Mar. 24th, 2020

MIE443

Mechatronics Systems: Design and Integration

Contest 2

Team 2

Name	**Student Number**
Zian Zhuang	1002449870
Jinxuan Zhou	1002350582
Rui Cheng Zhang	1002301948
Xu Han	999263434
Tianzhi Huang	1002076807

Table of Contents

1.0 Problem Definition

The goal of Contest 2 is to implement an algorithm that allows the Turtlebot to spot and navigate to five randomly located objects and subsequently recognize their feature tags [1]. The enclosed contest environment has a total area of 4.87m*4.87m, in which each 'object' is made of two cardboard boxes of dimensions of 50*32*40 cm^3 , with a feature tag (image) on one face of the object. The exact coordinates of the five objects will be given on the contest day, therefore having a robust program that enables the Turtlebot to accurately locate itself with respect to these five destination points is critical. Along with the object positions defined by the coordinates of its center and orientation, a 2D map of the contest environment will also be provided using the gmapping package prior to the contest.

Moreover, the Turtlebot will be using the OpenCV library, which is commonly built for computer vision applications, to detect and identify the tag feature on the five objects. Feature tags are represented by images and attached to only one side of the object. There will be three objects with unique tags, one with a duplicate tag and one with a blank space. As a result, the Turtlebot should distinguish and save the detected images through the RGB camera within its Kinect sensor.

Throughout the contest, the Turtlebot is able to utilize built-in libraries to achieve localization and obstacle avoidance. A time limit of 5 minutes is imposed for the robot to finish all the tasks before traversing back to its starting point and indicating the complement of the contest. After completion, the program should produce an output file showing the recognized tags and their corresponding locations.

2.0 Overview of Strategies

Before execution of the navigation and image recognition algorithm, the Turtlebot is required to localize in the given environment using Adaptive Monte Carlo Localization (AMCL) which is based upon the map yaml file generated by gmapping (please refer to Section 4.1.2 for more details). Physically, it requires a few additional steps to help the Turtlebot recognize the environment by moving and orienting the turtlebot moderately in order to minimize localization errors. After localization, the program needs to load and modify the coordinates and image templates applicable for its subsequent navigation and image processing algorithms. The program takes the given box coordinates associated with the known map environment from coords.xml file. These coordinates represent the center of the boxes, and the program converts them to navigable

coordinates where the turtlebot can have a complete and clear view of the image tag on each box. The details about how the coordinates are modified is explained in Section 4.1.1. Additionally, the program loads the provided tag images from the templates folder, and then builds descriptors that are used in the later image recognition process for each tag image with the custom imagePipeline.loadObjects(boxes) function. The algorithm of how the descriptors are built for image recognition is illustrated in Section 4.2.2. Constructing image descriptors at the initialization stage instead of whenever the image processing function is called speeds up the process and optimizes the program as it avoids duplication of work. Ros::spinOnce() is executed at the start of the program to update the current robot pose. Upon robot initialization, the pose is recorded as the starting position and starting orientation. The main strategy to solve the tasks is then executed after robot localization and initialization.

The core of robot navigation is to use the nearest neighbour greedy method, and image recognition is achieved through the SURF algorithm. More specifically, the navigation algorithm is established by computing coordinates of the closest box with respect to the current robot coordinates. This requires the program to set all boxe statuses as “unviewed” in the beginning, and then the program alters the boxe statuses as the box is examined one by one. This ensures all boxes can be visited and prevents the Turtlebot from examining each box more than once. The detailed implementation and justification of using this algorithm is illustrated in Section 4.2.1. Once the nearest box has been found, the Turtlebot will travel to the goal coordinates/pose of the box by utilizing the movetoGoal function.

After the turtlebot reaches the desired location, ros::spinOnce() is run to refresh the image feed. The imagePipeline.getTemplateID(boxes) function then matches the key descriptors on the scene image with the tag descriptors built at the start of the program. Once the image is recognized, the program stores its templateID along with its corresponding box number into a results_vector, and changes the examined boxe status to “viewed”. The method of image processing is elaborated in Section 4.2.2.

At the end, the program checks if all boxes have been visited by iterating through the boxe statuses. If any one of the statuses is “unviewed”, the program repeats itself by finding the closest box among unvisited boxes with respect to the current robot position. If all boxes have been examined, the iteration process breaks and the turtlebot will travel to the starting position using the movetoGoal function. As the program terminates, it interprets the results_vector, a vector of integer pairs, into strings by reading the templates.xml file included. The result text file is then exported to the user-defined directory, through the writeResults(results_vector) function.

2.1 Simulation Results

We conducted extensive testing on our program through Gazebo simulations and received promising results. The simulation environment is illustrated in Figure 1 below, along with the output image and messages.

The robot is able to calculate the goal positions and orientations, and subsequently calculate the paths with respect to its current position. It then navigates to the closest unexplored box. Upon arrival at each destination, the robot stops to take a snapshot of its current view through the camera, and the image gets passed on for further processing. An example of the captured image is shown in Figure 2. The target box is in the center of the view. The program then attempts to match the image to one of the 3 given templates. Finally, after exploring all 5 boxes, the robot goes back to the original starting position. The program terminates after writing the results to a txt file ("Results.txt") correctly listing the templates matched and the location they were discovered at.

The full video of the simulation can be viewed here:

https://drive.google.com/file/d/1U1rrSMnYqr5uiRTeNn_JtLm0G0biuszL/view?usp=sharing [2]

