Contest 2 Report

MIE443 Project

Savo Bajic - 1003051485
Maximilian Glidden - 1002277396
Catherine Kucaba - 1003278026

1.0 Problem Definition and Design Requirements

The objectives and requirements for this portion of the project are outlined in the Contest 2 Manual [1]. The primary goal of this contest is to develop an algorithm that allows a robot to autonomously navigate a known environment to locate and identify ten objects placed at different locations. The robot must return to its starting location when it has finished tagging the objects and must complete the entire contest within a certain time limit. An image (referred to as a tag) is placed on each object for identification purposes. The team was provided a practice map of a practice environment, shown in Figure 1. One aspect to note is that a "known" environment does not refer to the team knowing in advance the trial environment and being able to program with it in mind. Instead, it refers to a map being provided to the robot prior to executing the code for localization purposes. Due to the Covid-19 pandemic, this project is completed only using software, as it would typically involve a physical robot navigating a real environment.

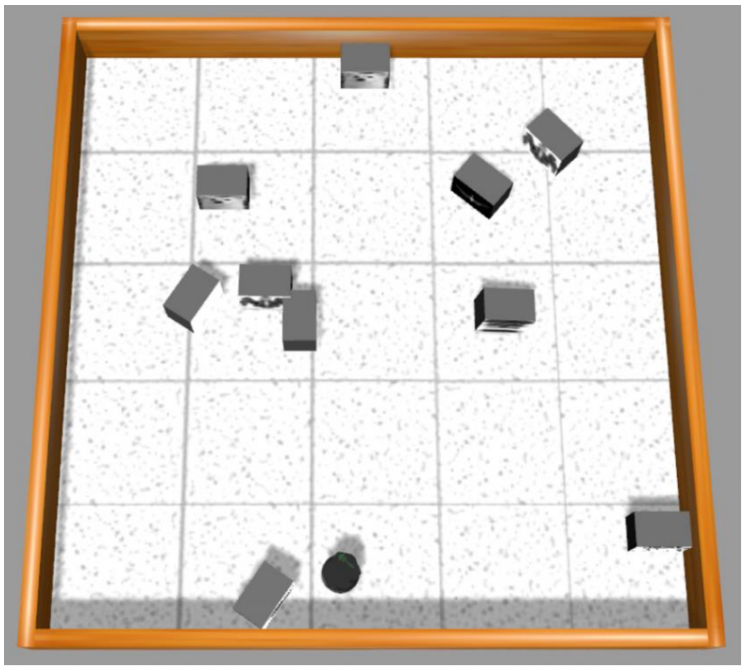


Figure 1: Example of a practice environment with 10 objects and TurtleBot [1]

The robot is simulated using TurtleBot and it navigates an environment created in gazebo. The environment is contained within a walled enclosure and only includes the objects the robot is to navigate to and identify (i.e., there will not be additional objects or obstacles in the area that the robot needs to identify or avoid). There are three main components in the algorithm that correspond to what the robot must accomplish in this contest: navigation, path planning, and image processing/recognition. The robot is to travel to specific locations in the environment; this information is provided by the teaching team via a dataset of coordinates and orientations of the objects prior to running the code. In order to efficiently move to these positions, an algorithm must be created to solve the traveling salesman problem presented in this contest. Lastly, once a robot nears an object, it must capture an image of the tag on the object and correctly identify it.

The team utilized the AMCL (Adaptive Monte Carlo Localization) ROS package as a localization system for the robot. It is a probabilistic system that tracks the position of a robot, using a particle filter, against a known map. The team also used RVIZ with the localization system in order to initialize the starting position of the robot (with its starting position in gazebo) and to converge AMCL before running and testing the code. This task will be completed by the teaching team before testing the program during the trials. The robot will be tested in two trials, with the best counting towards the final mark. Scoring is based on the navigation to each object as well as the correct identification of image tags (this includes stating duplicates) or determining if no tag is present on the object (a "blank").

The concepts required for this project can be applied to many other industries that are starting to use and rely on robots to execute tasks typically performed by humans. For example, a similar scenario is presented for service robots in the food or medical industry. Robots that take orders and deliver food to tables must efficiently navigate to different locations in a restaurant and tag the corresponding tables correctly to ensure the patrons receive the correct orders. The same idea can be applied to medical settings, where robots are used to deliver medications throughout hospitals. Another industry that specifically utilizes computer vision is manufacturing, which requires robots to properly identify components to either sort them or construct an assembly/subassembly with them.

