University of Toronto

MIE443 Contest 2:

Finding Objects of Interest in an Environment

Group 4 Members:

Mariem Ahmed

Katrina Cecco

Kevin Han

Anh Kiet Nguyen

Wenhan Jiang

Date of Submission: March 23rd, 2021

Table of Contents

1. Problem Definition

1.1. Objective

The main objective of Contest 2 is to develop an algorithm that explores a specified environment and identifies objects of interest. The robot should be able to start at a random starting position, optimize and navigate the path to all ten objects of interest, recognize images on the objects that match the template images that are provided prior to the contest, and return to its starting position. Each image has a tag attached to it for recognizing the objects. The output will be a text file that contains the results of the robot's visits and identifications of all objects. Any objects with blank images or duplicate images will be noted in the output as well. Our team will implement an algorithmic design for the TurtleBot to efficiently navigate the 6m by 6m simulated environment and travel to specified locations in the map where it can identify and save each object's tag, location, and uniqueness all within 8 minutes.

1.2. Requirements and Constraints

- The object locations and robot poses are defined by a coordinate vector that defines three components shown below in Figure 1: x, y, and phi. It is an orientation vector that represents the object rotation about the z-axis as phi in radians. These locations are measured from the object's local frame (Figure 1) with respect to the world coordinate frame (Figure 2) at the origin of the map where the robot is initially spawned. Every object dimension is the same with 50cm by 32cm by 40cm in length, width, and height [1]. The object's coordinate provided will be in reference to the centre of the rectangular side that contains the image.

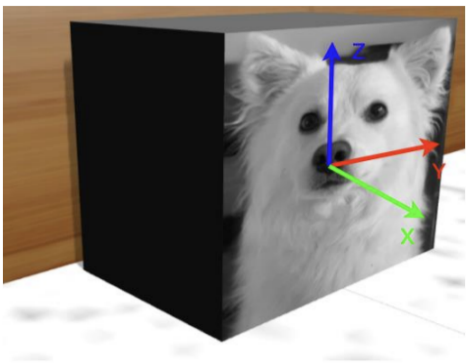

Figure 1: Object Coordinate Frame [1]

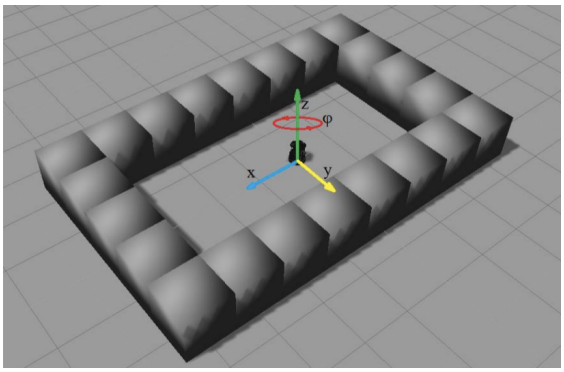

Figure 2: World Coordinate Frame [1]

- Our team must ensure that our code is robust enough to handle dynamic changes to the environment and the locations of the objects.

- Our team must optimize the robot's path to find the ten objects and identify them before returning to their starting location within the maximum time limit of 8 minutes.
- The TurtleBot must utilize the provided navigation library to move the robot around the map safely in its optimized path and use the camera input from the Kinect sensor to perform SURF feature detection in order to identify the tag on each object image.
- The TurtleBot must output to a text file that contains the order of visits, the tags it has found and their coordinate locations, and indicate whether there were any blank or duplicate image tags discovered.

2. Strategy

The strategy for this contest is initiated with the idea that the Turtlebot can first automatically adjust the base of the robot to locate itself inside the map before it is able to locate itself using laser readings. The robot localizes itself in the Gazebo setting in simulation by calling `spinOnce()` (found in `contest2.cpp` template) at the start to localize itself and recording the initial robot pose from the Adaptive Monte Carlo Localization (AMCL) package. This is given the assumption that upon launch, the AMCL is converged manually in RViz using 2D Pose Estimate and 2D Nav Goal functions to localize the robot.

For objects' locations, we are given the coordinates using the `boxes.coords`. For this Travelling Salesman Problem (TSP) of 11 vertices (ten objects + starting position), our team decided to utilize the brute-force approach from Lecture 4 [2] which computed the shortest path for our map in less than a second. We were aware of more efficient algorithms (further discussed in Section 4); but with the guaranteed optimal path and quick run time, we decided that this optimization strategy was perfectly suitable for the purposes of traversing across the map with ten objects. By setting up the distance as the weight, we represented all possible permutations in the matrix to compute the minimum-weight Hamilton circuit in the `optimizePath()` function. Then, our team used the `moveToGoal()` function (provided in the `Navigation.cpp` template) to move to the coordinate of interest in the order determined by path distance optimization.

For object identification, our team found from testing that the robot should ideally be 0.45m away from the centre of the object while facing straight ahead towards the image to ensure a high matching accuracy. However, there were instances where the ideal robot coordinate was unreachable due to it being out of bounds or obstructed by the wall or other objects. So for those situations, we implemented a strategy to allow the robot to adjust its desired coordinates to navigate towards. Our team made use of the `move_base/make_plan` service [3] to determine whether the goal location for each object is reachable (path can be planned) before

starting the navigation. If a valid plan cannot be made for an object, we utilize the `adjustGoal()` and `checkGoal()` functions to discover a new goal location with an angle offset. By checking every 5° incremental position from the original straight-ahead position, we sweep both the counterclockwise and clockwise direction up to 75° each until a valid plan can be made. Once the robot navigates to either the initial or adjusted reachable coordinate, the camera takes in the scene image from the environment and utilizes SURF feature matching and homography [4] to match the template tag images with the input image. In order to accurately identify the best match, our strategy involves looking at two key criteria: the number of key points and the homography projective box area formed on the scene image. Through hundreds of testing runs, we have decided that a certain threshold number of key points should be matched for a template image to be considered as a match. The area of the homography box is also set to have an expected range, where the robot can filter out falsely matching images that are in close proximity of the object at hand or appear far away in the background. The thresholds will be discussed in detail in Section 3.2. Within these two bounding factors, the template image that results in the maximum number of keypoint matches is determined as the tag of the object. If no template image passes these criteria, then the image is determined to be blank and the appropriate tag is saved. In each visit to the objects, the resulting information is stored in vectors and written in the output `Results.txt` file with the proper tag, location, visit order and existence of duplicates as shown in the `main()` function. The robot returns to its starting position after traversing all images and identifying the labels of each tag image while following the optimal path found.

3. Robot Design and Implementation

In Contest 2, the hardware of the robot is already provided. This contest focuses on integrating with existing hardware and developing a controller to successfully complete the objectives for this project. This section will address the use of particular hardware and controller architecture for sensing, navigation, and mapping.

