

MIE 443 Mechatronics Systems
Contest 2 Report
Team 7

Name	Student Number
Amanat Dhaliwal	1003425377
Jianfei Pan	1003948115
Qiwei Zhao	1003579950
Yuhang Song	1002946510

1.0 Problem Definition and Objective of Contest 1

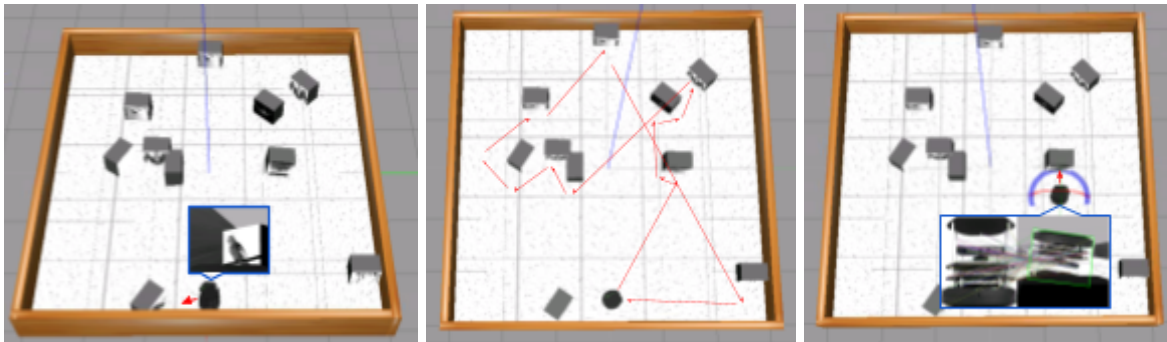


Figure 1. Sample environment, Navigation and Image Recognition

The main objective of contest 2 is to design a programme that enables the turtlebot to traverse the environment and navigate to and identify 10 different objects placed on the map. The general layout of the map including the location and orientation of the 10 objects will be provided at the beginning of the run. The turtlebot is required to use the given map to travel to each of the 10 objects, and identify the tags placed on each of these objects. The robot is also expected to return back to its starting position after identifying all 10 tags. All of this needs to be accomplished within the allotted time of 8 minutes. The team is responsible for designing an algorithm that can use the map provided to navigate towards each of the 10 objects in a time efficient manner and also utilize the robot RGB sensor to identify the tags and their corresponding locations for all 10 images before returning to the starting location.

1.1 Project Requirements

There some specific project requirements that the team needs keep in mind in order to accomplish the above mentioned objective. All the given requirements for this contest have been summarized below:

- The contest environment will be limited to a $6 \times 6 \text{ m}^2$ map with a completely flat surface. The environment will contain 10 objects each with a dimension of $50 \times 20 \times 40 \text{ cm}^3$.
- The team will be provided with 3 different sets of data before the start of the run. First, a 2-D map of the overall environment without the presence of any objects. Second, coordinates of the objects located in the environment. And Lastly, a list of data sets of different tags will be provided, which can be used to match each set with the image identified on each different object.
- The coordinates of the objects will be given relative to the origin of the map which will be native to each different map. The coordinates will be in the form (x, y, ϕ) ,

where x and y give the centre position of the object and the ϕ gives the orientation of the object relative to the origin of the map, as shown in Figure 2.

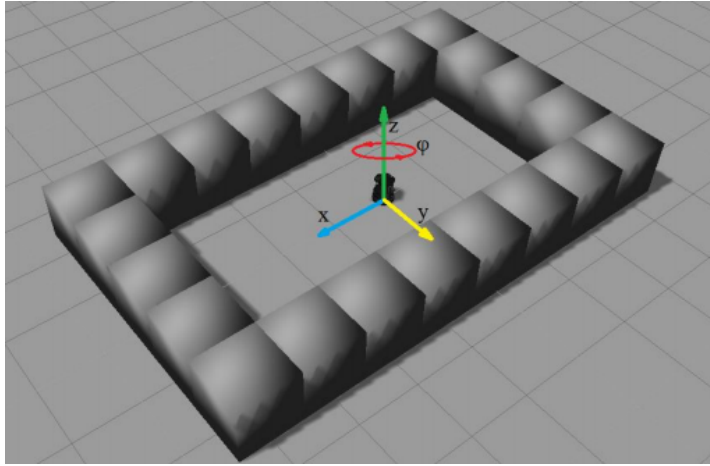


Figure 2: Contest 2 coordinate system

- The tags placed on each object will be a high contrast image. The image will be located on one of the long sides of the object. A data set containing all potential image tags will be provided at the beginning of the run.
- One or two objects in the environment will have duplicate tags. While one or two objects will also have no tags.
- Upon completion of the run, the robot must output to a file the tag names and their respective locations in the order in which it identified them. The basic format for the output should look like tagname + location.
- The robot will be given 2 trial runs and the run with the best score will be counted towards the teams grades.

1.2 Constraints

Apart from the several requirements listed above, there are also certain rules/constraints that apply to contest 2. These constraints have been summarized and are listed below:

- The robot will be given 8 minutes to navigate to all 10 objects, identify the tags on each one and return to its initial position.
- The turtlebot also must use the navigation library provided to accomplish the navigation goals of this contest.
- The turtlebot also must use the RGB camera on the Kinect Sensor, in order to identify all the tags and match them with the given data set.

- To complete the run, the robot must return back to its original position at the beginning of the run.

2.0 Robot Design Strategy and Implementation

The overall aim of the simulation design will be focused on navigation, path planning and image classification aspect of the trial. The navigation algorithms will allow the robot to traverse the environment, navigate to each object and return to the starting position safely within the allotted 8 mins. The path planning algorithms will allow the turtlebot to travel and acquire desired image tags in the most time efficient manner. The image classification algorithms are crucial for acquiring the correct image tags by matching the object image with the correct tag given in the data set. The strategies for these three major algorithms are detailed in the following sections.

2.1 Navigation Algorithm

The purpose of the navigation algorithm is to ensure that the turtlebot can reach the desired location while successfully avoiding the obstacles. Since we are given coordinates of the image objects, we need to plan the path and send the turtlebot to desired locations to acquire image tags. Unlike mapping problems, we need a more destination oriented navigation algorithm with robust obstacle avoidance ability. Therefore, we need a navigation algorithm that can take in the desired location coordinates and yaw and enable the turtlebot to safely travel to that location and orientation. We need to revise the existing MoveToGoal algorithm by checking if the turtlebot can move to its destination before executing the navigation algorithm.

2.2 Path Planning Algorithm

The purpose of path planning is to utilize the given coordinates of the image objects to accomplish the given task in a shortest amount of time. Based on the requirements, each object will be located in a random position with a variety of possible orientations. Navigating to each location randomly to acquire the image tags one by one may not be optimal or even possible to accomplish given the time constraint. Creating an algorithm that is able to find the most efficient path possible to traverse to all 10 objects within the allotted time will save a lot of time for image classifications and thus is one of our main goals. Therefore, the team plans to create a smart algorithm that optimizes the sequence of objects to visit after it reaches the first object so that the turtlebot can spend the least possible time of traversing the map.

2.3 Image Identification Algorithm

The purpose of image identification is to identify the image tags on the objects by matching the images in the scene with the prestored images. The contest requires identifying all ten images and their locations in the map upon returning to its origin. Therefore, the team needs to build an effective and accurate image classification algorithm. Since all images are quite distinct, the team's strategy is to use the feature detectors to extract the key points and descriptors from each image and match them with the key features in the scene image. Since the images can be placed in random orientation on the object, we plan to use the projective transformation to transform the matching features and allow us to infer the location of the image in the scene. Thus, in order to be robust to various image orientations and scales, we will compute and compare the distances between key features in the object image with the scene image and utilize homography to achieve reliable image classification. Since this process is time consuming, the team will only activate the algorithm to meet our time constraint when the turtlebot has reached the desired location and faced towards the images.

3.1 Sensory Design

3.1.1 Odometry

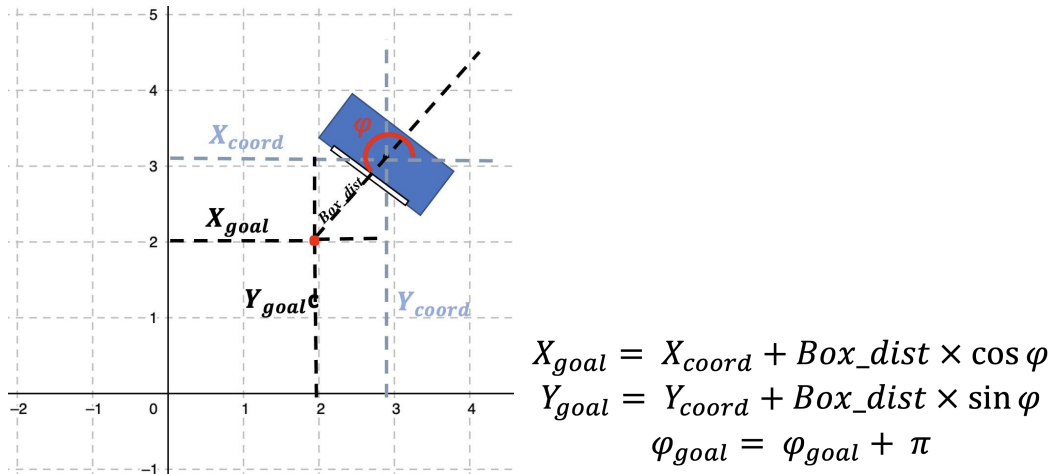


Figure 3. The use of Odometry in Path Planning Algorithm

Odometry is used in our sensory design because it keeps track of the location and orientation of the turtlebot. In the contest, it is crucial to identify turtlebot's current location and also understand the desired location and orientation. Odometry can keep track of the turtlebot's location and orientation so that we can instruct the turtlebot to go to anywhere on the map with any orientations. It is extremely important during image recognition, where the turtlebot can adjust its yaw and face towards the image on the object. Figure 3, has demonstrated a way of using odometry to keep track of the

turtlebot's location and orientation, which also allows us to pass on commands to adjust the turtlebot to the desired location and orientation. Additionally, we will also use odometry in our Adjustment algorithm, where we calculate the possibility of several orientations and locations. The location and yaw information provided by the odometry are key to the Adjustment algorithm.

