Contest 2 Report

MIE443 Mechatronics Systems: Design & Integration
Contest 2: Finding Objects of Interest in an Environment
Due: March 19th, 2024

Team #4	
Team Member	**Student Number**
Henry (Hua Hao) Qi	1005758039
Harry Park	1005674405
Alastair Sim	1006460287
Yi Lian	1005709333

1.0 Problem Definition/ Objective

The goal of MIE443: Contest 2 is to navigate an environment to find and identify five objects placed at different locations in an environment. This navigation task is carried out using a TurtleBot, which will drive autonomously in the environment using the coordinates of each object and the map of the environment generated with ROS gmapping, which are both provided in advance. The robot will rely on its Kinect sensor, which includes an RGBD camera, to conduct this task. Each object will have a feature tag which can be identified and interpreted using the OpenCV library. All coordinates are given with respect to the origin of the 2D map, and while the robot will not necessarily start at that origin, its position will still be measured with respect to this origin. The task is complete once the Turtlebot has visited all five objects and returned back to its starting position. The navigation task is defined to be fixed timed, with a robot-based subject of action, and a movement goal comprising of a combination of convergence and movement-while. The fixed time constraint is placed as the robot must navigate the environment and identify the specified objects within a time limit. In a real-world scenario, this time constraint may be due to scheduling restrictions (ie. gathering people at an office for a meeting before 2 pm). Although the task is fixed-timed, there is a preference for minimizing the navigation time. This is beneficial as the extra time could be used to account for unforeseen circumstances (ie. blockage) or get a head start on the next task. The subject of action is robot-based since the idea of the task is to have the robot place itself in the environment. Lastly, the movement goal is convergence and movement-while since the robot is identifying different objects while navigating to various locations in the environment before converging back to its starting location.

1.1 Contest Requirements

The contest requirements are outlined as follows:

- The contest environment will be 4.87 ✕ 4.87 m^2. For simplicity, there will be no additional objects in the environment other than the five objects to be examined as shown in Figure 1. Each object is represented by a cardboard box of dimensions 50x16x40 cm^3 (l x w x h). The coordinates of these objects, measured with respect to the world coordinate frame of the map, will be provided by the Instructor/TAs on contest day.
- Our team will be provided with: 1) a 2D map of the contest environment generated with gmapping, 2) test locations for the 5 objects within that environment, and 3) the image tags that will be used during the contest.
- An object's location in the environment map is defined by the coordinates of its centre and its orientation (x,y,φ), where φ is about the z-axis, with respect to the origin of the given 2D map.

- The TurtleBot does not have to start at the origin of the world coordinate frame when the map is loaded; however, its pose within the map, just like the objects, will be measured with respect to this origin.
- The object locations are defined by two vectors: 1) a coordinate vector which defines each object's location in x and y, and 2) an orientation vector which contains the object rotation about the z axis. These locations are measured from the object's local frame with respect to the world coordinate frame at the origin of the map.
- The image tags are high-contrast images with many unique features. These tags will be placed in the centre of one of the long faces of the objects
- On contest day, our team will receive a new set of object locations and a map to utilize for the contest.
- During the contest, the TurtleBot will start at a location in the map of the Instructor/TAs choosing. The robot will have a maximum time limit of 5 minutes to traverse to and identify all the objects provided, before returning to its starting location and indicating that it is done.
- Our TurtleBot must utilize the provided navigation library to drive the robot base safely and use the RGB camera on the Kinect sensor to perform SURF feature detection in order to find and identify the image tag on each object.
- There will be three objects in the environment with unique tags, one object with a duplicate tag, and one object without a tag, giving us a total of 5 objects.
- Once the TurtleBot has traversed to all the objects and returned to its starting location, it must output to a file all the tags it has found and at which object location

2.0 Strategy

Although the coordinates of the objects and the 2D map of the environment are provided in advance of the contest trials, the team will not have access to this information until the day of the contest. Therefore, the team has decided to develop a generalized algorithm that allows the robot to localize, calculate the feasible viewing locations for object detection from the given set of coordinates, plan out an efficient path of travel, navigate to the viewing locations, and identify the detected objects. The overall algorithm is sequential as the code is divided into five states that fall under two stages. The first stage involves deliberate planning with minimal reactive movements, which includes the first three states: coordinate calculation, startup, and path planning. The second stage involves reactive actions resulting from the last two states: navigation and object detection.

2.1 Planning Stage

To start the planning stage, the robot will first enter the coordinate calculation state where it calculates a feasible viewing location for each object coordinate given using trigonometry. It will then undergo its localization process in the startup state with the help of Adaptive Monte Carlo Localization (AMCL). Once the robot is localized, the brute force method will be used to solve for the most efficient travel path.

2.1.1 Coordinate Calculation

In the coordinate calculation state, the robot will calculate a feasible object viewing location (in coordinate format) from the object coordinate for each object. There are several requirements that the viewing location must satisfy to be considered feasible:

1. The viewing location must be inside the 4.87 x 4.87 m^2 contest environment and be free of obstacles.
    a. The turtlebot has a radius of 0.18 m so the viewing location must be at least 0.19 m (safety tolerance included) away from the wall and obstacles.
2. The viewing location must provide a clear view of the object for ideal object detection.
    a. The angle of view must not be greater than a 30-degree difference from the object's orientation about the z-axis.
    b. The viewing location must be within a distance of 0.5 m - 0.8 m from the object.

With these requirements set, the steps of coordinate calculation begin with a simple trigonometric manipulation of the given object coordinate. The object coordinate is defined by the coordinates of its centre and its orientation (x, y, φ), where φ is about the z-axis with respect to the origin of the given 2D map. Using the most optimal experimental viewing distance of 0.5 m, the viewing location coordinate can be determined as shown in Figure 2.1.1.

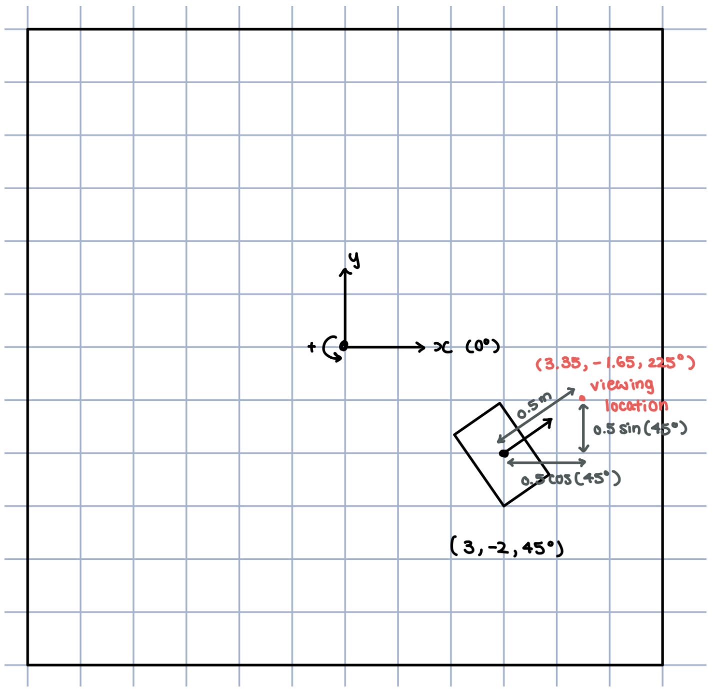

Figure 2.1.1 Example of viewing location coordinate calculation.