3.1. Sensory Design

The TurtleBot used in this course is divided into 2 main components: the Kobuki mobile base and the Simulated Microsoft Kinect 360. Each of these two contains multiple sensors. The Kobuki base includes 2x wheel drop sensors, 3x front bumpers, odometry, a gyroscope and 3 Cliff sensors, while the Kinect 360 is composed of an RGB camera and a depth sensor, shown in Figure 3. In Contest 2, the key sensors are the gyroscope, odometry, and Kinect RGB camera, which will be used for the localization, navigation, and image identification of the TurtleBot.

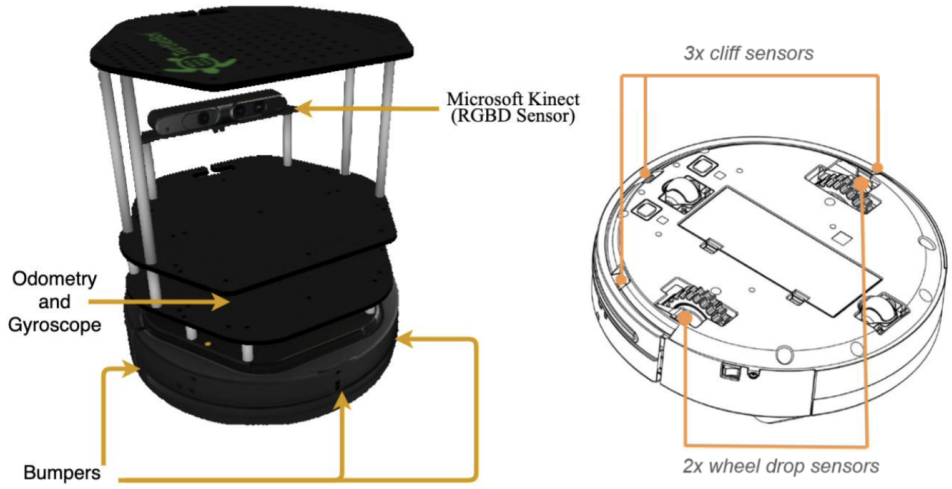

Figure 3: TurtleBot Hardware Components Overview [5]

3.1.1. Odometry

The built-in gyroscope and wheel encoders are used to determine the robot's location and orientation information relative to its initial position. The robot initially determines a proper coordinate framework based at the start of each run and the pose information is accessed by the Adaptive Monte Carlo Localization (AMCL) algorithm which utilizes a particle filter to localize the robot pose within a map in real-time. The odometry sensor will allow the simulated robot's pose (x, y, phi) values to be published in order to update the orientation and enable it to move precisely to coordinates while avoiding any obstacles, as shown in Figure 4.

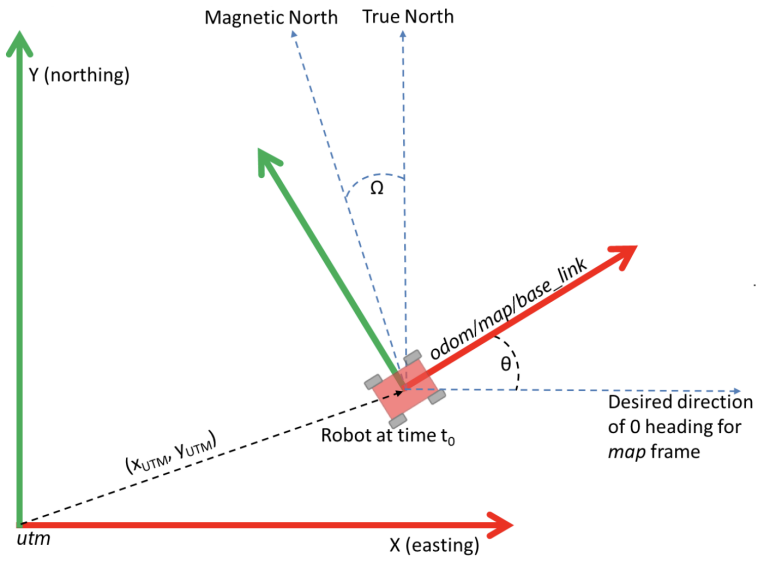

Figure 4: Overview of the Odometry: Position and Orientation of the Robot [6]

3.1.2. Kinect Sensor

The Kinect sensor will be used to locate the scene and capture images on each of the objects. The Kinect Sensor includes IR Emitter, Color Sensor and IR Depth Sensor, shown in Figure 5.

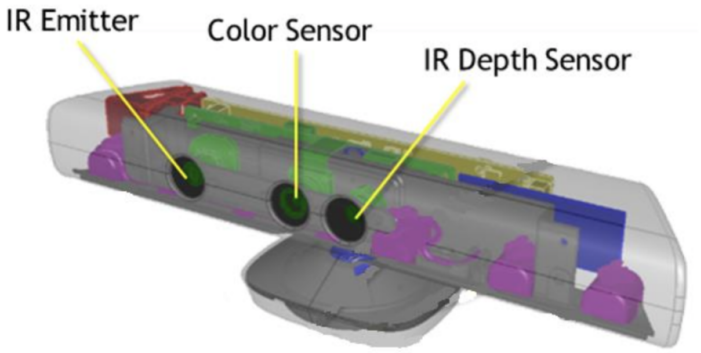

Figure 5: Microsoft Kinect 360 sensors [7]

The TurtleBot is equipped to access the images on the objects with the RGB Camera on Microsoft Kinect 360. The images are retrieved from the image topic of the Kinect sensor in the `imagePipeline.cpp` file and it will allow the algorithm to apply SURF feature matching and homography between the template and scene images. This will be discussed further in Section 3.2.2.

3.2. Controller Design

3.2.1. Navigation

The provided `Navigation::moveToGoal` is called to drive the robot to desired coordinates determined by our algorithm. The function is used to follow the path plan and controls the movement from one location to the next. It is integrated with the navigation package with the AMCL particle filter localization for obstacle avoidance. It utilizes the sensory input from the TurtleBot and the surrounding map environment to detect obstacles in its direct path and discovers a clear path to go around the obstacle safely. Figure 6 demonstrates the overall movement of the entire navigation algorithm process implemented in `contest2.cpp`.

