MIE 443 Contest 2:

Finding Object of Interest in an Environment

Yu-Tung Chen - 1005024910
Kevon Seechan - 1004941708
Diego Gomez - 1005139690
Miguel Gomez - 1005138628

March 23rd, 2023

1.0 INTRODUCTION

1.1 Objectives

The objective of this contest is to have a Turtlebot autonomously navigate to five different locations in a known environment and identify the feature tags placed on objects at these locations. Once all feature tags have been identified, the robot must return to its starting position. The environment map and object coordinates will be provided beforehand and as such the goal for the team is to develop a path planning algorithm that enables the robot to reach all five objects within a specified time limit. The team must also create a matching algorithm in order to successfully identify the tags on the object.

1.2 Requirements and Constraints

The robot will be randomly placed and given 5 minutes to find and identify the feature tags on the objects and return to its starting position. It is necessary that the team utilizes the provided navigation library to drive the robot base and the RGB camera on the Kinect sensor to perform SURF feature when identifying the image tag.

The contest environment is constrained to a 4.87 x 4.87 m^2 area. Each object will be represented by cardboard boxes with the dimension of 50 x 16 x 40 cm^3. Out of the five boxes presented, three will be labeled with unique tags, one will be a duplicate, and the other will not have any label.

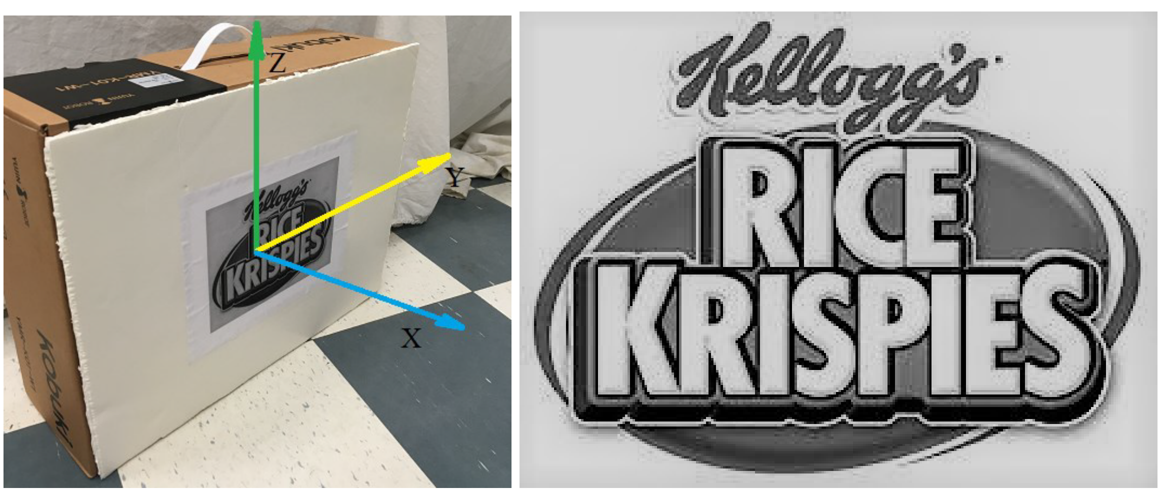

Figure 1: Example of Box with an attached feature tag

2.0 STRATEGY

The flowchart below describes the general flow of the algorithm.

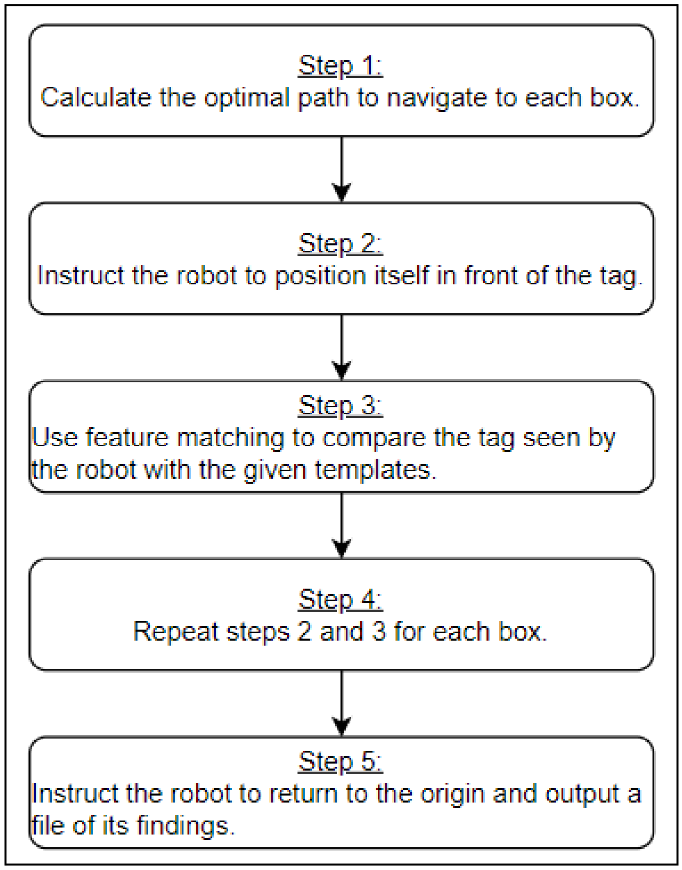

Figure 2: Flowchart of general strategy

The process can be divided into two portions, path planning, and feature matching.

