three perspectives will be taken in front of each box. In other words, there are a total of 21 images taken and compared one by one with the 15 potential tags. The reason for aving a large number of iterations is because of the many inaccuracies that could occur with only a single image. The hope is that by combining the results from multiple samples the aggregate results will lower the probability of errors.

Homography Algorithm
Homography Algorithm is used to identify the four edge vertices and transform them into a fully connected polygon.

An example is given in the Figure 34 below.

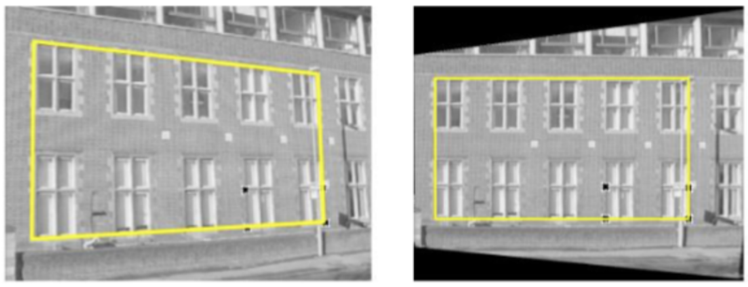


Figure 34: Homography Algorithm Illustration

With the rectification process of the image captured, even when the captured images are resized, rotated or upside down, Homography algorithm is still able to capture the four edges and return a relatively accurate result. However,

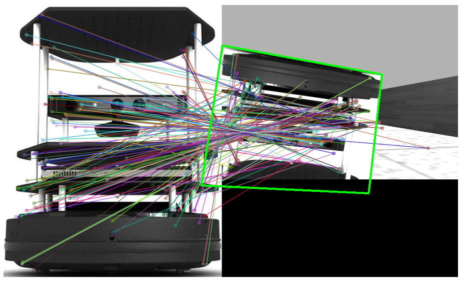


Figure 35: Homography Algorithm Dealing With Upside Down Images

the algorithm could return different number of good matches, as well as a different bounding area whilst the robot still remained in the same location. As shown in the image below, the results of good matches are different each time the photo is taken, even when the position remains the same.

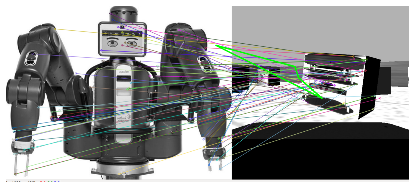


Figure 36: First image capture

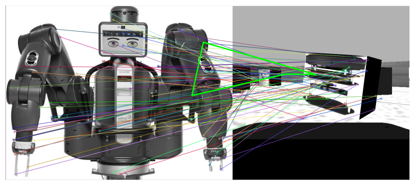


Figure 37: Second image capture, at the exact same location as the previous image

Therefore, three different perspectives of the same box is needed to reduce the error created by the inaccuracy of Homograph transformation. After reaching to the most ideal location in front of each box, TurtleBot will rotate plus and minus a certain degree to get three different perspectives. Initially, our team considered moving to a different point for two more diverse perspectives. However, AMCL move_base algorithm tends to malfunction when it moves around narrow spaces and even likely getting stuck. Moreover, even when the area is clear, it still takes some time for the TurtleBot to move to the correct pose position and rotate to the right orientation. Therefore, a slight angle shaft of $\frac{\pi}{32}$ is chosen as our angle increment for three different perspectives.

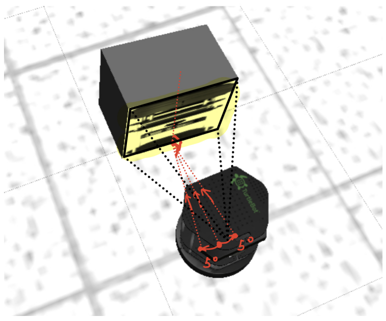


Figure 38: Three Different Perspective Change in Front of Each Box

Finally, by combining the SURF and Homography Algorithm, a set of useful indicators can be created to compute the likely-hoods of the image being each tag. The reason why several indicators are used and a probability is computed, instead of simply returning the tag with the most number of matches is that this increases the versatility of the function. There are edge cases where the image did not receive the highest number of matches from a particular tag, yet it belongs to that tag nevertheless. More details on the indicators will be explained in the "Image Pipeline Algorithm"

4.5.4 Image Pipeline Algorithm

The image pipeline algorithm takes in the processed images from the "Image Augmentation and Transformation Algorithm" and outputs a list of potential classified results based on the indicators we set. The overall flow diagram is shown below:

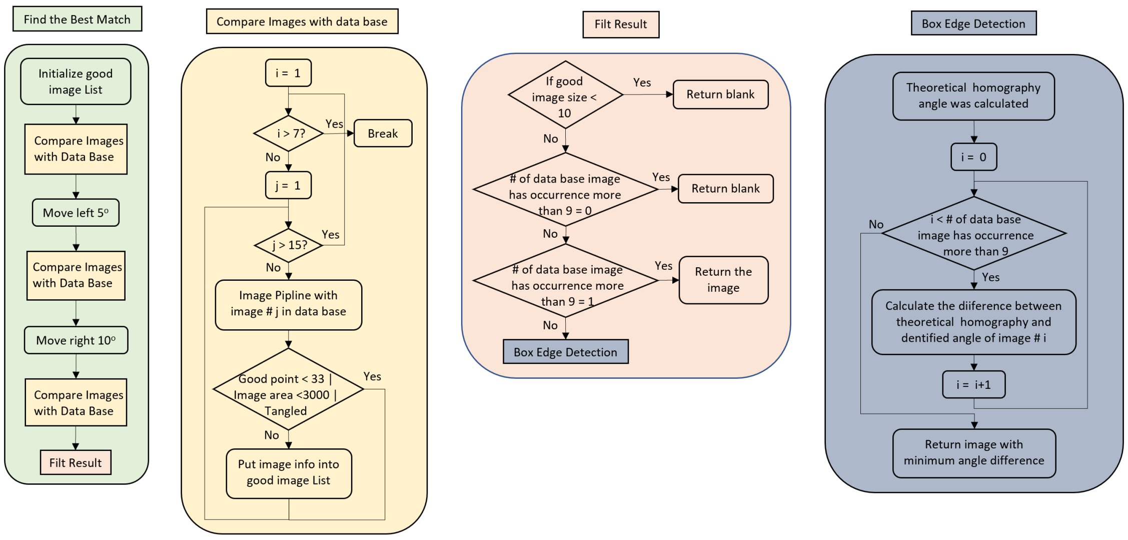

Find the Best Match
Initialize good image List
Compare Images with Data Base
Move left 5°
Compare Images with Data Base
Move right 10°
Compare Images with Data Base
Filt Result
Compare Images with data base
i = 1
i > 7?
Yes
Break
No
j = 1
j > 15?
Yes
No
Image Pipline with image # j in data base
Good point < 33 | Image area <3000 | Tangled
Yes
No
Put image info into good image List
Filt Result
If good image size < 10
Yes
Return blank
No
# of data base image has occurrence more than 9 = 0
Yes
Return blank
No
# of data base image has occurrence more than 9 = 1
Yes
Return the image
No
Box Edge Detection
Box Edge Detection
Theoretical homography angle was calculated
i = 0
No
i < # of data base image has occurrence more than 9
Yes
Calculate the diiference between theoretical homography and dentified angle of image # i
i = i+1
Return image with minimum angle difference

Figure 39: Find the Best Match Algorithm Flow Diagram

Overall, there are 12 indicators that are generated through the Image Pipeline Algorithm:

1. Tag Number shown in the image database
2. Box Index
3. The Area Bounded by the Four Vertices Found Via Homograph
4. Ratio Between the Good Matched Points and Total Feature Points

5. The (x,y) Coordinates of the Four Vertices Found
6. If The Vertices Tangle When Trying To Be Connected To A Polygon
7. The Total Number of Key Points on the Captured Image
8. The Total Number of Good Matches on the Captured Image
9. The Four Angles' Magnitudes for Each Point on the Polygon
10. The Location of the Bottom Left Point (among the four vertices)
11. The Effective Area Bounded Inside the Raw Image Boundary
12. Ground Truth of the Orientation Between the TurtleBot and the Box

To Begin with, based on the four vertices from Homography Algorithm, the bounded area is computed based on the OpenCV function and stored inside indicator 3, if the total area is less than 3000, it is then eliminated:

$$cv :: coutourarea(); \tag{6}$$

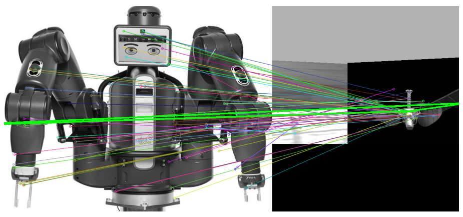


Figure 40: An Example When the Bounded Area is Too Small but With Large Good Matches

Based on the good matched feature points and the total key points on each tag database, a ratio can be computed as indicator 4. This indicator eliminated the fact that different tag image contains different number of key feature points. Solely eliminate based on the number of good match can eliminate the ideal match by accident. In the contest, the match is only eliminated when the ratio is less than 0.1

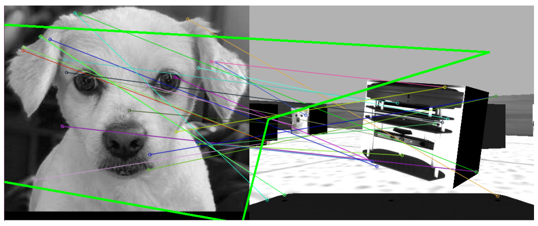


Figure 41: An Example When the Keypoints Matched Ratio is Too Small but Area is Too Big

Another boolean value is used to check if the four vertices are tangled together when connected and is stored in indicator 6. When it returns true, the match is also eliminated.

Those three conditional checks above are able to eliminate most of the undesired cases and in most cases return a single result. Otherwise, it is still capable of narrowing down the candidates to 2 3 numbers. When there is still no definitive result, it is categorized as the edge cases and mostly happens when multiple images are captured in one take. The remaining unmatched candidates are then feed into the more sophisticated algorithms below. The edge case algorithm filters out the mixed in images and only returns the results of the targeted one.

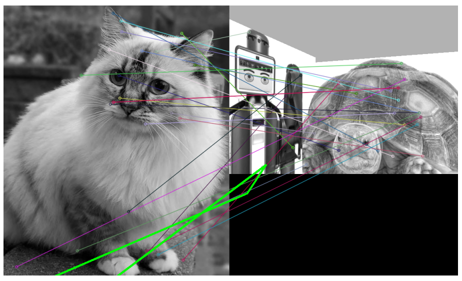


Figure 42: An Example When the Four Vertices Are Tangled

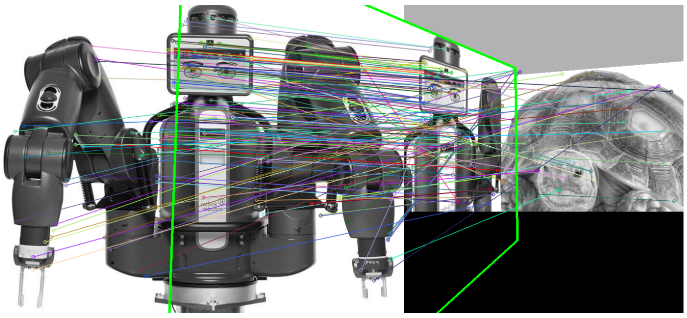


Figure 43: Edge Case When Multiple Images Are Captured in One Take

1. **Box Edge Angle Detection Algorithm:**

   This detection algorithm is used to compare the identified homography edges' angle and the estimated theoretical angle derived from the distance and angle offset between the TurtleBot and the box (from "Target Point Finding Algorithm"). Based on the comparison, it can help filter out the edge case mentioned in "Image Pipeline Algorithm" Section, where two boxes are clustered together.