If the calculated viewing location satisfies the requirements, the robot will continue on the calculation of the next object. However, if any of the requirements are not satisfied then the robot will calculate a new viewing location by drawing an arc with a radius of 0.5 m - 0.8 m and searching for a feasible location closest to the ideal viewing point on that arc. This process is shown in Figure 2.1.2.

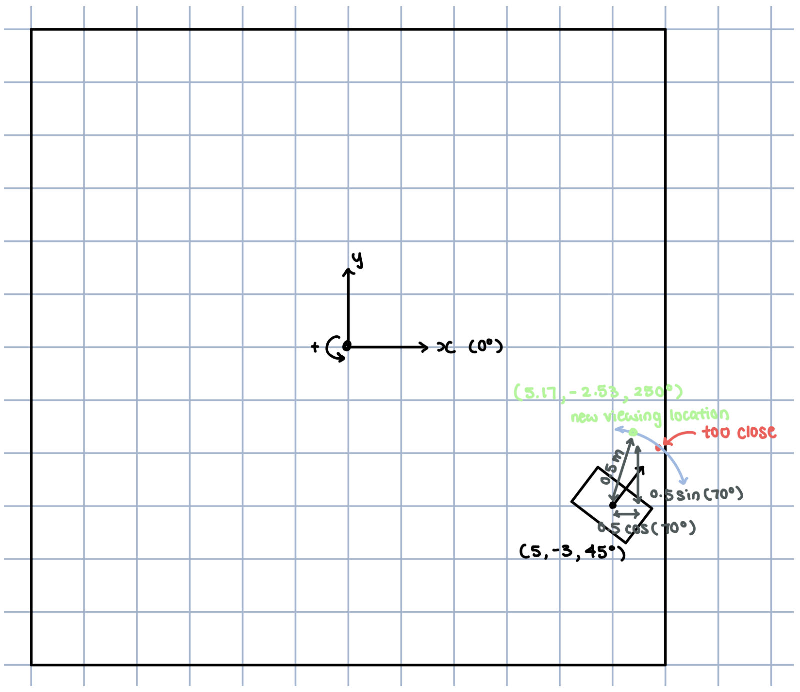

Figure 2.1.2 Recalculated viewing location coordinate.

This process is applied to each object until a feasible viewing location has been determined for all objects.

2.1.2 Startup State

In the startup state, the robot is prompted to scan 180 degrees on the spot to get sensor data of its surroundings. This information is used alongside the 2D Pose Estimate in Rviz by the AMCL particle filter algorithm to estimate the robot's position and orientation with respect to the origin of the 2D map given to localize.

2.1.3 Path Planning State

In the path planning state, the robot will take all the viewing location coordinates and starting coordinates and treat them as part of the Travelling Salesman Problem (TSP). Since there are only 6 nodes in total, the brute force method is used to solve for the shortest path. The cost of each path is represented using Euclidean distance rather than Manhattan distance. This is because the turtlebot can travel diagonally on the map and is not only limited to x and y-axis movements. Once the lowest cost path has been determined, the robot will then move to the action stage.

2.2 Action Stage

Once the planning stage has been completed, the robot will proceed to navigate to each object following the predetermined path from the path planning state. Upon arrival at each object, the robot will use the camera from its Kinect sensor to view the object and extract the words shown as features. These words will then be exported into a text file as the final deliverable. In the action stage, the navigation and object detection states will be alternately called repeatedly until all objects have been identified and the robot returns to its original starting position.

2.2.1 Navigation State

In the navigation state, the robot will use the coordinates provided by the coordinate calculation state to navigate to each object in the order provided by the path planning state. In some cases, the robot's path may interfere with the costmap, causing the robot to get stuck. To prevent such situations, the navigation algorithm adds an intermediate step to the right or left side of the target object, if reachable. This allows the turtlebot to travel around the target object and prevent it from getting stuck in the costmap. With the added clearance, the turtlebot can navigate through the maze with minimal disruption.

2.2.2 Object Detection State

In the object detection state, the robot will use its RGB camera on the Kinect sensor to register the image of the cereal box. This image is then processed using OpenCV and compared to the provided object images through speeded-up robust features (SURF) feature detection. Once the image tag of the object has been identified, this information will be outputted into a text file containing the object coordinate and the cereal box brand.

3.0 Robot Design

3.1 Sensory Design

The sensory design of the robot consists of proprioceptive and exteroceptive sensors. The proprioceptive sensors, which include the internal gyroscope and motor encoders, will be discussed in Section 3.1.1 and the exteroceptive sensors, which include the Kinect sensor and bumpers, will be discussed in Section 3.1.2.

3.1.1 Proprioceptive Sensors

Proprioceptive sensors are used to determine the internal state of the robot. They can help provide internal information about the robot, such as its linear and angular velocities as well as orientation. In our algorithm, we rely on using odometry to track the robot's position and orientation relative to the origin provided. The robot's position is given as an x and y coordinate while its orientation is an angle around the z-axis. When it travels from object to object, odometry helps update this information and collaborates with the localization module to confirm the robot's exact location. The first set of proprioceptive sensors used is the robot's motor encoders. A motor encoder is used to provide information on a rotary motor's speed and position. With the help of motor encoders, the robot's linear velocity can be specified and maintained. This is useful as the velocity can be used to calculate the distance travelled. As well, the navigation code also relies on the motor encoders to control the speed the robot travels in the environment.

The second set of proprioceptive sensors used is the internal gyroscope. With the help of the gyroscope, the robot's orientation can be tracked, which will help the navigation module confirm whether the robot is travelling correctly or if it is in the correct destination position. The navigation module also relies on the gyroscope to specify a certain angle for the robot to turn and determine when the robot is done turning by comparing the yaw of the initial and final positions of the robot. This can ensure that the intended turns for the robot are carried out accurately. As well, the gyroscope is also used to maintain the robot's angular velocity, which is something that the navigation module specifies.