2.1 Path Planning Strategy

The path planning portion of the contest is essentially a Travelling Salesman Problem (TSP). The main challenge of the TSP is to find the shortest possible path that a salesman can use to travel to a set of cities and return to the origin. In this case, the challenge is to find the shortest possible path the robot can take to navigate to each tag and return to its starting point. There are only 5 boxes present in the environment and as such the team decided to employ a brute force algorithm. Additionally, the brute force algorithm was selected because it guarantees an optimal solution. The algorithm works by first making a list of all possible solutions. These solutions are known as hamilton circuits. In this case, each circuit is a graph where each edge is weighted based on the certain variable. See below for an example of a weighted graph:

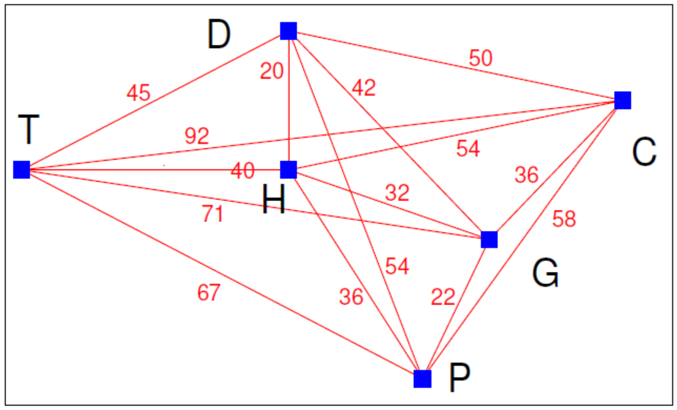

Figure 3: Weighted Graph

The team decided to weight the graph based on the distance between the boxes as this was simpler to implement in code than any other type of weight. Once each solution was calculated, the solution with the smallest total weight (in this case, distance) was chosen. To summarize, the brute force algorithm provided the robot with the optimal order in which to visit the boxes.

2.2 Feature Matching Strategy

The general feature matching strategy is described in the steps below:

1. Convert the image to grayscale.
2. Detect the keypoints and calculate descriptors using the Speeded Up Robust Features (SURF) detector.
3. Match descriptor vectors using the Fast Library for Approximate Nearest Neighbors (FLANN) matcher and filter the matches using Lowe's ratio test.
4. Localize the object in the input image by finding the homography matrix using the RANdom SAmple Consensus (RANSAC) algorithm.
5. Define a contour using the input corners and calculate its area.
6. Check if the good match is inside the contour.
7. Return the number of matches.

The specifics of this strategy is discussed in more detail in section *3.2 Controller Design*.

3.0 DETAILED ROBOT DESIGN AND IMPLEMENTATION

3.1 Sensory Design

The robot contains various sensors such as encoders, 3 bumper sensors, 3 cliff sensors, 2 wheel drop sensors, and a Kinect laser sensor. Apart from the Kinect sensor module, none of the other sensors are used directly by the team. They are used indirectly via the move_base package. This package is explained in section 3.2 of this document. Below, three of the main sensors are described.

3.1.1 RGB Camera

The Kinect sensor module is equipped with an RGB camera and it was the only sensor directly used by the team for the contest. Its primary function was to capture images of tags on the boxes to use as an input for the feature matching algorithm. The camera has a pixel resolution of 640x480 and a frame rate of 30 fps. However, during testing, the team discovered that the resolution was insufficient for performing matching from a distance. To ensure precise matching, the team decided to have the robot approach each box up close and at different angles.

3.1.2 IR Laser Depth Scanner

In addition to the RGB camera, the Kinect laser sensor also includes IR laser scanners that are used to detect distances to surfaces. Although the laser depth sensor is not directly used in the algorithm for this contest, it is used in the move_base package provided, to help the robot localize in the environment. Localization is important since the robot is expected to travel to a given point. If localization is not accurate, the robot will not be able to move to the exact position. With the laser depth scanner, the robot is able to generate a depth map that is compared to the map provided which helps the robot to localize itself.

3.1.3 Odometry

Similar to the laser depth scanner, while the odometry is not directly used in the algorithm, it is also used in the move_base package to help the robot localize itself. The odometry data is collected using a series of encoders in the Turtlebot. These track changes in position and orientation as the robot maneuvers.

3.2 Controller Design

The design of the controller for contest 2 can be described as a looping sequential control system. This is because there are three main steps for the robot to complete: drive to the box, take a picture and match the image with a given template. Then, the three steps are repeated until all boxes are visited. The diagram below describes the control architecture and the link between high level control and low level control.