1.1 Requirements and Constraints

For this contest, there are some design requirements and constraints that the team must take into account when developing the algorithm for the robot. These requirements can be found in the Contest 2 Manual [1] and are also presented and explained below.

1. The contest environment will be a three-dimensional simulated environment with static objects contained in a 6x6 m^2 area.
2. The initial location of the robot, when the map was created, determines the origin of the world coordinate frame.
    a. The origin is an inherent characteristic of the map.
    b. All distances in the map coordinate frame are measured with respect to this origin.
    c. The robot may not start at the origin of the world coordinate frame when the map is loaded.
3. There are 10 objects enclosed in the environment that the robot needs to navigate to and identify (example shown in Figure 1).
    a. There are no additional objects or obstacles in the environment.
    b. 1-2 objects will have duplicate tags, 1-2 objects will have no tag ("blank"), and the remainder will have unique tags.
    c. Each object is represented by a box of dimensions 50x32x40 cm^3 (lxwxh).
4. The location of an object is defined by the coordinates of its centre and orientation (x,y,φ), where φ is about the z-axis (see Figure 2).
    a. Locations are measured from the object's local frame (Figure 3) with respect to the world coordinate frame (Figure 2) at the origin of the map.

   b. Locations are defined by two vectors: a coordinate vector that defines the object's position in x and y, and an orientation vector that contains the object's rotation about z.
5. The tags are high contrast images (in greyscale) with unique features and are placed in the centre of one of the long faces of the objects (see Figure 3 for an example).
6. For testing, the team was provided with the following:
   a. A 2D map of the contest environment (generated with gmapping), including the 10 testing objects.
   b. Test locations (object coordinates/orientations) of the 10 objects, measured with respect to the world coordinate frame of the practice map.
   c. A dataset of 15 potential image tags that could be used during the contest, including a "blank" tag. The tags for the contest are strictly limited to this dataset.
7. The teaching team will generate a new set of objects and object locations (coordinates) to test the code in the trials that is not known to the team in advance.
   a. The program must be robust enough to handle object location changes.
8. The teaching team will choose the starting location of the robot in the map and will initialize the robot position/orientation in RVIZ prior to running the code.
9. The robot must use the provided navigation library and the RGB camera on the Kinect sensor to perform SURF feature detection.
10. The robot must navigate to and identify all provided objects, return to its starting position, and indicate it is finished within a maximum time limit of 8 minutes.
11. The program must output to a file all of the tags it identified in the exact order they were found including the object location and if the tag is a duplicate.

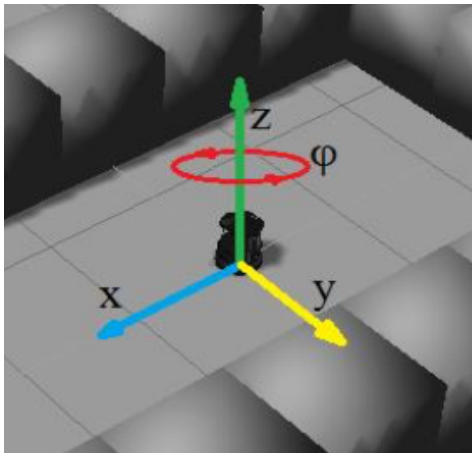


Figure 2: World coordinate frame [1]

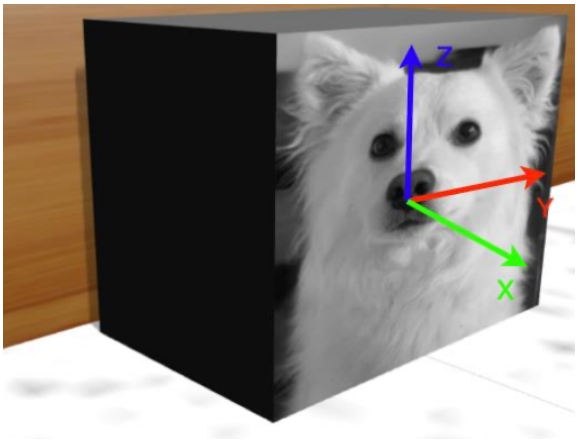


Figure 3: Object local frame and example of an image/tag [1]

2.0 Methods and Strategy

The team implemented a program, prior to creating the array to solve the traveling salesman problem (TSP), which determines the exact location the robot should navigate to for each object. Due to the system utilized, the object coordinates provided to the robot are not valid locations to travel to as they are located at/within the object (and the robot can only travel to regions of empty space). The code looks to find a valid position in front of or near the image side of an object in order to ensure the robot has a good enough view of the box to capture an image. This creates a robust system that will always pick the most optimal viewing locations (close to and directly looking at the image side of the object) before resorting to other, less effective positions that may result in worse capture images for the vision system.