3.1.2 Exteroceptive Sensors

Exteroceptive sensors are used to determine the external state of the robot. They can help provide external information about the environment, such as the layout of surrounding structures and object detection. In our algorithm, we rely on the Kinect depth sensor attached to the robot to understand the surrounding environment and localize itself in the map provided. The Kinetic depth sensors use laser scanning by subscribing to the scan topic and retrieving laser distances. The lasers are produced using an infrared projector and a camera that can see the tiny dots that the projector produces. Using the 3D depth sensor, the robot has a 45-degree vertical field of view, a 58-degree horizontal field of view, and a nominal range of 0.8-3.5 meters, which is useful for localization, object detection, and obstacle avoidance. At startup, the AMCL algorithm localizes by simulating particles based on the sensor readings from the depth sensor. During travel, the navigation module uses sensor feedback to determine if the robot's environment corresponds to the map or if it needs to travel in another direction to avoid an unexpected obstacle. In addition to the depth sensor, the Kinect sensor is also equipped with an RGB camera which allows for object detection and identification.

On top of the Kinect sensor, the robot is equipped with three bumpers located on the left, center, and right sides of the base platform of the robot. Since the map environment is given prior to navigation, the turtlebot is aware of the obstacles, hence bumper sensors are not explicitly used in the scope of contest 2.

3.2 Controller Design

To implement the strategy described in section 2.0, the controller was designed to operate the robot in two different main stages: planning and action (detection). The robot follows a simple sequential controller design, where the stages are executed one after the other. The first stage involves deliberate planning with minimal reactive movements, which includes the first three states: coordinate calculation, startup, and path planning. The second stage involves reactive actions resulting from the last two states: navigation and object detection. In both stages, callback functions and custom functions are used to ensure the individual states within each stage are completed successfully and in their respective orders. The details of the algorithm are explained in the following sections along with a visual representation of the overall control structure in Figure 3.2.1.

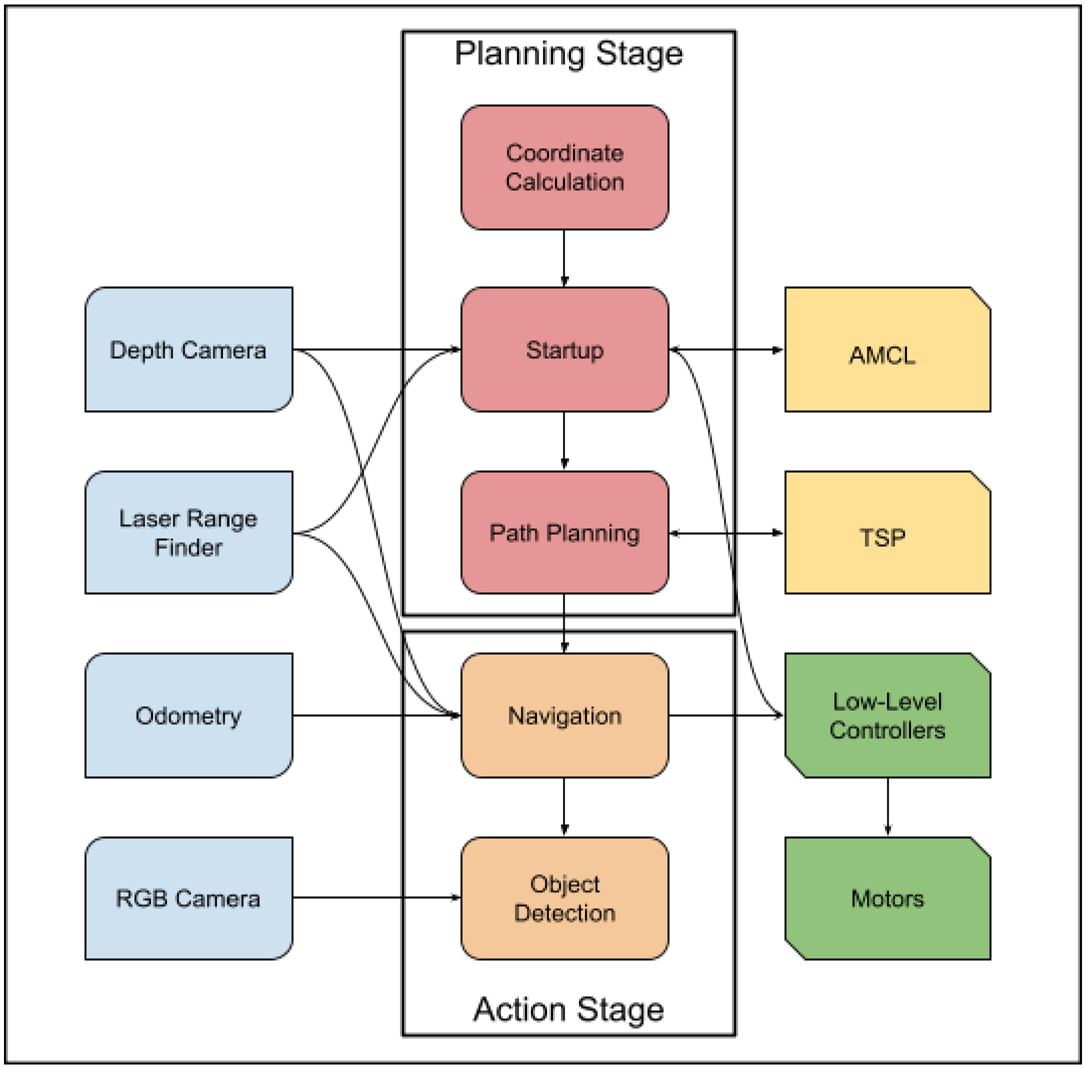

Figure 3.2.1 Overview of controller.

3.2.1 Main Contest Code (contest2.cpp)

This is the main code that initializes ROS by initializing object classes and functions as well as subscribing to all necessary ROS topics. These include classes like *Boxes* and *ImagePipeline*, which provide the templates for boxes and object detection methods, functions like *moveToGoal()* and *getPlan()*, which are navigation functions used to help the robot travel in the environment, and topics like *amcl_pose*, which uses odometry information from the robot to provide an estimation of the robot's position and orientation.