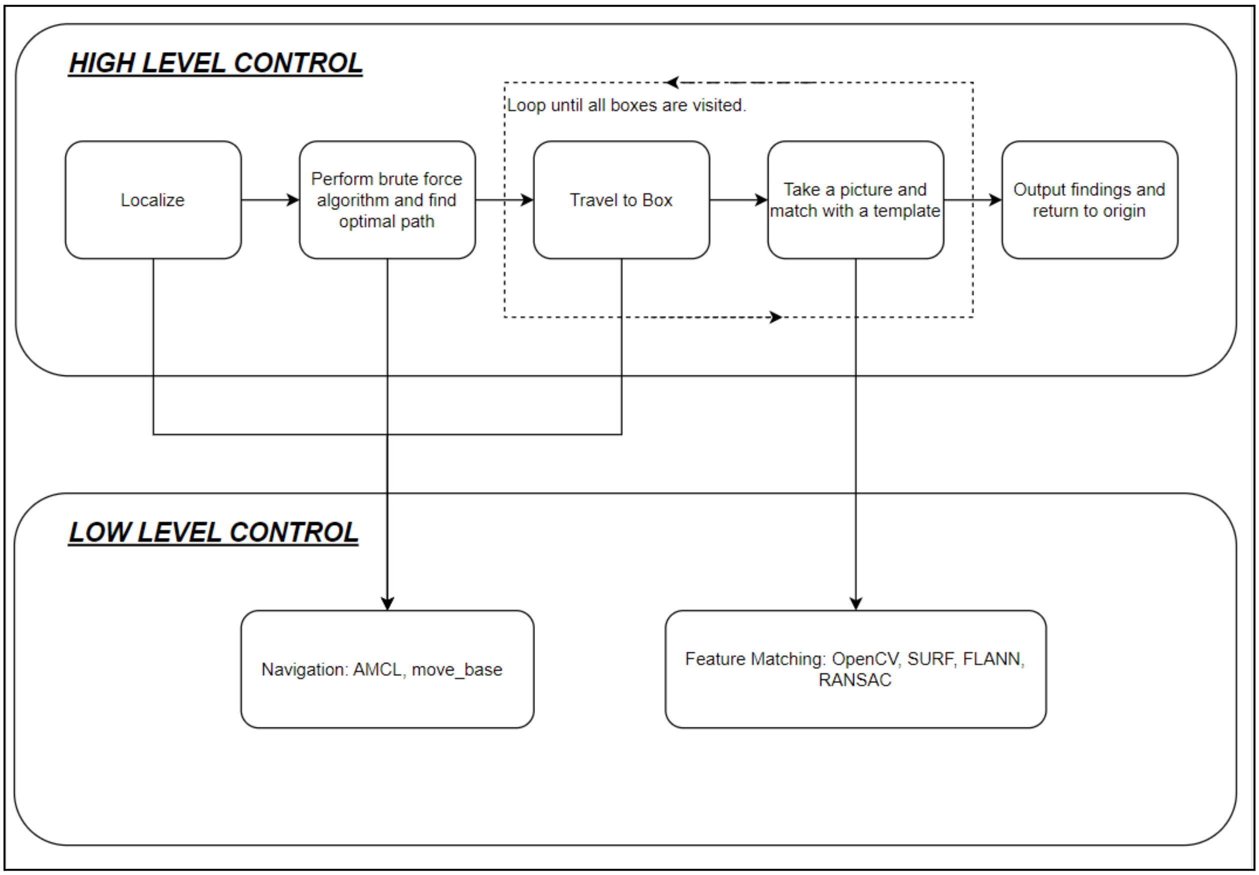

Figure 4: Diagrams of high and low level control

High level control involves the functions that the team wrote to implement the brute force algorithm, image processing and robot movement control. They make use of the different packages within the low level control. The low-level controller of the system includes the navigation and feature matching code that retrieves data from the sensors such as odometry and Kinect laser sensor directly. Additionally, there is a lower level of control that deals with the code surrounding the sensors themselves. This controller structure was adapted from the intelligent control lecture slides. It is a hybrid of deliberative (planner-based) control and reactive control. The code is deliberative in the sense that the team created an algorithm to feed the robot the optimal path, however it is simultaneously reactive in the sense that the code relies on the built in navigation systems of ROS to guide and allow the robot to adapt to its environment.

3.2.1 Navigation

3.2.1.1 Localization

For this contest, the Adaptive Monte Carlo Localization (AMCL) ROS package is used for localization which uses a particle filter to keep track of the robot's pose. The particle filter algorithm works by maintaining a probability distribution of all possible robot poses. The distribution is continuously updated by comparing each particle to the Turtlebot's current sensor state and assigning a weight that is proportional to how well the particle and sensor state correspond to each other. The process is repeated until the particle state converges to the true robot state.

3.2.1.2 Path Planning and Optimization

The path planning aspect of the code first begins by loading two arrays. The first array is loaded by a function called *loadDriveCoords ()*, this function utilizes the information given in 'boxes' to assign coordinates for the robot to drive to. The box coordinates given describe a location directly in front of the box. To ensure that the robot is able to take a proper picture, the drive coordinates are created with an offset. This offset was expressed in terms of a constant multiplied by sines and cosines, either being added or subtracted based on whether the image was facing up (add y offset), down (subtract y offset), left (subtract x offset), or right (add x offset). Given an image can also be looking diagonally, a mix of these offset additions and subtractions was used so that the robot will always be directly in front of the image.

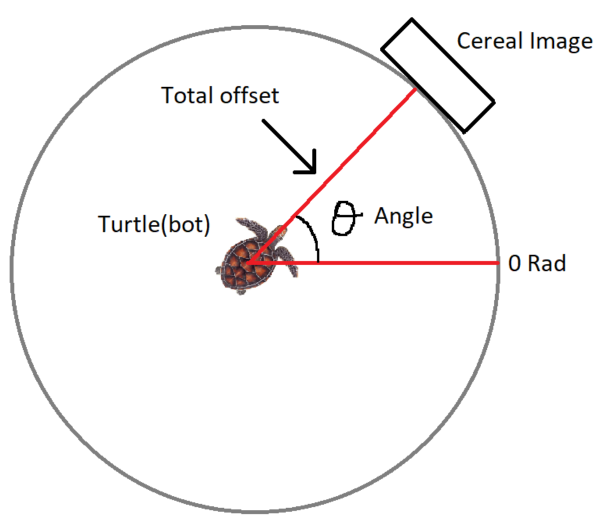

Figure 5 - Explanation of offset and angle of approach

The second array is loaded by a function called *loadMultDriveCoords ()*. This function is similar to the one described above, however, it contains 3 sets of driving instructions per box. This is because the team decided to take 3 pictures of each box at different angles. The *loadMultDriveCoords ()* uses the original box coordinates and a combination of linear and angular offsets to create coordinates that allow the robot to drive head on, to the left and to the right of the box.

Next, in order to begin the brute force algorithm, the distance between each box was calculated. This was performed by the function: *distanceStorage()* which filled an array with euclidean distance between each box. The brute force algorithm requires a start point and this is calculated by the function *nearestBoxToStart()*. This function returns the index of the box that is closest to the robot at the beginning of the contest.

Finally, to get the optimal path, the function: *findOptimalPath()* was used. This function cycled through each possible permutation in which the boxes can be visited. For each permutation, the total distance is calculated, the function returns the order of boxes that resulted in the least distance travelled. Note, the optimal path may not be physically possible for the robot to execute. This is because the brute force algorithm does not account for additional obstacles such as walls. Therefore, to ensure the path generated is possible, the function *isPathValid()* is used. This function uses the 'make_plan' service of the move_base package to determine if the path between two points is valid.

3.2.1.3 Robot Movement

The ROS move_base package is used to send the Turtlebot to the designated location for this contest. The package provides an implementation of an action that, given a goal in the world, will attempt to reach it with a mobile base. The move_base node links together a global and local planner to accomplish its global navigation task. Figure below shows the overview of the function.

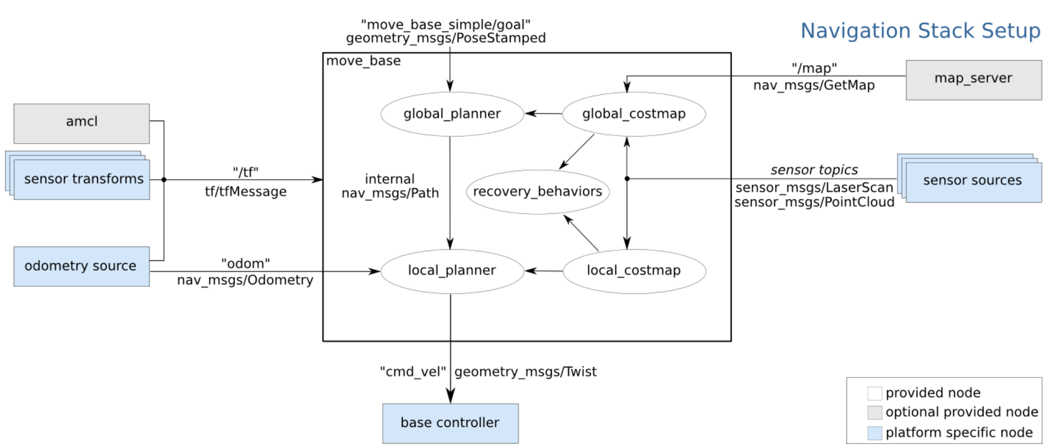

Figure 6: ROS move_base Overview Flowchart

The move_base node has a default recovery behavior that it would perform when the robot thinks it is stuck. Figure below shows the actions the robot would attempt when recovery behavior is actuated.