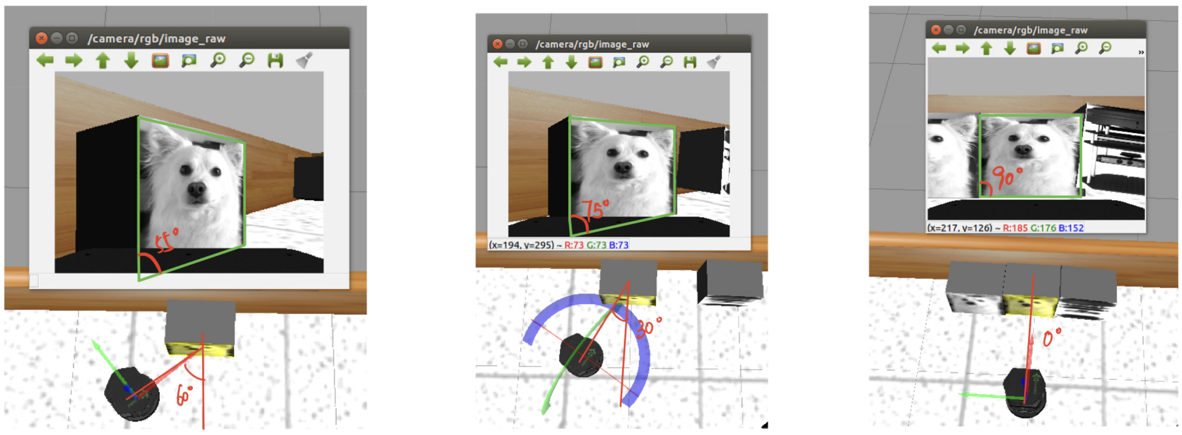


Figure 44: Ground Truth Predicted Angles from Navigation Results

Because the angle difference between the TurtleBot's orientation and targeted image orientation are known as the TurtleBot moves to the targeted location. This can be considered as the ground truth to perdict the projective angle of the image edges. The bottom left corner's angle can be then calculated using cosine rule:

$$\theta = \cos^{-1}(\frac{a^2 + b^2 - c^2}{2ab}) \tag{7}$$

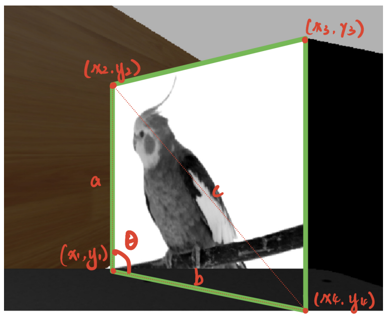


Figure 45: Angle Calculation of the Captured Image Bounded Box Edges

By calculating the sum of the differences between each result's bottom left angle and the ground truth angle for each image detection iteration, we are able to keep the one with the smallest sum and eliminate the interrupted images captured.

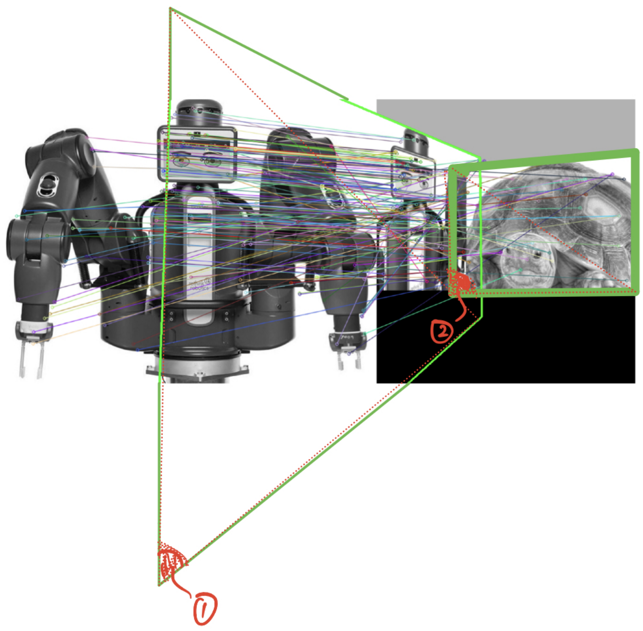


Figure 46: Angle Calculated Comparision Between Two Candidates in One Image; 1 is the faulty match and 2 is the correct match

2. **Effective Area Checking Algorithm:**

   Since the size of the image from the camera is always 640 by 480 pixels, the team decides to implement a further area checking filter. The team had noticed that sometimes, the homography returns four points which can be outside of the effective range of the image from the camera. Therefore, this filter rewards one point for each pixel that is inside the range of the camera image, and deducts half a point for each pixel that lies outside the boundary of the camera image. By comparing the total area between potential candidates, the one with the largest area will be considered the final result.

Finally, if the two edge case algorithms output a different result (which never happened in our testings), the program will assign weights to both the summedAngleDifference and summedTotalArea, sum it together, and select the best result.

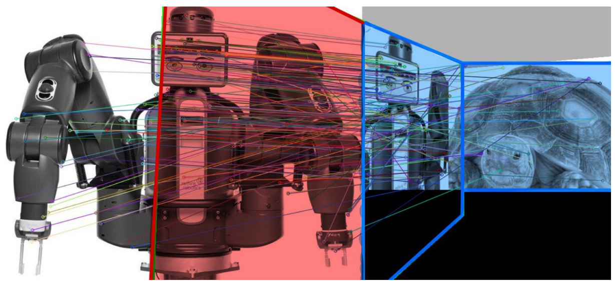
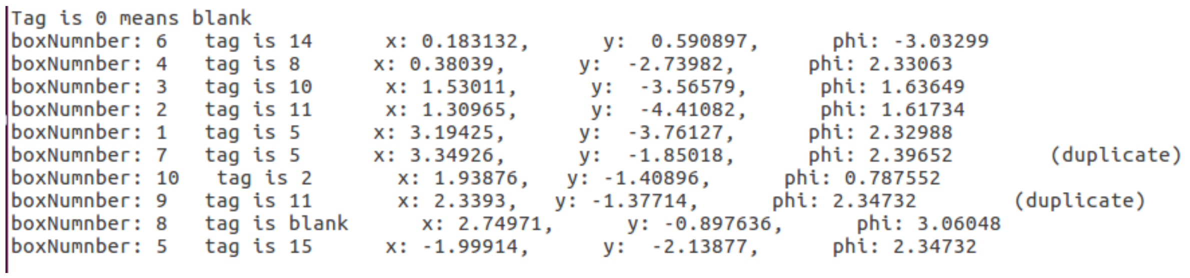


Figure 47: Blue areas are the pixels within the camera image range and therefore are rewarded, red areas are penalized

5 Final Results

Once the simulated TurtleBot has traversed to all objects and has returned to its starting location, the algorithm is set to output a file called the "changeEngine.txt" with all the tags it has found and their respective location in the exact order they are found. The output to the trial completed in the practice world is depicted below.



```
Tag is 0 means blank
boxNumnber: 6   tag is 14      x: 0.183132,      y:  0.590897,      phi: -3.03299
boxNumnber: 4   tag is 8      x: 0.38039,      y:  -2.73982,      phi: 2.33063
boxNumnber: 3   tag is 10      x: 1.53011,      y:  -3.56579,      phi: 1.63649
boxNumnber: 2   tag is 11      x: 1.30965,      y:  -4.41082,      phi: 1.61734
boxNumnber: 1   tag is 5      x: 3.19425,      y:  -3.76127,      phi: 2.32988
boxNumnber: 7   tag is 5      x: 3.34926,      y:  -1.85018,      phi: 2.39652        (duplicate)
boxNumnber: 10   tag is 2       x: 1.93876,   y: -1.40896,      phi: 0.787552
boxNumnber: 9   tag is 11       x: 2.3393,   y: -1.37714,      phi: 2.34732        (duplicate)
boxNumnber: 8   tag is blank      x: 2.74971,      y: -0.897636,      phi: 3.06048
boxNumnber: 5   tag is 15      x: -1.99914,      y:  -2.13877,      phi: 2.34732
```

Figure 48: Algorithm Results in the Practice World

As shown in the algorithm's output within the practice world, the TurtleBot is able to identify all ten objects accurately and return to its initial position within a span of approximately 6 minutes.

6 Future Recommendations

This paper presents the proposed navigation and image recognition algorithm for the TurtleBot that enables it to collect 10 image tags within 8 minutes. Throughout the course of this project, the team was able to develop a foundational navigation and object detection algorithm; however, there are ways to improve the current algorithm.

Similar to the last contest, there are some add-on features such as reinforcement learning (RL) the team would like to include, even though it is not required by the instructors, as it will assist with run time reduction. Although the bot is able to successfully complete the tasks within the assigned time, the team would like to improve the efficiency and robustness of the image recognition as well as the path planning algorithms.

Also, with respect to machine learning, since there is such a small dataset, the team believes that it might be better to implement a convolutional neural network, where we perform many data augmentations, and send it to the neural network for training. Since there is no validation set, the team believes that the training accuracy can be very high if implemented correctly.

As for the navigation algorithm, it currently employs a repetitive Nearest Neighbour algorithm which is an efficient but non-optimal solution. It is a quick (approximately 0.001 seconds) and straightforward solution but does not always find the lowest-weight Hamilton circuit. On the other hand, Brute-Force algorithm is optimal but inefficient solution which guarantees the shortest path. This would be achievable in this contest, since there are only 10 target points (9! or 362880 Hamilton circuit paths) to examine, but it is time consuming (about 4 seconds). For the purpose of this contest, however, the Nearest Neighbour approach is sufficient. The team, however, did not consider running time between the two option and only considered the path planning initialization time. In future projects, the team would like to implement both options once all algorithms are finalized in order to compare the results of the different path planning algorithms and select the one most suited to the project descriptions, if time permits.

The image processing system could still be enhanced for future projects. Currently, the team utilizes SURF detector to extract key points from the objects in the scene images. For future improvements, different detector algorithms such as getAffineTransform and warp Perspective features are to be tested and employed in the final image identification algorithm [3]. The most efficient and accurate feature combinations could be compared and contrasted by testing them in the finalized algorithm. In addition, the area and cosines bounded by the transformed image's four corners are calculated using homography. In the future, the team would like to maximize the usage of homography to eliminate false identifications, as the TurtleBot is prone to identify incorrect image tags due to the out of shape transformed four corners, even though the area and cosines of the images are logical.

7 Contribution Table

The contribution table is denoted as follow:

Sections	Qilong Cheng	Maryam Hosseini	Yuntao Cai	Yicheng Wang
Research	RS	RS	RS	RS
Coding: High Level Controller	WD, MR, DE, FP		WD,MR,DE,FP	
Coding: Point Validation	WD, MR, DE, FP		WD,DE,FP	
Coding: Target Point Location	WD, MR, DE, FP			
Coding: Path Planning	MR, FP		WD	WD, MR, DE, FP
Coding: Edge Cases – Tight Spaces	WD, MR, DE, FP		WD	WD, DE
Coding: Edge Cases – Corridors	WD, MR, DE, FP			
Coding: Image Pipeline	MR	WD	WD,MR,DE,FP	
Coding: Data Storage			WD, MR, DE, FP	
Report : Problem Definition and Design Objectives		WD,MR,FP, ET	FP	FP
Report : Strategy	WD, MR, FP, ET	WD,MR,FP, ET	FP	FP
Report : Sensory Design	WD, MR, FP, ET	WD,MR,FP ,ET	FP	WD,FP
Report : ROS Control Architecture	WD, MR, FP, ET	WD,MR,FP, ET	FP	FP
Report : High Level Controller Design	WD, MR, FP, ET	MR,FP, ET	FP	FP
Report : Navigation Controller Design	WD, MR, FP, ET	WD,MR,FP, ET	FP	WD, MR, FP
Report : Path Planning Algorithm	FP, ET	MR, ET,FP	MR, FP	WD, MR, FP, ET
Report : Image Classification	MR, FP, ET	MR,FP, ET	WD,MR,ET,FP	WD, MR, FP
Future Recommendation, Objectives and Constraint	MR, FP	WD,MR,FP, ET	WD,FP	FP

RS - research
ET - edited for grammar and spelling
WD - Wrote Draft
FP - final proofread of assignment
MR - Major revision
DE - Major debugging

References

[1] Contest 2 Finding Objects of Interest in an Environment. Toronto: University of Toronto, 2022.