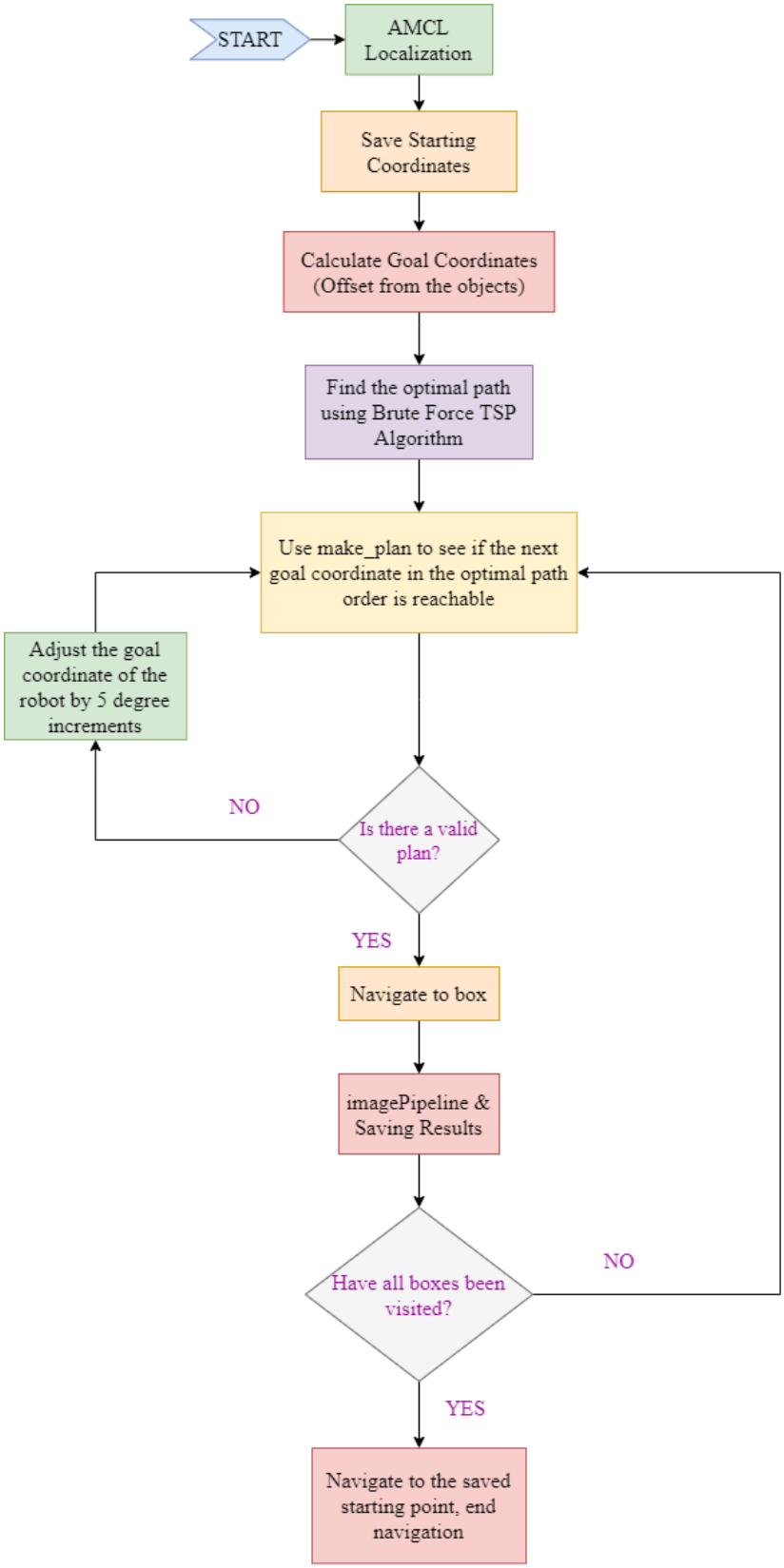

Figure 6: Overview of the Navigation Flow Chart

Firstly, our team will start by locating our starting point. This is achieved by the `spinOnce()` which is called while the robot remains static. The initial pose (x, y, phi) of the robot will be reported from the Adaptive Monte Carlo Localization (AMCL) package for facilitating localization and navigation. The starting coordinate information is also saved in order to return when all objects have been visited and identified.

Using the coordinates of the objects are provided, the robot then calculates the optimal position of scanning and identifying the images, which was determined to be 0.45m away from the centre of the object. These desired poses were stored as `goal_x, goal_y` and `goal_phi.` Then the `optimizePath()` function is used to figure out the optimal shortest path to visit all ten objects and return to the starting point via brute-force algorithm as explained in Section 2. The robot then proceeds to follow the optimal order to visit each object, identifying the images and marking them as it goes.

For the cases where the `goal_x` and `goal_y` coordinates cannot be reached due to walls or other objects impeding the desired location, new offset positions will be calculated and replace the original goal poses using the `adjustGoal()` function.

The robot will check if all the boxes have been visited (YES or NO); if YES, the robot will use `ImagePipeline::getTemplateID` implemented in `ImagePipeline.cpp` to match the appropriate tags for each object. If NO, then the robot will move to the next object in the determined optimal path order that has not yet been visited, and again use the `getTemplateID()` function to get the object's image identification.

The process above is accomplished with the following functions:

- **`goalPosition():`** this function utilizes the coordinates of the centre of the objects to calculate a desirable robot pose (`goal_x, goal_y,` and `goal_phi`) that is facing straight towards the images at a distance of 0.45m away to allow the RGB camera to pick up the scene accurately.

- **`optimizePath():`** this function figures out the optimal shortest path to the 11 vertices in the map (ten objects + starting point) via brute-force algorithm learned in class. In this function, the distances between all boxes and starting position are measured and stored in a matrix (`pathMatrix`). Another variable `pathLength` is used for representing the total distance of each path found. All the permutations of the orders of the boxes that the robot will go in are then formed and stored in another matrix (`goals`). The shortest path length is

stored as a variable (`minPath`). The total distances for all the permutations are then calculated one by one, and each time when a shorter path is found, the `minPath` variable is updated and the corresponding path order array `pathUlti` is used as the minimum-weight Hamilton circuit with the shortest optimal path for the robot.

- **`adjustGoal()`**: this function will be used to check whether the coordinates that the robot will go to (`goal_x, goal_y`) can be reached. It utilizes the make_plan service to check if a viable path can be planned to the goal locations, which is called in a helper function called **`checkGoal()`**. If not, the function will generate a new goal coordinate with a 5-degree angle offset counterclockwise from the original goal position. If this new position is still not available, it will continue the 5-degree sweep until it finds a reachable coordinate. If the sweep angle reaches 75 degrees and still cannot find a position that the robot can go to, the function will switch the offset angle to the other side (clockwise), and follow the same 5-degree sweep pattern as it does in the counterclockwise situation.

3.2.2. Image Identification

The RGB camera on the Kinect 360 is used to capture the scene, which is then processed with SURF to recognize labels attached to each box. For the template images provided, the aspect ratio is changed to 500 by 400 to match the box images in the environment. This significantly improved the accuracy of classification for images that were stretched out, especially for the vertically long tag_12.jpg. To extract scene features with increased accuracy, the scene image is transformed from colour (`BGR8`) to greyscale (`RGB2GRAY`).

There was a fatal error occurring from testing runs that came up while examining the scene, which would stop the entire run. This was uncovered to happen when the robot is oriented towards a wall and not lined up with the goal box, which would cause the scene descriptor matrix to be empty. Another error occurs when the object and scene keypoint matching vectors provide less than four points for finding homography. In both cases, the current iteration of matching a template image to the scene was skipped. If all 15 iterations of comparison are skipped as such or if the number of key points and homography projection area criteria explained in Section 3.2.2.2 are not fulfilled, the object is identified as having the blank image tag. Figure 7 below demonstrates the image identification process.