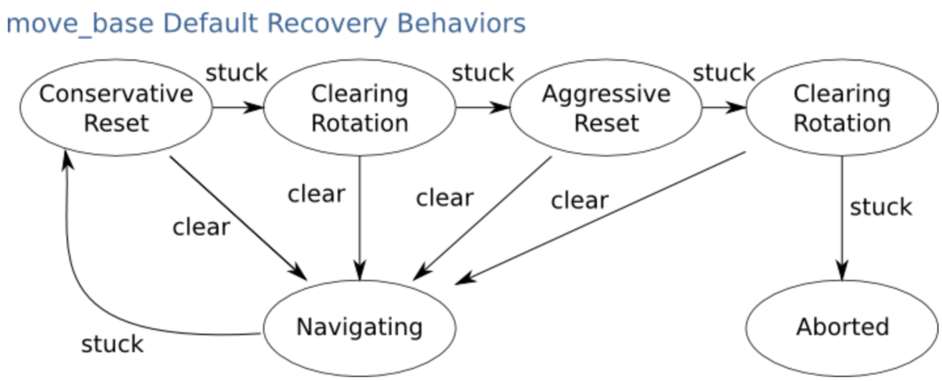

Figure 7: move_base Recovery Behaviors

The move_base package was implemented by the *Navigation::moveToGoal()* function. Drive coordinates were passed to this function in order to get the robot to move around the map.

3.2.2 Feature Matching

3.2.2.1 Image Pipeline

ImagePipeline is a ROS package that includes tools for image processing and manipulation. Its main purpose in the contest is to adapt the images taken by the RGB camera on the Kinect sensor, extracting the relevant information (the cereal box image) and formatting the image to be able to compare it to existing images. Some features used by ImagePipeline are homography (mapping points from the image taken by the camera to another image that can be interpreted), image resizing, and color conversion. These are used to be able to compare images more easily, allowing for the identification of each image captured by the robot.

3.2.2.2 SURF

Speeded up Robust Features (SURF) is used to detect the features of an image. These features are composed of keypoints (a 2D point in an image) and a descriptor (a vector containing values that characterize the area around the keypoint. The input to the SURF feature detector is an image (in our case, the cereal box image recognized by the robot) and the outputs are the features and descriptors that describe the image, which are compared against existing images to identify each one.

3.2.2.3 FLANN

FLANN (Fast Library for Approximate Nearest Neighbors) is a library providing algorithms for performing approximate nearest neighbor searches. This library is used in the contest to speed up feature matching between images, increasing the efficiency and accuracy of the process. Due to the reduced time limit in the contest, this is a useful library to use in order to increase the speed of image recognition and decrease the overall time taken by the robot to complete the trials.

3.2.2.4 Image Capturing

The robot tries to capture 3 images of each box, one from the left, one from the right and one directly from the front. These images are compared with the given templates using the *ImagePipeline::calculateSimilarityScore()* function. This function works by taking in an input image and a template image and calculating a similarity score between them. It first converts the input image to grayscale and resizes the template image to match the aspect ratio of the input image.

Next, it uses the SURF (Speeded Up Robust Features) detector to detect keypoints and calculate descriptors for both the template and input images. It then matches the descriptors using the FLANN (Fast Library for Approximate Nearest Neighbors) matcher and filters the matches using Lowe's ratio test to obtain good matches.

The function then localizes the object in the input image by finding the homography matrix using the good matches and calculates the contour of the object using the homography matrix. It calculates the area of the contour and assigns it a weight based on its size.

Finally, the function checks if each good match is inside the contour and multiplies the number of matches by the area weight to obtain the similarity score. If there are no key points detected in the input image or if the homography matrix is empty, an error message is printed and the similarity score is set to zero.

To get the actual template index that matches with the image, the function *ImagePipeline::getTemplateID()* is used. This function loops through each given template and returns the one with the most matches. The team decided to store the result of each position (left, right and center) of each box in an output file so that the accuracy of the template matching strategy can be analysed.

4.0 Explanation of Full Code Operation

The program initializes ROS, loads the coordinates and templates for the cereal boxes, and initializes variables for the driving coordinates, distances between the boxes, optimal path, and origin. It sets up the output CSV file, loads the specific and general driving coordinates, calculates distances between boxes and finds the closest box to start.

The program then enters into the main while loop: *while(ros::ok() && secondsElapsed <= 300)*. Within this loop, the code acts in a sequential manner, navigating to one box at a time, taking 3 images at each box and adding the results of these images to an output file. It begins by assigning the drive coordinates such that the robot approaches the left of the first box given by the optimal path algorithm. Once these coordinates pass the validity test, the robot drives to the box, takes an image and the imagePipeline processes are performed. The code then loops 2 more times such that the robot drives to the center and then the right of the box, similarly taking and analysing an image at each location. When the 3 images are taken for the first box, the robot is instructed to go to the next box given by the optimal path. The code repeats until all 5 boxes have been visited in the order of the optimal path and 3 images have been taken of each box.

Note that if the path is found to be invalid, the driving coordinates are repeatedly, slightly, adjusted until a valid path is found. Once the path is valid again, the code re-enters its regular routine. At the end of the code, an output file is sent to the mie443_contest2 folder with the details of the images taken and the templates that matched.