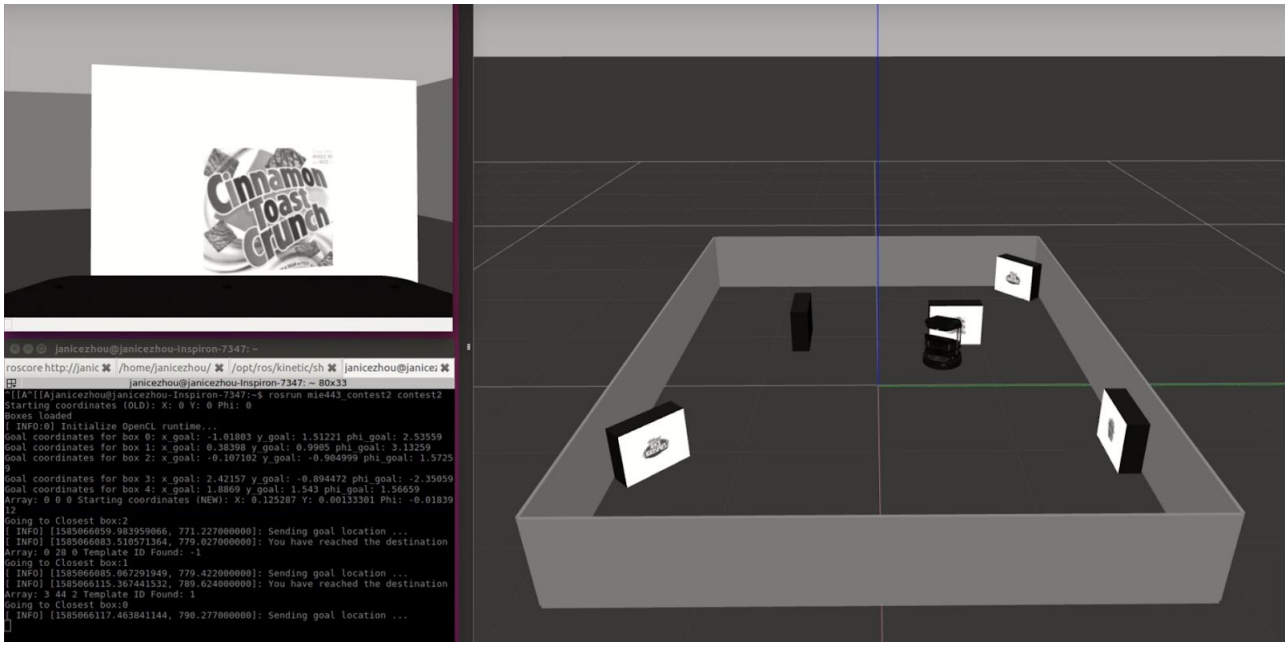

Figure 1: Simulation environment and output

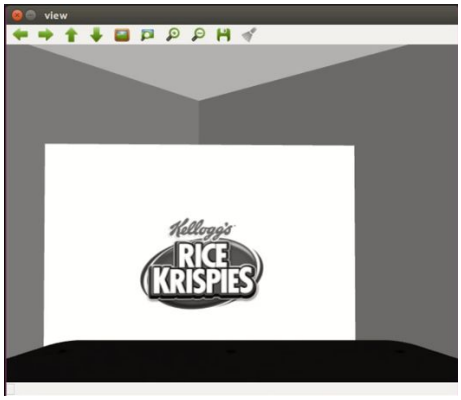

Figure 2: Example of captured image

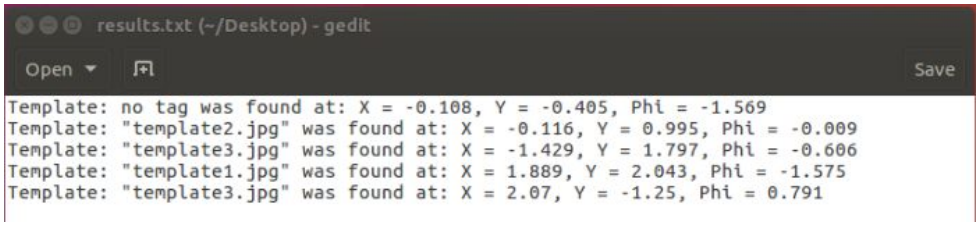

Figure 3: Results.txt file

3.0 Sensory Design

This section addresses the main sensors used in the development and testing of the algorithm for Contest 2. Figure 4 highlights the components of the Turtlebot and how they are connected to one another.

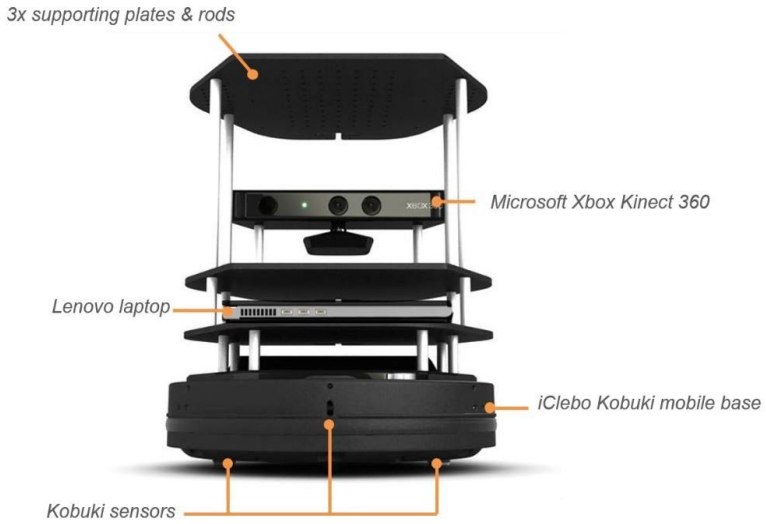

Figure 4. Turtlebot Schematics [3]

3.1 Kobuki Mobile Base

The Kobuki mobile base is equipped with cliff sensors, drop sensors, bumpers, a gyro encoder and wheel encoders. Due to the nature of Contest 2, only the gyro and wheel encoders are utilized by the Turtlebot to maneuver itself to the front of five designated object boxes. The feedback with regards to the odometry of the Turtlebot is retrieved from the iClebo's wheel encoders at each instance of time relative to its initial position, and it determines the estimated location in space of the robot with respect to the given world frame. The gyro sensor measures the linear and angular acceleration which also assists in computing the odometry information including robot coordinates and orientation. For Contest 2, the feature of encoders is used by low-level controls to achieve localization, map navigation and also adjustment of robot position and orientation once it arrives at the object.

3.2 Microsoft 360 Xbox Kinect Sensor

The Microsoft Kinect is composed of three sensory components --- a RBG camera, a depth sensor and a microphone array, as shown in Figure 5. In Contest 2, only the RBG camera is used for recognizing the image tags on the objects. As the Turtblebot locates and arrives at the destination points, it would then execute SURF feature detection for feature identification through the RBG camera, and the processed images will be verified by our program.

The RBG camera contains three channels of primary colours, namely, red, green and blue. After the communication with the Kinect sensor is established, the RGB pixel data obtained from scanning an image is broadcast by its camera node for the Kinect sensor. To transform the processed image in grayscale, the sensor employs its image-processing node and image-display node by converting pixel values.

In Contest 2, the OpenCV library is also used to facilitate image processing, where a large database of computer vision algorithms is available to interface with image frames captured by the Kinect sensor. Our algorithm is designed to examine the output of matched image features with respect to the given contest images. It is also suggested that laptop webcams are compatible with the OpenCV library to develop and refine a proof-of-concept algorithm related to object detection and recognition.