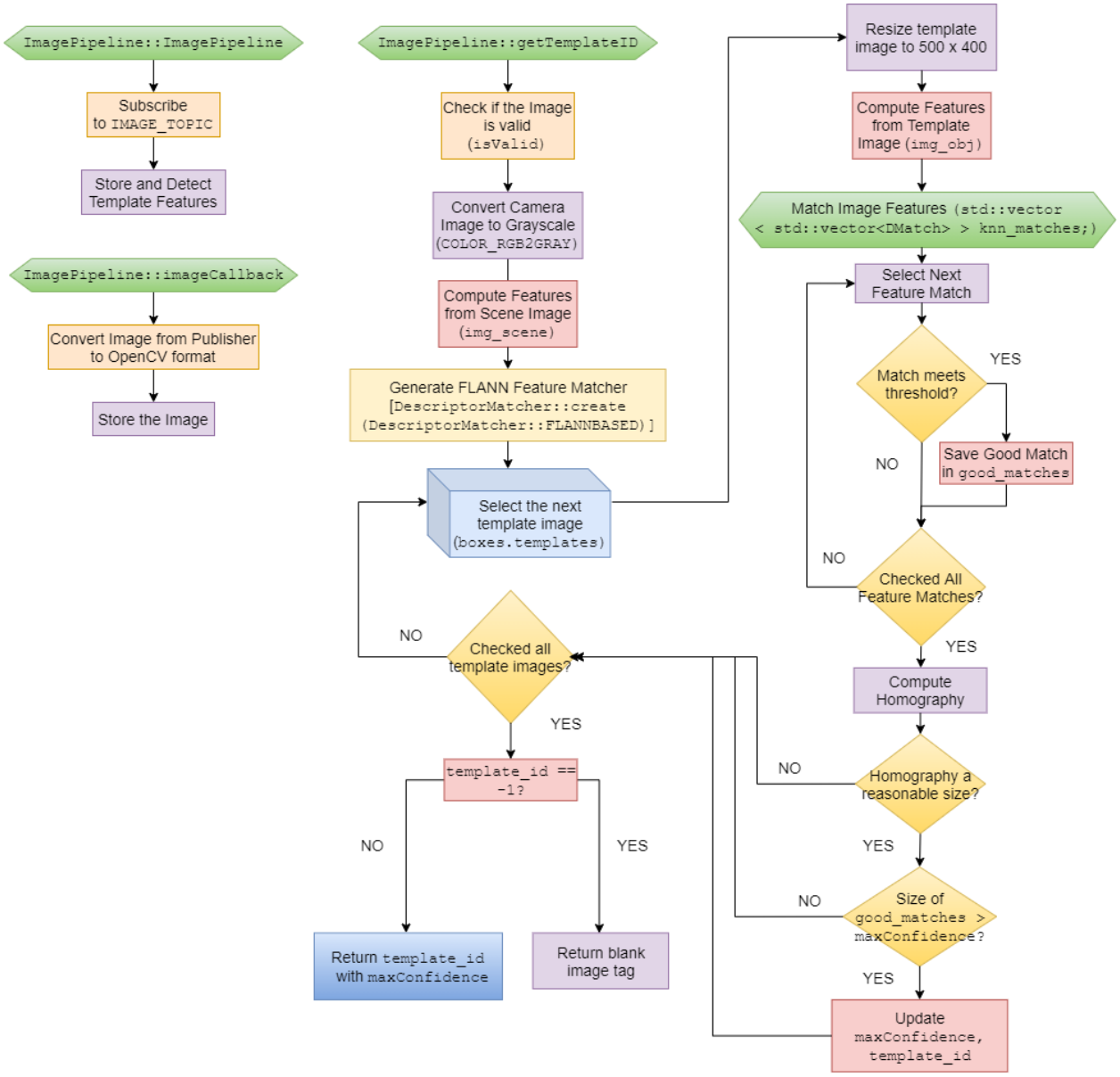

Figure 7: Image Identification Flow Chart

The process above is accomplished with the following key functions:

- **`ImagePipeline::getTemplateID():`** this function performs all of the image identification from start to finish. It first loads the template images resized at 500 by 400 to match the object dimensions, and it also converts the scene image taken from the RGB camera to grayscale. These two image augmentations greatly improved our algorithm's matching accuracy when testing. Then, the SURF feature detector is used to match the keypoints between the template and the object. The number of keypoints found is referred to as the

`confidence` variable in the code, and there is a minimum limit of 25 key points required to be matched in order for the template image to be considered as a reasonable identification. Then, the homography projected area variable called **`greenBoxSize`** is also calculated using the formula for the area of a polygon with the coordinates of its vertices [8]. After hundreds of trial runs, the most acceptable range of this area size came out to be between 80,000 and 375,000, depending on the varying orientation of the robot and objects after **`adjustGoal()`** is called. Finally, the template image that fulfills those criteria and has the maximum confidence value is returned as the best-matched tag for the visited object. If no template images satisfied those conditions to update `maxConfidence`, then the image was determined to be a blank image that did not have enough features to match or did not provide a reasonably projected homography box area.

- **`checkDuplicate()`**: this function will be used to check whether the tag that has been identified in the current object has been found previously in its trip. It goes through all of the elements in the tag storing vector that get filled as the robot visits the objects, and marks the current tag as a duplicate for the correct output text to be written in `StoreInfo()`.

- **`StoreInfo():`** this function is used to write and update the results according to the image identification algorithms. It notes the visit order, box number, matched tag, object coordinate, and the existence of duplicates for each of the ten boxes that are explored.

3.2.2.1. Feature Extraction

Images Features are extracted from all images using the Speed Up Robust Features (SURF) feature detector, which is included in OpenCV.

The SURF feature detector is able to detect features on different scales by filtering an image with a different size of a box filter. Since SURF produces feature descriptors that are an image scale function, it can match the features of input images that are of different sizes [9].

The `minHessian` controls the intensity of the feature SURF retains; a greater minHessian threshold will produce fewer but more outstanding features and vice versa [9]. For our algorithm, we settled on 800 as the `minHessian` parameter after extensive testing. On the lower level, since the SURF serves as a feature descriptor on the image gradients, the hessian is simply the gradient of the image gradients [9].

Images features from an RGB camera are calculated when an image is detected from the `ImagePipeline` file. The features extracted from the boxes are then passed on to the feature matcher and filter.

3.2.2.2. Feature Matching and Homography

The Fast Library for Approximate Nearest Neighbours (FLANN) library is used to match features from the camera image to each of the templates after extracting features from the templates and the scene [10].

In Contest 2, we are detecting key points from the images using `detectAndCompute()`. Based on the key points extracted, the image is checked against all the box templates given. Feature matchings are then completed using the kNN algorithm `knnMatch()`. The best matches (i.e. lowest computed distance) are kept in a vector `good_matches`.

After the good matches have been filtered, OpenCV's built-in `findHomography` function is used to measure the transformation of the image in the scene compared to the template.

A homography matrix is a matrix that transforms the image frame and the scene frame. After each template's homography has been computed, it is tested for validity by testing the size of the enclosed box [11]. After all of the template matches have been processed, the best match is chosen from the template with the most feature matches that have a valid homography, as shown in Figure 8.