5.0 FUTURE RECOMMENDATION

Due to time constraints, the team's strategy was to get a working code early in order to physically test the robot as quickly as possible. Code was selected based on its ease of implementation and modification rather than efficiency. Had there been more time to work on the project, more efficient code would have been implemented resulting in a more optimal solution to the problem. A Brute-Force Algorithm was used for path planning, which is generally inefficient and can potentially take a long time to execute. In the future, it could be worth it to explore different algorithms that might also work such as the Nearest-Neighbour algorithm. With more time, it could have also been interesting to consider combining map exploration with object recognition, eliminating the need to use a predetermined map with known locations.

Another consideration for the future would be to improve the feature matching strategy. The current strategy simply takes 3 pictures of each box, however, these pictures are not being compared. They are simply being stored meaning that for the 5 boxes, there are 15 outputs in the output file. With more time, the team would implement a strategy to compare these images before adding them to the output file.

6.0 APPENDIX

Appendix A - Contribution Table

Team Member	Task
Yu-Tung Chen	Responsible for testing the functionality of the robot. Wrote sections 1.0, 2.0, 3.1, 3.2.1.1, and 3.2.1.4.
Kevon Seechan	Responsible for developing the brute force algorithm and overall code implementation. Contributed to path planning and controller design sections of the report.
Miguel Gomez	Responsible for physical robot testing support. Wrote sections 3.2.2.1, 3.2.2.2, 3.2.2.3, contributed to 4.0.
Diego Gomez	Responsible for the initial navigation algorithm and aided in the debugging process of this algorithm. Wrote section *3.2.1.2* of the report and contributed to section 4.0.

Appendix B - CONTEST2.CPP CODE

```cpp
#include <boxes.h>
#include <navigation.h>
#include <robot_pose.h>
#include <imagePipeline.h>
#include <chrono>
#include <iostream>
#include <fstream>
#include <experimental/filesystem>
#include <ctime>
#include <locale>
#include <vector>
#include <algorithm>
#include <string>
#include <nav_msgs/GetPlan.h>

//to convert radians and degrees
#define RAD2DEG(rad) ((rad) * 180. / M_PI)
#define DEG2RAD(deg) ((deg) * M_PI / 180.)

const int numBoxes = 5;
const int numCoords = 3;
const int numPics = 3;
const int numBoxesXPics = numBoxes * numPics;

//sets angle to range of -Pi and Pi
float normalizeAngle(float angle){
    if (angle > M_PI){
        angle = angle -2*M_PI;
    }
    return angle;
}

float inbetweenDist (float x1, float y1, float x2, float y2){
    //gets the distanc between two points
    return sqrt(pow(x2-x1,2)+ pow(y2-y1,2));
}
```

```cpp
void loadDriveCoords (float driveCoords[numBoxes][numCoords], Boxes* boxesPtr, float offset){
    //loads the coordinates of the boxes using boxesPtr -> coords
    //New x, y, phi are calculated to be slightly offset from the box such that the robot faces the front of the box

    Boxes boxes = *boxesPtr;

    for (int i = 0; i < boxes.coords.size(); i++){
        float boxX = boxes.coords[i][0];
        float boxY = boxes.coords[i][1];
        float boxAngle = boxes.coords[i][2];
        float driveX = boxX + offset * cosf(boxAngle);
        float driveY = boxY + offset * sinf(boxAngle);
        float driveAngle = normalizeAngle(boxAngle + M_PI);
        driveCoords[i][0] = driveX;
        driveCoords[i][1] = driveY;
        driveCoords[i][2] = driveAngle;
        //ROS_INFO("Cereal Box Coords %d: (%.3f,%.3f,%.3f)", i, boxes.coords[i][0], boxes.coords[i][1], boxes.coords[i][2]);
        //ROS_INFO("Drive Coords %d: (%.3f, %.3f, %.3f)", i, driveCoords[i][0], driveCoords[i][1], driveCoords[i][2]);
    }
}
```

```cpp

//this function is similar to loadDriveCoords, however it forces the robot to drive to 3 spots in front of the robot
void loadMultDriveCoords (float driveMultCoords[numBoxesXPics][numCoords], Boxes* boxesPtr, float offset, float angleOffset){
    //loads the coordinates of the boxes using boxesPtr -> coords
    //New x, y, phi are calculated to be slightly offset from the box such that the robot faces the front of the box
    Boxes boxes = *boxesPtr;

    //fill driveCoords
    for (int i = 0; i < boxes.coords.size(); i++){
        float boxX = boxes.coords[i][0];
        float boxY = boxes.coords[i][1];
        float boxAngle = boxes.coords[i][2];
        float driveX = boxX + offset * cosf(boxAngle);
        float driveY = boxY + offset * sinf(boxAngle);
        float driveAngle = normalizeAngle(boxAngle + M_PI);

        // drive coordinates for directly in front of the box
        driveMultCoords[i*3][0] = driveX;
        driveMultCoords[i*3][1] = driveY;
        driveMultCoords[i*3][2] = driveAngle;

        // drive coordinates for slightly to the left of the box
        float leftAngle = normalizeAngle(boxAngle + angleOffset);
        driveMultCoords[i*3+1][0] = boxX + offset * cosf(leftAngle);
        driveMultCoords[i*3+1][1] = boxY + offset * sinf(leftAngle);
        driveMultCoords[i*3+1][2] = normalizeAngle(leftAngle + M_PI);

        // drive coordinates for slightly to the right of the box
        float rightAngle = normalizeAngle(boxAngle - angleOffset);
        driveMultCoords[i*3+2][0] = boxX + offset * cosf(rightAngle);
        driveMultCoords[i*3+2][1] = boxY + offset * sinf(rightAngle);
        driveMultCoords[i*3+2][2] = normalizeAngle(rightAngle + M_PI);

        //print out the drive instructions
        //ROS_INFO("Cereal Box Coords %d: (%.3f,%.3f,%.3f)", i, boxes.coords[i][0], boxes.coords[i][1], boxes.coords[i][2]);
        //ROS_INFO("Drive Coords %d - Directly in front: (%.3f, %.3f, %.3f)", i, driveMultCoords[i*3][0], driveMultCoords[i*3][1], driveMultCoords[i*3][2]);
        //ROS_INFO("Drive Coords %d - Left: (%.3f, %.3f, %.3f)", i, driveMultCoords[i*3+1][0], driveMultCoords[i*3+1][1], driveMultCoords[i*3+1][2]);
        //ROS_INFO("Drive Coords %d - Right: (%.3f, %.3f, %.3f)", i, driveMultCoords[i*3+2][0], driveMultCoords[i*3+2][1], driveMultCoords[i*3+2][2]);
    }
}