Once the algorithm determines the optimal viewing locations for each object, the coordinates are plugged into the TSP array. The team chose to implement a brute force algorithm in order to solve the TSP for this contest. This means that the program determines every possible Hamilton circuit for the given layout and chooses the ordering that creates the smallest weight for the circuit. The advantage of this system is that it is guaranteed to find the most optimal path regardless of object locations/orientations, and thus provides the most efficient path every time. The tradeoff is that the algorithm itself is inefficient in terms of computation time. As the number of boxes increases, the amount of time to complete the computations grows exponentially. For this contest, there are 10 objects, resulting in a computation time of 3-5 seconds. Given that the maximum time limit is 8 minutes, this computation takes, at most, 1% of the available time. Other methods, such as the Nearest Neighbour Algorithm, are more efficient in computation but produce non-optimal paths, which may take longer for navigation. Thus, the team opted for the brute force method, as it was quick and reliable in this situation.

For image processing and recognition, the team chose to take an image sample once the robot completed its motion and is stationary. Although not part of this contest, this approach is good for if the robot was traveling over uneven surfaces (where the camera may be shaky). The processing algorithm reduces redundant information in the captured image, such as removing sections of the image that are the sky/walls. This prevents these aspects of the image from slowing or impeding computation time, as well as possibly resulting in poor image recognition due to these regions acting as noise. During feature detection, the program compares features to find the most compatible match based on the quantity of shared features. However, in instances where multiple images are present (due to viewing angle, etc.), the team's algorithm also uses size as a factor during comparison. In the captured image, the code favours tags that are in the foreground, and thus the largest, while ignoring tags that may be picked up in the background by the camera.

3.0 Detailed Robot Design and Implementation

3.1 Sensory Design

The simulated Turtlebot platform is equipped with a variety of sensors to aid in navigation, including touch bumpers, a depth camera, cliff sensors, and odometry data. As in Contest 1, the environment the robot is exploring in Contest 2 is flat and enclosed. Cliff sensors do not produce much useful information; there are no edges or changes in elevation for the robot to detect. In contrast with Contest 1, a map of the environment for Contest 2 is provided to the robot, along with the position of objects in the space. Since these obstacle locations are known and visible to the robot prior to movement, the use of touch bumpers was not required for successful completion of the contest. The laser scanner and odometry were used through the *AMCL* and *move_base* ROS nodes to aid in localization and obstacle avoidance. The Microsoft Kinect's RGB camera was used directly for image acquisition. Of all the sensors, only the RGB camera's data is regularly used in the team's code, the other sensors are handled directly by the aforementioned ROS topics without any action required of the team's code to aid in their roles.

3.1.1 Depth Camera/Laser Scanner

The Turtlebot is equipped with a Microsoft Kinect camera system, which integrates both an RGB and depth camera. The depth camera can be used to approximate the functionality of a laser scanner, which generates a distance at each point in a one-dimensional array from the depth camera input. The motivation for using the depth camera in this way was to provide sensory input to the *move_base* and *amcl* nodes about the presence and proximity of obstacles before the robot. This depth data could then be used to avoid obstacles for *move_base*, as well as help plan motion and localize by recognizing particular geometry from the map relative to the robot for the *amcl* nodes.

3.1.2 RGB Video Camera

The Turtlebot's Kinect's integrated RGB camera provides a full colour video feed and is used primarily for visually identifying objects present in the environment (scene) before the robot. This ability to see allows the team to collect the visual data needed to identify the tags present on the side of boxes for this competition using image processing techniques and routines that are part of *imagePipeline*'s member functions.

3.1.3 Odometry

Encoders on the motors of the turtlebot provide information on the rotational velocity of each wheel, which is used with the forward differential kinematics model to produce linear and rotational velocities. Integrating these velocities over time produces an estimate of the robot's position and heading; this is the odometry data supplied by the *odom* topic.

Odometry data is used by the *move_base* node as an input to the local planner so it is aware of the rough distance travelled. Similarly, this is used by *amcl* to determine the estimated shift for the robot's position distribution which helps maintain accurate localization which in turn aids proper pathfinding and navigation for the robot between its destinations.