[2] "move_base", Library.isr.ist.utl.pt, 2011. [Online]. Available: http://library.isr.ist.utl.pt/docs/roswiki/move_base.html. [Accessed: 22- Mar- 2022].

[3] "Epipolar Geometry (CMSC426)."

Appendices

A Source Code on contest2.cpp

```
#include <boxes.h>
#include <navigation.h>
#include <robot_pose.h>
#include <imagePipeline.h>
#include <nav_msgs/Odometry.h>
#include <tf/transform_datatypes.h>
#include <random>
#include <iostream>
#include <bits/stdc++.h>
#include "std_msgs/String.h"
#include <sstream>
#include <fstream>
#include <float.h>
#include <tf2_ros/transform_listener.h>
#include <nav_msgs/OccupancyGrid.h>
#include <nav_msgs/MapMetaData.h>
#include <math.h>
#include <vector>
#include <ros/ros.h>
#include <nav_msgs/GetPlan.h>
#include <std_srvs/Empty.h>
#include "navigation.h"

#define RAD2DEG(rad) ((rad)*180. / M_PI)
#define DEG2RAD(deg) ((deg)*M_PI / 180.)
#define CCW (1)
#define CW (0)

std::ofstream myfileMap;
std::ofstream myfileLaser;
std::ofstream myfileAngle;
std::ofstream energyMatrix;
std::vector<std::vector<float>> targetPoints;
float defaultDistance = 0.3;
const float angularMax = 0.2;

std::vector<float> initPose;
std::vector<int> sortedBoxNum;
std::vector<float> sortedGotoPose;

std::vector<float> groundTruthAngle, groundTruthDistance;
float groundTruthAngleTemp, groundTruthDistanceTemp;
std::vector<std::vector <float> > unmatchedBoxeSArray;
std::vectoor<float> unmatchedBox;

Boxes boxes;
Navigation navigation;

enum orientation
{
    LEFT,
    RIGHT,
    FRONT
};

/////////////////////////////////////////////////////////////////////////////////////////
/////////////////////////**Function Declaration**////////////////////////////////////////
/////////////////////////////////////////////////////////////////////////////////////////

bool checkAvailbility(std::vector<float> goalCoord);
bool checkProximity(std::vector<float> boxCoord);
bool checkOrientation(std::vector<float> targetPoint);
float reversePhi(float phi);
std::vector<float> offsetPoint(std::vector<float> inputPoint, float offset);
std::pair<std::vector<float>, bool> findViablePoints(std::vector<float> inputPoint, float offset);
std::vector<std::vector<std::vector<float>>> sort_points(std::vector<std::vector<float>> coordinates);

/////////////////////////////////////////////////////////////////////////////////////////
/////////////////////////**Helper Functions**////////////////////////////////////////
/////////////////////////////////////////////////////////////////////////////////////////
```

```
float distance(std::vector<float> Pos1, std::vector<float> Pos2)
{
    return sqrt(pow((Pos1[0] - Pos2[0]), 2) + pow((Pos1[1] - Pos2[1]), 2));
}

//////////////////////////**two angle difference and the orientation: ccw or cw**/////////////////////////////////////

std::pair<bool, float> angleDifference(float angle1, float angle2)
{

    bool compare; // 0 if angle1 > angle2; 1 if angle1 < angle2;
    float deltaAngle = 0;
    bool direction = 0; // 0 as ccw; 1 as cw
    float path1, path2;

    if (angle1 > angle2)
    {
        compare = 0;
    }
    if (angle1 <= angle2)
    {
        compare = 1;
    }
    if (compare == 0)
    {
        path1 = angle1 - angle2;
        path2 = 2 * M_PI - path1;
        if (path1 < path2 && compare == 0)
        {
            direction = 1;
            deltaAngle = path1;
        }
        if (path1 < path2 && compare == 1)
        {
            direction = 0;
            deltaAngle = path1;
        }
        if (path1 > path2 && compare == 0)
        {
            direction = 0;
            deltaAngle = path2;
        }
        if (path1 > path2 && compare == 1)
        {
            direction = 1;
            deltaAngle = path2;
        }
    }
    else
    {
        path1 = angle2 - angle1;
        path2 = 2 * M_PI - path1;
        if (path1 < path2 && compare == 0)
        {
            direction = 1;
            deltaAngle = path1;
        }

        if (path1 < path2 && compare == 1)
        {
            direction = 0;
            deltaAngle = path1;
        }

        if (path1 > path2 && compare == 0)
        {
            direction = 0;
            deltaAngle = path2;
        }
        if (path1 > path2 && compare == 1)
        {
            direction = 1;
            deltaAngle = path2;
        }
    }

    //// for debugging /////
    if (direction == 0)
    {
        // ROS_INFO("Turing Counter-Clockwise");
    }
    else
    {
        // ROS_INFO ("Turning Clockwise");
    }
    // ROS_INFO("The current angle is: %f, target angle is: %f", RAD2DEG(angle1), RAD2DEG(angle2));
    // ROS_INFO ("The angle difference is: %f", RAD2DEG(deltaAngle));
    return std::make_pair(direction, deltaAngle);
}

/////////////////////////**reverse the orientation 180 degrees**///////////////////////////////////////

float reversePhi(float phi)
{
    if (phi > 0)
    {
        return (phi - M_PI);
    }
    else
    {
        return (phi + M_PI);
    }
}

/////////////////////////**two angle additions**///////////////////////////////////////

float angleAddition(float angle1, float angle2)
{
    float angle3 = angle1 + angle2;
    if (RAD2DEG(angle3) >= 180)
    {
        angle3 = M_PI - angle3;
    }
    else
    {
        angle3 = -angle3 - M_PI;
    }
    return angle3;
}

/////////////////////////**check if there are boxes close by**///////////////////////////////////////
bool checkProximity(std::vector<float> boxCoord, int boxIndex)
{
    for (int i = 0; i < boxes.coords.size(); i++)
    {
        std::cout << boxes.coords[i][4] << std::endl;
    }
}

/////////////////////////**check if the boxes are facing the same orientation**///////////////////////////////////////
bool checkOrientation(std::vector<float> targetPoint)
{

    targetPoint[2] = reversePhi(targetPoint[2]);
    std::vector<float> checkPoint = offsetPoint(targetPoint, 0.2);
    bool clearance;

    float checkPointX = checkPoint[0];
    float checkPointY = checkPoint[1];
    float dist;

    for (int i = 0; i < boxes.coords.size(); i++)
    {
        dist = distance(boxes.coords[i], checkPoint);
        if (dist < 0.15)
        {
            clearance = false;
        }
    }
    return clearance;
}

bool checkFacing(std::vector<float> point1, std::vector<float> point2)
{
    std::pair<bool, float> anglePair = angleDifference(point1[2], point2[2]);
    float angleDifference = anglePair.second;
    bool orientation = anglePair.first;

    if (fabs(angleDifference) < M_PI)
    {
        return true;
    }
    else
    {
        return false;
    }
}

bool clearCostMap()
{
    // Clear cost map using the service
    std_srvs::Empty srv;
    ros::NodeHandle nh;
    ros::ServiceClient clear = nh.serviceClient<std_srvs::Empty>("move_base/clear_costmaps");
    bool success = clear.call(srv);

    if (success)
        ROS_INFO("Cleared cost map");

    return success;
}

bool checkAvailbility(std::vector<float> goalCoord)
{

    bool possibility = false;
    RobotPose robotPose(0, 0, 0);
    ros::NodeHandle nh;

    // Set start position
    geometry_msgs::PoseStamped start;
    geometry_msgs::Quaternion phi = tf::createQuaternionMsgFromYaw(robotPose.phi);
    start.header.frame_id = "map";
    start.pose.position.x = robotPose.x;
    start.pose.position.y = robotPose.y;
    start.pose.position.z = 0;
    start.pose.orientation.x = 0;
    start.pose.orientation.y = 0;
    start.pose.orientation.z = phi.z;
    start.pose.orientation.w = phi.w;

    // Set goal position
    geometry_msgs::PoseStamped goal;
    phi = tf::createQuaternionMsgFromYaw(goalCoord[2]);
    goal.header.frame_id = "map";
    goal.pose.position.x = goalCoord[0];
    goal.pose.position.y = goalCoord[1];
    goal.pose.position.z = 0;
    goal.pose.orientation.x = 0;
    goal.pose.orientation.y = 0;
    goal.pose.orientation.z = phi.z;
    goal.pose.orientation.w = phi.w;

    // Set up the service and call it
    nav_msgs::GetPlan srv;
    ros::ServiceClient checkPath = nh.serviceClient<nav_msgs::GetPlan>("move_base/NavfnROS/make_plan");

    srv.request.start = start;
    srv.request.goal = goal;
    srv.request.tolerance = 0.0;
    bool success = checkPath.call(srv);

    // Output print statments
    if (!success)
    {
        // ROS_ERROR("ROS SERVICE Make plan service failed! ");
        // ROS_INFO(" ");
    }

    if (srv.response.plan.poses.size() > 0)
    {
         possibility = true;
         // ROS_INFO("ROS SERVICE Valid Point!");
         // ROS_INFO(" ");
    }
    else
    {
         possibility = false;
         // ROS_INFO("ROS SERVICE The point is not reachable! ");
         // ROS_INFO(" ");
    }

    return possibility;
```

```
}

/////////////////////////**publish velocity to move the turtlebot**///////////////////////////////////////
inline void publishVelocity(float linear, float angular)
{
    ros::NodeHandle nh;
    ros::Publisher vel_pub = nh.advertise<geometry_msgs::Twist>("cmd_vel_mux/input/navi", 1);
    geometry_msgs::Twist vel_msgs;
    vel_msgs.linear.x = linear;
    vel_msgs.angular.z = angular; // if angular is positive --> counter-clockwise; if angular is negative --> clock
    vel_pub.publish(vel_msgs);
    ros::spinOnce();
}

/////////////////////////////////////////////////////////////////////////////////////////
/////////////////////////**Point Offsetting and Finding**///////////////////////////////////////
/////////////////////////////////////////////////////////////////////////////////////////

/////////////////////////**Point Offsetting**///////////////////////////////////////
std::vector<float> offsetPoint(std::vector<float> inputPoint, float offset)
{
    std::vector<float> outputPoint(3, 0);

    float initOrientation = inputPoint[2];
    float initX = inputPoint[0];
    float initY = inputPoint[1];

    outputPoint[0] = initX + offset * cosf(initOrientation);
    outputPoint[1] = initY + offset * sinf(initOrientation);
    outputPoint[2] = reversePhi(initOrientation);

    return outputPoint;
}

/////////////////////////**Rotate and Find the Ideal Point**///////////////////////////////////////

// /////////////////////////**ROS Param to be tuned **/////////////////////////////////////////////

float angleIncrement = M_PI / 6;
float defaultOffset = 0.5;
float offsetIncrement = 0.1;

/////////////////////////**find a good point near one box**///////////////////////////////////////
std::pair<std::vector<float>, bool> findViablePoints(std::vector<float> inputPoint, float offset)
{
    std::vector<float> targetPoint;
    bool availbility;
    float targetPhi;
    bool availabilty = true;
    bool clearance = true;

    targetPoint = offsetPoint(inputPoint, offset);
    availbility = checkAvailbility(targetPoint);
    clearance = checkOrientation(targetPoint);
    float originalBoxPhi = inputPoint[2];

    if (!availbility || !clearance)
    {
        // ROS_WARN("The default point is not reachable");
        inputPoint[2] += M_PI / 6;
        targetPoint = offsetPoint(inputPoint, offset);
        availbility = checkAvailbility(targetPoint);
        clearance = checkOrientation(targetPoint);

        if (!availbility || !clearance)
        {
            // ROS_WARN("The ccw 30 point is not reachable");
            inputPoint[2] += M_PI / 6;
            targetPoint = offsetPoint(inputPoint, offset);
            availbility = checkAvailbility(targetPoint);

            if (!availbility)
            {
                // ROS_WARN("The ccw 60 point is not reachable");
                inputPoint[2] -= M_PI / 2;
                targetPoint = offsetPoint(inputPoint, offset);
                availbility = checkAvailbility(targetPoint);
                clearance = checkOrientation(targetPoint);

                if (!availbility || !clearance)
                {
                    // ROS_WARN("The cw 30 point is not reachable");
                    inputPoint[2] -= M_PI / 6;
                    targetPoint = offsetPoint(inputPoint, offset);
                    availbility = checkAvailbility(targetPoint);

                    if (!availbility)
                    {
                        // ROS_WARN("The cw 60 point is not reachable");
                        availabilty = false;
                    }

                    else
                    {
                        groundTruthAngleTemp = M_PI / 6;
                        // ROS_INFO("CW 60 degree point is a Good point! ");
                        // std::cout << "Target coordinate: " << std::endl;
                        // std::cout << i << " x: " << targetPoint[0] << " y: " << targetPoint[1] << " z: " << targ
                    }
                }
                else
                {
                    groundTruthAngleTemp = M_PI / 3;
                    // ROS_INFO("CW 30 degree point is a Good point! ");
                    // std::cout << "Target coordinate: " << std::endl;
                    // std::cout << i << " x: " << targetPoint[0] << " y: " << targetPoint[1] << " z: " << targetPo
                }
            }
            else
            {
                groundTruthAngleTemp = 5 * M_PI / 6;
                // ROS_INFO("CCW 60 degree point is a Good point! ");
                // std::cout << "Target coordinate: " << std::endl;
                // std::cout << i << " x: " << targetPoint[0] << " y: " << targetPoint[1] << " z: " << targetPoint[
            }
        }
        else
        {
            groundTruthAngleTemp = 2 * M_PI / 3;
            // ROS_INFO("CCW 30 degree point is a Good point! ");
            // std::cout << "Target coordinate: " << std::endl;
            // std::cout << i << " x: " << targetPoint[0] << " y: " << targetPoint[1] << " z: " << targetPoint[2] <<
        }
    }
    else
    {
        groundTruthAngleTemp = M_PI / 2;
        // ROS_INFO("Default point is a Good point! ");
        // std::cout << "Target coordinate: " << std::endl;
        // std::cout << i << " x: " << targetPoint[0] << " y: " << targetPoint[1] << " z: " << targetPoint[2] << st
    }
    groundTruthDistanceTemp = offset;

    return std::make_pair(targetPoint, availabilty);
}
// std::pair <std::vector<float>, bool> findViablePoints(std::vector<float> inputPoint, float offset){
//     std::vector<float> targetPoint;
//     bool availbility;
//     float targetPhi;
//     bool availabilty = true;

//     targetPoint = offsetPoint(inputPoint, offset);
//     availbility = checkAvailbility(targetPoint);
//     float originalBoxPhi = inputPoint[2];

//     if(!availbility){
//         ROS_INFO("The default point is not reachable");
//         inputPoint[2] = angleAddition(originalBoxPhi, angleIncrement);
//         targetPoint = offsetPoint(inputPoint, offset);
//         availbility = checkAvailbility(targetPoint);

//         if(!availbility){
//             ROS_INFO("The ccw %f point is not reachable", RAD2DEG(angleIncrement));
//             inputPoint[2] = angleAddition(originalBoxPhi, angleIncrement*2);
//             targetPoint = offsetPoint(inputPoint, offset);
//             availbility = checkAvailbility(targetPoint);

//             if(!availbility){
//                 ROS_INFO("The ccw %f point is not reachable", 2*RAD2DEG(angleIncrement));
//                 inputPoint[2] = angleAddition(originalBoxPhi, -angleIncrement);
//                 targetPoint = offsetPoint(inputPoint, offset);
//                 availbility = checkAvailbility(targetPoint);
//                  if(!availbility){
//                      ROS_INFO("The cw %f point is not reachable", -RAD2DEG(angleIncrement));
//                      inputPoint[2] = angleAddition(originalBoxPhi, -angleIncrement*2);
//                      targetPoint = offsetPoint(inputPoint, offset);
//                      availbility = checkAvailbility(targetPoint);

//                      if(!availbility){
//                          ROS_INFO("The cw %f point is not reachable", -2*RAD2DEG(angleIncrement));
//                          availabilty = false;
//                      }

//                      else{
//                          ROS_INFO("CW 60 degree point is a Good point! ");
//                          // std::cout << "Target coordinate: " << std::endl;
//                          // std::cout << i << " x: " << targetPoint[0] << " y: " << targetPoint[1] << " z: " << t
//                      }

//                  }
//                  else{
//                      ROS_INFO("CW 30 degree point is a Good point! ");
//                      // std::cout << "Target coordinate: " << std::endl;