```

```cpp


//this function fills a matrix with the distances between each box
void distanceStorage (float distMatrix[numBoxes][numBoxes], float driveCoords[numBoxes][numCoords]){
    for (int i=0; i < numBoxes; i++){
        for (int j=0; j < numBoxes; j++){
            distMatrix[i][j] = inbetweenDist(driveCoords[i][0], driveCoords[i][1], driveCoords[j][0], driveCoords[j][1]);
            //ROS_INFO("Distance between (%d, %d): %.3f",i, j, distMatrix[i][j]);
        }
    }
}

//finds the closest box to the robot at the start of the contest
int nearestBoxToStart (float driveCoords[numBoxes][numCoords]){
    float minDist = std::numeric_limits<float>::infinity();   //set to largest possible value
    float currentDist;
    int boxIndex = -1;
    for (int i=0; i < numBoxes; i++){
        currentDist = inbetweenDist(0, 0, driveCoords[i][0], driveCoords[i][1]);
        //ROS_INFO("Distance to box %d: %.3f",i, currentDist);
        if (currentDist < minDist){
            minDist = currentDist;
            boxIndex = i;
        }
    }
    ROS_INFO("Nearest Box: %d", boxIndex);
    return boxIndex;
}
```

```cpp

float findOptimalPath(float driveCoords[numBoxes][numCoords], float distMatrix[numBoxes][numBoxes], int startPoint, std::vector<int> &optimalPath){
     //calculates the optimal path distance and box order using the brute force algorithm
    //driveCoords: array of box coordinates
    //distStorage: array of distances between boxes
    //startPoint: start of box order
    //optimalPath: vector of box indices in order of the optimal path.
    std::vector<int> cerealBoxes;

    //create the cerelaBoxes vector without the start point
    for(int i = 0; i < numBoxes; i++){
        if(i != startPoint){
            cerealBoxes.push_back(i);
        }
    }

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

    do {
        float currentPathDist = 0;
        std::vector<int> currentSequence = {startPoint};
        int j = startPoint;

        for (int cerealBox : cerealBoxes) {
            currentPathDist += distMatrix[j][cerealBox];
            currentSequence.push_back(cerealBox);
            j = cerealBox;
```

```
        }

        currentPathDist += distMatrix[j][startPoint];

        if (currentPathDist < smallestPathDist){
            smallestPathDist = currentPathDist;
            optimalPath = currentSequence;
        }

    } while (std::next_permutation(cerealBoxes.begin(), cerealBoxes.end()));

    //std::cout << " Distance: " << smallestPathDist << std::endl;
    std::cout << "Printing Visiting Order: ";

    for (int cerealBox : optimalPath) {
        std::cout << cerealBox << " ";

    }

    std::cout << std::endl;
    return smallestPathDist;
}
```

```cpp
//we need to check whether the path to the boxes is valid
//to do this, we can check the current x,y,phi position and compare to the next x,y,phi recommend by the 'optimalPath' vector
bool isPathValid (ros::NodeHandle& nh, float curX, float curY, float curPhi, float nextX, float nextY, float nextPhi ){
    geometry_msgs::PoseStamped start, goal;
    start.header.frame_id = goal.header.frame_id = "map";
    start.header.stamp = goal.header.stamp = ros::Time::now();
    start.pose.position.x = curX;
    start.pose.position.y = curY;
    start.pose.position.z = goal.pose.position.z = 0.0;
    start.pose.orientation = tf::createQuaternionMsgFromYaw(curPhi);
    goal.pose.position.x = nextX;
    goal.pose.position.y = nextY;
    goal.pose.orientation = tf::createQuaternionMsgFromYaw(nextPhi);

    //call make_plan service
    nav_msgs::GetPlan srv;
    srv.request.start = start;
    srv.request.goal = goal;
    srv.request.tolerance = 0.0;

    if(nh.serviceClient<nav_msgs::GetPlan>("move_base/NavfnROS/make_plan").call(srv)){
        ROS_INFO("Sending call to check if path is valid");
        return srv.response.plan.poses.size() > 0;
    }
    else{
        ROS_INFO("Failed to make call to check path validity");
        return false;
    }
}
```

```cpp
int main(int argc, char** argv) {
    // Setup ROS.
    ros::init(argc, argv, "contest2");
    ros::NodeHandle n;
    // Robot pose object + subscriber.
    RobotPose robotPose(0,0,0);
    ros::Subscriber amclSub = n.subscribe("/amcl_pose", 1, &RobotPose::poseCallback, &robotPose);
    // Initialize box coordinates and templates
    Boxes boxes;
    if(!boxes.load_coords() || !boxes.load_templates()) {
        std::cout << "ERROR: could not load coords or templates" << std::endl;
        return -1;
    }
    for(int i = 0; i < boxes.coords.size(); ++i) {
        std::cout << "Box coordinates: " << std::endl;
        std::cout << i << " x: " << boxes.coords[i][0] << " y: " << boxes.coords[i][1] << " phi: " << boxes.coords[i][2] << std::endl;
    }
    // Initialize image objectand subscriber.
    ImagePipeline imagePipeline(n);

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

```cpp
    //initialize all variables
    float driveCoords[numBoxes][numCoords], driveMultCoords[numBoxesXPics][numCoords];
    float nextX, nextY, nextPhi, shortestPathDist, offset = 0.6, angleOffset = DEG2RAD(20);
    float distMatrix[numBoxes][numBoxes];
    int startPoint, template_ID, currentBox = 0;
    std::vector<int> optimalPath, pastID;
    std::array<float,3> origin;
    bool validPath, moveCompleted, dupTemplate = false, doneExploring = false;
    float changePhi = DEG2RAD(20);
    std::string filename, timeStr;
    int maxAngle = 61, angleInc = 15;
    int count = 0;
    std::string duplicateText;


    //writes the output file
    // Get timestamp for output file name
    time_t t = time(0);   // get time now
    struct tm * now = localtime(&t);
    char timestamp[150];
    strftime(timestamp, 150, "%Y-%m-%d %H-%M-%S", now);
    timeStr = std::string(timestamp);
    //use this filename when running on MIE443 Laptop
    filename = "/home/thursday2023/catkin_ws/src/mie443_contest2/Group 4 Output - " + timeStr + ".csv";