Once initialization has been completed, the coordinate calculation state begins by creating a new class *newBoxes* to store a second set of coordinates that correspond to feasible viewing locations for the objects. This is done by loading each box coordinate and applying the *getPlan()* function. This function first calculates for the most optimal viewing location which is described in Section 2.1.1 and shown in Figure 2.1.1. This location is 0.5 m away from the center of the object, which from experimental results was the most optimal distance to minimize unnecessary background features. Once a location has been calculated, the function will attempt to navigate to the viewing location through simulation. If the path is valid, then the location is stored. Otherwise, the function attempts to recalculate for a new feasible viewing location as described in Section 2.1.1 and shown in Figure 2.1.2.

Following the coordinate calculation state, a while loop is constructed to execute the rest of the states. The while loop is separated into two sections, the planning stage and the action stage, using a conditional statement. When the while loop first begins, the robot will enter the startup state in the planning stage and obtain positional information from AMCL, which will provide a set of coordinates indicating the estimation of the robot's position and orientation. This is done through *ros::spinOnce()* to update information on subscribed topics. The robot is then prompted to turn 180 degrees using the *moveToGoal()* function so it can get more sensory data from its environment and be fully localized.

After the robot is localized, it will then use its starting location as well as the new coordinates stored in *newBoxes* to calculate the shortest path of travel. In this path-planning state, the coordinates will first be sorted into a matrix using the *sortGraph()* function. The *sortGraph()* function calculates the Euclidean distance between each pair of locations as the cost of travel by calling the *euclideanDist()* function and stores the costs in their respective locations in the matrix. Once the matrix has been filled, the brute force method is applied to solve for the minimum cost path using the *travellingSalesmanProblem()* function. The minimum path is stored as the order of travel of the location indices.

Once the shortest path has been determined, the robot is prompted to navigate to each location by using the *Navigation::moveToGoal()* function. By indexing through *newBoxes* using the order stored in the minimum path, the robot is able to travel efficiently. Upon arrival at the viewing location, it executes *ros::spinOnce()* to update sensory information and calls the *ImagePipeline* class (explained in Section 3.2.2). By using the function *imagePipeline.getTemplateID()*, the robot can use SURF to recognize the cereal brand of the object, compare it to the given templates, and return the template ID that gives the closest match. A *box_count* variable is used to store how many objects have been successfully navigated to and identified. Once *box_count* has reached five, the robot will be prompted to navigate back to its initial starting position using the *Navigation::moveToGoal()* function and break out of the while loop. Upon the exit of the while loop, the final output text file is then generated by exporting the box number, cereal brand, and box coordinates while noting any repeats.

3.2.2 Image Pipeline Code (imagePipeline.cpp)

Image pipeline is used to detect and match features of interest of the given objects within a scene. The matching features are transformed and calculated to infer the location of the object in the scene image. In contest two, the objects are the given three cereal logo templates, and the scene is an image taken by the Turtlebot upon arrival at each box location in the Myhal maze. *ImagePipeline::imageCallback()* is first initialized in *contest2.cpp*, in which the Turtlebot subscribes to an image topic, which in this case, is the RGB camera on the Kinetic sensor. This callback function provides the input image.

To determine the template ID of the image, *ImagePipeline::getTemplateID()* takes the input image and identifies the logo through feature detection and matching. Within *ImagePipeline::getTemplateID()*, a SURF detector is created using *SURF::create(minHessian)*, useful in determining the key points of both the object and the scene, which are interest points selected based on the chosen minHessian threshold. Then, a nearest neighbour's keypoint matcher uses the acquired keypoint descriptions and attempts to estimate which key points have the greatest probability of being a match between the object and the scene. This is done based on the distance between the key points and the matches are filtered using the Lowe's ratio test. The best matches are stored in a vector named *good_matches*. Image pipeline also separates key points from *good_matches* into the object and scene, then utilizing *findHomography()*, it locates the transformed object corners and draws a box around where the object is believed to be in the scene image.

Upon receiving the image from the Kinect sensor's RGB camera, it will be iterated three times to compare the input image with the three given cereal logo templates. The logo with the most matches stored inside the *good_matches* vector is determined to be the cereal logo it sees, in which case *ImagePipeline::getTemplateID()* returns a value of 0, 1, or 2. This corresponds to the cereal logo templates provided to match the input image with the correct template number. In this contest, a template ID of 0, 1, and 2 corresponds to raisin bran, cinnamon toast, and rice krispies, respectively. If the box is blank, meaning there is no logo pasted onto the box, *good_matches* will only hold a handful of matching key points. Below a selected threshold of 25 good matches, the box is deemed blank and without an image, in which case *ImagePipeline::getTemplateID()* will return a value of -1. Lastly, each template ID is counted to keep track of repeating IDs to determine whether a repeat has occurred. The same ID number will be returned by *ImagePipeline::getTemplateID()*, but a counter ensures that the repeating ID is noted.

4.0 Future Recommendations

Future recommendations for enhancing the robot's performance are as follows:

- Actual Distance Measurement: Rather than relying solely on Euclidean distances, we would calculate the actual travel distance to the viewing location coordinates by investigating the cost map. The actual distances would then be used in the path planning algorithm to decide a true optimal path.

- Image Processing Enhancements: We could implement an edge detection algorithm to improve the recognition of the object boundaries. This could be potentially better and would need to be compared with SURF's speed and nearest neighbour algorithm. Furthermore, we could consider capturing images from multiple angles. This would allow the robot to have greater certainty that the object being identified is correct by allowing us to take the highest average features of the angles.

- Background and Object Isolation: We could develop algorithms to remove unnecessary background elements from the robot's camera feed. This would involve cropping the images to display areas with features of interest. For example, removing the ceiling and unnecessary regions in the box can greatly improve the accuracy of image detection.

- Confidence Bounds: We could implement a system to calculate confidence bounds for the image tags. This would allow us to have a measure of the certainty of the robot which would be used to decide whether or not to re-examine the object.

- Optimized Path Planning: If more nodes were needed to be visited, we may need to transition to a less computationally expensive path planning than the brute force approach, such as nearest neighbours or genetic algorithms. These methods would offer a decent solution for path optimization with the benefit of reducing computational costs. Additionally, we could parallelize the path-planning process and initiate it through a feedback loop. This would allow for real-time adjustments to the robot's planned path based on any changes that are not mapped on the environment.