//                      // std::cout << i << " x: " << targetPoint[0] << " y: " << targetPoint[1] << " z: " << target
//                  }

//              }
//              else{
//                  ROS_INFO("CCW 60 degree point is a Good point! ");
//                  // std::cout << "Target coordinate: " << std::endl;
//                  // std::cout << i << " x: " << targetPoint[0] << " y: " << targetPoint[1] << " z: " << targetPoi
//              }

//          }
//          else{
//              ROS_INFO("CCW 30 degree point is a Good point! ");
//              // std::cout << "Target coordinate: " << std::endl;
//              // std::cout << i << " x: " << targetPoint[0] << " y: " << targetPoint[1] << " z: " << targetPoint[2
//          }

//      }
//      else {
//          ROS_INFO("Default point is a Good point! ");
//          // std::cout << "Target coordinate: " << std::endl;
//          // std::cout << i << " x: " << targetPoint[0] << " y: " << targetPoint[1] << " z: " << targetPoint[2] <<
//      }

//      return std::make_pair(targetPoint, availabilty);

// }

bool checkOrientation(std::vector<float> originPoint, std::vector<float> targetPoint)
{
}

//////////////////////////**sort out all the targeted points near each box**///////////////////////////////////////

std::vector<int> targetNum;
std::vector<std::vector<float>> targetPointList;

void sortTargetPoints()
{
    std::vector<float> targetPoint;
    bool availbility;
    float offset = defaultOffset;
    std::pair<std::vector<float>, bool> viablePoint;
    for (int i = 0; i < boxes.coords.size(); i++)
    {
        viablePoint = findViablePoints(boxes.coords[i], offset);
        availbility = viablePoint.second;
        targetPoint = viablePoint.first;

        if (availbility)
        {
            targetPointList.push_back(targetPoint);
            targetNum.push_back(i);
            groundTruthAngle.push_back(groundTruthAngleTemp);
            groundTruthDistance.push_back(groundTruthDistanceTemp);
            // ROS_INFO("Default point is a Good point! ");
            // std::cout << "Target coordinate: " << std::endl;
            // std::cout << i << " x: " << targetPoint[0] << " y: " << targetPoint[1] << " z: " << targetPoint[2] <<
        }
```

```
else
{ // code optimization is needed here and more tests needed here to determine how to approach tight spaces
    offset = defaultOffset - offsetIncrement;
    viablePoint = findViablePoints(boxes.coords[i], offset);
    availbility = viablePoint.second;
    targetPoint = viablePoint.first;
    // ROS_INFO("Distance is: 0.4 %f ", offset);

    if (availbility)
    {
        targetPointList.push_back(targetPoint);
        targetNum.push_back(i);
        groundTruthAngle.push_back(groundTruthAngleTemp);
        groundTruthDistance.push_back(groundTruthDistanceTemp);
        // std::cout << "Target coordinate: " << std::endl;
        // std::cout << i << " x: " << targetPoint[0] << " y: " << targetPoint[1] << " z: " << targetPoint[
    }
    else
    { // code optimization is needed here and more tests needed here to determine how to approach tight spa
        offset = defaultOffset - 2 * offsetIncrement;
        viablePoint = findViablePoints(boxes.coords[i], offset);
        availbility = viablePoint.second;
        targetPoint = viablePoint.first;
        // ROS_INFO("Distance is: 0.3 %f ", offset);

        if (availbility)
        {
            targetPointList.push_back(targetPoint);
            targetNum.push_back(i);
            groundTruthAngle.push_back(groundTruthAngleTemp);
            groundTruthDistance.push_back(groundTruthDistanceTemp);
            // std::cout << "Target coordinate: " << std::endl;
            // std::cout << i << " x: " << targetPoint[0] << " y: " << targetPoint[1] << " z: " << targetPo
        }
        else
        { // code optimization is needed here and more tests needed here to determine how to approach tight
            offset = defaultOffset - 3 * offsetIncrement;
            viablePoint = findViablePoints(boxes.coords[i], offset);
            availbility = viablePoint.second;
            targetPoint = viablePoint.first;
            // ROS_INFO("Distance is: 0.2 %f ", offset);

            if (availbility)
            {
                targetPointList.push_back(targetPoint);
                targetNum.push_back(i);
                groundTruthAngle.push_back(groundTruthAngleTemp);
                groundTruthDistance.push_back(groundTruthDistanceTemp);
                // std::cout << "Target coordinate: " << std::endl;
                // std::cout << i << " x: " << targetPoint[0] << " y: " << targetPoint[1] << " z: " << targ
            }
            else
            { // code optimization is needed here and more tests needed here to determine how to approach t
                offset = defaultOffset + offsetIncrement;
                viablePoint = findViablePoints(boxes.coords[i], offset);
                availbility = viablePoint.second;
                targetPoint = viablePoint.first;
                // ROS_INFO("Distance is: 0.6 %f ", offset);

                if (availbility)
                {
                    targetPointList.push_back(targetPoint);
                    targetNum.push_back(i);
                    groundTruthAngle.push_back(groundTruthAngleTemp);
                    groundTruthDistance.push_back(groundTruthDistanceTemp);
                    // std::cout << "Target coordinate: " << std::endl;
                    // std::cout << i << " x: " << targetPoint[0] << " y: " << targetPoint[1] << " z: " <<
                }
                else
                { // code optimization is needed here and more tests needed here to determine how to approa
                    offset = defaultOffset + offsetIncrement;
                    viablePoint = findViablePoints(boxes.coords[i], offset);
                    availbility = viablePoint.second;
                    targetPoint = viablePoint.first;
                    // ROS_INFO("Distance is: 0.7 %f ", offset);

                    if (availbility)
                    {
                        targetPointList.push_back(targetPoint);
                        targetNum.push_back(i);
                        groundTruthAngle.push_back(groundTruthAngleTemp);
```

```
                                groundTruthDistance.push_back(groundTruthDistanceTemp);
                                // std::cout << "Target coordinate: " << std::endl;
                                // std::cout << i << " x: " << targetPoint[0] << " y: " << targetPoint[1] << " z: "
                            }
                            else
                            { // code optimization is needed here and more tests needed here to determine how to ap
                                offset = defaultOffset + 2 * offsetIncrement;
                                viablePoint = findViablePoints(boxes.coords[i], offset);
                                availbility = viablePoint.second;
                                targetPoint = viablePoint.first;
                                // ROS_INFO("Distance is: 0.8 %f ", offset); //////// CODE????
                                targetPointList.push_back(targetPoint);
                                targetNum.push_back(i);
                                groundTruthAngle.push_back(groundTruthAngleTemp);
                                groundTruthDistance.push_back(groundTruthDistanceTemp);
                                // std::cout << "Target coordinate: " << std::endl;
                                // std::cout << i << " x: " << targetPoint[0] << " y: " << targetPoint[1] << " z: "
                            }
                        }
                    }
                }
            }
        }
    }
}
////////////////////////**check the nearby box and find the closest ones**/////////////////////////////////////
// std::vector<std::vector<float> > unsortedBoxes = boxes.coords;

// std::vector<int> boxOrder;

// std::vector<float> findNearestPoint(std::vector<std::vector<float> > &unsortedBoxes, RobotPose &robotPose){
//      std::vector <float> initPoint {robotPose.x, robotPose.y};
//      std::vector <float> nextPoint, targetPoint, currentPoint;
//      float minDist = std::numeric_limits<float>::infinity();
//      float dist;
//      int boxNum;

//      for (int i; i < unsortedBoxes.size(); i++){
//          nextPoint = unsortedBoxes[i];
//          dist = distance(initPoint, nextPoint);

//          if (dist < minDist){
//              targetPoint = nextPoint;
//              minDist = dist;

//          }
//      }
// }

std::vector<std::vector<float>> sortedTargetPoints;
std::vector<int> sortedTargetIndex;
std::vector<float> sortedAngle, sortedLength;

void plan(std::vector<std::vector<float>> coordinates, std::vector<float> position, std::vector<int> index, std::ve
{
    std::vector<std::vector<float>> next_coordinates = coordinates;
    std::vector<int> next_index = index;
    std::vector<float> next_angle = angle;
    std::vector<float> next_length = length;
    // next index = copy index;

    if (coordinates.size() > 0)
    {
        std::vector<float> distance_list(coordinates.size(), 0);
        for (int i = 0; i < coordinates.size(); i++)
        {
            std::vector<float> pos_to_calculate = coordinates[i];
            distance_list[i] = distance(position, pos_to_calculate);
        }
        int min_distance_index = std::min_element(distance_list.begin(), distance_list.end()) - distance_list.begin
        std::vector<float> next_position = std::vector<float>(coordinates[min_distance_index].begin(), coordinates[
        sortedTargetPoints.push_back(coordinates[min_distance_index]);
        sortedTargetIndex.push_back(index[min_distance_index]);
        sortedAngle.push_back(angle[min_distance_index]);
        sortedLength.push_back(length[min_distance_index]);
        // sorted.index pushback
        next_coordinates.erase(next_coordinates.begin() + min_distance_index);
        next_index.erase(next_index.begin() + min_distance_index);
        next_angle.erase(next_angle.begin() + min_distance_index);
        next_length.erase(next_length.begin() + min_distance_index);
        // next index erase
        // ROS_INFO("index:%d ", min_distance_index);

        plan(next_coordinates, next_position, next_index, next_angle, next_length);
    }
    else
    {
        for (int i; i < sortedTargetPoints.size(); i++)
        {
            // ROS_INFO("The sorted index is: %d", sortedTargetIndex[i]);
            // ROS_INFO("goal_position: x: %f, y: %f, z: %f", sortedTargetPoints[i][0], sortedTargetPoints[i][1], s
        }
    }
}

std::vector<std::vector<std::vector<float>>> sort_points(std::vector<std::vector<float>> coordinates)
{
    std::vector<std::vector<std::vector<float>>> sorted(coordinates.size(), std::vector<std::vector<float>>(coordin
    for (int i = 0; i < coordinates.size(); i++)
    {
        std::vector<float> distance_list(coordinates.size(), 0);
        for (int j = 0; j < coordinates.size(); j++)
        {
            std::vector<float> current_pos = coordinates[i];
            std::vector<float> pos_compare = coordinates[j];
            // ROS_INFO("y1: %f", pos_compare[1]);
            // ROS_INFO("y2: %f", current_pos[1]);
            distance_list[j] = distance(current_pos, pos_compare);
        }

        // ROS_INFO("distance1: %f", distance_list[0]);
        // ROS_INFO("distance2: %f", distance_list[1]);
        // ROS_INFO("distance3: %f", distance_list[2]);

        // std::vector<std::vector<float>> nearest_list(coordinates.size(),std::vector<float>(3,0));
        std::vector<std::vector<float>> coordinates_copy = coordinates;
        for (int k = 0; k < coordinates.size(); k++)
        {
            // int size = nearest_list.size();
            // ROS_INFO("size: %d", size);
            int min_distance_index = std::min_element(distance_list.begin(), distance_list.end()) - distance_list.b
            sorted[i][k] = coordinates_copy[min_distance_index];
            distance_list.erase(distance_list.begin() + min_distance_index);
            coordinates_copy.erase(coordinates_copy.begin() + min_distance_index);
            // int size = coordinates_copy.size();
            // ROS_INFO("size: %d", size);
        }
        // sorted[i] = nearest_list;
    }
    // ROS_INFO("x1: %f", sorted[0][0][0]);
    // ROS_INFO("y1: %f", sorted[0][0][1]);
    // ROS_INFO("x1: %f", sorted[0][1][0]);
    // ROS_INFO("y2: %f", sorted[0][1][1]);
    // ROS_INFO("x3: %f", sorted[0][2][0]);
    // ROS_INFO("y3: %f", sorted[0][2][1]);
    // ROS_INFO("2x1: %f", sorted[1][0][0]);
    // ROS_INFO("2y1: %f", sorted[1][0][1]);
    // ROS_INFO("2x2: %f", sorted[1][1][0]);
    // ROS_INFO("2y2: %f", sorted[1][1][1]);
    // ROS_INFO("2x3: %f", sorted[1][2][0]);
    // ROS_INFO("2y3: %f", sorted[1][2][1]);
    // ROS_INFO("3x1: %f", sorted[2][0][0]);
    // ROS_INFO("3y1: %f", sorted[2][0][1]);
    // ROS_INFO("3x2: %f", sorted[2][1][0]);
    // ROS_INFO("3y2: %f", sorted[2][1][1]);
    // ROS_INFO("3x3: %f", sorted[2][2][0]);
    // ROS_INFO("3y3: %f", sorted[2][2][1]);

    return sorted;
}

/////////////////////////**Go to the specific point -- modified moveToPoint**/////////////////////////////////////

bool gotoPoint(std::vector<float> targetPoint)
{
    bool success = navigation.moveToGoal(targetPoint);
    int trailNum = 0;

    while (!success && trailNum < 5)
    {
        ROS_INFO("Initial Path planning failed; try again...");
        // bool finishClearMap = clearCostMap();
        success = navigation.moveToGoal(targetPoint);
        trailNum++;
    }

    if (!success)
    {
        ROS_INFO("Failed to reach the point: x %f, y %f, phi %f", targetPoint[0], targetPoint[1], targetPoint[2]);
    }
    else
    {
        ROS_INFO("Target is reached successfully! ");
    }
}

//////////////////////////** execuate the below function if and only if there are boxes nearby -- check proximity r
std::vector<float> alternativePointOne;
std::vector<float> alternativePointTwo;
float angularOffset = M_PI / 40;
std::vector<std::vector<std::vector<float>>> alternativePointsArray;
std::vector<std::vector<float>> alternativeGroundTruthAngle, alternativeGroundTruthDistance;

std::vector<std::vector<std::vector<float>>> alternativePoints()
{

    for (int i = 0; i < 10; i++)
    {
        float alternativeX, alternativeY, alternativePhi, alternativePhiOne, alternativePhiTwo, alternativeAngle, a
        alternativeX = sortedTargetPoints[i][0];
        alternativeY = sortedTargetPoints[i][1];
        alternativePhi = sortedTargetPoints[i][2];
        alternativeAngle = sortedAngle[i];
        alternativeDistance = sortedLength[i];

        alternativePhiOne = alternativePhi + angularOffset;
        alternativePhiTwo = alternativePhi - angularOffset;
        alternativeAngleOne = alternativeAngle + angularOffset;
        alternativeAngleTwo = alternativeAngle - angularOffset;
        alternativePointOne = {alternativeX, alternativeY, alternativePhiOne};
        alternativePointTwo = {alternativeX, alternativeY, alternativePhiTwo};

        int j = 0;

        std::vector<std::vector<float>> oneAlternativeSet;
        std::vector<float> oneAngleSet, oneDistanceSet;

        oneAlternativeSet.push_back(alternativePointOne);
        oneAngleSet.push_back(alternativeAngleOne);
        oneDistanceSet.push_back(alternativeDistance);
        // if(checkAvailbility(alternativePointOne)){
        //     oneAlternativeSet.push_back(alternativePointOne);
        //     j++;
        // }

        oneAlternativeSet.push_back(sortedTargetPoints[i]);
        oneAngleSet.push_back(alternativeAngle);
        oneDistanceSet.push_back(alternativeDistance);
        // j++;
        oneAlternativeSet.push_back(alternativePointTwo);
        oneAngleSet.push_back(alternativeAngleTwo);
        oneDistanceSet.push_back(alternativeDistance);

        // if(checkAvailbility(alternativePointOne)){
        //     oneAlternativeSet.push_back(alternativePointTwo);
        //     j++;
        // }

        alternativePointsArray.push_back(oneAlternativeSet);
        alternativeGroundTruthAngle.push_back(oneAngleSet);
        alternativeGroundTruthDistance.push_back(oneDistanceSet);
    }
    return alternativePointsArray;
}
//////////////////////////** Take 3 Snap Shots of the same target point -- main function control logic ??? **///////