    //use this filename when running on Kevon Laptop
    //filename = "/home/turtlebot/catkin_ws/src/mie443_contest2/Group 4 Output - " + timeStr + ".csv";
    // Write output file if it doesn't exist yet with headers
    std::ofstream output(filename);
    output << "\"Path Taken (Box ID)\"" << ", " << "\"Image Tag\"" << ", " << "\"Is Duplicate\"" << ", " << "\"Location (x, y, angle)\"" << std::endl;
```

```cpp

    //load the general driving coordinates
    loadDriveCoords (driveCoords, &boxes, offset);

    //load the specfic drive coordiantes
    loadMultDriveCoords (driveMultCoords, &boxes, offset, angleOffset);

    //calculate the distances between the cereal boxes
    distanceStorage (distMatrix, driveCoords);

    //Find the closest box to the robot at the beginning of the contest
    startPoint = nearestBoxToStart(driveCoords);

    //find the optimal path and shortest distance to navigate to the boxes.
    shortestPathDist = findOptimalPath (driveCoords, distMatrix, startPoint, optimalPath);

    //store origin
    origin = {robotPose.x, robotPose.y, robotPose.phi};
    ROS_INFO("Starting position:\n\tx: %5.3f\ty: %5.3f\tyaw: %5.3f", origin[0], origin[1], origin[2]);

```

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

        if (currentBox <= numBoxes){
            while(count < numPics){
                ROS_INFO("Count: %d", count);
            //set the x,y,phi coordinates to go to three locations in front the box
                if(count == 0){
                    nextX = driveMultCoords[optimalPath[currentBox]*numPics+1][0];
                    nextY = driveMultCoords[optimalPath[currentBox]*numPics+1][1];
                    nextPhi = driveMultCoords[optimalPath[currentBox]*numPics+1][2];
                }
                if(count == 1){
                    nextX = driveMultCoords[optimalPath[currentBox]*numPics][0];
                    nextY = driveMultCoords[optimalPath[currentBox]*numPics][1];
                    nextPhi = driveMultCoords[optimalPath[currentBox]*numPics][2];
                }
                if(count == 2){
                    nextX = driveMultCoords[optimalPath[currentBox]*numPics+2][0];
                    nextY = driveMultCoords[optimalPath[currentBox]*numPics+2][1];
                    nextPhi = driveMultCoords[optimalPath[currentBox]*numPics+2][2];
                }
                //need to check if the set driving coordinates are actually valid
                validPath = isPathValid (n, robotPose.x, robotPose.y, robotPose.phi, nextX, nextY, nextPhi);

```

```

                //if the path is not valid, recalculate the new drive coordinates until there is a valid path:
                //the while loop varies the angle by 15 degrees and checks if the path is valid, if it is still not valid, it changes the angle again
                //the loop goes +-15,30,45,60 degrees
                if (!validPath) {
                    while (fabs(changePhi) <= DEG2RAD(maxAngle) && currentBox < numBoxes) {
                        nextX = boxes.coords[optimalPath[currentBox]][0] + offset * cosf(boxes.coords[optimalPath[currentBox]][2] + changePhi);
                        nextY = boxes.coords[optimalPath[currentBox]][1] + offset * sinf(boxes.coords[optimalPath[currentBox]][2] + changePhi);
                        nextPhi = normalizeAngle(boxes.coords[optimalPath[currentBox]][2] + changePhi + M_PI);

                        ROS_INFO("Testing new drive coords %d (original %d) with offset %.1f. (%.3f, %.3f, %.3f)", currentBox, optimalPath[currentBox],
                        validPath = isPathValid(n, robotPose.x, robotPose.y, robotPose.phi, nextX, nextY, nextPhi);

                        if (validPath) {
                            break;
                        }
                        changePhi = (changePhi > 0) ? -changePhi : -changePhi + DEG2RAD(angleInc);
                    }
                }

```

```cpp

            //if the path is valid: execute the following
            if (validPath) {
                moveCompleted = Navigation::moveToGoal(nextX, nextY, nextPhi);
                ROS_INFO("Arrived at box");

                if (moveCompleted && currentBox < numBoxes) {
                    // Wait to receive the picture from the camera
                    ros::Duration(0.10).sleep();
                    ros::spinOnce();
                    ros::Duration(0.60).sleep();
                    ros::spinOnce();

                    // Get template ID and check for duplicates
                    template_ID = imagePipeline.getTemplateID(boxes);
                    dupTemplate = imagePipeline.Duplicate(pastID, template_ID);

                    duplicateText = (dupTemplate) ? "True" : "False";

                    if (!dupTemplate) {
                        pastID.push_back(template_ID);
                        ROS_INFO("Appended %i to pastID", template_ID);
                    }

                    // Append data to output file
                    ROS_INFO("Appending to output file");
                    output << "\"" << optimalPath[currentBox] << "\", \"" << imagePipeline.tagIndexToString(template_ID) << "\", \"" << duplicateText << "\", \"" << "("
                    << boxes.coords[optimalPath[currentBox]][0] << ", " << boxes.coords[optimalPath[currentBox]][1] << ", " << boxes.coords[optimalPath[currentBox]][2] <<
                    << std::endl;
                    count++;
```

```cpp

                }
                else{
                    if(currentBox < numBoxes){
                        ROS_INFO("PLAN VALID BUT NAVIGATION FAILED");
                        output << currentBox+1 << "\", \"" << optimalPath[currentBox] << "\", \"" << "" << "\", \"" << "" << "\", \"" << "\"" << std::endl;
                    }
                    count++;
                }
            }
            if (count == 3){
                currentBox ++;
                if (currentBox == numBoxes){
                    doneExploring = true;
                }
                count = 0;
                break;
            }
        }
        if(fabs(changePhi) > DEG2RAD(maxAngle) || !validPath){
            ROS_INFO("Could Not Find Any Path to Node");
        }
    }
    if (doneExploring){
        //all boxes have been explored
        ROS_INFO("Returning to Origin");
        Navigation::moveToGoal(origin[0], origin[1], origin[2]);
        break;
    }
    secondsElapsed = std::chrono::duration_cast<std::chrono::seconds>(std::chrono::system_clock::now()-start).count();
    ros::Duration(0.01).sleep();
  }
  output.close();
  return 0;
}
```