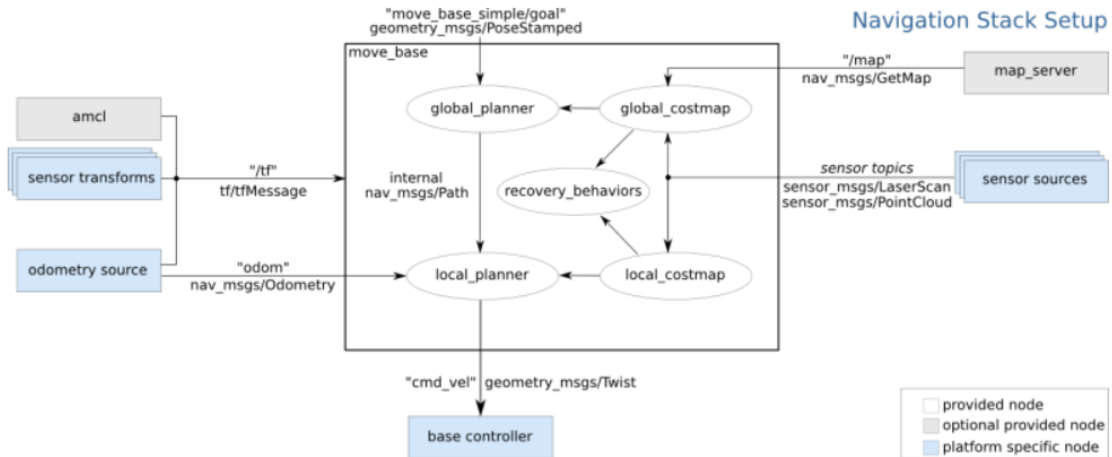


Figure 4. Application of Odometry and LaserScan in Navigation[1]

Moreover, odometry is also extremely important in our navigation and obstacle avoidance algorithm. Figure 4 has shown that the move_base algorithm utilizes both odometry and laserscan to achieve successful navigation. We used the move_base algorithm to build the MoveToGoal algorithm that allows turtlebot to move to the desired location and orientation based on our input. All in all, odometry is an important sensor as it provides the backbone information to many of our algorithms.

3.1.2 Microsoft Kinect Laser Scan (Depth Sensor)

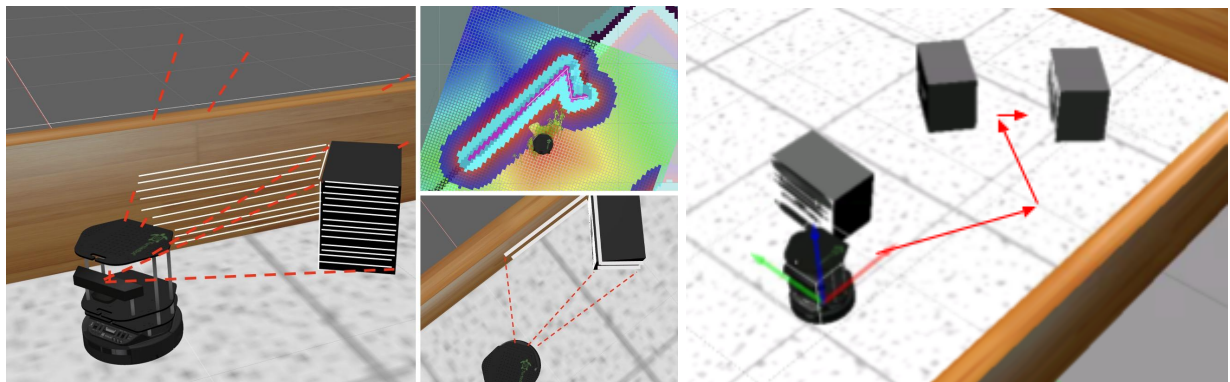


Figure 5. Turtlebot uses the Laser Scan for Localization and Obstacle Avoidance

Laser scan is a crucial sensor in our sensory design, because it provides the distance information to the surrounding obstacles and thus helps us identify the environment and key features in the map. The laser scan has several important applications in our algorithms. Firstly, it helps localization and enables particle filter algorithms. Because we use the AMCL algorithm to localize the turtlebot in the given map, laser scan provides depth information to the AMCL algorithm which will be used in calculating the probability of existence using particle filters as shown in Figure 5. As a result, the turtlebot can recognize its current location in the map, which is critical to all subsequent operations. Secondly, the navigation and obstacle avoidance algorithm rely on the depth information provided by the laser scan to identify the obstacles around the turtlebot and to plan out the path to travel safely to the desired location as illustrated in Figure 5. The laser scan is incorporated in Figure 5, which shows that it helps guide the motion of the turtlebot. Thirdly, laser scan enables our Adjustment algorithm by providing us with control over the distance to the images that tremendously impact our image identification accuracy. When the turtlebot is planning locations to acquire scene images in the Adjustment algorithm, the turtlebot calculates different locations and orientations around the image with the same distance. When it reaches the desired location, laser scan will be used to control the distance of the turtlebot to the image so that the turtlebot can scan the scene according to the plan. Overall, the laser scan sensor is an imperative in our sensory design, because it provides the depth information about the surrounding environment that the turtlebot can use to achieve accurate positioning, distance control, and obstacle avoidance.

3.1.3 Microsoft Kinect RGB sensor

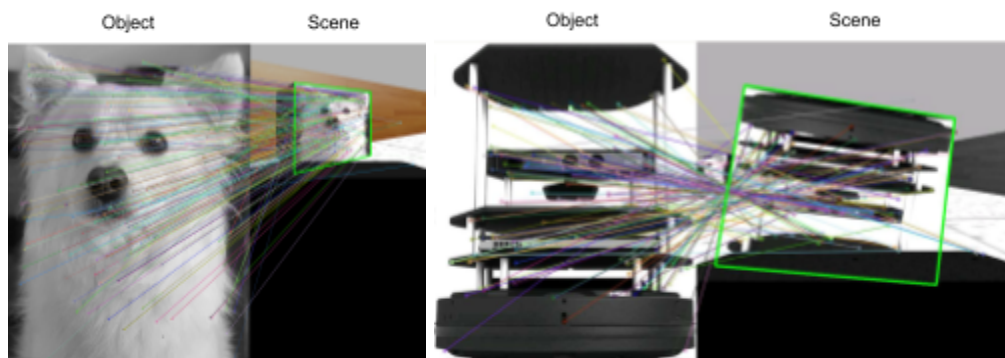


Figure 6. Turtlebot performs image recognition tasks using the RGB Camera

The RGB sensor is the most important sensor in our image identification algorithm, because it can capture and convert the scene to a matrix of pixels that can be manipulated by our image identification algorithm. The RGB sensor allows us to access the image on the object, which we will store and compute key features with. The RGB sensor receives the information of the scene in RGB format, which the team converts to grayscale for better feature matching. We will apply a feature detector and matcher to

the data acquired from the RGB sensor and perform image identification and classification with images in our database. Moreover, the RGB sensor allows us to perform homography on the object image so that we can transform the key points from the database to the scene and thus correctly locate the image in the scene. Figure 6 has illustrated our use of the RGB sensor to acquire the scene image and perform a series of image processing techniques and eventually achieve correct image identification. As a result, the image identification can be successfully implemented.

3.2 Controller Design

3.2.1 High Level Controller

Sense-Plan-Act Hybrid Deliberative Controller

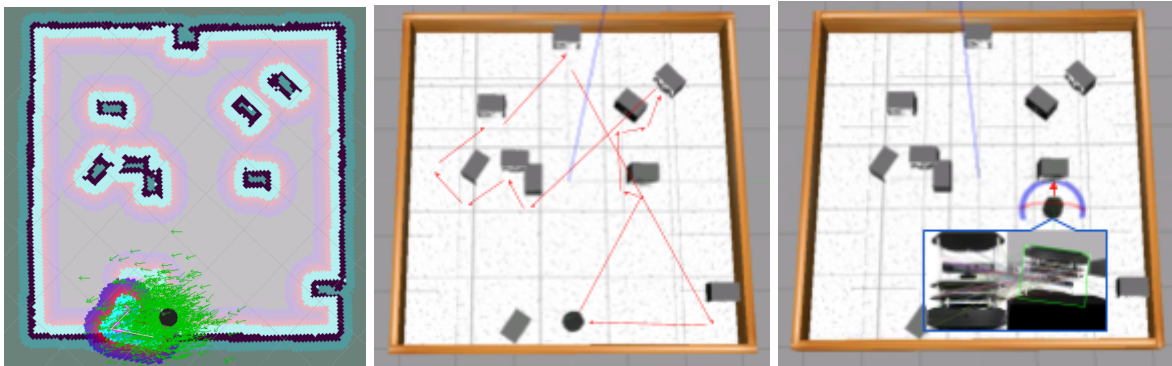


Figure 7. Sense (Left), Plan (Mid), Act (Right)

The team uses the Sense-Plan-Act control architecture as the high level controller that involves forecasting the future, planning the actions, and completing a series of tasks in an optimal order to successfully complete contest 2. The sensing, planning and acting are implemented in parallel and synergy with each other during our navigation, path planning and image classification algorithms. These three components are summarized at a high level as follows.

- Sensing
 - The odometer provides the yaw and location of the turtlebot to aid localization. The adaptive monte carlo localization algorithm will be applied with the sensory data to localize the turtlebot successfully.
 - The RGB camera takes in the image of the scene and provides it to the image pipeline algorithm that will categorize the image from the scene.
 - The depth sensor will provide information about the distances to the surrounding obstacles to achieve obstacle avoidance.
- Planning

- Turtlebot will plan the path to the closest object after successful localization.
- After moving to the first image location, the turtlebot will plan the shortest path to travel past all ten locations using the nearest neighbour algorithm. The sequence of destinations will be determined for turtlebot to follow to gather the image data in the most time efficient manner.
- Before moving to the location, the turtlebot will calculate and determine the location and orientation to reach subsequently in order to get a clear view of the image while avoiding the surrounding obstacles.
- The image recognition algorithm will extract important features and calculate similarities between the stored images and scenes and determine the image that is a counterpart to the scene and store its tag and coordinates.
- The path planning process will continue until turtlebot returns to the origin. Then, turtlebot will identify the locations that it has not reached or images were not successfully acquired.
- Acting
 - Once localization and path planning are complete, the turtlebot will move to the nearest neighbor.
 - After reaching the first destination, the turtlebot will perform the image pipeline algorithm and acquire the image tag.
 - The turtlebot will move to the next destination by following the planned path and repeat the image acquisition tasks.
 - Once the turtlebot has visited all the image locations, it will return to its starting location and output the required file that contains the image tags and their corresponding coordinates in the map.
 - The turtlebot will revisit the locations selectively based on its needs.