A homography box is then drawn to check the boundary area (`drawMatches()`). The homography is used as a first pass to eliminate unsuitable template matches. If the computed homography `greenBoxSize` is far larger or smaller than the expected image size in the scene, the corresponding template is not considered a match.

For each template that has a suitable `greenBoxSize`, we take the template that has the maximum number of elements in `good_matches` (using `maxConfidence` as our counter variable). Images that do not meet the minimum `conf_threshold` (i.e. too few keypoints) are also skipped. This prevents the assignment of a tag for a blank box.

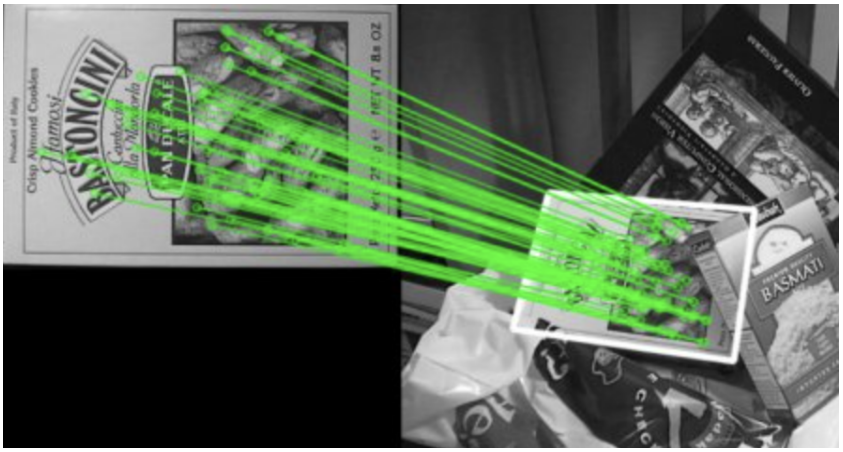

Figure 8: Example of Feature Matching + Homography to find Objects [4]

4. Future Recommendation

We decided to use the brute-force algorithm for the Traveling Salesman Problem (TSP), which ran in under a second in our testing runs. It was satisfactory for the conditions of Contest 2, while being simple to implement and always guaranteeing the optimal solution. However, the brute force algorithm can introduce serious issues if the number of vertices to visit was increased: for Contest 2 it was fine since we only have ten boxes to check. However, the time consumption grows factorially if more objects are added due to the intrinsic feature of the brute force algorithm. Therefore, our code would not be robust enough for that kind of change to the map environment. If we had more time, we would have liked to try implementing a more efficient optimal path planning algorithm that can deal with many more vertices. One of the examples of an efficient method in consideration was the Repetitive Nearest Neighbour algorithm. In this approach, we would try each vertex as the starting point, choose the nearest neighbour each time, and determine the shortest path among the 10 different paths before navigating the environment. This algorithm would not guarantee the optimal path like the brute-force algorithm, but would not be as computationally expensive and allow the code to be efficient and robust enough for a much higher number of objects in the environment.

There are also further improvements that could be made to the path optimization, which is to calculate the actual distance of the path that would be taken from each object to object, instead of the Euclidean distances between the provided object coordinates. This would determine a more accurate shortest path length since it takes the actual navigated path lengths. However, this was pushed back in priority for this contest for its complexity and time it would take to figure out how to fill the matrix with accurate numbers. It was also unclear if this would actually provide a different path order that is more optimized. Also, the robot pose information was not accurately tuned at all times with a margin of errors after navigation, so the time it took to get to the same

object in different iterations and from different directions resulted in varying experiences during testing.

Moreover, we can particularly design a function for the robot to move through a narrow passageway with less time consumption. At the moment, the robot movement is based on the AMCL particle filter, which facilitates localization and obstacle avoidance. Thus, it sometimes takes the robot a longer time to process and move through a narrow pathway. To improve this movement efficiency, we can design a function that calculates the distances between the TurtleBot and the nearby obstacles, and figures out an optimal moving path when these distances are smaller than a chosen threshold.

Last but not least, we can also tune the various parameters in the code to test different combinations. If time allowed, we could have tested with many more different sets of parameters such as the `minHessian`, `conf_threshold`, `ratio_threshold`, `greenBoxSize`, `boxDimension`, `tolerance`, and increments of `goal_phi` adjustment to determine what results in the most efficient and accurate runs.

5. References

[1] MIE443: Contest 2 (2021). *Finding Objects of Interest in an Environment*. Toronto: University of Toronto.

[2] G. Nejat. (2021). Lecture 4: Intelligent Control. Toronto: University of Toronto.

[3] "move_base - ROS Wiki", *Wiki.ros.org*, 2021. [Online]. Available: http://wiki.ros.org/move_base. [Accessed: 23- Mar- 2021].

[4] "OpenCV: Features2D + Homography to find a known object", *Docs.opencv.org*, 2021. [Online]. Available: https://docs.opencv.org/3.4.4/d7/dff/tutorial_feature_homography.html. [Accessed: 23- Mar- 2021].

[5] *Kobuki User Guide*. 2017.

[6]"When using navsat_transform_node, isn't it a problem to use IMU yaw to compute transform between map and base_link ? - ROS Answers: Open Source Q&A Forum", Answers.ros.org, 2021. [Online]. Available: https://answers.ros.org/question/265240/when-using-navsat_transform_node-isnt-it-a-problem-to-use-imu-yaw-to-compute-transform-between-map-and-base_link/. [Accessed: 22- Mar- 2021].

[7] Microsoft, “Kinect Sensor Components and Specifications.” [Online]. Available:https://msdn.microsoft.com/en-us/library/jj131033.aspx.

[8]"Area of any polygon (Coordinate Geometry) - Math Open Reference", *Mathopenref.com*, 2021. [Online]. Available: https://www.mathopenref.com/coordpolygonarea.html. [Accessed: 23- Mar- 2021].

[9]"Introduction to SURF (Speeded-Up Robust Features) — OpenCV-Python Tutorials 1 documentation", Opencv-python-tutroals.readthedocs.io, 2021. [Online]. Available: https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_feature2d/py_surf_intro/py_surf_intro.html#surf-in-opencv. [Accessed: 22- Mar- 2021].

[10]"Fast Approximate Nearest Neighbor Search — OpenCV 3.0.0-dev documentation", Luckytaylor.top, 2021. [Online]. Available: https://luckytaylor.top/modules/flann/doc/flann_fast_approximate_nearest_neighbor_search.html. [Accessed: 22- Mar- 2021].

[11]"OpenCV: Basic concepts of the homography explained with code", Docs.opencv.org, 2021. [Online]. Available: https://docs.opencv.org/master/d9/dab/tutorial_homography.html. [Accessed: 22- Mar- 2021].