3.2 Controller Design

The team's code for this contest focuses on autonomous navigation and image processing to achieve success in this contest. To ensure success in as many cases possible the robot may find itself in, the team did not design any behaviours to have special cases for specific circumstances that may pose an issue, and instead the team instead focused on developing a solid set of behaviours to operate well in the service environment in most circumstances. This prevents the possibility of unpredictable behaviour brought in by the unexpected triggering of special cases.

To this end, the team prescribes four primary states for the robot executed sequentially, listed below:

1. Initialization
2. Movement to boxes (and return to start once complete)
3. Image processing
4. Closing

Stages two and three are repeated until the robot visits (or attempts to visit) all the boxes, at which point the robot will return its starting location as required by this contest and proceed to stage four. The file responsible for controlling the state of the robot is *contest2.cpp*.

The robot control code begins by initializing itself. This begins by recording the start time (to ensure the robot knows when it is running out of time) and loading in files containing information about the boxes in the environment such as their positions and reference images for the tags. Following the box data collection, ROS is initialized with a subscriber prepared for positional information to be passed on to *robotPose* from the AMCL service provided by ROS. After the positional data service is configured the image processing system is initialized so it knows what to look for when needed and it has a subscription to the video from the Kinect cameras. This is explained in greater detail in section 3.2.2.

With the input systems initialized, the pathfinding is started to plan the motions the robot should undertake based on the starting point and the locations of all the boxes. This path planning determines the viable stops for the robot to visit to best see each box's tag and then finds an optimal path between them. This is covered in greater detail in section 3.2.1.

Once the path is determined, under normal operation, the robot will begin to execute this plan and begin moving to the boxes (going from the initialization state to the motion state). If there is testing enabled, the tests are performed instead on these initialized systems (refer to section 3.2.4 for greater detail).

The movement stage is simple, and makes use of the team's *goToStop* function to attempt reaching the next box in the environment according to the planned route (if it is returning to the start location *goToCoords* is used instead), these functions are explained more in section 3.2.1. If it was trying to reach a box, the code progresses to image processing. If it was returning to the start point at the end of a run, it would exit the main loop and proceed to the closing stage.

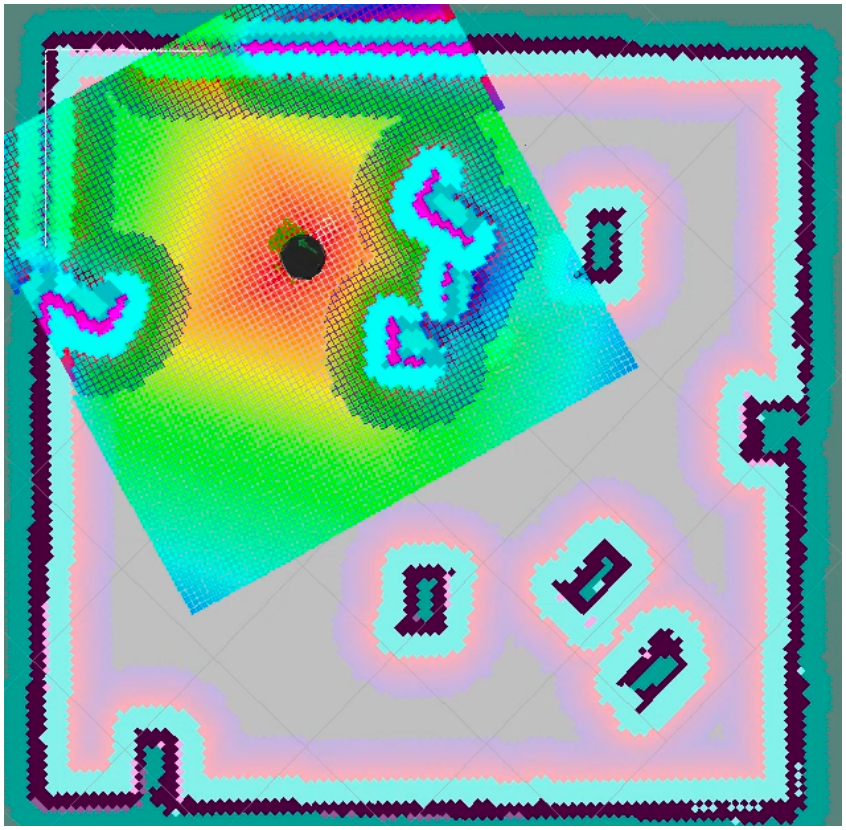


Figure 4: Rviz Visualization for the mapping, localization, and path planning used by moveToGoal*. The robot is the black circle, the colour gradient emanating from its location is its pathfinding "warmth" (red is the cheapest way to approach its goal). There is a second layer of colours faintly visible across the entire map, the cost layer, showing the cost incurred by travelling through these regions based on the proximity to an obstacle (black shapes).*