In summary, the team utilizes the hybrid deliberative control architecture in both our path planning and image classification algorithms. In the navigation and path planning algorithm, the turtlebot will plan out where and at what yaw to orient if it can reach the destination before executing the motion command. The turtlebot will firstly localize itself in the maze using odometry and laserscan and go to the image that is closest to it. The turtlebot will not start moving until he finds a safe place to go to. During the navigation process, the turtlebot will acquire the information from amcl and laserscan to adjust its movement to avoid obstacles. When turtlebot reaches the first desired image, turtlebot will plan out the path it will follow to mostly efficiently acquire all necessary image tags. In our image classification algorithm, the turtlebot will scan the scene and at the same time identify key points and features to match with the given object images. During this process, the turtlebot continues to perceive, analyze, and classify until getting a final

image tag. The turtlebot has performed sensing, planning, and acting throughout our entire control architecture.

3.2.2 Low Level Controller and Algorithms

1. Adaptive Monte Carlo Localization Algorithm (AMCL)

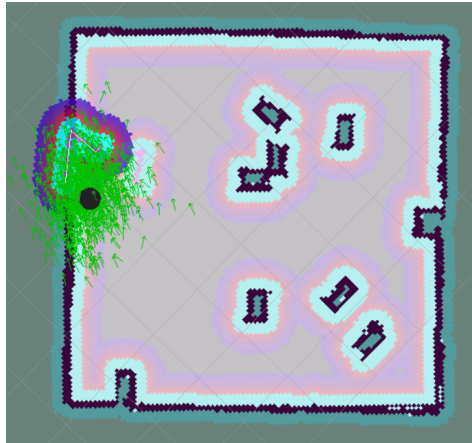
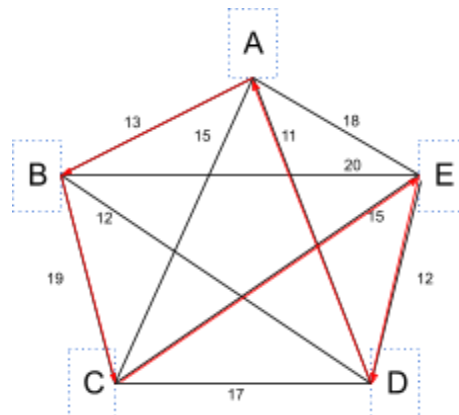


Figure 8. Visualization of Localization that utilizes the AMCL Algorithm

The AMCL is a probabilistic algorithm that uses particle filters to estimate the turtlebot's real-time location and orientation as the turtlebot moves in the given map and senses the surrounding.[2] It retrieves the information published by the laser-scan node and performs particle filters so that its location and orientation in the given map can be determined.[2] AMCL is the backbone of our navigation algorithms, because it provides its estimated location and orientation that will be used for path planning as well as obstacle avoidance. The team uses the given amcl package in ROS and retrieves the estimated pose information for the subsequent navigation and detection algorithm. The Figure 8 illustrates the implementation of the AMCL algorithm to achieve localization and green arrows indicate probabilistic particle clouds for pose and location estimation.

2. The Repetitive Nearest-Neighbour Algorithm (RNN)



Path 1: Weighted(ADBCEA) = $11+12+19+15+18 = 75$
 Path 2: Weighted(BDACEB) = $12+11+15+15+20 = 73$
 Path 3: Weighted(CEDABC) = $15+12+11+13+19 = 70$
 Path 4: Weighted(DABCED) = $11+13+19+15+12 = 70$
 Path 5: Weighted(EDABCE) = $12+11+13+19+15 = 70$

Figure 9. Demonstration of the RNN Algorithm

Due to the limited execution time, the team applied and developed the repetitive nearest-neighbour algorithm to optimize the path for completing the entire mission in a time efficient manner. Figure 9 demonstrates an example of the RNN algorithm of five destinations. The five characters represent five different destinations and the numbers on the path indicates the distance to travel from one point to the other. The idea of the nearest neighbour is to find the sequence of destinations to travel that will minimize the total travel distance. Specially, this algorithm repeats computing the total cost of the Hamilton circuits with each destination as the vertex using nearest neighbour principle. Then, the path that yields the lowest total distance will be the final choice. As we can see from Figure 9, path 3, 4, 5 are essentially equivalent and yield the smallest distance and therefore will be selected as the optimal path. However, the only difference in our problem is that instead of five destinations, we will have 10 destinations. After turtlebot reaches the first destination closest to its original location, it will compute and plan the path using RNN algorithm and follow the algorithm during subsequent data acquisitions. The RNN algorithm does not guarantee the absolute optimized path compared to the Brute-Force algorithm, but it is chosen because it consumes less time to execute and relatively easy to code compared to other models.

3. The MoveToGoal Algorithm

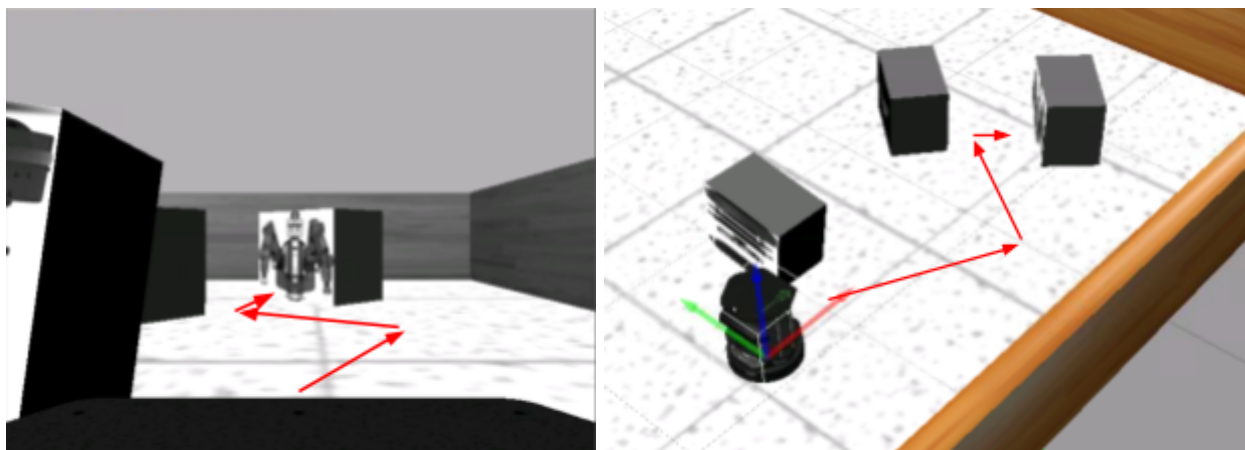


Figure 10. Demonstration of Navigation

The team is provided with the MoveToGoal algorithm that allows turtlebot to move to the required location and yaw while avoiding the obstacles. This algorithm receives the

desired location and yaw as input and performs the navigation, during which the depth sensor is used to keep the turtlebot a safe distance to the surrounding obstacles. Figure 10 has illustrated the implementation of the algorithm. However, if the destination is inside of an obstacle, the turtlebot will not move and abort the entire program.

4. Path Planning Algorithm

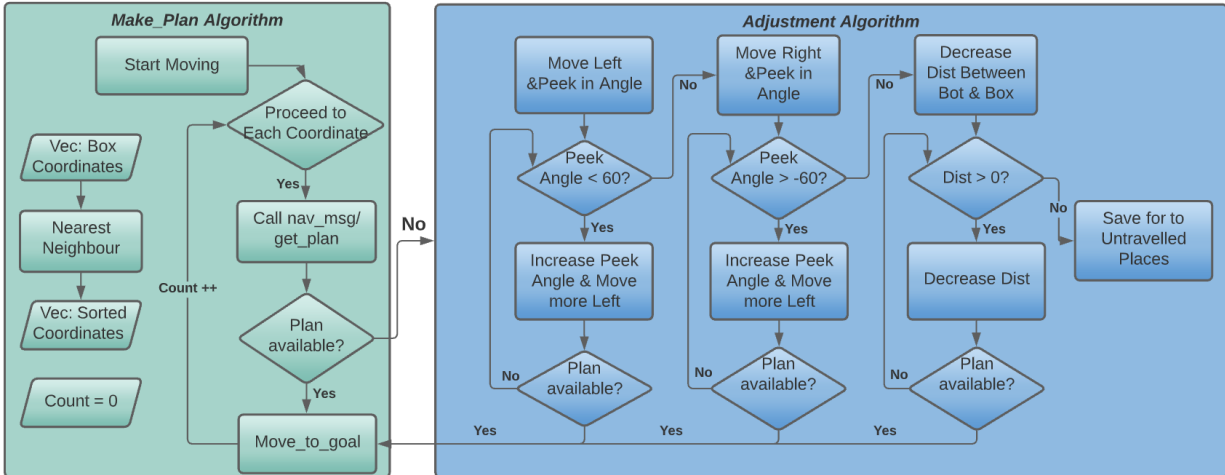


Figure 11. Flowchart of the Path Planning and Adjustment Algorithm

After successful localization, the turtlebot will find and move to the image that is closest to its initial location. However, the team notices that the obstacle is slightly bigger in the localization simulation than in the actual scene. This creates the problem that we need to provide an offset to the image that is outside of the range of perceived obstacle. Moreover, sometimes turtlebot can not face the image directly due to the surrounding obstacles, because we are given only the coordinates of the images, not the coordinates of all obstacles. Thus, these scenarios can create execution problems for the MoveToGoal algorithm that has been discussed previously, because impossible destinations will halt the program. Therefore, the team has developed the adjustment algorithm to ensure that the destination is valid before executing the MoveToGoal algorithm. Figure 11 shows the flowchart of the algorithm. Before calling MoveToGoal, turtlebot firstly checks if the destination can be reached by calling the get_plan function. The turtlebot will only move to the desired location if get_plan confirms positive.

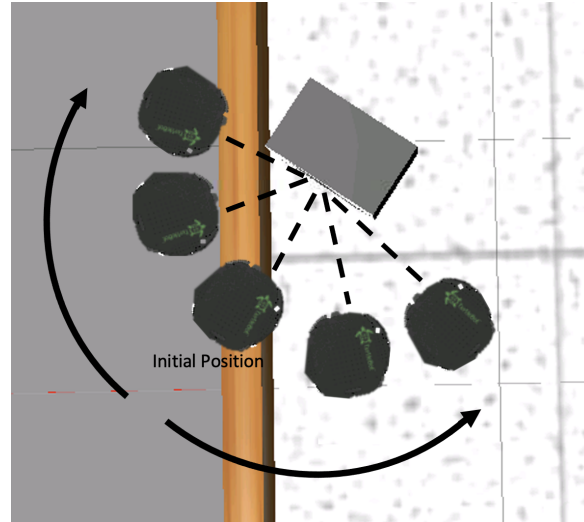


Figure 12. Adjustment Algorithm

Our first priority is always to get right in front of the image, a location we denote as the Initial Position shown in Figure 12. If get_plan confirms negative, then, we will search if turtlebot can reach to locations besides the initial location from the leftmost to the rightmost. The adjustment algorithm is explained in the flowchart in Figure 11 and illustrated graphically in Figure 12. We will calculate if the turtlebot will bump into any obstacles when it locates besides the initial location and scan the image at an angle. If there is still no possible location, the turtlebot will move slightly towards the image and recalculate the above situations until a reachable location is found and its corresponding yaw is calculated. If our adjustment algorithm confirms the location can be reached, the MoveToGoal algorithm will be called to navigate the turtlebot to the required location and turn to the precalculated yaw so that turtlebot can successfully scan the image directly or at an angle to acquire the image tags.