//////////////////////////**check the nearby box and find the closest ones**//////////////////////////////////////

// int checkNum = 0;
// std::vector <float> currentPose = initPose;
// std::vector <float> nextPose = boxes.coords[checkNum];
// std::vector <int> list;
// std::vector<int> getNearestPath(){

//      float minDist = std::numeric_limits<float>::infinity();
//      int box_id = -1;
//      float dist;

//      for (int i = 0; i < boxes.coords.size(); i++){
//          dist = distance (currentPose, boxes.coords[i]);
//          if(dist < minDist){
//              box_id = i;
//              minDist = dist;
//          }
//      }

//      list.push_back(box_id);
//      checkNum++;
//      currentPose = nextPose;

//      return (checkNum == (boxes.coords.size()-1) ? list : getNearestPath());

// }

// std::vector<std::vector<float> > sort(std::vector<std::vector<float> > coordinates, std::vector<float> position)
//      std::vector<std::vector<float>> next_coordinates = coordinates;
//      std::vector<std::vector<float>> sorted;

//      if (coordinates.size() >0) {
//          std::vector<float> distance_list(coordinates.size(),0);
//          for(int i=0; i< coordinates.size() ;i++){
//              std::vector<float> pos_to_calculate = std::vector<float>(coordinates[i].begin(), coordinates[i].begi
//              distance_list[i] = distance(position,pos_to_calculate);
//          }
//          int min_distance_index = std::min_element(distance_list.begin(),distance_list.end()) - distance_list.beg

//          std::vector<float> next_position = std::vector<float>(coordinates[min_distance_index].begin(),coordinate

//          sorted.push_back(position[min_distance_index]);
//          next.coordinates.erase(next.coordinates.begin()+min_distance_index);
//          sort(next_coordinates,next_position);
//      }
//      else{;
//          return sorted;
//      }
// }

// std::vector<std::vector<float>> nearest_neighbour( std::vector<std:vector<float> > goal_coordinates){
//      std::vector current_position = {current.position.x,current.position.y};
//      std::vector sorted;
//      sort(goal_coordinates,current_position);
// }

/////////////////////////////////////////////////////////////////////////////////////
////////////////////////**Navigation Part**//////////////////////////////////////////
/////////////////////////////////////////////////////////////////////////////////////
void printToFile(std::vector<std::vector<float>> workableArray, int indexin)
{
    myfileMap.open("MapText.txt", std::ios_base::app);
    // myfileMap<<"indexin:   "<<indexin<<"\n";
    for (int tempRowCounter = 0; tempRowCounter < workableArray.size(); tempRowCounter++)
    {
        if (workableArray[0].size() > 0 && workableArray[tempRowCounter][2] > 0 && workableArray[tempRowCounter][4]
        {
            for (int tempColCounter = 0; tempColCounter < 6; tempColCounter++)
            {

                myfileMap << workableArray[tempRowCounter][tempColCounter] << ",␣␣␣␣␣";
            }

            // myfileMap <<"\n("<< workableArray[tempRowCounter][6] << ",      ";
            // myfileMap << workableArray[tempRowCounter][7] << ")\n";
            // myfileMap <<"("<< workableArray[tempRowCounter][8] << ",      ";
            // myfileMap << workableArray[tempRowCounter][9] << ")\n";
            // myfileMap <<"("<< workableArray[tempRowCounter][10] << ",      ";
            // myfileMap << workableArray[tempRowCounter][11] << ")\n";
            // myfileMap <<"("<< workableArray[tempRowCounter][12] << ",      ";
            // myfileMap << workableArray[tempRowCounter][13] << ")\n";

            myfileMap << workableArray[tempRowCounter][14] << ",␣␣␣␣␣";
            myfileMap << workableArray[tempRowCounter][15] << ",␣␣␣␣␣";
            myfileMap << workableArray[tempRowCounter][16] << ",␣␣␣␣␣";
            myfileMap << workableArray[tempRowCounter][17] << ",␣␣␣␣␣";
            myfileMap << workableArray[tempRowCounter][18] << ",␣␣␣␣␣";
            myfileMap << workableArray[tempRowCounter][19] << ",␣␣␣␣␣";
            myfileMap << workableArray[tempRowCounter][20] << ",␣␣␣␣␣";
            myfileMap << "\n";
        }
    }
    myfileMap << "CONNECTOR\n";
    myfileMap.close();
}

int main(int argc, char **argv)
{

    // Setup ROS.
    ros::init(argc, argv, "contest2");
    ros::NodeHandle nh;
    // Robot pose object + subscriber.
    RobotPose robotPose(0, 0, 0);
    // Initialize image objectand subscriber.
    ImagePipeline imagePipeline(nh);

    ros::Subscriber amclSub = nh.subscribe("/amcl_pose", 1, &RobotPose::poseCallback, &robotPose);

    ////////////////////////////////////////////////////////////////////////////////////////////////////
    /////////////////////////**Initial Localization Part**/////////////////////////////////////////////
    ////////////////////////////////////////////////////////////////////////////////////////////////

    std::chrono::time_point<std::chrono::system_clock> start;
    start = std::chrono::system_clock::now();
    float deltaTime = 0;

    while (deltaTime < 3)
    {
        deltaTime = std::chrono::duration_cast<std::chrono::seconds>(std::chrono::system_clock::now() - start).coun
        publishVelocity(0, angularMax);
    }
    ROS_INFO("Finished␣up␣initialization!␣");
    publishVelocity(0, 0);

    ////////////////////////////////////////////////////////////////////////////////////////////////////
    /////////////////////////**Getting the robot initial pose**////////////////////////////////////////
    ////////////////////////////////////////////////////////////////////////////////////////////////////
    // record the initial position of the turtlebot

    std::vector<float> currentPose = {robotPose.x, robotPose.y, robotPose.phi};
    initPose = currentPose;

    ROS_INFO("The␣initial␣position␣is␣at:␣x:%f,␣y:%f␣phi:␣%f", robotPose.x, robotPose.y, robotPose.phi);

    // Initialize box coordinates and templates

    ////////////////////////////////////////////////////////////////////////////////////////////////////
    /////////////////////////**Box Coordinates Loading**///////////////////////////////////////////////
    ////////////////////////////////////////////////////////////////////////////////////////////////////

    if (!boxes.load_coords() || !boxes.load_templates())
    {
        std::cout << "ERROR:␣could␣not␣load␣coords␣or␣templates" << std::endl;
        return -1;
    }

    for (int i = 0; i < boxes.coords.size(); i++)
    {
        std::cout << "Box␣Index:" << i << "␣Box␣Coordinates"
                  << "␣x:␣" << boxes.coords[i][0] << "␣y:␣" << boxes.coords[i][1] << "␣z:␣" << boxes.coords[i][2] <
    }

    std::cout << "␣___________________␣" << std::endl;
    std::cout << "" << std::endl;

    ////////////////////////////////////////////////////////////////////////////////////////////////////
    /////////////////////////** Calculating the target coordinates**//////////////////////////////////
    ////////////////////////////////////////////////////////////////////////////////////////////////////

    sortTargetPoints();

    for (int i = 0; i < targetPointList.size(); i++)
    {
        std::cout << "Target␣Index:" << targetNum[i] << "␣Targeted␣Point␣Coordiates:␣x:␣" << targetPointList[i][0] <
        // navigation.moveToGoal(targetPointList[i]);
    }
```

```
std::cout << "--------------------␣" << std::endl;
std::cout << "" << std::endl;

////////////////////////////////////////////////////////////////////////////////////////////////////////
/////////////////////////** Path Planning Algorithm **////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////////////////////

plan(targetPointList, initPose, targetNum, groundTruthAngle, groundTruthDistance);

for (int i = 0; i < alternativePointsArray.size(); i++)
{
    int j = 0;
    do
    {
        std::cout << "Alternative␣Index:" << targetNum[i] << "␣Alternative␣Point␣Coordiates:␣x:␣" << alternative
        std::cout << "␣The␣Angle␣Ground␣Truth␣is:␣" << alternativeGroundTruthAngle[i][j] << std::endl;
        // navigation.moveToGoal(alternativePointsArray[i][j]);
        ros::Duration(2.5).sleep();
        j++;
    } while (j < alternativePointsArray[i].size());
    std::cout << "--------------------␣" << std::endl;
}

// for (int i =0; i< sortedTargetPoints.size(); i++){
//     std::cout << "Going to Point: " << sortedTargetIndex[i] << " at the Coordiates: x: " << sortedTargetPoin
//     //navigation.moveToGoal(sortedTargetPoints[i]);

// }

////////////////////////////////////////////////////////////////////////////////////////////////////////
/////////////////////////** Alternative Three Points around the Box **////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////////////////////
alternativePoints();

/////////////////////** Move back to the origin ** ////////////////////////////////////////////////////
// navigation.moveToGoal(initPose);
//  std::cout << "Moved back to the original position " << std::endl;
//  Execute strategy.
int spinInntialTwice = 0;

int spinInitialThres = 1;

myfileMap.open("MapText.txt");
myfileMap << "␣Starting␣Fusoc\n";
myfileMap.close();
myfileLaser.open("LaserText.txt");
myfileLaser << "hjhgfjgjh:␣"<< "\n";
myfileLaser.close();
energyMatrix.open("changeEngine.txt");
energyMatrix << "";
energyMatrix.close();
int currentCounter = 0;
// 0=travellingmode
// 1=detectionmode
int currentFlag = 0;
int numOimgPipeline = 3;
int imgPipelineCouner = 0;
std::reverse(targetPointList.begin(), targetPointList.end());
// std::vector<std::vector<float>> tmpesTesint2 = imagePipeline.getTemplateID(boxes);
int isBlank = 0;
int theArrow = -1;
int totalNotIns = 0;
//std::vector<std::pair<std::vector<KeyPoint>, Mat>> preCompuatedKeyandMat = imagePipeline.preComputekeyFeature
//imagePipeline.preCompuitedVector= imagePipeline.preComputekeyFeatures(boxes);

while (ros::ok() && spinInntialTwice < spinInitialThres)
{
    ros::spinOnce();
    /***YOUR CODE HERE***/
    // Use: boxes.coords
    // Use: robotPose.x, robotPose.y, robotPose.phi
    // imagePipeline.getTemplateID(boxes);
    std::vector<std::vector<float>> tmpesTesint2;
    std::vector<std::vector<std::vector<float>>> emparorTensor;
    ros::spinOnce();
    int howManyImagePerPosition = 7;

    for (int whichBoxCounter = 0; whichBoxCounter < alternativePointsArray.size(); whichBoxCounter++)
    {
        int whichBoxJCounter = 0;
        isBlank = 0;
```

```
theArrow = −1;
do
{
    std::cout << "Alternative␣Index:" << targetNum[whichBoxCounter] << "␣Alternative␣Point␣Coordiates:␣x
    std::cout << "␣The␣Angle␣Ground␣Truth␣is:␣" << alternativeGroundTruthAngle[whichBoxCounter][whichBo
    navigation.moveToGoal(alternativePointsArray[whichBoxCounter][whichBoxJCounter]);

    myfileMap.open("MapText.txt", std::ios_base::app);
    myfileMap << "Visited␣the:␣␣" << sortedTargetIndex[whichBoxCounter] << "times:␣" << whichBoxJCounte
    myfileMap.close();

    ros::spinOnce();
    for (int imagePerPosCounter = 0; imagePerPosCounter < howManyImagePerPosition; imagePerPosCounter++
    {
        ros::spinOnce();
        tmpesTesint2 = imagePipeline.getTemplateID(boxes);
        for (int pusasnCounter = 0; pusasnCounter < tmpesTesint2.size(); pusasnCounter++)
        {
            tmpesTesint2[pusasnCounter][21] = RAD2DEG(alternativeGroundTruthAngle[whichBoxCounter][which
        }
        printToFile(tmpesTesint2, imagePerPosCounter);
        emparorTensor.push_back(tmpesTesint2);
    }

    myfileMap.open("MapText.txt", std::ios_base::app);
    myfileMap << "\n−end";
    for (int asefs = 0; asefs < 128; asefs++)
    {
        myfileMap << "−";
    }
    myfileMap << "\n\n";
    myfileMap.close();
    ros::spinOnce();

    // ros::Duration(2.5).sleep();
    whichBoxJCounter++;
} while (whichBoxJCounter < alternativePointsArray[whichBoxCounter].size());
std::cout << "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣" << std::endl;

myfileLaser.open("LaserText.txt", std::ios_base::app);
myfileLaser << "boxes:␣" << whichBoxCounter << "\n";
myfileLaser.close();

std::vector<std::vector<float>> turtleHmmaerThrow;
for (int i2 = 0; i2 < emparorTensor.size(); i2++)
{
    for (int j2 = 0; j2 < emparorTensor[i2].size(); j2++)
    {
        turtleHmmaerThrow.push_back(emparorTensor[i2][j2]);
    }
}
// for(int putAngleIncounter=0;putAngleIncounter<3  ;putAngleIncounter++)
// {
//     emparorTensor[whichBoxCounter][putAngleIncounter][14] =  alternativeGroundTruthAngle[whichBoxCou

// }

std::vector<std::pair<int, int>> selectionArraty;

for (int pickselectCounter = 0; pickselectCounter < turtleHmmaerThrow.size(); pickselectCounter++)
{

    int ifItsINThere = 0;

    for (int interlpickCOinter = 0; interlpickCOinter < selectionArraty.size(); interlpickCOinter++)
    {

        if (turtleHmmaerThrow[pickselectCounter][0] == selectionArraty[interlpickCOinter].first)
        {
            ifItsINThere = 1;
            selectionArraty[interlpickCOinter].second += 1;
        }
    }
    if (ifItsINThere == 0)
    {
        std::pair<int, int> tempsecsds(roundf(turtleHmmaerThrow[pickselectCounter][0]), 1);
        selectionArraty.push_back(tempsecsds);
    }
}

if (turtleHmmaerThrow.size() < 10)
```

```
            {
                isBlank = 1;
            }
            std::vector<int> goodu;
            for (int remveLessThwanCounter = 0; remveLessThwanCounter < selectionArraty.size(); remveLessThwanCount
            {
                if (selectionArraty[remveLessThwanCounter].second > 7)
                {
                    goodu.push_back(selectionArraty[remveLessThwanCounter].first);
                }
            }
            if (goodu.size() == 1)
            {
                theArrow = goodu[0];
            }

            if (isBlank == 1)
            {
                energyMatrix.open("changeEngine.txt", std::ios_base::app);
                energyMatrix << "boxNumnber:␣" << sortedTargetIndex[whichBoxCounter]+1 << "␣␣␣tag␣is␣"
                            << "blank\n";
                energyMatrix.close();
            }
            else if (theArrow != -1)
            {
                //might be blank
                //cherry
                energyMatrix.open("changeEngine.txt", std::ios_base::app);
                energyMatrix << "boxNumnber:␣" << sortedTargetIndex[whichBoxCounter]+1 << "␣␣␣tag␣is␣" << theArrow -
                energyMatrix.close();
            }
            else
            {
                for (int i2 = 0; i2 < emparorTensor.size(); i2++)
                {
                    for (int j2 = 0; j2 < emparorTensor[i2].size(); j2++)
                    {
                        myfileLaser.open("LaserText.txt", std::ios_base::app);
                        myfileLaser << emparorTensor[i2][j2][0] << ",␣␣␣␣␣";
                        myfileLaser << emparorTensor[i2][j2][1] << ",␣␣␣␣␣";
                        myfileLaser << emparorTensor[i2][j2][2] << ",␣␣␣␣␣";
                        myfileLaser << emparorTensor[i2][j2][3] << ",␣␣␣␣␣";
                        myfileLaser << emparorTensor[i2][j2][4] << ",␣␣␣␣␣";
                        myfileLaser << emparorTensor[i2][j2][5] << ",␣␣␣␣␣";
                        myfileLaser << emparorTensor[i2][j2][16] << ",␣␣␣␣␣";
                        myfileLaser << emparorTensor[i2][j2][17] << ",␣␣␣␣␣";
                        myfileLaser << emparorTensor[i2][j2][18] << ",␣␣␣␣␣";
                        myfileLaser << emparorTensor[i2][j2][19] << ",␣␣␣␣␣";
                        myfileLaser << emparorTensor[i2][j2][20] << ",␣␣␣␣␣";
                        myfileLaser << emparorTensor[i2][j2][21] << ",␣␣␣␣␣";
                        myfileLaser << "CONNECTOR\n";
                        myfileLaser.close();
                    }
                }

                totalNotIns += 1;
                unmatchedBoxeSArray.push_back(alternativePointsArray[whichBoxCounter][whichBoxJCounter]);
            }
            emparorTensor.clear();
        }

        energyMatrix.open("changeEngine.txt", std::ios_base::app);
        energyMatrix << "unmatched:␣" << totalNotIns << "\n";
        energyMatrix.close();

        std::cout << "-------------------␣" << std::endl;
        std::cout << "-------------------␣" << std::endl;
        std::cout << "" << std::endl;

        navigation.moveToGoal(initPose);
        std::cout << "Moved␣back␣to␣the␣original␣position␣" << std::endl;

        spinInntialTwice += 1;

        ros::Duration(0.01).sleep();
    }
    return 0;
}
```