Regardless of the success the robot had in reaching its destination, it scans the environment to attempt spotting a tag on the box, using a process explained in section 3.2.2, and then records the result for this stop. The choice to scan regardless of if the motion was successful or not is grounded in the team's observation that the robot faces the direction it is travelling in, and that a majority of these failures occurred in the final approach to the box. So should it fail to reach a box, there is a good chance the robot is at least roughly facing it and thus can potentially make a positive match, even if it is not in the ideal position relative to the box.

Following the image scan, the results the robot has collected up to that point are recorded in the output file using the output system explained in section 3.2.3. These results are recorded after each stop so all data is not lost in the event the robot's execution is interrupted, so there is at least a portion of submittable results that can be used.

The main loop concludes by incrementing to the next box and checking the robot has not passed the time limit. Should the robot be running overtime, it immediately proceeds to the closing stage. Otherwise it begins the next loop iteration with the motion stage.

In the closing stage, the user is informed the robot is going to finish (either due to it finishing the route properly or it running out of time), one final set of results is recorded, the user is informed of the output file's location on their system, and then the code finishes.

3.2.1 Motion Planning and Execution

Given that the operating environment and locations of target objects are known prior to robot movement for contest 2, it is possible to pre-plan a route from coordinate to coordinate. Due to the tight time limit imposed for the contest, an optimal or near-optimal route was desired. The robot must visit every node exactly once, and then return to its starting place, forming a Hamilton circuit. This is analogous to solving the classical *travelling salesman problem* (TSP). The TSP can be solved optimally via brute force for small sets of nodes, but this approach is not computationally efficient. With *n* nodes and holding the starting point the same for all possible paths, the naive brute force algorithm must evaluate (n-1)! unique Hamilton circuits. Contest 2 consists of 10 target nodes and a fixed starting position (n = 11), resulting in ~4 million unique Hamilton circuits. This is a small enough number of permutations to make a brute force approach feasible for path planning in the time constrained environment of these contests.

The brute force algorithm for solving the TSP can be broken into two sections. A preprocessing step receives the pose of each object, generates a target node, and creates an adjacency matrix containing the cost between nodes. Object coordinates are supplied by the *boxCoords* vector. These coordinates are used to generate a target robot pose for image acquisition of each object in the *faceBoxPoint* function. This is necessary to ensure the robot is well-positioned to "see" the image; far enough away to capture most of the image and at an acceptable viewing angle. The function makes use of the *move_base* node's *make_plan* service to check for a reachable point in the neighbourhood of the target object. Starting from a point 0.35m directly in front of the image, the function evaluates points in a widening arc and distance, until it finds a feasible location for the robot to acquire images from. This generates a vector containing the pose of the robot at which it will view each object. The poses are the nodes for which the optimal Hamilton circuit will be found.

The poses generated by *faceBoxPoint* were used to fill an adjacency matrix. This is an 11 x 11 matrix, where the entry a_{ij} represents the cost of traversing between the i^{th} and j^{th} nodes. The cost associated with traversal was selected as the straight-line distance between points, which is reflective of the time taken to move between nodes given the mostly open environment. Each entry in the matrix could then be computed directly from the x and y-coordinates of the nodes as:

$$a_{ij} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

The second step for solving the TSP is to calculate the total cost of each possible route. A default *movePlan* vector containing the index of each node in numerical order is initialized. The cost of the default case is calculated by summing the entries of the adjacency matrix for the entries in *movePlan*. This cost is stored as the initial best cost. The set of all possible Hamilton circuits is iterated over using the *std::next_permutation* function. This function in the C++ standard library rearranges the elements in a range into the next lexicographically greater permutation, returning true if such a permutation exists, and false otherwise. Lexicographic ordering is an extension of

alphabetic ordering to include numbers and other symbols. The lexicographically smallest permutation has the elements sorted in ascending order, while the lexicographically largest permutation has the elements sorted in descending order. By starting with the lexicographically smallest ordering of the *movePlan* vector, *next_permutation* can be used to iterate through all possible Hamilton circuits. This function serves as the condition for a *while* loop, in which the cost of the current permutation is calculated and compared to the lowest cost. If the current cost is less than the lowest cost, the lowest cost becomes the current cost. This loop iterates until there are no lexicographically larger permutations of the set of nodes, at which point *next_permutation* returns *false*, and the loop terminates. The permutation with the lowest cost is passed out of the loop, cleaned up for use in the main control loop, and returned. The resulting output is a vector containing the indexes of the nodes in their optimal order of travel to, starting from the robot's starting position, then stopping to view each box, and then finally returning to its starting position.