5. SURF Feature Detectors & Homography

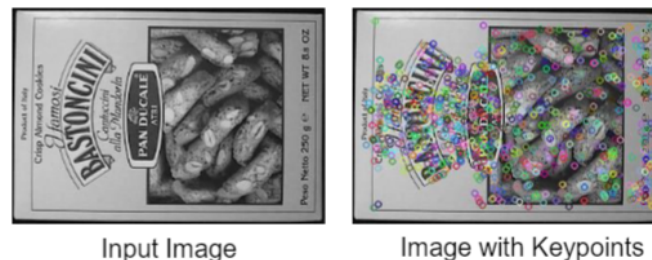


Figure 13. SURF Detection [3]

When the rover reaches the designated position, it will scan and acquire the scene image from the LaserScan. In order to identify the scene image, the turtlebot will use the

SURF detector to extract the unique keypoints and feature descriptors on object image and scene image. Then, the number of good matches between the image and scene is found by evaluating the key features' proximity with each other. The matches between scene and image are considered acceptable if their distances are smaller than a threshold value.

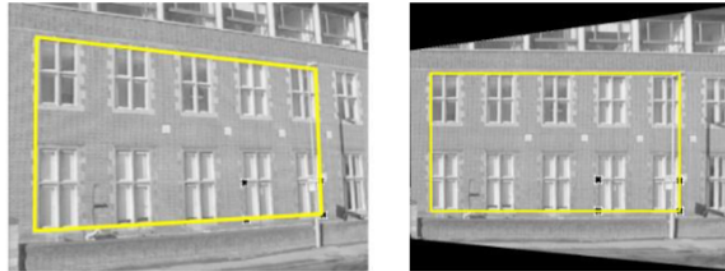


Figure 14. Illustration of Homography [3]

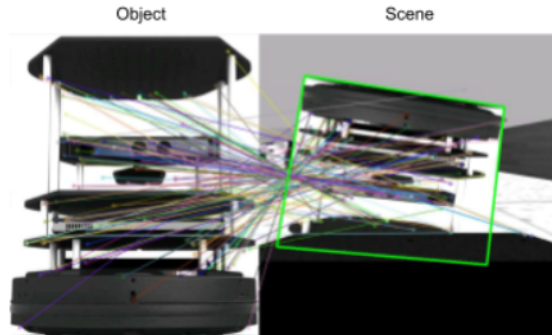


Figure 15: Matching Illustration

Additionally, in order to make our algorithm robust to image scaling, translation, rotation, the team uses the perspective transformation technique, also called homography. Homography is the process of transforming the perspective view of the image, which allows the turtlebot to recognize the image that is in the scene. As shown in figure 14, the initial look of the scene image is at an angle. After the homography, the object image perspective was corrected and can now be located in the scene. However, homography needs at least 4 input good matches from the scene and object images to perform the transformation. The team has found that error occurs when the turtlebot is facing a blank image resulting in fewer than 4 input points. Therefore, the team uses this strategy to identify the blank image by setting the condition on the number of inputs to the homography. Once we do the homography of an image, we can easily utilise the feature detection algorithm and find the best matched scene image. The matched images are shown in Figure 15. Lastly, the team calculates the area formed by the four corners drawn by the homography in order to eliminate the false positive images that have high number of matches but small transformed area.

6. The ImagePipeline Algorithm

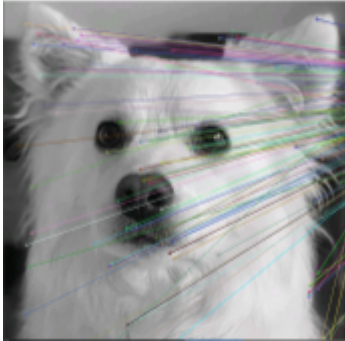
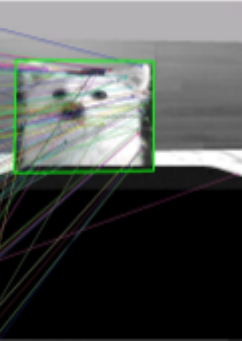
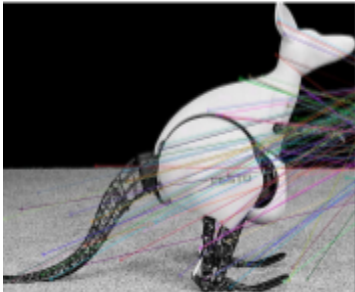
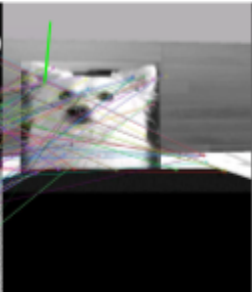
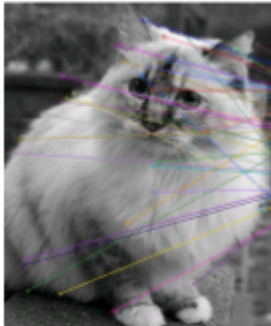
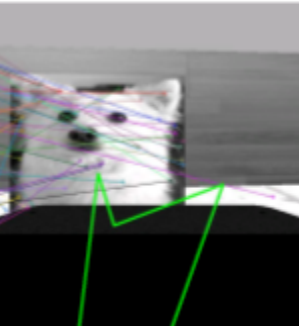
Successful Match	<div>Object </div> <div>Scene </div>	Number of Good matches : 109 Template ID : 4 Image Area: 100548
Failed Match	<div>Object </div> <div>Scene </div>	Number of Good Matches : 85 Template ID : 2 Image Area: 49.9756
Failed Match	<div>Object </div> <div>Scene </div>	Number of Good Matches : 53 Template ID : 5 Image Area: 136050

Figure 16. Illustrations of Results of the ImagePipeline Algorithm

Figure 16 has demonstrated that the team has utilized both the number of good matches and transformed area to identify the image. We initially utilized openCV and

the existing image pipeline code to do the image identification. After a few trials, we found the following pattern. When the openCV is comparing the images from the object and the scene. If they are the correct image, the number of good matches and the area bounded by the four corner vertices after homography are usually large. When the openCV fails to outline the homography on the scanned image, the image area is very low despite the high number of good matches. When the number of good matches is lower than 80, the openCV also fails to identify the correct image despite the high homography transformed area. Therefore, it is crucial to use both the number of good matches and area of the transformed image to acquire accurate results.

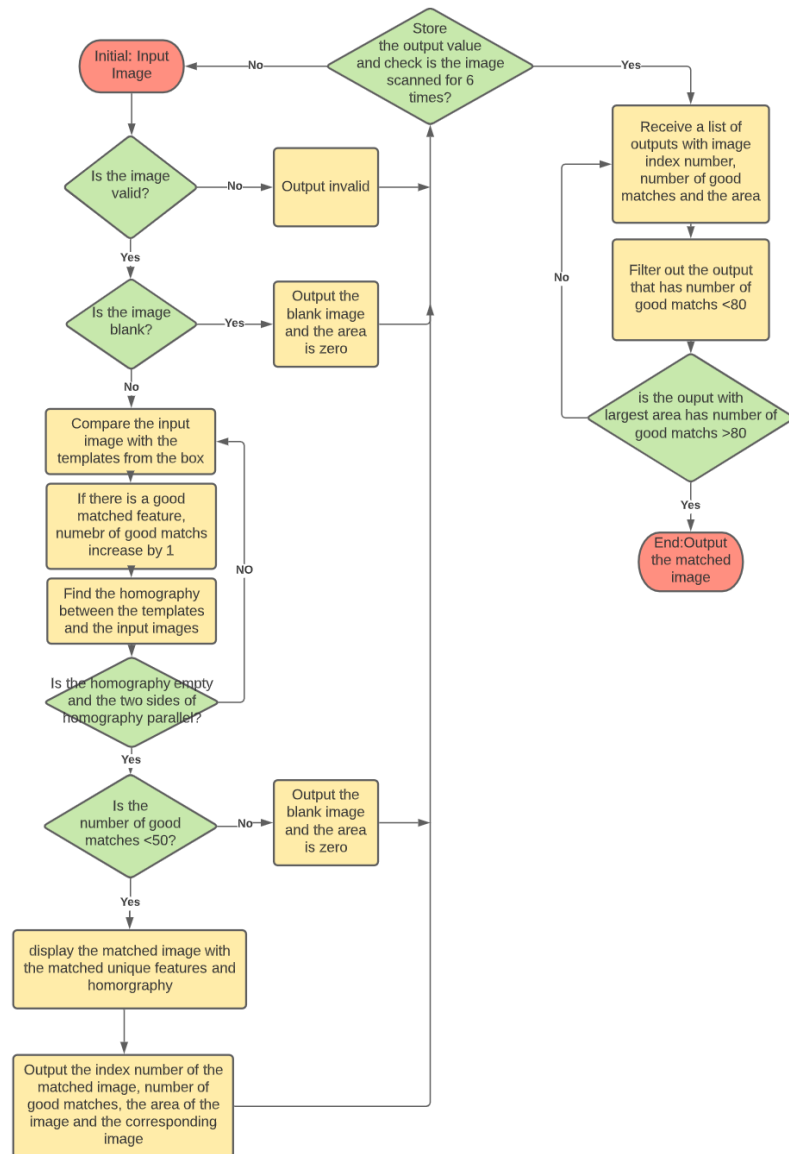


Figure 17. Flowchart of the ImagePipeline Algorithm

Figure 17 demonstrates a high level overview of the team's ImagePipeline algorithm. It basically demonstrates the entire process from acquiring the image to recognizing and classifying the image. Due to the noise around the target image and the sensor error, the accuracy of results from our algorithm fluctuates. Thus, it is necessary for the rover to stay at the same position and repeat the scanning process for at least 6 times. Then, the turtlebot will identify the object image from the prestored list that has the highest number of good matches and area to identify the correct image. If the compared image has a high number of good matches number and area, the turtlebot will consider the image as the correct match and output the matched image id along with its image area and good matches number. Otherwise, the turtlebot will output that the scanned image is blank. Since each image is scanned six times, there will be a list of six sets of outputs. Each set has a picture ID, number of good matches and image area, from which we will filter out the sets that have an extremely low good matches number and transformed area. Then, we will select the image that has the largest image area and acceptably large number of good matches as the final identified image. The iterations significantly improve the accuracy of our image identification algorithm to 90%.