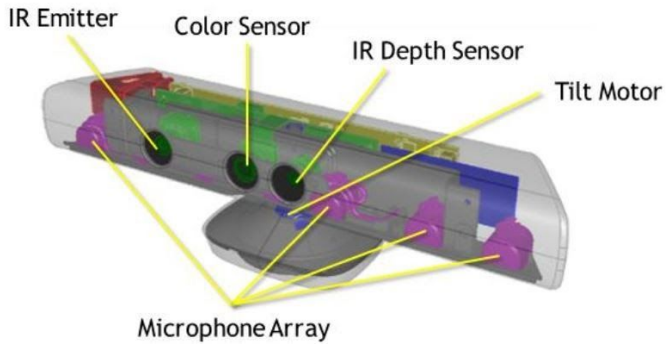

Figure 5. Microsoft 360 Xbox Kinect Sensor [3]

4.0 Controller Design and Strategy

The overall architecture can be summarized with a Nested Hierarchical Controller shown in Figure 6. Sensing and acting are achieved using low-level controllers introduced in Section 4.1, and Section 4.2 focuses on the planning and decision making tasks in the contest.

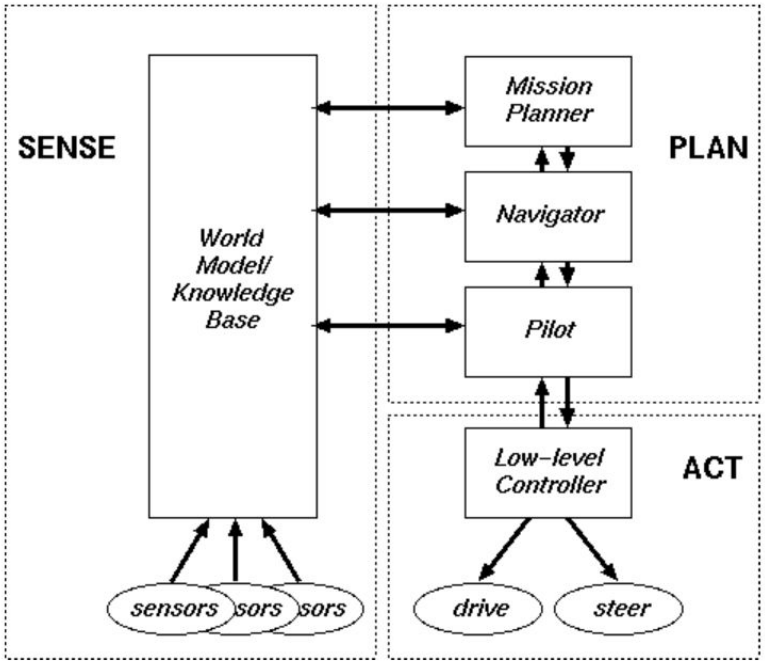

Figure 6. Nested Hierarchical Controller [4]

4.1 Low-Level Control

Low-level control involves using the modules and packages offered in the ROS environment and the OpenCV framework. The information gathered from the modules, along with output commands obtained in high-level control, was processed and analyzed. The following sections will cover four main modules: Robot Position and Orientation, Adaptive Monte Carlo Localization, Move Base Navigation, OpenCV framework and Write Results.

4.1.1 Robot Position and Orientation

The update_coords() function first evaluates the coordinates of the targeted objects, represented by (x, y, ϕ), and then sends a position signal of the destination point, represented by (x^{new}, y^{new}, ϕ^{new}), to the Turtblebot for navigation. It is important for the robot to move to the correct location and face the normal to the image while capturing the image within the camera frame. During the testing of camera image recognition, it was determined that a distance of 0.5m away from the boxes yields the best result. The (x^{new}, y^{new}, ϕ^{new}) labelled in Figure 7 is the desired location for the robot to perform image recognition.

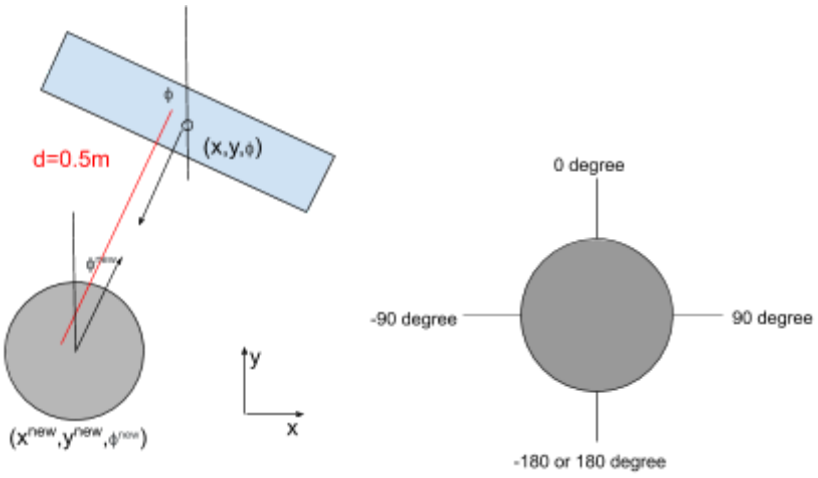

Figure 7. Robot Position and Orientation

As the "ϕ" orientation of the robot has the configuration shown in Figure 7, simple trigonometry can be employed to convert the given object coordinates to the desired destination location for the Turtlebot. The navigable coordinates (x^{new}, y^{new}, ϕ^{new}) are appended as three additional integer elements in the "boxes.coords" array. The structure of the array is shown in Section 4.2.1.

$$x^{new} = x + 0.5\ cos(\phi) \qquad y^{new} = y + 0.5\ sin(\phi)$$
$$\phi^{new} = \phi + 180^{o}\ for\ \phi < 0;\ \phi^{new} = \phi - 180^{o}\ otherwise$$

4.1.2 Adaptive Monte Carlo Localization (AMCL)

The AMCL is an open-source package for the Turtlebot to localize in a given map environment using a particle filter [5]. During the contest, the map of the arena generated by gmapping was inputted into the RVIS or Gazebo, and AMCL allows the robot to estimate its position and orientation, facilitating path planning and navigation. The initial initial filter state will be a moderately sized particle cloud centered about (0,0,0). The RVIS allows a "2D Pose Estimation," which is filled with potential particle clouds at the locations of the robot. With any movements from the odometry sensor and kinetic sensor, the particle cloud will converge into a single point in which the robot is localized within the map. The AMCL package is implemented with the move_base package which enables the navigation within the map with given coordinates.

4.1.3 Move Base Navigation

The move_base package provides an implementation of an action that, given a goal in the world, will attempt to reach it with a mobile base. In this contest, this is the primary function utilized to maneuver the robot in this given environment. The move_base package utilizes AMCL map information and sensor feedback to develop a route to the goal location. The package acquires information for obstacle avoidance ih real-time. However, the robot sometimes encounters issues where it undergoes its recovery behaviour. The default robot recovery behaviour is outlined in Figure 8 below.