6. Contribution Table

		Katrina	Kevin	Anh	Mariem	Wenhan
Codes	`goalPosition()`	TC	TC	WC,TC,AD		
	`optimizePath()`	WC, TC, AD	WC, TC			
	`checkDuplicate()`	TC	WC, AD, TC			
	`storeInfo()`	TC	WC, TC	WC,AD		
	`checkGoal()`	TC	WC, AD, TC			
	`adjustGoal()`	WC, TC, AD	WC, AD, TC	WC		
	`getTemplateID()`	WC, TC, AD	WC, AD, TC			
Report	Problem Objective		MR		RD	
	Strategy		MR		RD	
	Sensory Design		MR		RD	
	Controller Design		MR		RD	
	Navigation		MR	MR	RD, MR	RD
	Image Identification	MR	MR		RD, MR	RD
	Future Recommendation		MR	MR		RD
	Overall Document		ET, FP		ET, FP	

RS – research **RD** – wrote first draft **MR** – major revision **ET** – edited for grammar and spelling **FP** – final read **WC** - Written Code **TC** - Testing Code **AD** - Algorithmic Design

7. Appendix - C++ Code

7.1. contest2.cpp

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

#include <vector>
#include <cmath>
#include <chrono>
#include <iostream>
#include <fstream>

#include <ros/console.h>
#include "ros/ros.h"
#include <nav_msgs/GetPlan.h>
#include <tf/transform_datatypes.h>

#define RAD2DEG(rad) ((rad) * 180./M_PI)
#define DEG2RAD(deg) ((deg) * M_PI / 180.)

float xStart = 0.0, yStart = 0.0, phiStart = 0.0;
float boxDimension = 0.45;
float goal_x[11]; // 10 boxes + starting position
float goal_y[11];
float goal_phi[11];
int goalBoxNumber = -1; // initial starting point
bool gotPlan;

Boxes boxes;
RobotPose robotPose(0,0,0);

// store values for result output
std::vector<float> storeTag;
// store location in order of visit
std::vector<int> storeBoxNumber;
std::vector<float> storeX;
```

```
std::vector<float> storeY;
std::vector<float> storePhi;

// Create array of 0, change to 1 if that box has a duplicate tag
int duplicate[10] = {};
// Create array of 0, corresponding to unexamined object, and 1 for examined
object
int examined[10] = {};
// Ultimate TSP path between boxes
int pathUlti[10] = {};



//Replace the coords of the image's center with coords that the robot can go to
void goalPosition(){
    for (int i = 0; i < boxes.coords.size(); ++i){
        goal_x[i] = boxes.coords[i][0] + boxDimension * cos(boxes.coords[i][2]);
        goal_y[i] = boxes.coords[i][1] + boxDimension * sin(boxes.coords[i][2]);

        if (boxes.coords[i][2] < 0.0)
        {
            goal_phi[i] = (boxes.coords[i][2] + M_PI);
        }
        else
        {
            goal_phi[i] = (boxes.coords[i][2] - M_PI);
        }
    }
    goal_x[10] = xStart;
    goal_y[10] = yStart;
    goal_phi[10] = phiStart;
}


void optimizePath(){

    //create a matrix of distance lengths between boxes (and starting point - 11)
    float pathsMatrix [11][11] = {0.0};
```

```
    for (int i = 0; i < 11; i++){
        for (int j = 0; j < 11; j++){
            pathsMatrix[i][j] = std::sqrt(std::pow(goal_x[i] - goal_x[j], 2) +
std::pow(goal_y[i] - goal_y[j], 2));
        }
    }

    // permutation matrix for orders of boxes to travel to
    int goals[10];
    for (int k = 0; k < 10; k++){
        goals[k] = k;
    }

    // check all possible permutations for shortest path
    // documentation / example for next_permutation here:
http://www.cplusplus.com/reference/algorithm/next_permutation/
    float minPath = std::numeric_limits<float>::infinity();
    float pathLength = 0.0;
    do {
        //first path length from start Position to first box
        pathLength += pathsMatrix[10][goals[0]];

        //Calculate path between boxes
        for (int k = 0; k < 9; k++){
            pathLength += pathsMatrix[goals[k]][goals[k+1]];
        }

        //last path length from last box to start Position
        pathLength += pathsMatrix[10][goals[9]];

        //new shortest path detected! Save length + order of boxes
        if (minPath > pathLength){
            minPath = pathLength;
            std::copy(goals, goals + 10, pathUlti);
        }
        pathLength = 0.0; //reset for next permutation
    } while (std::next_permutation(goals, goals+10)); //check all possible
permutations, holding starting point steady
```

```
    ROS_INFO("The shortest path length is: %f", minPath);
    ROS_INFO("Visit boxes in order: ");

    for (int i = 0; i < 10; i++){
        ROS_INFO("%i", pathUlti[i]);
    }
}


void checkGoal(ros::ServiceClient &serviceClient, std::vector<float> goal,
RobotPose robotPose){

    nav_msgs::GetPlan srv;

    geometry_msgs::Quaternion phi =
tf::createQuaternionMsgFromYaw(robotPose.phi);//initial position phi quaternion
coordinate
    srv.request.start.header.frame_id ="map";
    srv.request.start.pose.position.x = robotPose.x;//initial position x
coordinate
    srv.request.start.pose.position.y = robotPose.y;//initial position y
coordinate
    srv.request.start.pose.position.z =  0.0;
    srv.request.start.pose.orientation.x = 0.0;
    srv.request.start.pose.orientation.y = 0.0;
    srv.request.start.pose.orientation.z = phi.z;
    srv.request.start.pose.orientation.w = phi.w;//Orientation

    geometry_msgs::Quaternion phiGoal = tf::createQuaternionMsgFromYaw(goal[2]);
    srv.request.goal.header.frame_id = "map";
    srv.request.goal.pose.position.x = goal[0];//End point coordinates
    srv.request.goal.pose.position.y = goal[1];
    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 = 0.0;//If the goal cannot be reached, the nearest
available constraint

    if (serviceClient.call(srv)) {
                //srv.response.plan.poses is the container for storing the
results, traversed and taken out
        if (!srv.response.plan.poses.empty()) {
//
std::for_each(srv.response.plan.poses.begin(),srv.response.plan.poses.end(),myfun
ction);
            ROS_INFO("make_plan success!");
            gotPlan = true;
        }
    }
    else {
        ROS_WARN("Got empty plan");
        gotPlan = false;
    }
    return;
}


void adjustGoal(ros::ServiceClient make_plan_client){
    int iter;
    // Check if the goal location of the box is valid (use make_plan to check if
path exists)
    gotPlan = false;
    std::vector<float> tryGoal;
    tryGoal.push_back(goal_x[goalBoxNumber]);
    tryGoal.push_back(goal_y[goalBoxNumber]);
    tryGoal.push_back(goal_phi[goalBoxNumber]);
    // Iteration count for how many new goals tried if initial goal is not valid
    iter = 1;
    // Check positive direction first
    std::cout << "initial x: " << tryGoal[0] << ", initial y: " << tryGoal[1] <<
", initial phi: " << tryGoal[2] << std::endl;

    //Check until valid plan found, or full +-75 degrees is swept
    while (gotPlan == false && iter >= -15){
```