7.0 Data Storing Algorithm

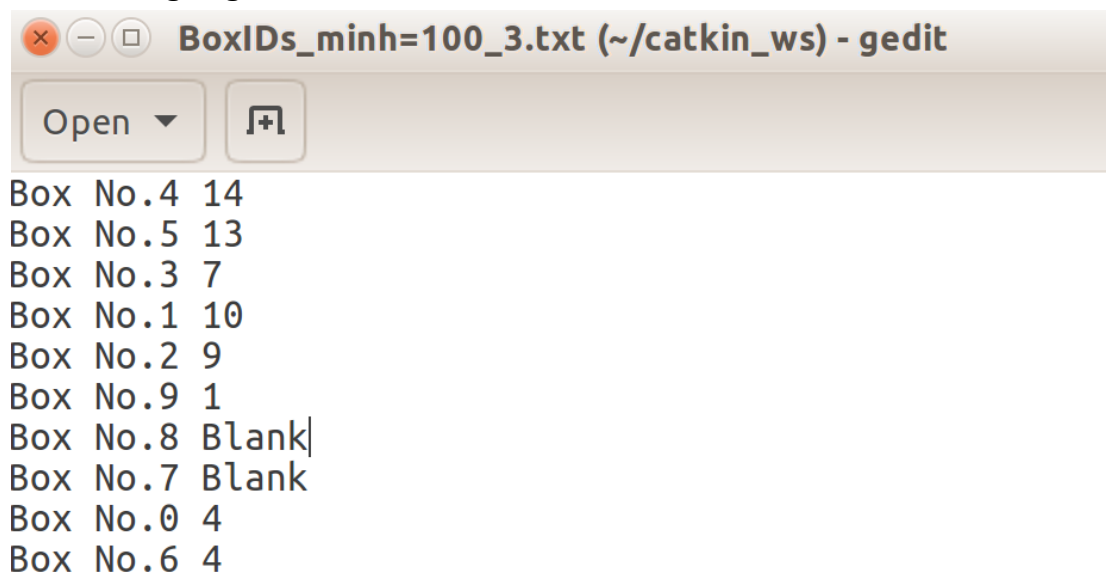


Figure 18. Illustration of the stored Image Tags and their Coordinates

After the turtlebot has successfully traveled to and identified the image tag on one obstacle, it will store the name of the image tag and its current location in global coordinates in a txt file called BoxIDs. The team takes the template ID from our image pipeline algorithm, corresponds the ID to the tag name, and stores the information. The image is identified as a blank image if the template ID is -1. The coordinates are gathered and stored in the BoxIDs.txt as well. As a result, after turtlebot has traveled in

the map and acquired the image tags from all images, the file will contain all the images tags and their locations as shown in Figure 18, which will be reviewed by the teaching team.

4.0 Future Recommendation

The current program enables turtlebot to successfully acquire image tags within 8 minutes. But if the team had more time to improve the existing programme then we would focus on improving the efficiency of the robot navigation algorithm as well as making the tag identification process much more robust. The future recommendations for each section are discussed as follows.

Navigation Algorithm

- The team used the repetitive nearest neighbor algorithm in our path optimization. However, this algorithm does not guarantee the shortest path. Therefore, our future recommendation is to try different path planning algorithms that can further improve our navigation efficiency. For example, the brute force algorithm guarantees the absolute shortest path but is very time consuming for a large number of destinations. We can also try the cheapest-link algorithm that is an efficient solution. Since this contest did not require the most optimal solution, the RNN algorithm was sufficient. But in a contest with a different set of parameters, we would compare the results of different path planning algorithms and determine the one that suits most to our problem.

Image Identification Algorithm

- The team used the SURF detector in extracting key points from the scene and given object images. We recommend trying different detector algorithms in the future such as AffineFeature, AKAZE detectors and descriptor extractors.[4] We can compare the effectiveness of different feature extractors and detectors and then select the one that is most efficient and accurate to implement in our algorithm.
- The team used the FLANNBASED matcher in matching the key points from the images. However, if we are given more time, we can also try using different matchers such as brute force BF matcher or Descriptor matcher.[5] We can determine the effects of different matchers and choose the one that is most efficient and accurate.
- The team calculated the area and cosines enclosed by the transformed image's four corners using homography. These are effective measures and help improve our accuracy. However, the team thinks we can still seek out more effective use of homography by not just looking at the four corners and also identifying the

geometry of the bounded contour in order to eliminate false identifications. For example, the turtlebot identified the wrong image before because the shape of the transformed four corners of the image is erratic although the cosines and area are reasonable.

5.0 Contribution Table

The contribution table is summarized below.

Section	Student Names			
	Eugene Song	Amanat Dhaliwal	Richard Zhao	Jianfei Pan
The problem definition/objective of the contest		RS,RD,MR,CM		
Strategy		RS, ET		RD,MR
Detailed Robot Design and Implementation	RD	ET,	RD,MR,RS,CM	RD,MR,CM,RS
Future Recommendation		RS,ET.		RD,MR
Coding	RD,MR,MI,TS,RS,CM	RS,MI	MI,MR,TS,CM	MI,MR,TS
All	FP	FP	FP	FP

Fill in abbreviations for roles for each of the required content elements. You do not have to fill in every cell. The “all” row refers to the complete report and should indicate who was responsible for the final compilation and final read through of the completed document.

RS – research

RD – wrote first draft

MR – major revision

ET – edited for grammar and spelling

MI – minor revision in codes

TS – test and debug the codes

FP – final read through of complete document for flow and consistency

CM – responsible for compiling the elements into the complete document

OR – other

References

- [1] “move_base - ROS Wiki.” [Online]. Available: http://wiki.ros.org/move_base. [Accessed: 24-Mar-2021]
- [2] “amcl - ROS Wiki.” [Online]. Available: <http://wiki.ros.org/amcl>. [Accessed: 24-Mar-2021]
- [3] “weblogin idpz | University of Toronto - MIE 443 Tutorial 4 OpenCV Features 2021.pdf”, Q.utoronto.ca, 2021. [Online]. Available: <https://q.utoronto.ca/courses/198595/files/folder/Tutorials?preview=12861095>. [Accessed: 13- Mar- 2021].
- [4] “OpenCV: Feature Detection and Description.” [Online]. Available: https://docs.opencv.org/3.4/d5/d51/group__features2d__main.html. [Accessed: 24-Mar-2021]
- [5] “OpenCV: Descriptor Matchers.” [Online]. Available: https://docs.opencv.org/3.4/d8/d9b/group__features2d__match.html. [Accessed: 24-Mar-2021]

Appendix

Full Code:

1. Contest2.cpp

```
// *****
// MIE 443 Contest2 Code
// Authors: Eugene Song, Richard Zhao, Alan Pan
// Date March 24, 2021
// Move Plan Algorithm adapted from:
// https://u.cs.biu.ac.il/~yehoshr1/89-685/Fall2013/demos/lesson7/MakePlan.cpp
// *****

#include <boxes.h>
#include <navigation.h>
#include <robot_pose.h>
#include <imagePipeline.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>
#define forEach BOOST_FOREACH
#define defaultboxdist 0.7
#define proxthresh 0.75

double g_GoalTolerance = 0.0;
std::string g_WorldFrame = "map";
float xgoal,ygoal,phigoal, phigoalactual;
```

```
float phigoalorg, xgoalorg, ygoalorg;  
float boxdist = 0.6;  
bool change_search_dir = false;
```

```
enum Proxstate  
{  
    Tight = 3,  
    CW = 2,  
    CCW = 1,  
    Normal = 0,  
  
};
```

```
class boxpos{  
public:  
    float phi;  
    int quadrant;  
    Proxstate prox;  
public:  
    boxpos(){  
        phi = 0;  
        quadrant = 0;  
        prox = Proxstate::Normal;  
    }  
    boxpos(float phi_in){  
        phi = phi_in;  
        prox = Proxstate::Normal;  
        if (phi>=0 && phi<M_PI/2){  
            quadrant = 1;  
        }else if (phi>M_PI/2 && phi<M_PI){  
            quadrant =2;  
        }else if (phi<0 && phi<-M_PI/2){  
            quadrant = 3;  
        }else{  
            quadrant = 4;  
        }  
        phi = fmod(phi,2*M_PI);  
    }  
}
```



```
};
```

```
void fillPathRequest(nav_msgs::GetPlan::Request &request, float xgoal, float ygoal, float  
phigoal, RobotPose &robotPose);
```

```
bool callPlanningService(ros::ServiceClient &serviceClient, nav_msgs::GetPlan &srv);
```

```
float absdist(std::vector<float> Pos1, std::vector<float> Pos2){  
    float x = Pos1[0] - Pos2[0]; //calculating number to square in next step  
    float y = Pos1[1] - Pos2[1];  
    return sqrt(pow(x, 2) + pow(y, 2));
```

```
}
```

```
std::vector<int >Nearest_neighbour(std::vector<std::vector<float> > &coords,  
    RobotPose &robotPose);
```

```
Proxstate checkProx2(boxpos box1, boxpos box2, boxpos box3, bool middle){
```

```
if ((box1.phi - box2.phi > 0 && box1.phi - box2.phi < M_PI  
    && box3.phi - box1.phi > 0 && box3.phi - box1.phi < M_PI) ||  
    (box2.phi - box1.phi > 0 && box2.phi - box1.phi < M_PI  
    && box1.phi - box3.phi > 0 && box1.phi - box3.phi < M_PI)){  
    return Proxstate::Tight;  
}
```

```
if (box2.prox == Proxstate::CW && middle){
```

```
    if(box1.quadrant != box3.quadrant){  
        if (box1.quadrant - box3.quadrant == -1 ||  
            box1.quadrant - box3.quadrant == 3){
```

```
            return Proxstate::Tight;}  
        else{
```

```
            return Proxstate::CCW;  
        }  
    } else {
```

```

        return box1.phi < box3.phi ? Proxstate::Tight : Proxstate::CCW;
    }

} else if(box2.prox == Proxstate::CCW && middle){

    if(box1.quadrant != box3.quadrant){
        if (box1.quadrant - box3.quadrant == 1 ||
            box1.quadrant - box3.quadrant == -3){

            return Proxstate::Tight;}
        else{
            return Proxstate::CW;}
    }
    else {
        return box1.phi < box3.phi ? Proxstate::Tight : Proxstate::CCW;
    }

} else if(box2.prox == Proxstate::Tight && middle){

    if(box1.quadrant != box2.quadrant){
        if (box1.quadrant - box2.quadrant == 1 ||
            box1.quadrant - box2.quadrant == -3){

            return Proxstate::CCW;

            } else if (box1.quadrant - box2.quadrant == -1 ||
                box1.quadrant - box2.quadrant == 3){

                return Proxstate::CW;
            }
        } else {
            return box1.phi > box2.phi ? Proxstate::CCW : Proxstate::CW;
        }
    }

} else {
    if(box1.quadrant != box2.quadrant){
        if (box1.quadrant - box2.quadrant == 1 ||
            box1.quadrant - box2.quadrant == -3){

            return Proxstate::CCW;
        }
    }
}