5.0 Appendix (Full C++ ROS code)

Appendix A: Contribution Table

Section	Subsection	Henry (Hua Hao) Qi	Harry Park	Alastair Sim	Yi Lian
1.0			✔		✔
2.0		✔	✔		✔
3.0	**3.1**	✔			✔
	3.2	✔		✔	✔
4.0			✔	✔	
Code		✔	✔	✔	✔

Appendix B: *contest2.cpp*

```
#include <boxes.h>
#include <navigation.h>
#include <robot_pose.h>
#include <imagePipeline.h>
#include <chrono>
#include <cmath>

#include <nav_msgs/GetPlan.h>
#include <move_base_msgs/MoveBaseAction.h>
#include <tf/transform_datatypes.h>

#include "path_planning.cpp"

#include <iostream>
#include <fstream>
#include <string>
using namespace std;

float start_x, start_y, start_z, x,y,z;

bool first = true; // check if first iteration or not

//std::string                         outputPath                         =
"/home/tuesday2023/Oogway/catkin_ws/src/mie443_contest2/boxes_database/Con
test_2_Submission_0318.txt";
std::string                           outputPath                         =
"/home/harryp/catkin_ws/src/mie443_contest2/boxes_database/Contest2.txt";

float incAngle = 1, incNorm = 0.1;
float angStart=1, angEnd=45;
float normStart=0.48, normEnd=0.8, normSide=0.7;
float checkRight=M_PI/6, checkLeft=-M_PI/6;
float deltAngle=angStart, normDist=normStart;

bool  getPlan(float  xStart,  float  yStart,  float  phiStart,  float  xGoal,
float yGoal, float phiGoal){
   // Set up and wait for actionClient.
   // Initialize node
   ros::NodeHandle n;
                        ros::ServiceClient         check_path          =
n.serviceClient<nav_msgs::GetPlan>("/move_base/NavfnROS/make_plan");
   nav_msgs::GetPlan srv;

   // Set start
                        geometry_msgs::Quaternion        phistart          =
tf::createQuaternionMsgFromYaw(phiStart);
   srv.request.start.header.frame_id = "map";
   srv.request.start.pose.position.x = xStart;
   srv.request.start.pose.position.y = yStart;
   srv.request.start.pose.position.z = 0.0;
   srv.request.start.pose.orientation.x = 0;
   srv.request.start.pose.orientation.y = 0;
   srv.request.start.pose.orientation.z = phistart.z;
   srv.request.start.pose.orientation.w = phistart.w;

                        geometry_msgs::Quaternion        phigoal          =
tf::createQuaternionMsgFromYaw(phiGoal);
   srv.request.goal.header.frame_id = "map";
   srv.request.goal.pose.position.x = xGoal;
   srv.request.goal.pose.position.y = yGoal;
   srv.request.goal.pose.position.z = 0.0;
   srv.request.goal.pose.orientation.x = 0.0;
   srv.request.goal.pose.orientation.y = 0.0;
   srv.request.goal.pose.orientation.z = phigoal.z;
   srv.request.goal.pose.orientation.w = phigoal.w;
   srv.request.tolerance = 0.1;

   check_path.call(srv);



   return srv.response.plan.poses.size()>0;
}

bool results;
string getFinalOutput(int id);
int rb_repeat=0, ct_repeat=0, rk_repeat=0;
```

```
// Converts template_id values into chosen template
string getFinalOutput(int id) {

   if (rb_repeat>0 && id == 0) {
       return "(REPEAT) RAISIN_BRAN";
   } else if (ct_repeat>0 && id == 1) {
       return "(REPEAT) CINNAMON_TOAST";
   } else if (rk_repeat>0 && id == 2) {
       return "(REPEAT) RICE_KRISPIES";
   } else if (rb_repeat==0 && id == 0) {
       rb_repeat++;
       return "RAISIN_BRAN";
   } else if (ct_repeat==0 && id == 1) {
       ct_repeat++;
       return "CINNAMON_TOAST";
   } else if (rk_repeat==0 && id == 2) {
       rk_repeat++;
       return "RICE_KRISPIES";
   } else if (id == -1) {
       return "BOX IS BLANK!";
   } else {
       return "unidentified...";
   }

}


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);
   ros::spinOnce();

   // Setup output text file
   std::ofstream contest2_file(outputPath);
   // 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] << std::endl;
   }


   // Initialize image objectand subscriber.
   ImagePipeline imagePipeline(n);
   int final_output[5];    // stores template_id of each box in order
   array<string,5> template_names = {"N/A", "N/A", "N/A", "N/A", "N/A"};

   // Initialize path vector
   std::vector<int> min_path;

   // contest count down timer
   std::chrono::time_point<std::chrono::system_clock> start;
   start = std::chrono::system_clock::now();
   uint64_t secondsElapsed = 0;
   int box_count = 0;


     // Initialize second set of box coordinates to check for valid
coordinates
   Boxes newBoxes;
   if (!newBoxes.load_coords() || !boxes.load_templates())
   {
         std::cout << "ERROR: could not load new coords or templates"<<
std::endl;
       return -1;
```

```
    }
    // Calculate valid coordinates
    for (int i=0; i < newBoxes.coords.size(); i++)
    {
        ROS_INFO("Starting Box coordinate %d", i);
        z = newBoxes.coords[i][2] - M_PI;
                                    x = newBoxes.coords[i][0] +
normDist*std::cos(newBoxes.coords[i][2]);
                                    y = newBoxes.coords[i][1] +
normDist*std::sin(newBoxes.coords[i][2]);


        while (!getPlan(robotPose.x, robotPose.y, robotPose.phi, x,y,z))
        //while (true)
        {
            ros::spinOnce();

            z = newBoxes.coords[i][2]+deltAngle/180*M_PI - M_PI;
                                        x = newBoxes.coords[i][0] +
normDist*std::cos(newBoxes.coords[i][2]+deltAngle/180*M_PI);
                                        y = newBoxes.coords[i][1] +
normDist*std::sin(newBoxes.coords[i][2]+deltAngle/180*M_PI);

            if (getPlan(robotPose.x, robotPose.y, robotPose.phi, x,y,z))
            {
                    ROS_INFO("Valid Path! Updating coordinates to %f, %f,
%f...", x,y,z);
                 ROS_INFO("Delta angle is: %f and normal distance is: %f",
deltAngle, normDist);
                //break;
            }
            else
            {
                      ROS_INFO("Invalid Path to %f, %f, %f, not updating
coordinates", x,y,z);
                 ROS_INFO("Delta angle is: %f and normal distance is: %f",
deltAngle, normDist);
            }
```

```
              // break statement when delta angles exceeds 15 and normal
exceeds 0.8
           if (deltAngle > angEnd)
           {
               normDist += incNorm;
               deltAngle = angStart;
               ROS_INFO("Updating normal distance");

           }

           // increment delta angle
           else
           {
                                  if  (int(abs(deltAngle))%2==1)  deltAngle  =
(abs(deltAngle)+incAngle)*(-1); //odd, positive deltangle
                         else deltAngle = (abs(deltAngle)+incAngle); //even,
negative deltangle
           }

           if (deltAngle >angEnd && normDist >normEnd)
           {
               deltAngle = angStart;
               normDist = normStart;
               break;
           }

       }

       // Update boxes.coords with valid poses
       newBoxes.coords[i][0] = x;
       newBoxes.coords[i][1] = y;