```
        //Check if current config results in a traversable goal
        checkGoal(make_plan_client, tryGoal, robotPose);
        //Valid goal is found - set it
        if (gotPlan == true){
            goal_x[goalBoxNumber] = tryGoal[0];
            goal_y[goalBoxNumber] = tryGoal[1];
            goal_phi[goalBoxNumber] = tryGoal[2];
        }
        //Valid goal is not found - increment and try again
        else{
            tryGoal[2] = goal_phi[goalBoxNumber] + iter*DEG2RAD(5);
            // Adjust goal angle to within +-M_PI
            if (tryGoal[2] < -M_PI){
                tryGoal[2] += 2*M_PI;
            }
            else if (tryGoal[2] > M_PI){
                tryGoal[2] -= 2*M_PI;
            }

            tryGoal[0] = boxes.coords[goalBoxNumber][0] - boxDimension *
cos(tryGoal[2]);
            tryGoal[1] = boxes.coords[goalBoxNumber][1] - boxDimension *
sin(tryGoal[2]);

            std::cout << "testing x: " << tryGoal[0] << "testing y: " <<
tryGoal[1] << "testing phi: " << tryGoal[2] << std::endl;
        }

        // If we checked both left and right angle increments, we try a larger
angle offset goal position
        if (iter > 0){
            //sweep 5 degrees CCW
            iter += 1;
        }
        else{
            //sweep 5 degrees CW
            iter -=1;
        }
```

```
        //Once we have exceeded 75 degrees CCW sweep
        if (iter >=15){
            // Switch direction of checking to CW
            iter = -1;
        }

        std::cout << "gotPlan? (0 = false, 1 = true) " << gotPlan << std::endl;
    }
}


//Mark the stored tags that appear more than once as 1, to show in Results.txt
void checkDuplicate(std::vector<float> storeTag){
    for (int i =0; i < storeTag.size(); ++i){
        for (int j =0; j < storeTag.size(); ++j){
            //if (i != j){
            if (i < j){
                if (storeTag[i] == storeTag[j]){
                    duplicate[j] = 1;
                }
            }
        }
    }
}


//Open, write, and save the stored results after the robot finishes scanning
void storeInfo(std::vector<int> storeBoxNumber, std::vector<float> storeTag,
std::vector<float> storeX, std::vector<float> storeY, std::vector<float>
storePhi){

    std::ofstream
output("/home/turtlebot/catkin_ws/src/mie443_contest2/Results.txt");

    for (int i =0; i < storeTag.size(); ++i){
        std::string blankTag = "tag_blank.jpg";

        output << "Visit #";
        output << i+1;
```

```
        output << ": ";
        output << "For box number ";
        output << storeBoxNumber[i];
        output << ", ";

        if (storeTag[i] == 0){
            output << blankTag;
            output << " was found at X = ";
            output << storeX[i];
            output << ", Y = ";
            output << storeY[i];
            output << ", Phi = ";
            output << storePhi[i];
            if (duplicate[i] == 1){
                output << ", and this is a duplicate tag.";
            }
            output << "\n";
        }

        else{
            output << "tag_";
            output << storeTag[i];
            output << ".jpg";
            output << " was found at X = ";
            output << storeX[i];
            output << ", Y = ";
            output << storeY[i];
            output << ", Phi = ";
            output << storePhi[i];
            if (duplicate[i] == 1){
                output << ", and this is a duplicate tag.";
            }
            output << "\n";
        }
    }
    output.close();
}
```

```
int main(int argc, char** argv) {
    // Setup ROS.
    ros::init(argc, argv, "contest2");
    ros::NodeHandle n;
    // Robot pose object + subscriber.
    //RobotPose robotPose(0,0,0);
    ros::Subscriber amclSub = n.subscribe("/amcl_pose", 1,
&RobotPose::poseCallback, &robotPose);
    // Initialize box coordinates and templates
    //Boxes boxes;

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

    // Keep track of time for 8 minute limit
    std::chrono::time_point<std::chrono::system_clock> start;
    start = std::chrono::system_clock::now();
    uint64_t secondsElapsed = 0;

    // give time for initial starting position to be saved properly
    while (secondsElapsed < 1)
    {
        ros::spinOnce();
        xStart = robotPose.x;
        yStart = robotPose.y;
        phiStart = robotPose.phi;
        secondsElapsed =
std::chrono::duration_cast<std::chrono::seconds>(std::chrono::system_clock::now()
- start).count();
    }

    std::cout << "Initial Position to return to: " << "x: " << xStart << " y: "
<< yStart << " phi: " << phiStart << std::endl;
```

```
    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 object and subscriber.
    ImagePipeline imagePipeline(n);
    int templateID;

    //Replace the coords of the image's center with coords that the robot can go
to
    goalPosition();
    optimizePath();
    ROS_INFO("goal positions calculated");
    int goalBoxIndex = 0;

    // Execute strategy, within 8 minute time limit
    while(ros::ok() && secondsElapsed <= 480) {
        ros::spinOnce();

        // break out of box-to-box navigation loop and go back to start once the
path is complete
        if (goalBoxIndex == -1){
            std::cout << "Returning to the starting coodinates..." << std::endl;
            Navigation::moveToGoal(xStart,yStart,phiStart);
            secondsElapsed =
std::chrono::duration_cast<std::chrono::seconds>(std::chrono::system_clock::now()
-start).count();
            std::cout << "Final Time: " << secondsElapsed << " seconds" <<
std::endl;
            std::cout << "Finished! The results have been saved at
/home/turtlebot/catkin_ws/src/mie443_contest2/Results.txt" << std::endl;
```

```
            break;
        }

        // Set current goal box from TSP list
        goalBoxNumber = pathUlti[goalBoxIndex];
        ROS_INFO("Goal box number: %i", goalBoxNumber);

        if (examined[goalBoxNumber] != 1){
            //Check to see if the goal coordinate is reachable, adjust with
angled offset if needed
            adjustGoal(make_plan_client);
            //Move to box location

Navigation::moveToGoal(goal_x[goalBoxNumber],goal_y[goalBoxNumber],goal_phi[goalB
oxNumber]);
            ros::spinOnce();
            //Get image info
            templateID = imagePipeline.getTemplateID(boxes);

            // store result vectors
            storeBoxNumber.push_back(goalBoxNumber);
            //storeInfo(templateID) (goes from 0 to 14 -> tag 1 to 15)
            storeTag.push_back(templateID+1);
            //store the object location
            storeX.push_back(boxes.coords[goalBoxNumber][0]);
            storeY.push_back(boxes.coords[goalBoxNumber][1]);
            storePhi.push_back(boxes.coords[goalBoxNumber][2]);
            //Update the "examined" status of the box
            examined[goalBoxNumber] = 1; //box is now examined

            // Write and save Results.txt every visit just in case of force-quit
or unexpected error.
            // Mark the duplicate tags in storeTag
            checkDuplicate(storeTag);
            // Output the info to text file
            storeInfo(storeBoxNumber, storeTag, storeX, storeY, storePhi);
        }

        //Increment to next goal box in list
```

```
        if (goalBoxIndex == 10) {
            goalBoxIndex = -1;
        }
        else {
            goalBoxIndex++;
        }