Motion is executed using the provided *moveToGoal* function which attempts to autonomously drive the robot to a specified position and orientation. It was not altered by the team, however it was overloaded so the team could pass in vectors to represent the goal position instead of three separate floating point numbers.

A separate function *goToCoords*, wraps the use of *moveToGoal* so the team's motion failure recovery method could be implemented and edited in one place instead of multiple locations in the main code. Should *moveToGoal* fail to reach its destination in this function, the user is informed and the cost map is cleared before the motion is reattempted. Should that fail the user is informed before the function returns. The team was limited in its options to try a different "recovery" from the motion failure as motion command is limited to the provided *moveToGoal* function which does not provide facilities for direct control of the robot.

A derivative of *goToCoords* is *goToStop*. This simply takes in the index of which box point (stop) to go to, uses that to look up the coordinate vector for it and passes it to *goToCoords* to execute the motion with failure recovery.

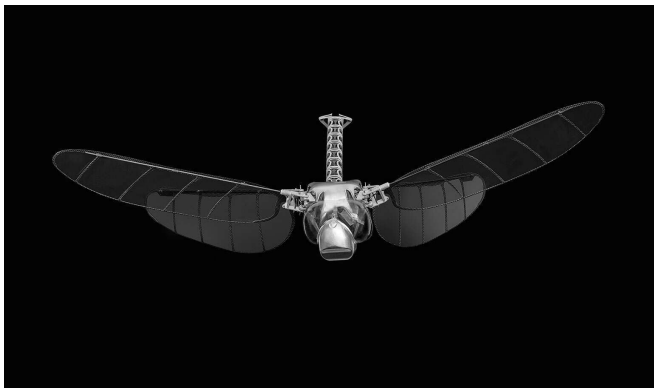
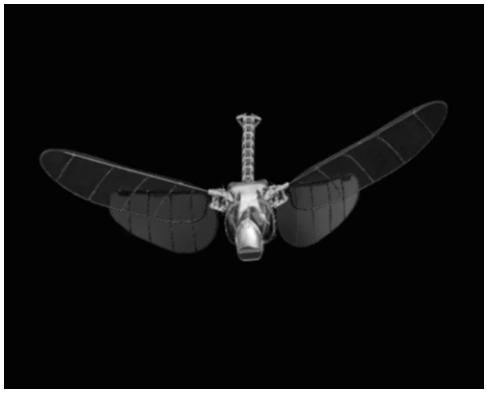
3.2.2 Image Processing and Recognition

Vision processing is an integral part of this robot as it is required to identify the tags on the boxes in the robot's operating environment. Vision is provided to the robot by the mounted Kinect sensor's visible light camera, which provides a colour (RGB) image 640 pixels wide by 480 pixels tall, at a rate of roughly 30 images (frames) per second. These images are then fed the team's own code based on the OpenCV library to locate tags. All code related to vision systems is contained in the heavily modified *imagePipeline.cpp* and *imagePipeline.h* files.

The team tailored the vision code for the robot for its service environment (a simulation) and task (identifying grayscale tags). This guided the team to exploit the very predictable and noise-free properties inherent to a simulation, namely the way lighting is modelled.

The vision recognition system begins with its initialization when the robot code is first executed. When initialized, the system subscribes to the raw video feed from the Kinect through ROS so it ensures that it has access to the most up to date images of the environment. It then pre-processes

and stores the reference image for each possible tag. This preprocessing consists of loading them in grayscale format, resizing the images to a ratio of 5 wide by 4 tall, then a slight Gaussian blur is applied.