       newBoxes.coords[i][2] = z;
   }

   // print updated box coordinates
   for(int i = 0; i < newBoxes.coords.size(); ++i) {
       std::cout << "Updated Box coordinates: " << std::endl;
         std::cout << i << " x: " << newBoxes.coords[i][0] << " y: " <<
newBoxes.coords[i][1] << " z: "
               << newBoxes.coords[i][2] << std::endl;
```

```
    }




    // Execute strategy.
    while(ros::ok() && secondsElapsed <= 300) {

        ros::spinOnce();
        ros::Duration(0.1).sleep();


        // Use: boxes.coords
        // Use: robotPose.x, robotPose.y, robotPose.phi

        if (first){
            ros::spinOnce();
            ros::Duration(0.1).sleep();
            ros::spinOnce();
            ros::Duration(0.1).sleep();
            ros::spinOnce();
            ros::Duration(0.1).sleep();
            first = false;
            ROS_INFO("First Iteration!");
            start_x = robotPose.x;
            start_y = robotPose.y;
            start_z = robotPose.phi;
                  ROS_INFO("Starting Coordinates: (%f, %f, %f)", start_x,
start_y, start_z);

            ROS_INFO("Localizing...");
                       ROS_INFO("At: %f,%f,%f", robotPose.x, robotPose.y,
robotPose.phi);
            Navigation::moveToGoal(start_x, start_y, start_z+ M_PI);

            ros::spinOnce();

            start_x = robotPose.x;
            start_y = robotPose.y;
            start_z = robotPose.phi- M_PI;
            ROS_INFO("Finished Localizing");
             ROS_INFO("Actual Starting Coordinates: (%f, %f, %f)", start_x,
start_y, start_z);

            //Calculate shortest path
            ROS_INFO("Calculating Shortest path");
            std::vector<float> temp_vec = {start_x, start_y, start_z};
            std::vector<std::vector<float>> view_coords = newBoxes.coords;
            view_coords.push_back(temp_vec);
                                std::vector<vector<float>>   sorted_graph   =
sortGraph(view_coords);
            int start_node = view_coords.size() - 1;
            min_path = travellingSalesmanProblem(sorted_graph, start_node);
            std::cout << "MIN PATH SIZE:" << min_path.size() << std::endl;

        }

        else if (box_count == 5)
        {
                        ROS_INFO("At:  %f,%f,%f",  robotPose.x,  robotPose.y,
robotPose.phi);
            ROS_INFO("GOING to: %f,%f,%f", start_x, start_y, start_z);

            Navigation::moveToGoal(start_x, start_y, start_z);
                                                              secondsElapsed     =
std::chrono::duration_cast<std::chrono::seconds>(std::chrono::system_clock
::now()-start).count();

            std::cout << "Finished in:" << std:: endl;
            std::cout << secondsElapsed << std::endl;
            break;
        }

        else
        {

            // replace box_count with min_path[box_count] from here on
```

```
           // If available, check and create waypoint at right side of box
           normDist = normSide;
           deltAngle = M_PI/2;
           z = boxes.coords[min_path[box_count]][2];
                            x   =   boxes.coords[min_path[box_count]][0]   +
normDist*std::cos(boxes.coords[min_path[box_count]][2]+deltAngle);
                            y   =   boxes.coords[min_path[box_count]][1]   +
normDist*std::sin(boxes.coords[min_path[box_count]][2]+deltAngle);

           ROS_INFO("Checking right side of box...");
           if (getPlan(robotPose.x, robotPose.y, robotPose.phi, x, y, z))
           {
               ROS_INFO("Right side available!");
               Navigation::moveToGoal(x,y,z);
               ROS_INFO("Reached right side");
           }

           else // If right side not available, check left side of box
           {
                 ROS_INFO("Right side not available, checking left side of
box...");
               deltAngle = -M_PI/2;
               z = boxes.coords[min_path[box_count]][2];
                             x  =  boxes.coords[min_path[box_count]][0]  +
normDist*std::cos(boxes.coords[min_path[box_count]][2]+deltAngle);
                             y  =  boxes.coords[min_path[box_count]][1]  +
normDist*std::sin(boxes.coords[min_path[box_count]][2]+deltAngle);
                if (getPlan(robotPose.x, robotPose.y, robotPose.phi, x, y,
z))
               {
                   ROS_INFO("Left side available!");
                   Navigation::moveToGoal(x,y,z);
                   ROS_INFO("Reached left side");
               }
               else
               {
                   ROS_INFO("Left side not available either");
               }
           }
```

```
          ros::spinOnce();
          ROS_INFO("Heading to normal of box...");

          z = newBoxes.coords[min_path[box_count]][2];
          x = newBoxes.coords[min_path[box_count]][0];
          y = newBoxes.coords[min_path[box_count]][1];

                    ROS_INFO("At:  %f,%f,%f",  robotPose.x,  robotPose.y,
robotPose.phi);
          ROS_INFO("GOING to: %f,%f,%f", x, y, z);

          if (Navigation::moveToGoal(x,y,z))
          {
              // image pipeline
              ros::spinOnce();
                                  final_output[min_path[box_count]]   =
imagePipeline.getTemplateID(boxes);

              box_count++;

          }
      }



                                              secondsElapsed        =
std::chrono::duration_cast<std::chrono::seconds>(std::chrono::system_clock
::now()-start).count();
      std::cout << "Time Elapsed:" << std:: endl;
      std::cout << secondsElapsed << std::endl;
      ros::Duration(0.01).sleep();

   }
   // make text file of final output (coords and template_id
   std::cout<< "Final output 1: " << final_output[0] << std::endl;
   std::cout<< "Final output 2: " << final_output[1] << std::endl;
   std::cout<< "Final output 3: " << final_output[2] << std::endl;
   std::cout<< "Final output 4: " << final_output[3] << std::endl;
   std::cout<< "Final output 5: " << final_output[4] << std::endl;

   for (int i=0;i<5;i++) {
      template_names[i] = getFinalOutput(final_output[i]);

      // Checks for repeats by keeping count
      if (final_output[i]==0) {
          rb_repeat++;
      } else if (final_output[i]==1) {
          ct_repeat++;
      } else if (final_output[i]==2) {
          rk_repeat++;
      }
   }

     contest2_file  <<  "Names:  Henry,  Harry,  Cherry,  Alastair\n"  <<
std::endl;
   for (int i=0;i<5;i++) {
      contest2_file << "Box "<< i+1 << "Tag: " << template_names[i] <<
std::endl;
      contest2_file << "Coordinates (x,y, phi): (" << boxes.coords[i][0]
<< ", " << boxes.coords[i][1] << ", " << boxes.coords[i][2] << ") \n" <<
std::endl;
   }

   //contest2_file.close();


   return 0;
}

// Edit costmap parameters

//roscd turtlebot_navigation/param
//sudo gedit costmap_common_params.yaml
```