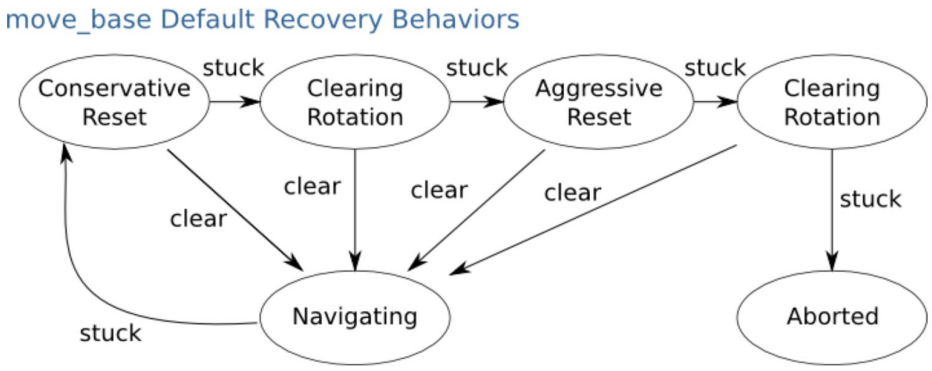

Figure 8: Move Base Navigation Expected Robot Behaviour [6]

In the contest, there are a few situations which will trigger the recovery behaviours. When the robot position and goal point is set outside of the given map or in an area with obstacles, the robot will be stuck in the clearing rotation and result in an abort function outlined in Figure 8. In other scenarios, when the robot approaches an edge of the map,

the robot will enter the recovery behaviours until the robot moves outside the trapped environment. After the move_base undergoes the abort command in its recovery behaviours, the program will default resume. With the situation given above, the team added additional requirements to check the current position before initiating the image recognition process. The detailed high-level program architecture is outlined in Section 4.2.

4.1.4 Image streaming from Camera

The camera embedded in the Kinect sensor of the Turtblebot is capable of receiving both depth and RGB information from the surroundings. In Contest 2, however, only the RGB imagery is deemed useful for image recognition of the feature tags. The schematics below illustrate how the program is constituted with various interlinked nodes. Under the ROS Master, three nodes are registered --- camera node, image processing node, and image display node. The visual data feeds into the camera node from the Kinect sensor and gets transferred to the image processing node in the form as RGB pixel values. The program then sends the processed image in either RGB or grayscale to the image display node so that the users can visually verify the image identifiability of the algorithm as proof-of-concept testing.

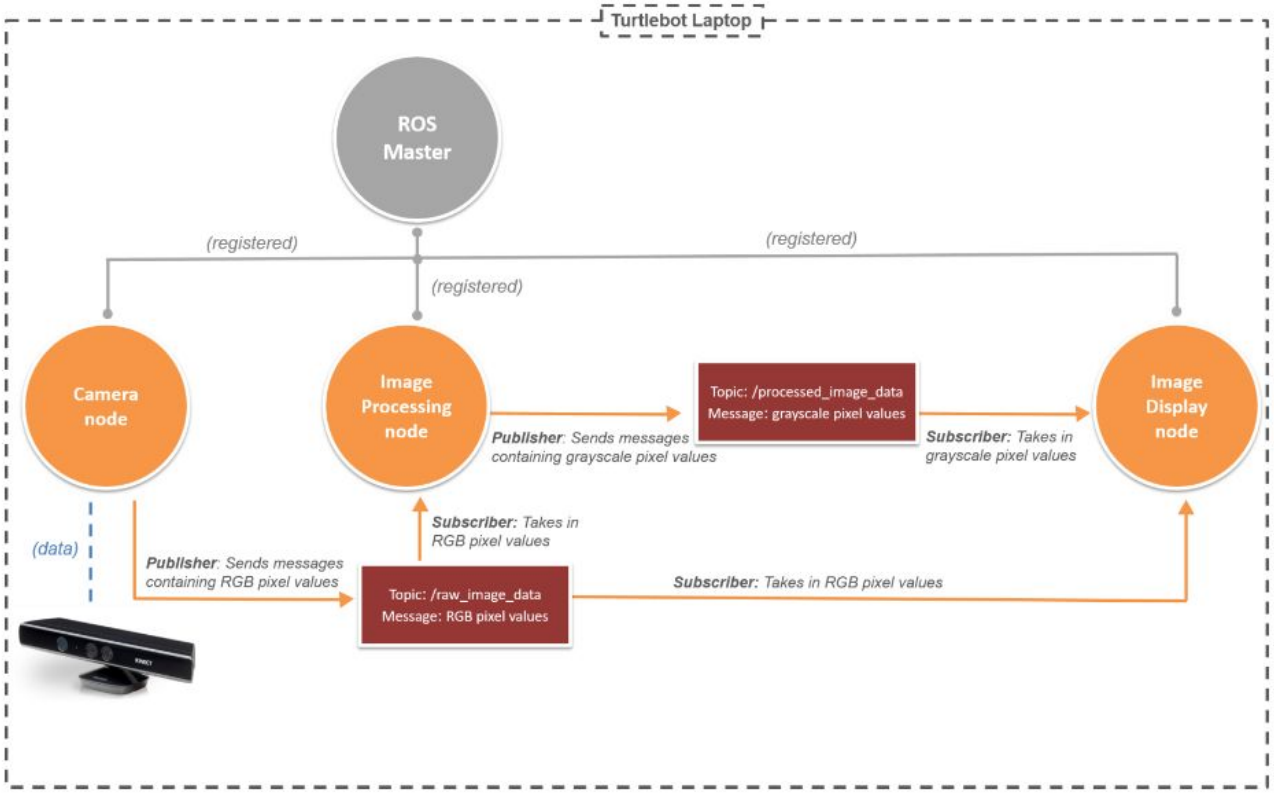

Figure 9: Image Processing Schematics [7]

4.1.5 OpenCV Library

The OpenCV library [8], which contains an immense database of optimized real-time computer vision algorithms, is utilized for the recognition of the image tags. The code that initializes the library has been provided, where the program has the ability to perceive matching features using the SURF detection algorithm between a provided image and a scene image. The matching process is invariant to scaling, translation and rotation of the scene image as the target object within a scenery can be located based on calculated transformation between matching features. To extend beyond the provided code, we implemented a logic by imposing a threshold to ensure the quality of matching between the provided image and the scene image is adequate. Details of implementation can be found below in Section 4.2.2.

4.1.6 Write Result to Text

Our WriteResults() function is the algorithm to create a text file with the following format below. With each line as an individual box noting which template was found at this respective box and the X, Y, and Phi coordinates of the box.