B Source Code on imagepipeline.cpp

```
#include <imagePipeline.h>
#include <iostream>

#include "opencv2/core.hpp"
#include "opencv2/calib3d.hpp"
#include "opencv2/highgui.hpp"
#include "opencv2/imgproc.hpp"
#include "opencv2/features2d.hpp"
#include "opencv2/xfeatures2d.hpp"
#include "std_msgs/String.h"
#include <sstream>
#include <fstream>
#include <float.h>
#include <vector>
#include <bits/stdc++.h>

#include <algorithm>
#include <iterator>
#include "dataStructure.h"

using namespace cv;
using namespace cv::xfeatures2d;

#define IMAGE_TYPE sensor_msgs::image_encodings::BGR8
#define IMAGE_TOPIC "camera/rgb/image_raw" // kinect:"camera/rgb/image_raw" webcam:"camera/image"

std::ofstream myfile2;

ImagePipeline::ImagePipeline(ros::NodeHandle &n)
{
    image_transport::ImageTransport it(n);
    sub = it.subscribe(IMAGE_TOPIC, 1, &ImagePipeline::imageCallback, this);
    isValid = false;
}

void ImagePipeline::imageCallback(const sensor_msgs::ImageConstPtr &msg)
{
    try
    {
        if (isValid)
        {
            img.release();
        }
        img = (cv_bridge::toCvShare(msg, IMAGE_TYPE)->image).clone();
        cv::cvtColor(img, img, cv::COLOR_BGR2GRAY);

        isValid = true;
    }
    catch (cv_bridge::Exception &e)
    {
        std::cout << "ERROR: Could not convert from " << msg->encoding.c_str()
                  << " to " << IMAGE_TYPE.c_str() << "!" << std::endl;
        isValid = false;
    }
}

bool ImagePipeline::isHeartBroken(float x1, float y1,
                                  float x2, float y2,
                                  float x3, float y3,
                                  float x4, float y4)
{
    std::vector<float> xList = {x1, x2, x3, x4};
    std::vector<float> yList = {y1, y2, y3, y4};
    float maxX = std::max({x1, x2, x3, x4});
    float maxY = std::max({y1, y2, y3, y4});
    float minX = std::min({x1, x2, x3, x4});
    float minY = std::min({y1, y2, y3, y4});

    float newX;
    float newY;

    float m14 = (y4 - y1) / (x4 - x1);
    float b14 = y4 - x4 * m14;
    float m23 = (y3 - y2) / (x3 - x2);
    float b23 = y3 - x3 * m23;

    newX = (b23 - b14) / (m14 - m23);
    newY = m14 * newX + b14;

    if (newX > minX && newX < maxX && newY > minY && newY < maxY)
    {

        return 0;
    }
    else
    {

        float m12 = (y2 - y1) / (x2 - x1);
        float b12 = y1 - x1 * m12;
        float m34 = (y4 - y3) / (x4 - x3);
        float b34 = y3 - x3 * m34;
        newX = (b34 - b12) / (m12 - m34);
        newY = m34 * newX + b34;

        if (newX > minX && newX < maxX && newY > minY && newY < maxY)
        {
            return 0;
        }
        return 1;
    }
}
float ImagePipeline::isHeartTwisted(float x1, float y1,
                                    float x2, float y2,
                                    float x3, float y3)
{

    float pivotAngleX = x1;
    float pivotAngleY = y1;
    float sideAngle1X = x2;
    float sideAngle1Y = y2;
    float sideAngle2X = x3;
    float sideAngle2Y = y3;

    float distPivotTo1 = sqrt((pivotAngleX - sideAngle1X) * (pivotAngleX - sideAngle1X) + (pivotAngleY - sideAngle1Y
    float distPivotTo2 = sqrt((pivotAngleX - sideAngle2X) * (pivotAngleX - sideAngle2X) + (pivotAngleY - sideAngle2Y
    float dist1to2 = sqrt((sideAngle2X - sideAngle1X) * (sideAngle2X - sideAngle1X) + (sideAngle2Y - sideAngle1Y) *

    float tempStsces = -distPivotTo1 * distPivotTo1 - distPivotTo2 * distPivotTo2 + dist1to2 * dist1to2;

    float tmepsfset2 = tempStsces / (-2 * distPivotTo1 * distPivotTo2);
    float finAgnle1 = acos(tmepsfset2);

    return finAgnle1;
}



// std::vector<std::pair<std::vector<KeyPoint>, Mat>> ImagePipeline::preComputekeyFeatures(Boxes &boxes)
// {
//     std::vector<std::pair<std::vector<KeyPoint>, Mat>> returnaARts;
//     Mat imgBoxes;
//     Mat img_object;

//     for (int indi = 0; indi < boxes.templates.size(); indi++)
//     {
//         img_object = boxes.templates[indi];

//         int minHessian = 300;
//         Ptr<SURF> detector = SURF::create(minHessian);
//         std::vector<KeyPoint> keypoints_object;
//         Mat descriptors_object;
//         detector->detectAndCompute(img_object, noArray(), keypoints_object, descriptors_object);
//         std::pair<std::vector<KeyPoint>, Mat> tempputinLater(  keypoints_object,descriptors_object );
//         returnaARts.push_back(tempputinLater);
//     }

//     return returnaARts;
// }



std::vector<std::vector<float>> ImagePipeline::getTemplateID(Boxes &boxes)
{
    cv::waitKey(10);
    // myfile2 << "Starting the stream :\n";

    std::vector<std::vector<float>> theJuice;
    for (int i = 0; i < boxes.templates.size(); i++)
```

```
{
    theJuice.push_back(std::vector<float >());
    for (int j = 0; j < 25; j++)
    {
        theJuice[i].push_back(-69.420);
    }
}

for (int i = 0; i < boxes.templates.size(); i++)
{

    theJuice[i][0] = i;
}



if(preCompuitedVector.size()==0)
{
    std::vector<std::pair<std::vector<KeyPoint>, Mat>> returnaARts;
    Mat imgBoxes;
    Mat img_object;

    for (int indi = 0; indi < boxes.templates.size(); indi++)
    {
        img_object = boxes.templates[indi];

        int minHessian = 300;
        Ptr<SURF> detector = SURF::create(minHessian);
        std::vector<KeyPoint> keypoints_object;
        Mat descriptors_object;
        detector->detectAndCompute(img_object, noArray(), keypoints_object, descriptors_object);
        std::pair<std::vector<KeyPoint>, Mat> tempputinLater(  keypoints_object,descriptors_object );
        returnaARts.push_back(tempputinLater);
    }
    preCompuitedVector= returnaARts;
}
// Indexing
// 0: tagid
// 1: numberOfMatches
// 2: area
// 3:

// if it is theJuice[0][1]  -70, it means that it doesnt even work
int template_id = -1;
if (!isValid)
{
    std::cout << "ERROR: INVALID IMAGE!" << std::endl;
    template_id = -1;
    theJuice[0][1] = -70;
    return theJuice;
}
else if (img.empty() || img.rows <= 0 || img.cols <= 0)
{
    std::cout << "ERROR: VALID IMAGE, BUT STILL A PROBLEM EXISTS!" << std::endl;
    std::cout << "img.empty():" << img.empty() << std::endl;
    std::cout << "img.rows:" << img.rows << std::endl;
    std::cout << "img.cols:" << img.cols << std::endl;

    template_id = -1;
    theJuice[0][1] = -70;
    return theJuice;
}
else
{
    // fuck off
    // cherry

    myfile2.open("example2.txt", std::ios_base::app);
    myfile2 << "\nsucs\n\n\n";
    myfile2.close();

    Mat imgFromCamera = img;
    Mat imgBoxes;
    Mat img_object;

    int minHessian = 300;
    Ptr<SURF> detectorTemp = SURF::create(minHessian);
    std::vector<KeyPoint> keypoints_scene;
    Mat descriptors_scene;
    detectorTemp->detectAndCompute(imgFromCamera, noArray(), keypoints_scene, descriptors_scene);
```

```
ROS_INFO("istasef");
for (int indi = 0; indi < boxes.templates.size(); indi++)
{
    img_object = boxes.templates[indi];

    //-- Step 1: Detect the keypoints using SURF Detector, compute the descriptors
   // int minHessian = 300;
    //Ptr<SURF> detector = SURF::create(minHessian);
    //ROS_INFO("preCompuitedVectorsize: %d",preCompuitedVector.size());
    std::vector<KeyPoint> keypoints_object=preCompuitedVector[indi].first; //, keypoints_scene;
    Mat descriptors_object=preCompuitedVector[indi].second;              //, descriptors_scene;
    //detector->detectAndCompute(img_object, noArray(), keypoints_object, descriptors_object);

    // if(indi==1)
    // {
    //      myfile2.open("example2.txt",std::ios_base::app);
    //      int sefssfsef=keypoints_object.size();
    //      if(keypoints_object.size()>100)
    //      {
    //          sefssfsef=100;
    //      }
    //      for(int efef=50;efef<sefssfsef;efef++)
    //      {

    //          myfile2<<"("<<keypoints_object[efef].pt.x<<","<<keypoints_object[efef].pt.y<<"),";
    //      }
    //      myfile2<<"size: "<<keypoints_object.size()<<"\n";
    //      myfile2.close();
    // }

    // detector->detectAndCompute(imgFromCamera, noArray(), keypoints_scene, descriptors_scene);

    //-- Step 2: Matching descriptor vectors with a FLANN based matcher
    // Since SURF is a floating-point descriptor NORM_L2 is used
    Ptr<DescriptorMatcher> matcher = DescriptorMatcher::create(DescriptorMatcher::FLANNBASED);
    std::vector<std::vector<DMatch>> knn_matches;
    matcher->knnMatch(descriptors_object, descriptors_scene, knn_matches, 2);
    //-- Filter matches using the Lowe's ratio test
    // cherry 0.77f
    const float ratio_thresh = 0.77f;
    std::vector<DMatch> good_matches;
    for (size_t i = 0; i < knn_matches.size(); i++)
    {
        if (knn_matches[i][0].distance < ratio_thresh * knn_matches[i][1].distance)
        {
            good_matches.push_back(knn_matches[i][0]);
        }
    }
    // ROS_INFO("heinehinsyeyoo");
    //  ROS_INFO("good matches size  %d and  index  %d",good_matches.size(),i);
    // myfile2 << good_matches.size() << "      " << indi + 1 << "       ";
    theJuice[indi][2] = good_matches.size();
    theJuice[indi][3] = indi + 1;
    if (keypoints_object.size() > 0)
    {
        theJuice[indi][5] = (float)good_matches.size() / (float)keypoints_object.size();
    }
    else
    {
        theJuice[indi][5] = 0;
    }
    //-- Draw matches
    // Mat img_matches;
    // drawMatches(img_object, keypoints_object, imgFromCamera, keypoints_scene, good_matches, img_matches,
    //             Scalar::all(-1), std::vector<char>(), DrawMatchesFlags::NOT_DRAW_SINGLE_POINTS);

    //-- Localize the object
    std::vector<Point2f> obj;
    std::vector<Point2f> scene;
    for (size_t i = 0; i < good_matches.size(); i++)
    {
        //-- Get the keypoints from the good matches
        obj.push_back(keypoints_object[good_matches[i].queryIdx].pt);
        scene.push_back(keypoints_scene[good_matches[i].trainIdx].pt);
    }

    int verificationable = scene.size();
    if (verificationable <= 3)
    {
        // myfile2 << "less than 4 points\n";
        theJuice[indi][1] = -60;
        continue;
    }
    // cherry   checks with blanks, to see if there are less .

    Mat H = findHomography(obj, scene, RANSAC);

    // cherry   see if blank
    if (H.empty())
    {
        // myfile2 << "h is mepy\n";

        theJuice[indi][1] = -50;
        continue;
    }

    //-- Get the corners from the image_1 ( the object to be "detected" )
    std::vector<Point2f> obj_corners(4);
    obj_corners[0] = Point2f(0, 0);
    obj_corners[1] = Point2f((float)img_object.cols, 0);
    obj_corners[2] = Point2f((float)img_object.cols, (float)img_object.rows);
    obj_corners[3] = Point2f(0, (float)img_object.rows);
    std::vector<Point2f> scene_corners(4);
    perspectiveTransform(obj_corners, scene_corners, H);
    theJuice[indi][6] = scene_corners[0].x;
    theJuice[indi][7] = scene_corners[0].y;
    theJuice[indi][8] = scene_corners[1].x;
    theJuice[indi][9] = scene_corners[1].y;
    theJuice[indi][10] = scene_corners[2].x;
    theJuice[indi][11] = scene_corners[2].y;
    theJuice[indi][12] = scene_corners[3].x;
    theJuice[indi][13] = scene_corners[3].y;

    float zehoX = scene_corners[0].x;
    float zehoY = scene_corners[0].y;
    float uneX = scene_corners[1].x;
    float uneY = scene_corners[1].y;
    float tooX = scene_corners[2].x;
    float tooY = scene_corners[2].y;
    float treeX = scene_corners[3].x;
    float treeY = scene_corners[3].y;

    theJuice[indi][14] = 0;

    if (zehoX >= uneX)
    {
        theJuice[indi][14] += 1;
    }
    if (zehoY >= treeY)
    {
        theJuice[indi][14] += 2;
    }
    if (uneY >= tooY)
    {
        theJuice[indi][14] += 4;
    }
    if (treeX >= tooX)
    {
        theJuice[indi][14] += 8;
    }
    theJuice[indi][15] = 0;
    // if = 0 then bad
    if (isHeartBroken(zehoX, zehoY, uneX, uneY, tooX, tooY, treeX, treeY) == 0)
    {
        theJuice[indi][15] = -7;
    }

    theJuice[indi][16] = isHeartTwisted(zehoX, zehoY, uneX, uneY, treeX, treeY) * 180 / M_PI;
    theJuice[indi][17] = isHeartTwisted(uneX, uneY, zehoX, zehoY, tooX, tooY) * 180 / M_PI;
    theJuice[indi][18] = isHeartTwisted(tooX, tooY, treeX, treeY, uneX, uneY) * 180 / M_PI;
    theJuice[indi][19] = isHeartTwisted(treeX, treeY, zehoX, zehoY, tooX, tooY) * 180 / M_PI;

    // float distPivotTo1=sqrt(zehoX*zehoX-uneX*uneX+zehoY*zehoY-uneY*uneY);
    // float distPivotTo2=sqrt(zehoX*zehoX-sideAngle2X*sideAngle2X+zehoY*zehoY-sideAngle2Y*sideAngle2Y);
    // float dist1to2=sqrt(sideAngle2X*sideAngle2X-sideAngle1X*sideAngle1X+sideAngle2Y*sideAngle2Y-sideAngle

    // float tempStsces=-distPivotTo1*distPivotTo1-distPivotTo2*distPivotTo2+dist1to2*dist1to2;

    // float tmepsfset2=tempStsces/(-2*distPivotTo1*distPivotTo2);
    // float finAgnle1=acos(tmepsfset2);
```

```
            float whichOnesmallerwenHao0 = zehoX + zehoY;
            float whichOnesmallerwenHao1 = uneX + uneY;
            float whichOnesmallerwenHao2 = tooX + tooY;
            float whichOnesmallerwenHao3 = treeX + treeY;

            theJuice[indi][20] = 100;
            if (whichOnesmallerwenHao1 < whichOnesmallerwenHao0)
            {
                theJuice[indi][20] = 101;
            }
            if (whichOnesmallerwenHao2 < whichOnesmallerwenHao1 && whichOnesmallerwenHao2 < whichOnesmallerwenHao0)
            {
                theJuice[indi][20] = 102;
            }
            if (whichOnesmallerwenHao3 < whichOnesmallerwenHao2 && whichOnesmallerwenHao3 < whichOnesmallerwenHao1 &
            {
                theJuice[indi][20] = 103;
            }
            // myfile2 << cv::contourArea(scene_corners) << "\n";
            theJuice[indi][4] = cv::contourArea(scene_corners);
            //-- Draw lines between the corners (the mapped object in the scene - image_2 )
            // line( img_matches, scene_corners[0] + Point2f((float)img_object.cols, 0),
            //      scene_corners[1] + Point2f((float)img_object.cols, 0), Scalar(0, 255, 0), 4 );
            // line( img_matches, scene_corners[1] + Point2f((float)img_object.cols, 0),
            //      scene_corners[2] + Point2f((float)img_object.cols, 0), Scalar( 0, 255, 0), 4 );
            // line( img_matches, scene_corners[2] + Point2f((float)img_object.cols, 0),
            //      scene_corners[3] + Point2f((float)img_object.cols, 0), Scalar( 0, 255, 0), 4 );
            // line( img_matches, scene_corners[3] + Point2f((float)img_object.cols, 0),
            //      scene_corners[0] + Point2f((float)img_object.cols, 0), Scalar( 0, 255, 0), 4 );

            // line( img_matches, Point2f(0, 0) + Point2f((float)img_object.cols, 0), Point2f(0, 600) + Point2f((flo
            // line( img_matches, Point2f(0, 600) + Point2f((float)img_object.cols, 0),Point2f(450, 600) + Point2f(
            // line( img_matches, Point2f(450, 600) + Point2f((float)img_object.cols, 0),Point2f(450, 0) + Point2f(
            // line( img_matches, Point2f(450, 0) + Point2f((float)img_object.cols, 0),Point2f(0, 0) + Point2f((flo
            // fanle
            //-- Show detected matches
            // imshow("Good Matches & Object detection", img_matches );
            // ROS_INFO("the rwo%d", imgFromCamera.rows);
            // ROS_INFO("the cols%d", imgFromCamera.cols);
            cv::waitKey(10);

            // cv::imshow("view", img);
        }
        ROS_INFO("itsOber");

        /***YOUR CODE HERE***/
        // Use: boxes.templates
        cv::imshow("view", img);
        cv::waitKey(10);
    }

    std::vector<int> goodIndexs;
    for (int tempRowCounter = 0; tempRowCounter < theJuice.size(); tempRowCounter++)
    {
        if (theJuice[0].size() > 0 && theJuice[tempRowCounter][2] > 33 && theJuice[tempRowCounter][4] > 3000)
        {
            if (theJuice[tempRowCounter][15] != -7)
            {
                goodIndexs.push_back(tempRowCounter);
            }
        }
    }
    std::vector<std::vector<float>> theGooodJuice;
    for (int goodIndexCounter = 0; goodIndexCounter < theJuice.size(); goodIndexCounter++)
    {
        auto result1 = std::find(begin(goodIndexs), end(goodIndexs), goodIndexCounter);
        if (result1 != std::end(goodIndexs))
        {
            theGooodJuice.push_back(theJuice[goodIndexCounter]);
        }
    }

    return theGooodJuice;
}
```