Appendix C: *boxes.cpp*

```
#include <ros/package.h>
#include <boxes.h>

bool Boxes::load_coords() {
   std::string filePath = ros::package::getPath("mie443_contest2") +

std::string("/boxes_database/gazebo_coords.xml");
   cv::FileStorage fs(filePath, cv::FileStorage::READ);
   if(fs.isOpened()) {
       cv::FileNode node;
       cv::FileNodeIterator it, end;
       std::vector<float> coordVec;
             std::string  coords_xml[5]  =  {"coordinate1",  "coordinate2",
"coordinate3", "coordinate4",
                                    "coordinate5"};
       for(int i = 0; i < 5; ++i) {
           node = fs[coords_xml[i]];
           if(node.type() != cv::FileNode::SEQ) {
               std::cout << "XML ERROR: Data in " << coords_xml[i]
                           << " is improperly formatted - check input.xml"
<< std::endl;
           } else {
               it = node.begin();
               end = node.end();
               coordVec = std::vector<float>();
               for(int j = 0; it != end; ++it, ++j) {
                   coordVec.push_back((float)*it);
               }
               if(coordVec.size() == 3) {
                   coords.push_back(coordVec);
               } else {
                   std::cout << "XML ERROR: Data in " << coords_xml[i]
                                     << " is improperly formatted - check
input.xml" << std::endl;
               }
           }
       }
       if(coords.size() == 0) {
```

```
                 std::cout  <<  "XML  ERROR:  Coordinate  data  is  improperly
formatted - check input.xml"
                 << std::endl;
           return false;
       }
   } else {
        std::cout << "Could not open XML - check FilePath in " << filePath
<< std::endl;
       return false;
   }
   return true;
}

bool Boxes::load_templates() {
   std::string filePath = ros::package::getPath("mie443_contest2") +
                          std::string("/boxes_database/templates.xml");
   cv::FileStorage fs(filePath, cv::FileStorage::READ);
   if(fs.isOpened()) {
       cv::FileNode node = fs["templates"];;
       cv::FileNodeIterator it, end;
               if(!(node.type()  ==  cv::FileNode::SEQ  ||  node.type()  ==
cv::FileNode::STRING)) {
            std::cout << "XML ERROR: Image data is improperly formatted in
" << filePath
                     << std::endl;
           return false;
       }
       it = node.begin();
       end = node.end();
       std::string imagepath;
       for(; it != end; ++it){
           imagepath = ros::package::getPath("mie443_contest2") +
                       std::string("/boxes_database/") +
                       std::string(*it);
                                 templates.push_back(cv::imread(imagepath,
CV_LOAD_IMAGE_GRAYSCALE));
       }
   } else {
       std::cout << "XML ERROR: Could not open " << filePath << std::endl;
       return false;
```

```
    }
    return true;
}
```

Appendix D: *imagePipeline.cpp*

```
#include <imagePipeline.h>

// Henry added
#include <iostream>
#include "opencv2/core.hpp"
//#ifdef HAVE_OPENCV_XFEATURES2D
#include "opencv2/calib3d.hpp"
#include "opencv2/highgui.hpp"
#include "opencv2/imgproc.hpp"
#include "opencv2/features2d.hpp"
#include "opencv2/xfeatures2d.hpp"

using namespace cv;
using namespace cv::xfeatures2d;
using std::cout;
using std::endl;

// Keep track of repeats
int rb_count=0, ct_count=0, rk_count=0;

//input1: box -> template, input2: boxinscene -> img
const char* keys =
       "{ help h |                           | Print help message. }"
       "{ input1 | ../data/box.png           | Path to input image 1. }"
       "{ input2 | ../data/box_in_scene.png | Path to input image 2. }";

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

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();
       // imshow("test",img);
       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) {
   int template_id = -1;
   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 {
       /***YOUR CODE HERE***/

       // Initialize variables
       Mat img_object;
       Mat img_scene;
       //std::vector<KeyPoint> keypoints_object, keypoints_scene;
                         std::vector<KeyPoint>    keypoints_object_chosen,
keypoints_scene_chosen;
       Mat descriptors_object, descriptors_scene;

       int prev_good_matches=0;

       //std::vector<DMatch> good_matches;
       std::vector<DMatch> chosen_matches;
       int matches_threshold = 25;

          // Iterate 3 times through all the templates, initialize some
variables
       for (int counter=0;counter<3;counter++) {
           std::cout << "--template number "<< counter << std::endl;

            img_object = boxes.templates[counter];             // need to
cycle through all the templates?
           img_scene = img;
           // Show the image it sees on the box
           // imshow("cereal", img);
           cv::waitKey(10);
           if ( img_object.empty() || img_scene.empty() )
           {
               cout << "Could not open or find the image!\n" << endl;
               //parser.printMessage();
               return -1;
           }

            //-- Step 1: Detect the keypoints using SURF Detector, compute
the descriptors
           int minHessian = 600;
           Ptr<SURF> detector = SURF::create( minHessian );
           std::vector<KeyPoint> keypoints_object, keypoints_scene;
           // Mat descriptors_object, descriptors_scene;
                       detector->detectAndCompute( img_object, noArray(),
keypoints_object, 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_object, descriptors_scene,
knn_matches, 5 );
```

```
            //-- Filter quality of matches using the Lowe's ratio test
            const float ratio_thresh = 0.70f;
            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]);
                }
            }
                std::cout << "----matches size: "<< good_matches.size() <<
std::endl;

                 //  Template  with  most  matches  gets  id  saved,  important
variables saved
            if (good_matches.size() >= prev_good_matches) {
                prev_good_matches = good_matches.size();
                template_id = counter;
                chosen_matches = good_matches;
                keypoints_object_chosen=keypoints_object;
                keypoints_scene_chosen=keypoints_scene;
            }
        }

        if (chosen_matches.size() < matches_threshold) {
            std::cout << "----Box is BLANK!!---- " << std::endl;
            return -1;
        }

        std::cout << "----template id chosen: "<< template_id << std::endl;
        img_object = boxes.templates[template_id];
        //-- Draw matches - good keypoint matches visualized between object
and scene
        Mat img_matches;
             drawMatches(  img_object,  keypoints_object_chosen,  img_scene,
keypoints_scene_chosen, chosen_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( size_t i = 0; i < chosen_matches.size(); i++ )
        {
               //-- Get the keypoints from the good matches - separate into
obj and scene
                                      obj.push_back(   keypoints_object_chosen[
chosen_matches[i].queryIdx ].pt );
                                     scene.push_back(   keypoints_scene_chosen[
chosen_matches[i].trainIdx ].pt );
        }
        Mat H = findHomography( obj, scene, RANSAC );

        // -- Harry edit
        if (H.empty())
        {
            ROS_INFO("Empty object! Assuming this block is empty");
            return -1;
        }

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

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

       //-- Show detected matches
       // imshow("Good Matches & Object Detection", img_matches );
       // cv::waitKey(1000);
       //return 0;

       if (rb_count>0 && template_id==0) {
                  imshow("(REPEAT)  RAISIN  BRAN  -  Good  Matches  &  Object
Detection", img_matches );
           cv::waitKey(1000);

       } else if (ct_count>0 && template_id==1) {
                imshow("(REPEAT)  CINNAMON  TOAST  -  Good  Matches  &  Object
Detection", img_matches );
           cv::waitKey(1000);

       } else if (rk_count>0 && template_id==2) {
                 imshow("(REPEAT)  RICE  KRISPIES  -  Good  Matches  &  Object
Detection", img_matches );
           cv::waitKey(1000);

       } else if (template_id==0) {
                imshow("RAISIN  BRAN  -  Good  Matches  &  Object  Detection",
img_matches );

           cv::waitKey(1000);
           rb_count++;

       } else if (template_id==1) {
               imshow("CINNAMON TOAST - Good Matches & Object Detection", 
img_matches );
           cv::waitKey(1000);
           ct_count++;

       } else if (template_id==2) {
               imshow("RICE KRISPIES - Good Matches & Object Detection", 
img_matches );
           cv::waitKey(1000);
           rk_count++;

       }

       // Use: boxes.templates
       //cv::imshow("view", img);
       cv::waitKey(10);

   }
   return template_id;
}
```