```

```

    } else if (box1.quadrant - box2.quadrant == -1 ||
               box1.quadrant - box2.quadrant == 3){

        return Proxstate::CW;
    }
} else {
    return box1.phi > box2.phi ? Proxstate::CCW : Proxstate::CW;
}
}
}

void CheckProximity(std::vector<std::vector<float> >
&sortedcoords, std::vector<boxpos> &Boxposs){
    for (int i = 0; i< sortedcoords.size(); i++){
        boxpos Boxpos(sortedcoords[i][2]);
        Boxposs.push_back(Boxpos);}
    for (int i = 0; i< sortedcoords.size(); i++){
        std::vector<float> currpos = std::vector<float>(sortedcoords[i].begin(),
sortedcoords[i].begin()+2);
        std::vector<float> nextpos;
        std::vector<float> endpos;
        boxpos boxbegin = Boxposs[i];
        boxpos boxnext;
        boxpos boxend;
        bool middle = false;
        if(i == 0)
        {
            nextpos = std::vector<float>(sortedcoords[i+1].begin(),
sortedcoords[i+1].begin()+2);
            endpos = std::vector<float>(sortedcoords[i+2].begin(),
sortedcoords[i+2].begin()+2);
            boxnext = Boxposs[i+1];
            boxend = Boxposs[i+2];
        } else if ( i == sortedcoords.size()-1)
        {
            nextpos = std::vector<float>(sortedcoords[i-1].begin(),
sortedcoords[i-1].begin()+2);
            endpos = std::vector<float>(sortedcoords[i-2].begin(),
sortedcoords[i-2].begin()+2);

```

```

        boxnext = Boxposs[i-1];
        boxend = Boxposs[i-2];
    } else{
        nextpos = std::vector<float>(sortedcoords[i-1].begin(),
sortedcoords[i-1].begin()+2);
        endpos = std::vector<float>(sortedcoords[i+1].begin(),
sortedcoords[i+1].begin()+2);
        boxnext = Boxposs[i-1];
        boxend = Boxposs[i+1];
        middle = true;
    }

    if (absdist(currpos,nextpos)<proxthresh || absdist(currpos,endpos)<proxthresh){
        ROS_INFO("Here, %i, absdist: %f", i,absdist(currpos,nextpos));
        ROS_INFO("box1 Quart, %i, box2 Quart, %i box3 Quart, %i",
boxbegin.quadrant,
                                boxnext.quadrant,boxend.quadrant);
        ROS_INFO("box1 P, %f, box2 P, %f box3 P, %f", boxbegin.phi,
                                boxnext.phi,boxend.phi);

        Boxposs[i].prox = checkProx2(boxbegin,boxnext,boxend,middle);
    }else{
        Boxposs[i].prox = Proxstate::Normal;
    }
}
}

```

```

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

    // Setup ROS.
    ros::init(argc, argv, "contest2");
    ros::NodeHandle n;
    // Robot pose object + subscriber.
    RobotPose robotPose(0,0,0);
    ros::Subscriber amclSub = n.subscribe("/amcl_pose", 1, &RobotPose::poseCallback,
&robotPose);

```

```

// Initialize box coordinates and templates
Boxes boxes;
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 coordinates: " << std::endl;
    std::cout << i << " x: " << boxes.coords[i][0] << " y: " << boxes.coords[i][1] << " z: "
        << boxes.coords[i][2] * 180/M_PI << std::endl;
}
// Initialize image object and subscriber.
ImagePipeline imagePipeline(n);

//*****
//my code for small adjustments
//*****
for (auto i = 0; i < 30; ++i)
{
    ros::spinOnce();
    ros::Duration(0.01).sleep();
}

// Get Localization initialized
for (auto i = 0; i < 5; ++i)
{
    ros::spinOnce();
    if (!Navigation::moveToGoal(robotPose.x + 0.01, robotPose.y, robotPose.phi))
    {
        return -1;
    }
}

int count = 0;
std::vector<std::vector<float>> coords = boxes.coords;
std::vector<int> sorted = Nearest_neighbour(coords,robotPose);
std::vector<std::vector<float>> sortedcoords;
std::vector<float> initPos {robotPose.x,robotPose.y};
for(uint8_t i = 0; i<sorted.size();i++){
    sortedcoords.push_back(coords[sorted[i]]);
}

```

```

}
initPos.push_back(robotPose.phi);
sortedcoords.push_back(initPos);
std::vector<boxpos> Boxposs;
Boxposs.reserve(sortedcoords.size());
CheckProximity(sortedcoords,Boxposs);
std::vector<std::vector<float> > coords_not_went;
//*****
//my code end
//*****
//rosservice list /move_base

//*****Make Plan*****//
// Init service query for make plan
phigoal = sortedcoords[count][2];
xgoal = sortedcoords[count][0]+ 0.55*std::cos(phigoal);
ygoal = sortedcoords[count][1]+ 0.55*std::sin(phigoal);
std::string service_name = "move_base/NavfnROS/make_plan";
while (!ros::service::waitForService(service_name, ros::Duration(3.0))) {
    ROS_INFO("Waiting for service move_base/make_plan to become
available");
}

    ros::ServiceClient serviceClient =
n.serviceClient<nav_msgs::GetPlan>(service_name, true);
    if (!serviceClient) {
        ROS_FATAL("Could not initialize get plan service from %s",
serviceClient.getService().c_str());
        return -1;
    }

    if (!serviceClient) {
        ROS_FATAL("Persistent service connection to %s failed",
serviceClient.getService().c_str());
        return -1;
    }
nav_msgs::GetPlan srv;
std::ofstream BoxIDs("BoxIDs.txt");
//*****Make Plan Finished*****//

```

```

// Execute strategy.
while(ros::ok()) {
    ros::spinOnce();

    //ROS_INFO("RobotPose: x:%f, y:%f, phi:%f", robotPose.x,robotPose.y,
robotPose.phi);
    if (count < sortedcoords.size()-1){
        phigoal = sortedcoords[count][2];
        xgoal = sortedcoords[count][0]+ boxdist*std::cos(phigoal);
        ygoal = sortedcoords[count][1]+ boxdist*std::sin(phigoal);
        phigoalorg = phigoal;
        xgoalorg = xgoal;
        ygoalorg = ygoal;
        fillPathRequest(srv.request,xgoal,ygoal,fmod(phigoal + 3.14, 6.28), robotPose);
        float searchangle = M_PI/3;
        boxdist = defaultboxdist;
        ROS_INFO("Original Coords: x:%f, y:%f, phi:%f", xgoal,ygoal, phigoal);
        while (!callPlanningService(serviceClient, srv)){

            xgoal = xgoalorg; ygoal = ygoalorg;
            if (!change_search_dir){
                if(phigoal - phigoalorg< searchangle){
                    phigoal += M_PI/9;
                    ROS_INFO("CCW Adjusting Angle");
                }else {
                    phigoal = phigoalorg;
                    change_search_dir = true;
                    ROS_INFO("CCW way no good");
                }
            } else{
                if(phigoal - phigoalorg > -searchangle){
                    phigoal -= M_PI/9;
                    ROS_INFO("CW Adjusting Angle");
                }else{
                    if(searchangle < M_PI/2){
                        ROS_INFO("Damn, No place to go?? Increase the search range!");
                        phigoal = searchangle;
                        searchangle += M_PI/4;
                        change_search_dir = false;
                    }
                }
            }
        }
    }
}

```



```

        boxdist -= 0.05;
    } else{
        ROS_INFO("Damn, I guess I will need to decrease the box dist");
        phigoal = phigoalorg;
        if (boxdist > 0){
            boxdist -= 0.05;
            phigoal = phigoalorg;
            change_search_dir = false;}
        else{
            break;
        }
    }
}
}

xgoal = sortedcoords[count][0]+
(boxdist-0.1*fabs(phigoal-phigoalorg)/searchangle) * std::cos(fmod(phigoal, 2*M_PI));
ygoal = sortedcoords[count][1]+
(boxdist-0.1*fabs(phigoal-phigoalorg)/searchangle) * std::sin(fmod(phigoal, 2*M_PI));
fillPathRequest(srv.request,xgoal,ygoal,fmod(phigoal + 3.14, 6.28),
robotPose);
}
}else if (count == sortedcoords.size()-1){
    ROS_INFO("Going Home");
    phigoal = sortedcoords[count][2];
    xgoal = sortedcoords[count][0];
    ygoal = sortedcoords[count][1];
    phigoalorg = phigoal;
    xgoalorg = xgoal;
    ygoalorg = ygoal;
    change_search_dir = false;
    fillPathRequest(srv.request,xgoal,ygoal,fmod(phigoal, 6.28), robotPose);
    while (!callPlanningService(serviceClient, srv)){
        if(!change_search_dir){
            if (xgoal - xgoalorg < 1){
                xgoal += 0.01;
                fillPathRequest(srv.request,xgoal,ygoal,fmod(phigoal, 6.28), robotPose);
                while (!callPlanningService(serviceClient, srv)){
                    if(!change_search_dir){
                        if (ygoal - ygoalorg < 1)