C Source Code on imagepipeline.h

```
#pragma once
```

```
#include <image_transport/image_transport.h>
#include <std_msgs/String.h>
#include <opencv2/core.hpp>
#include <cv.h>
#include <cv_bridge/cv_bridge.h>
#include <boxes.h>
#include <algorithm>
#include <iterator>



#include <iostream>

#include "opencv2/core.hpp"
#include "opencv2/calib3d.hpp"
#include "opencv2/highgui.hpp"
#include "opencv2/imgproc.hpp"
#include "opencv2/features2d.hpp"
#include "opencv2/xfeatures2d.hpp"
#include "std_msgs/String.h"
#include <sstream>
#include <fstream>
#include <float.h>
#include <vector>
#include <bits/stdc++.h>





class ImagePipeline {
    private:
        cv::Mat img;
        bool isValid;
        image_transport::Subscriber sub;

    public:
        std::vector<std::pair<std::vector<cv::KeyPoint>,cv::Mat>> preCompuitedVector;
        ImagePipeline(ros::NodeHandle& n);
        void imageCallback(const sensor_msgs::ImageConstPtr& msg);
        std::vector<std::vector<float>> getTemplateID(Boxes& boxes);
        //std::vector<std::pair<std::vector<KeyPoint>,Mat>> preComputekeyFeatures(Boxes &boxes);
        bool isHeartBroken(float x1, float y1,
                 float x2, float y2,
                 float x3, float y3,
                 float x4, float y4);
        float isHeartTwisted(float x1, float y1,
                 float x2, float y2,
                 float x3, float y3);
};
```