Appendix E: *navigation.cpp*

```
#include <navigation.h>
#include <actionlib/client/simple_action_client.h>
#include <move_base_msgs/MoveBaseAction.h>
#include <tf/transform_datatypes.h>
#include <nav_msgs/GetPlan.h>

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

Appendix F: *path_planning.cpp*

```
#include <iostream>
#include <math.h>
#include <chrono>
#include <bits/stdc++.h>
using namespace std;

// cost function: Euclidean Distance

float euclideanDist(vector<float> object1, vector<float> object2) {
   float x = object1[0] - object2[0];
   float y = object1[1] - object2[1];
   float dist;
   dist = pow(x, 2) + pow(y, 2);
   dist = sqrt(dist);
   return dist;
}

// path solver: brute force

vector<int> travellingSalesmanProblem(vector<vector<float>> graph, int s)
{

   // put all nodes other than start into a vector
   vector<int> vertex;
   for (int i = 0; i < graph.size(); i++)
       if (i != s)
           vertex.push_back(i);
   // save minimum path
   float min_path_cost = INT_MAX;
   vector<int> min_path = vertex;
   do {
       float current_pathweight = 0;
       int k = s;
       for (int i = 0; i < vertex.size(); i++) {
           current_pathweight += graph[k][vertex[i]];
           k = vertex[i];
       }
       current_pathweight += graph[k][s];
```

```
        if (current_pathweight < min_path_cost){
            min_path.clear();
            for (auto i: vertex)
                min_path.push_back(i);
        }

        // update minimum
        min_path_cost = min(min_path_cost, current_pathweight);

        // for (auto i: vertex)
        //     cout << i << ' ';
        // cout << "\n";
        // cout << current_pathweight;
        // cout << "\n";
    } while (
        next_permutation(vertex.begin(), vertex.end()));

    return min_path;
}

// sort euclidean distances into graph

vector<vector<float>> sortGraph(vector<vector<float>> objects) {
    int objects_size = objects.size();
    vector<vector<float>> graph;

    for (int row = 0; row < objects_size; row++) {
        vector<float> temp_row;
        for (int col = 0; col < objects_size; col++) {
            temp_row.push_back(euclideanDist(objects[row], objects[col]));
        }
        graph.push_back(temp_row);
    }

    return graph;

}
```

Appendix G: *robot_pose.cpp*

```
#include <robot_pose.h>
#include <tf/transform_datatypes.h>

RobotPose::RobotPose(float x, float y, float phi) {
   this->x = x;
   this->y = y;
   this->phi = phi;
}

void                                              RobotPose::poseCallback(const
geometry_msgs::PoseWithCovarianceStamped& msg) {
   phi = tf::getYaw(msg.pose.pose.orientation);
   x = msg.pose.pose.position.x;
   y = msg.pose.pose.position.y;
}
```

Appendix H: *webcam_publisher.cpp*

```
#include <ros/ros.h>
#include <image_transport/image_transport.h>
#include <opencv2/highgui/highgui.hpp>
#include <cv_bridge/cv_bridge.h>
#include <sstream> // for converting the command line parameter to integer

int main(int argc, char** argv)
{
 // Check if video source has been passed as a parameter
 if(argv[1] == NULL){
   std::cout << "****No Camera Selected****" << std::endl << "Input camera
number to use, ie. 0 for default laptop camera." << std::endl;
   return 1;
 }

 ros::init(argc, argv, "image_publisher");
 ros::NodeHandle nh;
 image_transport::ImageTransport it(nh);
 image_transport::Publisher pub = it.advertise("camera/image", 1);

  // Convert the passed as command line parameter index for the video
device to an integer
 std::istringstream video_sourceCmd(argv[1]);
 int video_source;
 // Check if it is indeed a number
 if(!(video_sourceCmd >> video_source)) return 1;

 cv::VideoCapture cap(video_source);
 // Check if video device can be opened with the given index
 if(!cap.isOpened()) return 1;
 cv::Mat frame;
 sensor_msgs::ImagePtr msg;

 ros::Rate loop_rate(30);
 while (nh.ok()) {
   cap >> frame;
   // Check if grabbed frame is actually full with some content
   if(!frame.empty()) {
```

```
                msg    =    cv_bridge::CvImage(std_msgs::Header(),    "bgr8",
frame).toImageMsg();
     pub.publish(msg);
     cv::waitKey(1);
   }

   ros::spinOnce();
   loop_rate.sleep();
 }
}
```