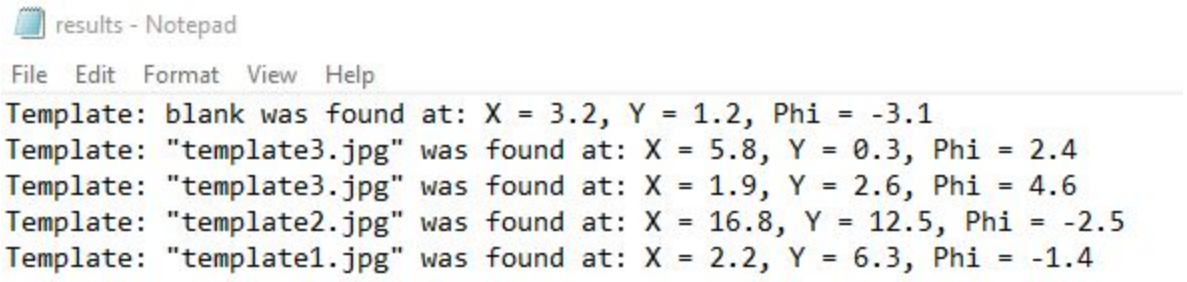

Figure 10: Results Text File Example Format

The function takes in a vector of a pair of integers, called results_vector. Each pair in the vector represents a box we have visited, with the first integer being the templateID found, and the second integer being the box number.

Using this information, the function creates a file called “results.txt” and iterates through the results_vector. The templateID is converted into a string through reading the appropriate line in templates.xml for objects 0, 1, and 2, or is simply set to “no tag” in the case of a blank.

The X, Y, and Phi information are retrieved through the global boxes variable. Given the box number, we access the coordinates of that particular box and write to the appropriate line in “results.txt”.

Once all 5 items in results_vector have been iterated over, we close and save the file.

4.2 High-level Control

The high-level control utilizes the data structures and logic from low-level control and performs additional computations and logic for the overall traversal algorithm to command the Turtlebot during Contest 2.

4.2.1 Path Planning

The navigation challenge in this contest is to solve the Travelling Salesman Problem (TSP) so that the turtlebot can complete the task in an efficient way and within the time constraint. Since the number of the desired locations is not tremendous, it is possible to solve the problem using brute force method [9]. However, the team noticed that the path planning algorithm, movetoGoal function, built in the Move_Base library does not guarantee the shortest route to the goal locations. The route generated by Move_Base library depends on real-time information and could experience several recovery adjustments if there is any localization error, as shown in Section 4.1.3. Due to the unpredictable route planning, the shortest path found by using brute force might not be the most optimal path. Therefore, the team decided to use the simplest and easiest debugging algorithm to solve the TSP problem by implementing the nearest neighbour greedy method.

Using nearest neighbour greedy method requires the program to set a "viewed/unviewed" status for each desired box so the robot will not visit the boxes that have been previously visited. This is accomplished by appending an additional integer element in the "boxes.coords" array indicating the boxes visiting status, as 0 means "unviewed" and 1 means "viewed". The structure of the "boxes.coords" array is demonstrated in the following Table 1. This algorithm is implemented in the update_coords function. Each box's status is initially set as "unviewed" until it is visited and the image processing about its tag is completed.

Table 1. Array Structure of "boxes.coords"

<table>
<tr><td>boxes</td><td colspan="7">coords</td></tr>
<tr><td>0</td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td></tr>
<tr><td rowspan="2">Box number</td><td>x</td><td>y</td><td>phi</td><td>x_goal</td><td>y_goal</td><td>phi_goal</td><td rowspan="2">View status</td></tr>
<tr><td colspan="3">From coords.xml</td><td colspan="3">Navigable coords</td></tr>
</table>

To determine the closest box respective to the current robot pose, ros::spinOnce() needs to be executed to update the odometry and get the current robot pose. The function first initializes the shortest distance variable to infinity and the closest box number to -1 as default. Then, the function runs iterations to compute distance between the current robot position and navigable coordinates for each unvisited box. If the distance computed is smaller than the shortest distance variable, the shortest distance variable and the closet box number is replaced. This algorithm is implemented in the find_closestBox function.

Once the nearest box has been determined, the turtlebot travels to the goal pose of the box using movetoGoal function, and performs image recognition. If there is no unvisited box, the closest box number would be the default number, -1. Accordingly, the turtlebot is set to travel to its recorded starting position for the next step. This algorithm is implemented in the goto_Goal function.

4.2.2 Image Recognition

The basis of our image recognition algorithm is the OpenCV feature finding framework. The templates of each image (in the form of a matrix of pixel values) for all 3 objects and the scene (image from camera feed) are first processed to extract a vector of keypoints. Keypoints are extracted to describe the position and coverage area of each feature found. Second, we create a matrix of descriptors from the objects and their key points. Descriptors assign a numerical description to the area of the image the keypoint refers to. This allows descriptors to be independent of keypoint position, robust against image transformations, and scale independent. In other words, descriptors can be used to match objects regardless of position, rotations, and size within the image. These are all characteristics we desire as the object to be recognized will unlikely be correctly rotated and completely covered by the robot camera.

Once the descriptors of the objects and the scene have been stored, we use the FLANN (Fast Library for Approximate Nearest Neighbors) based matcher to match the scene descriptor with the object descriptor, and then filter out the relatively “bad” matches based on a Euclidean distance threshold. After filtering, the matches are stored into the good_matches vector.

We repeat the FLANN matching for each of the 3 objects with the scene image. For each iteration, we simply analyze the good_matches vector for its overall number of

items and treat this number as a “score” of how closely the object matches the scene. These “scores” are pushed back into a match_points vector.

After all objects have been analyzed compared to the scene, we have a match_points vector storing the number of matches of object 0, 1, and 2 (respective of their positions in boxes.templates), respectively. We iterate through this vector to obtain the index of the maximum number (0, 1 or 2), and treat this index as the object which best matches the scene.

We then compare this with a threshold value. If the “score” is below the threshold value, we assume that the scene does not contain any of the 3 objects of interest, and the matches are simply “noise”, so the templateID returned will be -1 for “blank/noise”. If it is above the threshold, we assume a positive match with the respective object, and so the index of the object is stored in templateID to be returned. The templateID is then returned. The threshold value was determined iteratively through testing the robot in the contest arena.

With the change into the Gazebo platform, however, we noticed through testing that sometimes the blank template has unreasonably high values in our “score” calculation. Additionally, we also noticed that the actual tags have all 3 “scores” (matches with each of the 3 respective objects) greater than 0, the blank template would sometimes have high “scores” with template 2, and 0 “scores” with templates 1 and 3.

To mitigate variance and try and overcome this Gazebo-specific problem, we also added an upper threshold (any “scores” higher than this is assumed to be the blank template), and added another condition checking for 0 “scores” with templates 1 and 3, which also signify a blank template in Gazebo. We do not believe these measures are necessary for a real-world contest, but seemed to work better for us once we moved our testing to within Gazebo.