```

```
{
    ygoal += 0.01;}
else{
    change_search_dir = true;
    ygoal = ygoalog;
}

}else{
if (ygoal - ygoalog > -1){
    ygoal -= 0.01;
} else{
    change_search_dir = false;
    break;
}
}
}
}
}
}else{
    if (xgoal - xgoalog > -1){
        xgoal -= 0.01;
        fillPathRequest(srv.request,xgoal,ygoal,fmod(phigoal, 6.28), robotPose);
        while (!callPlanningService(serviceClient, srv)){
            if(!change_search_dir){
                if (ygoal - ygoalog < 1)
                {
                    ygoal += 0.01;}
                else{
                    change_search_dir = true;
                    ygoal = ygoalog;
                }

            }
        }
    }
    if (ygoal - ygoalog > -1){
        ygoal -= 0.01;
    } else{
        change_search_dir = false;
        break;
    }
}
}
```

```

        }else{
            change_search_dir = false;
            break;
        }

    }
    fillPathRequest(srv.request,xgoal,ygoal,fmod(phigoal, 2* M_PI), robotPose);
}
if (!Navigation::moveToGoal(xgoal, ygoal, phigoalactual))
{
    return -1;
}
return -1;
}

else if (!coords_not_went.empty()){
    ROS_INFO("Going to left_over coords");
    //Code?
    return -1;
}

else{
    return -1;
}

float phidev = phigoalorg - phigoal;

ROS_INFO("phi deviated: %f", phidev);
ROS_INFO("Current Coords: x:%f, y:%f, phi:%f", xgoal,ygoal, phigoal);
phigoalorg = phigoal;
float xgoalprev = xgoal;
float ygoalprev = ygoal;
float accumulatedist = 0;
float minimaldist = 0;
//Algorithm deal with situations with more than one pic around
switch(Boxposs[count].prox){
    case CCW:
        phigoal = fmod(phigoal+(M_PI/8*(1-2*phidev/M_PI)), 2*M_PI);
        minimaldist = 0.25;
        ROS_INFO("Other Objects Very Close, Rotate CCW");

```

```

break;

case CW:
    phigoal = fmod(phigoal-(M_PI/8*(1-2*phidev/M_PI)), 2*M_PI);
    ROS_INFO("Other Objects Very Close, Rotate CW");
    minimaldist = 0.25;
    break;

case Normal:
    ROS_INFO("Normal Conditions");
    minimaldist = 0.2;
    break;

case Tight:
    phigoal = sortedcoords[count][2];
    phigoalorg = phigoal;
    minimaldist = 0.2;
    ROS_INFO("More than two objects around");
    break;
}

fillPathRequest(srv.request,xgoal-0.01*std::cos(fmod(phigoalorg, 2*M_PI)),
    ygoal-0.01*std::sin(fmod(phigoalorg, 2*M_PI)),fmod(phigoal + 3.14, 6.28),
robotPose);
while(callPlanningService(serviceClient, srv)&& boxdist - accumulatedist
>minimaldist){
    ROS_INFO("Im here");
    xgoalprev = xgoal; ygoalprev = ygoal;
    xgoal -= 0.01*std::cos(fmod(phigoalorg, 2*M_PI));
    ygoal -= 0.01*std::sin(fmod(phigoalorg, 2*M_PI));
    accumulatedist += 0.01;
    fillPathRequest(srv.request,xgoal,ygoal,fmod(phigoal + 3.14, 6.28), robotPose);
}
xgoal = xgoalprev;
ygoal = ygoalprev;

phigoalactual = fmod(phigoal + 3.14, 6.28);
if (!Navigation::moveToGoal(xgoal, ygoal, phigoalactual))

```

```

{
    coords_not_went.push_back(sortedcoords[count]);
}else{

//Imagepipeline Starts Here
// Space for Image Pipeline

ImagePipeline imagePipeline(n);
std::vector<int> potential_pic;
std::vector<int> list_matches;
std::vector<int> list_area;

int suggested_id = 0;
int template_id = 0;
int num_iteration = 0;
int num_valid = 0;
int num_matches=0 ;
int area=0;
while(ros::ok()) {
    ros::spinOnce();
    /**YOUR CODE HERE***/
    // Use: boxes.coords
    // Use: robotPose.x, robotPose.y, robotPose.phi

    std::vector<int> received_value = imagePipeline.getTemplateID(boxes);
    suggested_id = received_value[0];
    num_matches = received_value[1];
    area = received_value[2];

    if (num_iteration<6 && suggested_id !=-2){

        ROS_WARN("Richard the god suggested_id: %i",suggested_id+1);
        potential_pic.push_back(suggested_id);
        list_matches.push_back(num_matches);
        list_area.push_back(area);
        num_iteration = num_iteration+1;
        ROS_INFO("THE Jianfei_iiiiiiiiiteration:%i", num_iteration);

    }
}

```

```

if (num_iteration==6)
{
    // template_id = mostFrequent(potential_pic,potential_pic.size());
    // ROS_INFO("THE PIC NUMBER IS: %i", template_id);
    // potential_pic.clear();

    for (int i = 0; i <potential_pic.size();i++){
        if (list_matches[i]<70){    // can tune
            list_area[i] =0;
        }
    }

    list_matches[0]=0;
    list_area[0]=0;

    int maxMatchesIndex =
std::max_element(list_matches.begin(),list_matches.end()) - list_matches.begin();
    int maxMatches = *std::max_element(list_matches.begin(), list_matches.end());

    int maxAreaIndex = std::max_element(list_area.begin(),list_area.end()) -
list_area.begin();
    int maxArea = *std::max_element(list_area.begin(), list_area.end());
    ROS_INFO("The max area :%i", maxArea);
    ROS_INFO("The max area index :%i", maxAreaIndex);
    ROS_INFO("The max area's Matches number:%i",
list_matches[maxAreaIndex]);

    ROS_INFO("#####
#####");

    ROS_INFO("The max matches :%i", maxMatches);
    ROS_INFO("The max Matches Index :%i", maxMatchesIndex );
    ROS_INFO("The max matches's area :%i", list_area[maxMatchesIndex]);

    ROS_INFO("#####
#####");

    if (list_matches[maxAreaIndex]>70){
        ROS_FATAL("The picture index number is :%i",
potential_pic[maxMatchesIndex]+1);

```

```

ROS_INFO("#####");
#####);
    if (count<sorted.size()){
        BoxIDs <<"Box No."<<sorted[count]<<"
tag_ "<<potential_pic[maxMatchesIndex] +1 << " Coordiantes:" <<
sortedcoords[count][0]<< "," << sortedcoords[count][1] << std::endl;
    }

}
else {
    ROS_INFO("It is possible blank");
    if (count<sorted.size()){
        BoxIDs <<"Box No."<<sorted[count]<<" tag_ "<< "Blank" << " Coordiantes:" <<
sortedcoords[count][0]<< "," << sortedcoords[count][1] <<std::endl;
    }
}

    potential_pic.clear();
    list_matches.clear();
    list_area.clear();
    break;
}

// std::cout<<'the selected pic_id:'<<template_id<<std::endl;

}
}

    count ++;
    ros::Duration(0.01).sleep();
    change_search_dir = false;
    ROS_INFO("Desitination %i reach", count);
}
return 0;
}

////////////////////////////////////
////////////////////////////////////

```



```

////////////////////////////////////
////////////////////////////////////
////////////////////////////////////**Functions**////////////////////////////////////
////////////////////////////////////
////////////////////////////////////
////////////////////////////////////

```

```

void fillPathRequest(nav_msgs::GetPlan::Request &request, float xgoal, float ygoal, float
phigoal, RobotPose &robotPose)

```

```

{
    request.start.header.frame_id = g_WorldFrame;
    request.start.pose.position.x = robotPose.x;
    request.start.pose.position.y = robotPose.y;
    request.start.pose.orientation.w = robotPose.phi;

    request.goal.header.frame_id = g_WorldFrame;
    request.goal.pose.position.x = xgoal;
    request.goal.pose.position.y = ygoal;
    request.goal.pose.orientation.w = phigoal;

    request.tolerance = g_GoalTolerance;
}

```

```

bool callPlanningService(ros::ServiceClient &serviceClient, nav_msgs::GetPlan &srv)

```

```

{
    // Perform the actual path planner call
    if (serviceClient.call(srv)) {
        if (!srv.response.plan.poses.empty()) {
            return true;
        }
        else {
            return false;
        }
    }
    else {
        ROS_ERROR("Failed to call service %s - is the robot moving?",
serviceClient.getService().c_str());
        return false;
    }
}

```

```

std::vector<int >Nearest_neighbour(std::vector<std::vector<float> > &coords,
                                   RobotPose &robotPose){

    std::vector<int> sorted(coords.size(),0);
    std::vector<float> initPos {robotPose.x,robotPose.y}; // save the initial pos
    std::vector<float> nextPos {robotPose.x,robotPose.y}; // the first pos will be the initial
pos

    std::vector<int> unsorted(coords.size(),0);
    for(uint8_t i =0; i<unsorted.size(); i++){unsorted[i]=i;} // initialize an vector starts from
0 to 9
    std::vector<float> costlist(unsorted.size(),0); // initialize a list that saves all the
eucliden distance
    int index = 0;

    while(unsorted.size()>0){ // continue the loop while there are unfinished coods

        for(uint8_t i = 0; i<unsorted.size();i++){ //calculate all the costs(abs distance)
            std::vector<float> currpos = std::vector<float>(coords[unsorted[i]].begin(),
coords[unsorted[i]].begin()+2);
            costlist[i] = absdist(currpos,nextPos);
        }
        //find the index of the box that has the minimum cost
        int minElementIndex = std::min_element(costlist.begin(),costlist.end()) -
costlist.begin();
        // update the next coord
        nextPos =
std::vector<float>(coords[unsorted[minElementIndex]].begin(),coords[unsorted[minElem
entIndex]].begin()+2);
        sorted[index] = unsorted[minElementIndex];
        unsorted.erase(unsorted.begin()+minElementIndex);
        costlist.pop_back();
        index ++;

    }
    return sorted;
}