Figures 5 (left) and 6 (right): The original tag image (5) provided to the robot and the processed result (6) stored for future use

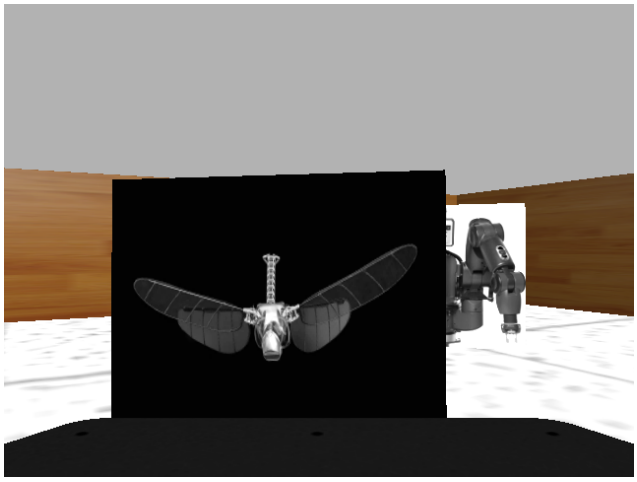
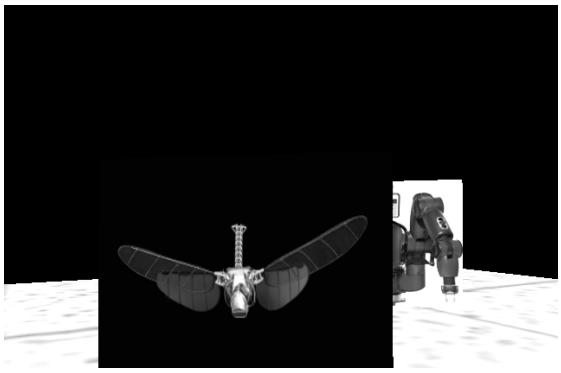
These effects create an image that will be closer to the expected input when encountered in the environment. Grayscaling is used to reduce the redundant information to process should it have been stored in RGB formatting. The 5:4 ratio is used to match the aspect ratio of the box sides (tags are stretched to this in the environment). Gaussian blur helps remove minor non-defining features that could cause issues in recognition such as false positive matches, and prepares the robot to deal with an off angle image.

These preprocessed tags are stored so that they do not need to be preprocessed with every instance of a scan, reducing computation time. Initially the tags were not stored and were dynamically prepared with every scan, this was slower and would cause segmentation faults when there were too many successive scans due to the memory intensive processes involved and there not being enough time for OpenCV to clear the irrelevant data.

When the robot needs to scan its view for potential tags, it takes in the most recent frame provided by the camera of the scene before the robot. This image is preprocessed before it goes into feature detecting and matching algorithms provided by OpenCV. The results are compared across all tags to determine which is the most likely match.

The image preprocessing is different to the preprocessing the tags undergo, although they both intend to simplify the matching process. While processing the tags focused on making them easier to identify in a scene from the simulator (generalizing them), the preprocessing of the environment image is focused on the removal of extraneous information for the algorithm so it may focus on potential tags. This is where the team exploits the simulator's lighting behaviour the most. It follows these steps:

1. Remove (crop) the lowest portion of the image.
2. Blank any coloured pixels. (The R, G, and B values for a gray pixel are all equal)
3. Convert to grayscale format.
4. Blank the skybox. (The simulator sky is a set uniform colour)

Figures 7 (left) and 8 (right): The original scene image the robot collected from the scene (7) and the result from preprocessing (8)

The bottom portion of the image is removed because it is always obstructed by the structure of the robot, thus providing the robot with no potential information on any tags. Removing it helps speed up the computations that follow. This portion is a set constant because the view is fixed within the robot, the Kinect does not get readjusted by users, nor is it reactive to any potential force that would adjust it in the simulation such as a collision.

The team exploited the uniform and noise free lighting as well as the "perfect" camera in the simulation to selectively blank out colour pixels. All the tags are grayscale, so their red, green, and blue pixel colour values are equal within a pixel. Any colour pixel (R, G, and B not all equal) is of no use to our task and can be safely removed to avoid it affecting the feature scanner and matcher. Shadows in the simulator do not affect this process as they proportionally darken all RGB values the same when a pixel is grayscale so they may go from all being 102 to 90, but they will all still be equal. These coloured pixels are set to pure black so they blend in with the black sides of the boxes (the sides without tags), thus avoiding creating unintentional edge features.

Conversion to grayscale is done to reduce the memory and processing needed to handle the image going forward. This is done after any colour is removed from the picture in the previous step so it has no effect other than reducing the amount of data needed to represent the image.

The skybox shade is removed from the top of the image (going from the top where the sky begins and going down each column until a non-sky pixel is encountered). Just with the coloured pixels, these are set to black to avoid creating additional features. This is possible only in the simulator since the skybox is a uniform mid-gray colour that does not blend into the edges of the tags (which are usually on light backgrounds).