One major modification we made to the imagePipeline.cpp file is the addition of a global descriptors_objects variable. Descriptors_objects store the matrices (cv::Mat) of each of the 3 objects to identify in a vector. During robot initialization, once all the object templates (cv::Mat) are loaded into boxes.templates, we run our custom member function loadObjects() located in imagePipeline.cpp. The function takes every template stored in boxes.templates and uses the aforementioned process in OpenCV to convert each object into a descriptor (cv::Mat), which is then pushed back into descriptors_objects.

Without the loadObjects() function, each call of getTemplateID() would repeat the keypoint and descriptor analysis for each of the 3 object images, which is unnecessary

processing as they remain unchanged through the competition. This way, we only run loadObjects() once throughout the entire run of the robot. Every call to getTemplateID() subsequently will only create descriptors for the scene, and reuse the descriptors from each object from the global vector, reducing the operations required for each call of the function. Accordingly, we also modified the imagePipeline.h file to include our loadObjects() function.

5.0 Future Recommendations

5.1 Current issues

Through testing in Gazebo simulations, the team has identified some issues which could affect the efficiency of the navigation and accuracy of the image identification results. First of all, as explained in Section 4.1.3, the algorithm in the move_base package involves a lot of robot rotations for recovery and adjustment. This feature sometimes results in inefficient path planning as the robot will spin in its spot or spiral towards its goal instead of going in a straight line. Another consequence of this rotation feature is that the robot sometimes reaches a destination and stops at an orientation (ϕ) that is slightly different from the desired ϕ. As a result, the target object will not be in the centre of the captured view or will be viewed at an angle, which can affect the accuracy of the template matching program.

5.2 Would you use different methods or approaches based on the insight you now have?

As mentioned in Section 4.2.1, the team decided to use the nearest neighbour greedy algorithm for navigation because the path planned by movetoGoal function could be inaccurate. As discussed above, the movetoGoal function involves a lot of robot rotation for recovery and incremental adjustment. Based on this insight, the algorithm can be retrofitted with additional functions to ensure the smooth and accurate robot movement. A potential improvement can be made by using visual data perceived by the Kinect sensor for navigation instead of relying heavily on odometry and AMCL results. Alternatively, the robot could break out from the movetoGoal function once the computer vision realizes the goal is close enough to perform image recognition. If the functionality of movetoGoal logic is guaranteed, implementing a shortest hamiltonian circuit with brute force could be more efficient than using nearest neighbour greedy algorithm.

5.3 What would you do if you had more time?

If we had more time to work on our current project, we would try and implement machine learning techniques for image recognition. The supervised learning can be determined with a training set data of the logo in different angles. With the image streamed from the camera, a sliding window method can be used to find the logo within the frame. However, this way, it will require more computational power and a considerable amount of training and cross-validation set to compute the accurate image logo. On the other hand, in the context of determining the brand logo for each cereal brand, photo optical character recognition (OCR) can be very effective in implementing and identifying the context of the image logo [10]. Through implementing a photo OCR pipeline, including text detection, character segmentation and character recognition, with the available online resources for training sets, image recognition can be effectively performed and the image converted to text. After the image is converted into text, the comparison will be straightforward for text matching. However, this method only works with the contexts of the cereal boxes, and is not generalizable for any increases in contest scope.

6.0 Attribution Table

Section	Student Names				
	Zian Zhuang	**Jinxuan Zhou**	**Xu Han**	**Rui Cheng Zhang**	**TianZhi Huang**
Robot Design					
Navigation Code	MR, DB		DB	RD, RS	RS
Image Processing Code		DB	RD, MR, DB	DB	RS
Gazebo Simulation	RD, DB	RD, MR, DB	DB		
Testing	DB	DB	DB	DB	DB
Report					
1.0 Problem					RD
2.0 Overall Strategy	RD	RD, MR		ET	ET
3.0 Sensor Design			ET	ET	RD
4.0 Controller Design	RS, RD, MR		RS, MR, DB	RD, ET	ET
5.0 Future Recommandations	MR	RD, MR		MR, ET	ET
Overall	ET	ET	ET	ET	ET

RS – major research

RD – wrote first draft

MR – major revision

DB - Debug and Testing

ET – edited for grammar and spelling

7.0 References

[1] MIE443_contest2.PDF | *University Of Toronto*. [online] Available at: <https://q.utoronto.ca/courses/139490/files/folder/Contests/Contest%202?preview=6327114>

[2] Google Drive. 2020. *Copy Of Contest2_Team2.Mp4*. [online] Available at: <https://drive.google.com/file/d/1U1rrSMnYqr5uiRTeNn_JtLm0G0biuszL/view?usp=sharing>

[3] MIE443 Turtlebot Student Manual 2020.pdf | *University Of Toronto*. [online] Available at: <https://q.utoronto.ca/courses/139490/files/folder/Manual?preview=5556150>

[4] Intelligent Control Lecture 4.pdf | *University Of Toronto*. [online] Available at: <https://q.utoronto.ca/courses/139490/files/folder/Notes?preview=5883300>

[5] MIE443 Tutorial 3 2020.pdf | *University Of Toronto*. [online] Available at: <https://q.utoronto.ca/courses/139490/files/folder/Tutorials?preview=6223300>

[6] Wiki.ros.org. 2020. *Move_Base - ROS Wiki*. [online] Available at: <http://wiki.ros.org/move_base>

[7] Intro to Ubuntu and ROS Lecture 2.pdf| *University Of Toronto*. [online] Available at: <https://q.utoronto.ca/courses/139490/files/folder/Notes?preview=5717842>

[8] MIE443 Tutorial 4 2020.pdf| University Of Toronto. [online] Available at: <https://q.utoronto.ca/courses/139490/files/folder/Tutorials?preview=6223183>

[9] Www2.cs.sfu.ca. 2020. 13. Case Study: Solving The Traveling Salesman Problem — CMPT 125 Summer 2012 1 Documentation. [online] Available at: <https://www2.cs.sfu.ca/CourseCentral/125/tjd/tsp_example.html>

[10] “Learning center,” *OCR - Optical Character Recognition Explained | Learning Center*. [Online]. Available: https://www.abbyy.com/en-ca/finereader/what-is-ocr/.