```

2. ImagePipeline.cpp

```
#include <imagePipeline.h>
#include <tuple>
#include <stdio.h>
#include <iostream>
#include "opencv2/core.hpp"
#include "opencv2/features2d.hpp"
#include "opencv2/highgui.hpp"
#include "opencv2/calib3d.hpp"
#include "opencv2/xfeatures2d.hpp"
#include <vector>
#include <math.h>

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

using namespace cv;
using namespace cv::xfeatures2d;

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;
    }
}
```

```
}
```

```
// use the number of 0 in the image in determine if the image is blank
```

```
std::vector<int> ImagePipeline::getTemplateID(Boxes& boxes) {  
    int template_id = -1;  
    std::vector<int> num_matches;  
    std::vector<int> num_matches_index;  
    std::vector<int> if_parallel_sides;  
    std::vector<int> currentarea;  
    std::vector<int> important_info;  
    important_info.clear();  
  
    if(!isValid) {  
        std::cout << "ERROR: INVALID IMAGE!" << std::endl;  
  
        important_info.push_back(-2);  
        important_info.push_back(0);  
        important_info.push_back(0);  
        return important_info;  
    } 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;  
        important_info.push_back(-1);  
        important_info.push_back(0);  
        important_info.push_back(0);  
        return important_info;  
    } else {  
        cv::imshow("view", img); // used to be at the back  
        cv::waitKey(10);  
        Mat img_scene = img;  
        Mat input;  
        Mat img_object;  
        float cos_hori_top;  
        float cos_hori_bot;  
        float cos_verti_left;  
        float cos_verti_right;
```

```

float tan_hori_top;
float tan_hori_bot;
float tan_verti_left;
float tan_verti_right;
bool notweirdshape = true;

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

    /**YOUR CODE HERE***/
    // Use: boxes.templates

    // Copy the input image to a local variable called input
    //img.copyTo(input);
    //img.release();
    //convert the input image to greyscale, save it as img_scene
    //cvtColor(input, img_scene, CV_BGR2GRAY);

    //--Step 1: Detect the keypoints using SURF Detector, compute the descriptors

    img_object = boxes.templates[ind];
    int minHessian=250;
    Ptr<SURF> detector =SURF::create( minHessian);
    std::vector<KeyPoint> keypoints_object, keypoints_scene;
    Mat descriptors_object, descriptors_scene;
    //Mat img_object = boxes.templates[ind];
    detector->detectAndCompute( img_object, Mat(), keypoints_object,
descriptors_object);
    detector->detectAndCompute( img_scene, Mat(), keypoints_scene,
descriptors_scene);

    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
    const float ratio_thresh=0.76f;
    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]);
        }
    }
    num_matches.push_back(good_matches.size());
    num_matches_index.push_back(ind);

    //-- Localize the object
    std::vector<Point2f> obj;
    std::vector<Point2f> scene;

    for( int 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 );
    }

    if (scene.size() < 4){
        //std::cout << "Blank Tag scene.size() < 4:" << std::endl;
        template_id = -1;    // if no contour can be found
        important_info.push_back(template_id);
        important_info.push_back(0);
        important_info.push_back(0);
        return important_info;
    }

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

    //-- Get the corners from the image_1 ( the object to be "detected" )
    std::vector<Point2f> obj_corners(4);
    obj_corners[0] = cvPoint(0,0);
    obj_corners[1] = cvPoint( img_object.cols, 0 );
    obj_corners[2] = cvPoint( img_object.cols, img_object.rows );
    obj_corners[3] = cvPoint( 0, img_object.rows );
    std::vector<Point2f> scene_corners(4);

    //std::cout << "transform matrix empty"<< H.empty()<< std::endl;
    if (H.empty()){
        //std::cout << "Blank Tag H.empty:" << std::endl;

```

```

        template_id = -1;    // if no contour can be found
        important_info.push_back(template_id);
        important_info.push_back(0);
        important_info.push_back(0);
        return important_info;
    }

    //std::cout << "scene.size_after:" << obj_corners << std::endl;
    //std::cout << "object.size_after:" << obj.size() << std::endl;
    perspectiveTransform( obj_corners, scene_corners, H);

    float x1 = scene_corners[1].x - scene_corners[0].x;
    float y1 = scene_corners[1].y - scene_corners[0].y;
    float x2 = scene_corners[2].x - scene_corners[3].x;
    float y2 = scene_corners[2].y - scene_corners[3].y;
    float x3 = scene_corners[3].x - scene_corners[0].x;
    float y3 = scene_corners[3].y - scene_corners[0].y;
    float x4 = scene_corners[2].x - scene_corners[1].x;
    float y4 = scene_corners[2].y - scene_corners[1].y;
    cos_hori_top = (abs(x1)) / (pow((x1 * x1 + y1 * y1), 0.5) );
    cos_hori_bot = (abs(x2)) / (pow((x2 * x2 + y2 * y2), 0.5) );
    cos_verti_left = (abs(x3)) / (pow((x3 * x3 + y3 * y3), 0.5) );
    cos_verti_right = (abs(x4)) / (pow((x4 * x4 + y4 * y4), 0.5) );

    tan_hori_top = std::atan(x1/y1);
    tan_hori_bot = std::atan(x2/y2);
    tan_verti_left = std::atan(x3/y3);
    tan_verti_right = std::atan(x4/y4);

    if ((abs(tan_hori_top) > 60 ) || (abs(tan_hori_bot) > 60 ) || (abs(tan_verti_left) >
60 ) || (abs(tan_verti_right) > 60 )){
        notweirdshape = false;
    }

    if (abs((cos_hori_top - cos_hori_bot) < 0.5) && (abs(cos_verti_left -
cos_verti_right) < 0.5) && notweirdshape){
        if_parallel_sides.push_back(1);
    }else{
        if_parallel_sides.push_back(0);
    }
}

```

```

        float current_area = cv::contourArea(scene_corners);
        currentarea.push_back(current_area);
    }

    img_object.release();

    size_t area = 0;
    size_t desired_matches = 0;
    for (size_t match_index = 0; match_index < num_matches.size(); match_index++)
    // find the maximum number of matches and its corresponding index number
    {
        if ((area < currentarea[match_index]) && (num_matches[match_index] > 80) &&
        (if_parallel_sides[match_index] = 1) && (currentarea[match_index] > 1000)) // Add one
        more constraint of homography
        {
            template_id = num_matches_index[match_index];
            desired_matches = num_matches[match_index];
            area = currentarea[match_index];
        }
    }

    std::cout << "number of best matches:" << desired_matches << std::endl;
    std::cout << "template_id" << template_id << std::endl;

    if (desired_matches < 30)
    {
        template_id = -1; // if match number is too small, it is the blank image
        std::cout << "Blank Tag <50:" << std::endl;
        important_info.push_back(template_id);
        important_info.push_back(0);
        important_info.push_back(0);
        return important_info;
    }

    img_object = boxes.templates[template_id];

```



```

int minHessian=250;
Ptr<SURF> detector =SURF::create( minHessian);
std::vector<KeyPoint> keypoints_object, keypoints_scene;
Mat descriptors_object, descriptors_scene;
//Mat img_object = boxes.templates[ind];
detector->detectAndCompute( img_object, Mat(), keypoints_object,
descriptors_object);
detector->detectAndCompute( img_scene, Mat(), keypoints_scene,
descriptors_scene);

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
const float ratio_thresh=0.75f;
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]);
    }
}

//--Draw matches
Mat img_matches;
drawMatches( img_object, keypoints_object, img_scene, keypoints_scene,
good_matches, img_matches, Scalar::all(-1),Scalar::all(-1), std::vector<char>(),
DrawMatchesFlags::NOT_DRAW_SINGLE_POINTS );

//-- Localize the object
std::vector<Point2f> obj;
std::vector<Point2f> scene;

for( int 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 );
}

```

```
}
```

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

```
//-- Get the corners from the image_1 ( the object to be "detected" )
std::vector<Point2f> obj_corners(4);
obj_corners[0] = cvPoint(0,0); obj_corners[1] = cvPoint( img_object.cols, 0 );
obj_corners[2] = cvPoint( img_object.cols, img_object.rows ); obj_corners[3] =
cvPoint( 0, img_object.rows );
std::vector<Point2f> scene_corners(4);
```

```
perspectiveTransform( obj_corners, scene_corners, H);
```

```
//-- Draw lines between the corners (the mapped object in the scene - image_2 )
line( img_matches, scene_corners[0] + Point2f( img_object.cols, 0),
scene_corners[1] + Point2f( img_object.cols, 0), Scalar(0, 255, 0), 4 );
line( img_matches, scene_corners[1] + Point2f( img_object.cols, 0),
scene_corners[2] + Point2f( img_object.cols, 0), Scalar( 0, 255, 0), 4 );
line( img_matches, scene_corners[2] + Point2f( img_object.cols, 0),
scene_corners[3] + Point2f( img_object.cols, 0), Scalar( 0, 255, 0), 4 );
line( img_matches, scene_corners[3] + Point2f( img_object.cols, 0),
scene_corners[0] + Point2f( img_object.cols, 0), Scalar( 0, 255, 0), 4 );
```

```
//std::cout << "scene_corners[0]" << scene_corners[0] << std::endl;
//std::cout << "scene_corners" << scene_corners << std::endl;
//-- Show detected matches
imshow( "Good Matches & Object detection", img_matches );
float x1 = scene_corners[1].x - scene_corners[0].x;
float y1 = scene_corners[1].y - scene_corners[0].y;
float x2 = scene_corners[2].x - scene_corners[3].x;
float y2 = scene_corners[2].y - scene_corners[3].y;
float x3 = scene_corners[3].x - scene_corners[0].x;
float y3 = scene_corners[3].y - scene_corners[0].y;
float x4 = scene_corners[2].x - scene_corners[1].x;
float y4 = scene_corners[2].y - scene_corners[1].y;
cos_hori_top = (abs(x1)) / (pow((x1 * x1 + y1 * y1), 0.5) );
cos_hori_bot = (abs(x2)) / (pow((x2 * x2 + y2 * y2), 0.5) );
cos_verti_left = (abs(x3)) / (pow((x3 * x3 + y3 * y3), 0.5) );
cos_verti_right = (abs(x4)) / (pow((x4 * x4 + y4 * y4), 0.5) );
```

```
float current_area = cv::contourArea(scene_corners);

//std::cout << "cos_hori_top" << cos_hori_top << std::endl;
//std::cout << "cos_hori_bot" << cos_hori_bot << std::endl;
//std::cout << "cos_verti_left" << cos_verti_left << std::endl;
//std::cout << "cos_verti_right" << cos_verti_right << std::endl;
std::cout << "final_current_area" << current_area << std::endl;
```

```
waitKey(10);
```

```
descriptors_object.release();
descriptors_scene.release();
H.release();
img_matches.release();
img_object.release();
img_scene.release();
good_matches.clear();
if_parallel_sides.clear();
currentarea.clear();
obj_corners.clear();
scene_corners.clear();
obj.clear();
scene.clear();
num_matches.clear();
num_matches_index.clear();
```

```
int convert_desired_matches = static_cast<int>(desired_matches);
int convert_area = static_cast<int>(area);
important_info.push_back(template_id);
important_info.push_back(convert_desired_matches);
important_info.push_back(convert_area);
return important_info;
```

```
}
```

```
}
```

3. 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 <vector>
```

```
class ImagePipeline {
```

```
    private:
```

```
        cv::Mat img;
```

```
        bool isValid;
```

```
        image_transport::Subscriber sub;
```

```
    public:
```

```
        ImagePipeline(ros::NodeHandle& n);
```

```
        void imageCallback(const sensor_msgs::ImageConstPtr& msg);
```

```
        std::vector<int> getTemplateID(Boxes& boxes);
```

```
};
```