D Source Code on navigation.cpp

```
#include <navigation.h>
#include <actionlib/client/simple_action_client.h>
#include <move_base_msgs/MoveBaseAction.h>
#include <tf/transform_datatypes.h>
#include <navigation.h>
#include <robot_pose.h>
#include <math.h>
#include <cmath>
#include <vector>
#include <algorithm>
#include <navigation.h>
#include <actionlib/client/simple_action_client.h>
#include <move_base_msgs/MoveBaseAction.h>
#include <tf/transform_datatypes.h>
#include <geometry_msgs/PoseStamped.h>
#include <nav_msgs/GetPlan.h>
#include <string>
#include <boost/foreach.hpp>
#include <fstream>
#include <iostream>
#include <stdlib.h>

bool Navigation::moveToGoal(float xGoal, float yGoal, float phiGoal){
        // Set up and wait for actionClient.
```

```
    actionlib::SimpleActionClient<move_base_msgs::MoveBaseAction> ac("move_base", true);
    while(!ac.waitForServer(ros::Duration(5.0))){
        ROS_INFO("Waiting for the move_base action server to come up");
    }
        // Set goal.
    geometry_msgs::Quaternion phi = tf::createQuaternionMsgFromYaw(phiGoal);
    move_base_msgs::MoveBaseGoal goal;
    goal.target_pose.header.frame_id = "map";
    goal.target_pose.header.stamp = ros::Time::now();
    goal.target_pose.pose.position.x =  xGoal;
    goal.target_pose.pose.position.y =  yGoal;
    goal.target_pose.pose.position.z =  0.0;
    goal.target_pose.pose.orientation.x = 0.0;
    goal.target_pose.pose.orientation.y = 0.0;
    goal.target_pose.pose.orientation.z = phi.z;
    goal.target_pose.pose.orientation.w = phi.w;
    ROS_INFO("Sending goal location ...");
        // Send goal and wait for response.
    ac.sendGoal(goal);
    ac.waitForResult();
    if(ac.getState() == actionlib::SimpleClientGoalState::SUCCEEDED){
        ROS_INFO("You have reached the destination");
        return true;
    } else {
        ROS_INFO("The robot failed to reach the destination");
        return false;
    }
}
// std::vector<float> Navigation::getGoals(){


// }

bool Navigation::moveToGoal(std::vector<float> goal){
    return Navigation::moveToGoal(goal[0], goal[1], goal[2]);
}
```

E Source Code on navigation.h

```
#pragma once
#include <vector>


class Navigation {
        public:
                static bool moveToGoal(float xGoal, float yGoal, float phiGoal);
                static bool moveToGoal(std::vector<float> goal);
};
```