        // Use: robotPose.x, robotPose.y, robotPose.phi
        ros::Duration(0.01).sleep();
        // Update timer
        secondsElapsed =
std::chrono::duration_cast<std::chrono::seconds>(std::chrono::system_clock::now()
-start).count();
        std::cout << "Time Elapsed: " << secondsElapsed << " seconds" <<
std::endl;
    }

    return 0;
}
```

7.2. imagePipeline.cpp

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

#include <cmath>
using namespace cv;
using namespace cv::xfeatures2d;
using std::cout;
using std::endl;

#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 is the scene input from the camera
        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) {
    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***/

        //Store the confidence of each template against the scene
        int confidence [boxes.templates.size()];
        float maxConfidence = 0.0;
        int conf_threshold = 25;
        // Set the scene as one of the tag images (e.g. "tag_2", "tag_9" etc. up
to 15, "tag_blank")
        //Mat img_scene = cv::imread("mie443_contest2/boxes_database/tag_2",
CV_LOAD_IMAGE_GRAYSCALE);
        //Set the scene from camera input
        Mat img_scene = img;
        // Convert to grayscale
        cvtColor(img_scene, img_scene, COLOR_RGB2GRAY);
        Mat img_obj;
        int minHessian = 800;
        Ptr<SURF> detector = SURF::create(minHessian);
        const float ratio_thresh = 0.7f;

        //Detect keypoints in scene
        std::vector<KeyPoint> kypt_scene;
        Mat des_scene;
        detector -> detectAndCompute(img_scene, noArray(), kypt_scene,
des_scene);
```

```
        Ptr <DescriptorMatcher> matcher =
DescriptorMatcher::create(DescriptorMatcher::FLANNBASED);

        // Check against all possible box templates
        for (int i = 0; i < boxes.templates.size(); i++){
            img_obj = boxes.templates[i];
            cv::resize(img_obj, img_obj, cv::Size(500,400)); //Match template
image ratio to box images

            //lines below from this tutorial:
https://docs.opencv.org/3.4.4/d7/dff/tutorial_feature_homography.html

            // Detect object keypoints and descriptors with SURF
            std::vector<KeyPoint> kypt_obj;
            Mat des_obj;
            detector -> detectAndCompute(img_obj, noArray(), kypt_obj, des_obj);

            // Use knn for feature matching, L2 norm
            std::vector< std::vector<DMatch> > knn_matches;
            // Prevent openCV error - check for empty scene descriptor matrix
(rare cases where robot looks into nowhere or wall)
            if (des_scene.empty()){
                std::cout << "empty scene descriptor matrix, skip to next box!"
<< std::endl;
                continue;
            }
            matcher->knnMatch(des_obj, des_scene, knn_matches, 2);

            //Filter best matches only
            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 << "tag number: " << i+1 << " -> Confidence: " <<
good_matches.size() << std::endl;
```

```
            // draw homography box, check the boundary area
            Mat img_matches;
            drawMatches(img_obj, kypt_obj, img_scene, kypt_scene, good_matches,
img_matches, Scalar::all(-1), Scalar::all(-1), std::vector<char>(),
DrawMatchesFlags::NOT_DRAW_SINGLE_POINTS);

            std::vector<Point2f> obj;
            std::vector<Point2f> scene;

            for( size_t i =0; i<good_matches.size(); i++){
                obj.push_back(kypt_obj[good_matches[i].queryIdx].pt);
                scene.push_back(kypt_scene[good_matches[i].trainIdx].pt);
            }

            // Prevent bad argument error
            if (obj.size() == 0 || scene.size() == 0){
            //if (obj.size() < 4){
                std::cout << "empty obj and scene vectors, skip to next image!"
<< std::endl;
                continue;
            }

            // Find homography
            Mat H = findHomography(obj, scene, RANSAC);
            // Prevent assertion error
            if (H.empty()){
                std::cout << "empty H matrix, skip to next image!" << std::endl;
                continue;
            }

            std::vector<Point2f> obj_corners(4);
            obj_corners[0] = Point2f(0,0);
            obj_corners[1] = Point2f( (float)img_obj.cols, 0);
            obj_corners[2] = Point2f( (float)img_obj.cols, (float)img_obj.rows);
            obj_corners[3] = Point2f(0, (float)img_obj.rows);

            std::vector<Point2f> scene_corners(4);
```

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

            line( img_matches, scene_corners[0] + Point2f((float)img_obj.cols,
0), scene_corners[1] + Point2f((float)img_obj.cols, 0), Scalar(0, 255, 0), 4 );
            line( img_matches, scene_corners[1] + Point2f((float)img_obj.cols,
0), scene_corners[2] + Point2f((float)img_obj.cols, 0), Scalar( 0, 255, 0), 4 );
            line( img_matches, scene_corners[2] + Point2f((float)img_obj.cols,
0), scene_corners[3] + Point2f((float)img_obj.cols, 0), Scalar( 0, 255, 0), 4 );
            line( img_matches, scene_corners[3] + Point2f((float)img_obj.cols,
0), scene_corners[0] + Point2f((float)img_obj.cols, 0), Scalar( 0, 255, 0), 4 );

            // Calculate the area of the green box formed using distances between
the points
            Point2f point1 = scene_corners[0] + Point2f((float)img_obj.cols, 0);
            Point2f point2 = scene_corners[1] + Point2f((float)img_obj.cols, 0);
            Point2f point3 = scene_corners[2] + Point2f((float)img_obj.cols, 0);
            Point2f point4 = scene_corners[3] + Point2f((float)img_obj.cols, 0);
            float greenBoxSize = fabs((point1.x*point2.y - point1.y*point2.x) +
(point2.x*point3.y - point2.y*point3.x) + (point3.x*point4.y - point3.y*point4.x)
+ (point4.x*point1.y - point4.y*point1.x))/2;
            // Display the keypoint matches
            float scale_percent = 0.4;
            Mat small_matches;
            resize(img_matches, small_matches,
Size(img_matches.cols*scale_percent, img_matches.rows*scale_percent), 0.5, 0.5,
CV_INTER_LINEAR);
            imshow("Keypoint detection", small_matches);
            std::cout << "greenBoxSize: " << greenBoxSize << std::endl;
            //Update most likely template_id if it has most matches & better than
some threshold
            //if (good_matches.size() > maxConfidence && good_matches.size() >
conf_threshold){
            if (good_matches.size() > maxConfidence && good_matches.size() >
conf_threshold && greenBoxSize > 80000 && greenBoxSize < 375000){
                maxConfidence = good_matches.size();
                template_id = i;
                std::cout << "maxConfidence updated" << std::endl;
                std::cout << "New best tag is: " << i+1 << std::endl;
            }
```

```
            cv::waitKey(10);
        }
        // Unless determined as blank image, display the best image match
        if (template_id > -1){
            imshow("Best template match", boxes.templates[template_id]);
        }
        else if (template_id == -1){
            std::cout << "empty H matrices, blank image!" << std::endl;
        }
        cv::waitKey(10);
    }
    return template_id;
}
```