Appendices:

Contest2.cpp

```
#include <boxes.h>
#include <robot_pose.h>
#include <navigation.h>
#include <imagePipeline.h>
#include <iostream>
#include <fstream>
#include <algorithm>
#include <vector>
#include <utility>
#include <string>
#include <regex>
#include <cmath>
#include <chrono>
#define RAD2DEG(rad) ((rad)*180. / M_PI)
#define DEG2RAD(deg) ((deg)*M_PI / 180.)

//                              GLOBAL                                VARAIBLES
================================================================================
=====
Boxes boxes;
RobotPose robotPose(0, 0, 0);
std::vector<std::pair<int, int>> results_vector;
float startPose_x = 0.0, startPose_y = 0.0, startPose_z = 0.0;
float blockDist = 0.5;
int goal = -1;
```

```
//                                                                    FUNCTIONS
================================================================================
============

float calcDistance(float x1, float y1, float x2, float y2)

{

   return std::sqrt(std::pow(x1 - x2, 2) + std::pow(y1 - y2, 2));

}


void update_coords()

{ // update coords [0-2] to navigatable coords [3-5]

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

   {

              boxes.coords[i].push_back(boxes.coords[i][0]  +  blockDist  *
cos(boxes.coords[i][2])); // goal x

              boxes.coords[i].push_back(boxes.coords[i][1]  +  blockDist  *
sin(boxes.coords[i][2])); // goal y

                                    if     (boxes.coords[i][2]     <     0.0)
// goal phi

       {

           boxes.coords[i].push_back(boxes.coords[i][2] + M_PI);

       }

       else

       {

           boxes.coords[i].push_back(boxes.coords[i][2] - M_PI);

       }

                  boxes.coords[i].push_back(0);   //   0/1   indicate   F(not
examed)/T(examed)

         std::cout << "Goal coordinates for box " << i << ": x_goal: " <<
boxes.coords[i][3] << " y_goal: " << boxes.coords[i][4] << " phi_goal: "
<< boxes.coords[i][5] <<  std::endl;

   }
```

```
}


void find_closestBox()
{ // find closest box with respective to current pose
   float minDist = std::numeric_limits<float>::infinity();
   float boxDist = 0.0;
   goal = -1;
   for (int i = 0; i < boxes.coords.size(); ++i)
   {
       if (boxes.coords[i][6] != 1)
       { // for unexamed box
                       boxDist  =  calcDistance(robotPose.x,  robotPose.y,
boxes.coords[i][3], boxes.coords[i][4]);
           if (minDist > boxDist)
           {
               minDist = boxDist;
               goal = i;
           }
       }
   }
}


void goto_Goal()
{
   if (goal == -1)
   { // go to startPose if no unexamed box
             std::cout  <<  "ERROR:  no  unexamed  box  -->  go  to  starting
coordinates" << std::endl;
       Navigation::moveToGoal(startPose_x, startPose_y, startPose_z);
```

```
    }
    else
    {
        std::cout << "Going to Closest box:" << goal << std::endl;
                                Navigation::moveToGoal(boxes.coords[goal][3],
boxes.coords[goal][4], boxes.coords[goal][5]);
       // ros::spinOnce();
                        //   if   (calcDistance(robotPose.x,   robotPose.y,
boxes.coords[goal][3], boxes.coords[goal][4]) > 0.2)
       // {
              //          Navigation::moveToGoal(startPose_x,  startPose_y,
startPose_z);
                 //           Navigation::moveToGoal(boxes.coords[goal][3],
boxes.coords[goal][4], boxes.coords[goal][5]);
       // }
    }
}


std::string trim(const std::string &str)
{
    const auto strBegin = str.find_first_not_of(" \t");
    if (strBegin == std::string::npos)
        return ""; // no content
    const auto strEnd = str.find_last_not_of(" \t");
    const auto strRange = strEnd - strBegin + 1;
    return str.substr(strBegin, strRange);
}


void writeResults(std::vector<std::pair<int, int>> results_vector)
{
```

```
   std::fstream templates_file;

       std::ofstream  results_file("/home/janicezhou/Desktop/results.txt");
//// update result file address

     for  (auto  it  =  results_vector.begin();  it  !=  results_vector.end();
it++)

   {

        results_file << "Template: ";

        std::string template_name;

        if (it->first == -1)

        {

            template_name = "no tag";

        }

        else

        {

            int i = 1;


templates_file.open("/home/janicezhou/catkin_ws/src/mie443_contest2/boxes_
database/templates.xml"); //// update result file address

            switch (it->first)

            {

            case 0:

                while (std::getline(templates_file, template_name))

                {

                    if (i == 4)

                    {

                        break;

                    }

                    i++;

                }

                templates_file.close();
```

```
        break;
    case 1:
        while (std::getline(templates_file, template_name))
        {
            if (i == 5)
            {
                break;
            }
            i++;
        }
        templates_file.close();
        break;
    case 2:
        while (std::getline(templates_file, template_name))
        {
            if (i == 6)
            {
                break;
            }
            i++;
        }
        templates_file.close();
        break;
    default:
        std::cout << "Error in switch statement\n";
        templates_file.close();
        break;
    }
```

```
        }
        std::string template_name_trimmed;
        template_name_trimmed = trim(template_name);
        results_file << template_name_trimmed;
        results_file << " was found at: X = ";
        results_file << boxes.coords[it->second][0];
        results_file << ", Y = ";
        results_file << boxes.coords[it->second][1];
        results_file << ", Phi = ";
        results_file << boxes.coords[it->second][2];
        results_file << "\n";
    }
    templates_file.close();
    results_file.close();
}

int main(int argc, char **argv)
{
    bool all_checked = true;
    uint64_t duration = 0;
    std::chrono::time_point<std::chrono::system_clock> start;
    ros::init(argc, argv, "contest2");
    ros::NodeHandle n;

    // wait to establish position for robot for 5 seconds
    start = std::chrono::system_clock::now();
    while ((startPose_x == 0.0 || startPose_y == 0.0 || startPose_z == 0.0)
&& duration < 5)
    { // wait for robot to get pose info or 5 seconds
```

```
        ros::spinOnce();

        startPose_x = robotPose.x;

        startPose_y = robotPose.y;

        startPose_z = robotPose.phi;

                                                                duration        =
std::chrono::duration_cast<std::chrono::seconds>(std::chrono::system_clock
::now() - start).count();

    }

    std::cout << "Starting coordinates (OLD):" << " X: " << startPose_x <<
" Y: " << startPose_y << " Phi: " << startPose_z << std::endl;


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


    // load boxes

    if (!boxes.load_coords() || !boxes.load_templates())

    {

            std::cout << "ERROR: could not load coords or templates" <<
std::endl;

        return -1;

    }

    std::cout << "Boxes loaded" << std::endl;

    /*

    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] << " phi: "

                  << boxes.coords[i][2] << std::endl;

    }

    */
```

```
    // set up imagePipeline
    ImagePipeline imagePipeline(n);
    imagePipeline.loadObjects(boxes);
    int templateID = -1;

    // add to boxes x y and phi of approriate robot position
    update_coords();

    // look at what the first image is
    ros::spinOnce();
    templateID = imagePipeline.getTemplateID(boxes);

    // save starting position
    startPose_x = robotPose.x;
    startPose_y = robotPose.y;
    startPose_z = robotPose.phi;
    std::cout << "Starting coordinates (NEW):" << " X: " << startPose_x <<
" Y: " << startPose_y << " Phi: " << startPose_z << std::endl;

    while (ros::ok())
    {
        ros::spinOnce();

        find_closestBox();

        if (boxes.coords[goal][6] != 1)
        { // if box is unexamined
            goto_Goal();
```

```
            // identify image
            ros::spinOnce();
            templateID = imagePipeline.getTemplateID(boxes);
            // push back image and box info
            results_vector.push_back(std::make_pair(templateID, goal));
            std::cout << "Template ID Found: " << templateID << std::endl;
            boxes.coords[goal][6] = 1; // box is now examined
        }
        all_checked = true;
        // check if all boxes have been examined
        for (int i = 0; i < boxes.coords.size(); ++i)
        {
            if (boxes.coords[i][6] != 1)
            {
                all_checked = false;
            }
        }
        if (all_checked)
        {
            break;
        }
        ros::Duration(0.01).sleep();
    }
     std::cout << "All boxes have been examined --> going to starting coordinates" << std::endl;
    ros::spinOnce();
    std::cout << "Current coordinates:" << " X: " << robotPose.x << " Y: " << robotPose.y << " Phi: " << robotPose.phi << std::endl;
    std::cout << "Starting coordinates:" << " X: " << startPose_x << " Y: " << startPose_y << " Phi: " << startPose_z << std::endl;
```

```
    std::cout << "Moving back to starting coordinates now..." << std::endl;


    Navigation::moveToGoal(startPose_x, startPose_y, startPose_z);

    ros::spinOnce();

    Navigation::moveToGoal(startPose_x, startPose_y, startPose_z);


    ros::spinOnce();

     std::cout << "Ending coordinates:" << " X: " << robotPose.x << " Y: "
<< robotPose.y << " Phi: " << robotPose.phi << std::endl;


    writeResults(results_vector);

    std::cout << "Contest 2 is completed. Results.txt saved to Desktop." <<
std::endl;

    return 0;

}
```