Appendix C - imagePipeLine.cpp CODE

```cpp
#include <imagePipeline.h>
#include <string>

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

bool ImagePipeline::Duplicate(std::vector<int> &pastID, int template_ID){
    return (std::find(pastID.begin(), pastID.end(), template_ID) != pastID.end());
}

```

```cpp
std::string ImagePipeline::tagIndexToString(int idx) {

    constexpr char prefix[] = "tag_";
    constexpr char suffix[] = ".jpg";
    constexpr char blankSuffix[] = "blank.jpg";

    // Check for the blank case
    if (idx == -1) {
        return std::string(prefix) + std::string(blankSuffix);
    }

    // Compute the required string length (not including null terminator)
    int numDigits = static_cast<int>(std::log10(idx + 1)) + 1;
    int length = sizeof(prefix) + numDigits + sizeof(suffix);

    // Allocate a buffer to hold the string
    std::string result;
    result.reserve(length);

    // Copy the prefix
    result.append(prefix, sizeof(prefix) - 1);

    // Copy the index value
    char indexStr[12]; // Enough space for a 32-bit integer
    int numChars = std::snprintf(indexStr, sizeof(indexStr), "%d", idx + 1);
    result.append(indexStr, numChars);

    // Copy the suffix
    result.append(suffix, sizeof(suffix) - 1);

    return result;
}

```

```cpp
// Resize template image to match input image aspect ratio
double ImagePipeline::calculateSimilarityScore(Mat template_image) {
    // Convert input image to grayscale
    cv::Mat gray_image;
    cv::cvtColor(img, gray_image, cv::COLOR_BGR2GRAY);
    cv::resize(template_image, template_image, cv::Size(500, 400));

    // Step 1 & 2: Detect the keypoints and calculate descriptors using SURF Detector
    int minHessian = 400;
    Ptr<SURF> detector = SURF::create(minHessian);
    std::vector<KeyPoint> keypoints_template, keypoints_input;
    Mat descriptors_template, descriptors_input;
    detector->detectAndCompute(template_image, Mat(), keypoints_template, descriptors_template);
    detector->detectAndCompute(gray_image, Mat(), keypoints_input, descriptors_input);

    // Lowe's ratio filer
    // Step 3: Matching descriptor vectors using FLANN matcher
    Ptr<DescriptorMatcher> matcher = DescriptorMatcher::create(DescriptorMatcher::FLANNBASED);
    std::vector< std::vector<DMatch> > knn_matches;
    matcher->knnMatch(descriptors_template, descriptors_input, knn_matches, 2);

    // Filter matches using the Lowe's ratio test
    const float ratio_thresh = 0.75f;
    std::vector<DMatch> good_matches;
    for (size_t i = 0; i < knn_matches.size(); i++) {
        if (knn_matches[i][0].distance < ratio_thresh * knn_matches[i][1].distance) {
            good_matches.push_back(knn_matches[i][0]);
        }
    }

```

```
    // Localize the object
    std::vector<Point2f> template_points;
    std::vector<Point2f> input_points;

    for (int i = 0; i < good_matches.size(); i++) {
        // Get the keypoints from the good matches
        template_points.push_back(keypoints_template[good_matches[i].queryIdx].pt);
        input_points.push_back(keypoints_input[good_matches[i].trainIdx].pt);
    }

    if( input_points.empty()){
        std::cout << "Error: No keypoints detected in the captured image." << std::endl;
        return 0.0;  //set similarity score to zero.
    }

    Mat homography = findHomography(template_points, input_points, RANSAC);

    // Get the corners from the template image
    std::vector<Point2f> template_corners(4);
    template_corners[0] = cvPoint(0, 0);
    template_corners[1] = cvPoint(template_image.cols, 0);
    template_corners[2] = cvPoint(template_image.cols, template_image.rows);
    template_corners[3] = cvPoint(0, template_image.rows);
    std::vector<Point2f> input_corners(4);

    // Define input_corners using homography
    if (!homography.empty()) {
        perspectiveTransform(template_corners, input_corners, homography);
    } else {
        std::cout << "Error: Homography matrix is empty." << std::endl;
    }
```

```cpp
    // Define contour using the input_corners
    std::vector<Point2f> contour;
    for (int i = 0; i < 4; i++) {
        contour.push_back(input_corners[i] + Point2f(template_image.cols, 0));
    }

    double area = contourArea(contour);
    double area_weight = 1.0;

    area_weight = (area > 1000 * 1000 || area < 5 * 5) ? 0.0 : 1.0;

    std::cout << "area: " << area << ", weight: " << area_weight << std::endl;

    double score;

    std::vector< DMatch > best_matches;

    // Check if the good match is inside the contour. If so, write in best_matches and multiply by area weight
    Point2f matched_point;
    for( int i = 0; i < good_matches.size(); i++ )
    {
        matched_point = keypoints_input[ good_matches[i].trainIdx ].pt + Point2f( template_image.cols, 0);
        score = pointPolygonTest(contour, matched_point, false);
        if(score >= 0) best_matches.push_back( good_matches[i]);
    }

    cv::waitKey(10);

    double similarityScore = best_matches.size()*area_weight;
    return similarityScore;
}
```

```cpp
int ImagePipeline::getTemplateID(Boxes& boxes) {
    int template_id = -1;
    double best_matches, matches;
    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 {
        //test value
        best_matches = 10;

        //loop through each box template
        for(size_t i = 0; i < boxes.templates.size(); ++i){
            matches = calculateSimilarityScore(boxes.templates[i]);

            if (matches > best_matches){
                best_matches = matches;
                template_id = i;
            }
        }
    }
    //show grayscale image in Rviz
    cv::Mat gray_img;
    cv::cvtColor(img, gray_img, cv::COLOR_BGR2GRAY);

    //display the scene image
    cv::imshow("view", img);
    cv::waitKey(10);

    if (template_id == -1) {
        std::cout << "No matches found" << std::endl;
    }
    else {
        std::cout << "The best template is " << template_id << " with " << best_matches << " matches" << std::endl;
    }

    return template_id;
}
```