imagePipeline.cpp

```
#include <imagePipeline.h>

#include <string>

#include <algorithm>

#include "opencv2/calib3d.hpp"

#include "opencv2/highgui.hpp"

#include "opencv2/imgproc.hpp"

#include "opencv2/features2d.hpp"

#include "opencv2/xfeatures2d.hpp"

#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;


// vector of matrices -> descriptors of each of the 3 objects

std::vector<cv::Mat> descriptors_objects;


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

{

   image_transport::ImageTransport it(n);

      sub  =  it.subscribe(IMAGE_TOPIC,  1,  &ImagePipeline::imageCallback,
this);

   isValid = false;

}


// new function to load descriptors for each object, only need to call
once each trial
```

```
void ImagePipeline::loadObjects(Boxes boxes)

{

     // iterate through each template in boxes.templates, and create descriptor matrices for each

    for (auto it = boxes.templates.begin(); it != boxes.templates.end(); it++)

   {

       // dereference it to img_object

       cv::Mat img_object = *it;


       // error checking

       if (!img_object.data)

       {

           std::cout << " --(!) Error reading images " << std::endl;

           return;

       }


            // Detect the keypoints using SURF Detector, compute the descriptors

       int minHessian = 400;

       cv::Ptr<SURF> detector = SURF::create(minHessian);

       std::vector<KeyPoint> keypoints_object;

       cv::Mat descriptors_object;

       detector->detectAndCompute(img_object, noArray(), keypoints_object, descriptors_object);


       // pushback to the descriptors_objects vector

       descriptors_objects.push_back(descriptors_object);

   }

}
```

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


int ImagePipeline::getTemplateID(Boxes &boxes)
{
   // initialize template_id to be -1
   int template_id = -1;


   // error checking
   if (!isValid)
   {
```

```
        std::cout << "ERROR: INVALID IMAGE!" << std::endl;
    }
    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;
    }
    else
    {
        cv::Mat img_scene = img;
         std::vector<float> match_points; // vector to store "scores" for
each object based on closeness of match to scene

        // iterate through each template in descriptors_objects vector, and
match with img
                for (auto it = descriptors_objects.begin(); it !=
descriptors_objects.end(); it++)
        {
            // error checking
            if (!img_scene.data)
            {
                std::cout << " --(!) Error reading images " << std::endl;
                return -1;
            }

             //-- Step 1: For scene imgage, detect the keypoints using SURF
Detector, compute the descriptors
```

```
            int minHessian = 400;
            Ptr<SURF> detector = SURF::create(minHessian);
            std::vector<KeyPoint> keypoints_scene, keypoints_object;
            cv::Mat descriptors_scene;
            cv::Mat descriptors_object;
                    descriptors_object = *it; // dereference it to get
descriptors_object
                            detector->detectAndCompute(img_scene,  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_scene,  descriptors_object,
knn_matches, 2);


            //-- Filter matches using the Lowe's ratio test
            const float ratio_thresh = 0.7f;
            std::vector<DMatch> good_matches;


             // iterate through total matches, and only add to good_matches
if within threshold
            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]);
```

```
                }
            }


              // count number of "good_matches" and append to match_points
vector
            match_points.push_back(good_matches.size());
        }


        // pick max value from match_points --> use the index of max value
         // if max value from match_points < threshold, assume no no tag at
box --> index of max value to -1
        // template_id = index of max value
        std::cout << "Array: ";
           for (auto it = match_points.begin(); it != match_points.end();
it++)
        {
            std::cout << *it << " ";
        }
          // in Gazebo simulation, the all black box produces (0, x, 0)
        if ((match_points[0] == 0) && (match_points[2] == 0))
        {
            cv::imshow("view", img);
            cv::waitKey(10);
            return template_id;
        }


        int lower_threshold = 20;
        int upper_threshold = 90;
                int  max_index  =  std::max_element(match_points.begin(),
match_points.end()) - match_points.begin();
```

```
        // only consider it a match, if the points are between the lower
and upper thresholds
                if ((match_points[max_index] >= lower_threshold) &&
(match_points[max_index] <= upper_threshold))
       {
           template_id = max_index;
       }

       //std::cout << std::endl << "ID: " << template_id;
       cv::imshow("view", img);
       cv::waitKey(10);
       // std::cout << "\n----------------------------\n";
   }
   return template_id;
}

cv::Mat ImagePipeline::getImg()
{
   return img;
}
```