Once the scene image is preprocessed it has its numerically identifying features detected and mapped. The team was constrained to using a SURF method for feature detections in this contest, although we were allowed to adjust its behavior.

The same detector is then applied to a tag to find and log its features in the same way. These sets of features are then passed to an OpenCV algorithm to find matching features between the two. If the portion of matching features to features present in a tag exceeds a threshold it is considered a possible match. If it is a possible match then an attempt is made to determine the bounding box of the tag in the scene using the matching features in homography, this box is used to determine the area a tag occupies in an image. If it fails to exceed a set threshold then the tag is no longer considered a possible match. If both thresholds are passed, the area and portion are combined into a match confidence value. This process is repeated for each potential tag.

The final step is to get the highest and second highest confidence values. The highest confidence is compared to the second highest value to ensure it exceeds it by a certain factor, so it is the "dominant" tag in the scene and recorded as the tag (match) identified in the scene. If there is no tag that exceeds the minimum confidence and is "dominant" enough, then a fail to identify is returned.

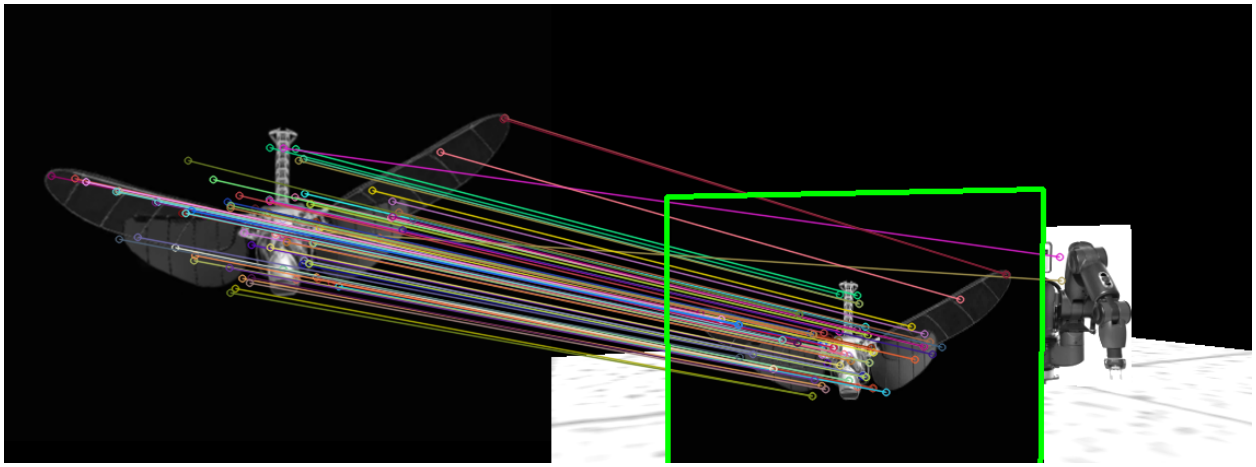


Figure 9: Side by side comparison of processed tag (left) identified as the dominant tag in the processed scene image (right). The green bounding box and lines connecting matching features were added to show the steps involved.

The figures used and additional intermediate images of the processing steps for vision processing are presented in higher resolution in Appendix A for tag processing and Appendix B for scene processing and matching. The figures shown with regards to vision processing are not immediately present when executing the code submitted for the competition to save time and resources, they can be made visible with minor edits to the code.

3.2.3 Outputting Data

As part of the contest requirements the team needed to prepare code to output the robot's findings. This was achieved with a function (*writeLog*) defined in the *fileWrite.cpp* and *fileWrite.h* files. The function receives the data relating to the boxes, the path taken, and the findings of the image scanning. It then prepares a text file (*.txt*) containing the results line by line for each box, in the order the robot visits them. As required by the competition, this data is the tag present on the box (if any), the coordinates of the box (x (m), y(m), and yaw (rad)), and if the robot is seeing it for the first time ("new") or a duplicate of a previous box encountered in its run

("dup"). For example, if tag 2 is first encountered in the run on a box at (1.94, -1.41, 0.788) the entry in the output file would be.

```
Tag  2 - ( 1.94, -1.41,  0.788) - new
```

This file is saved to the user's "Documents" folder in their Ubuntu home directory as *team22results.txt*. A full example output file is shown in Appendix C.

Initially the results were only recorded once the main loop was terminated under normal operation (either by completing the circuit between all boxes or exceeding the time limit). The team changed it to record its results after each vision scan so should the robot code execution be interrupted, there will at least be a file with the results collected prior to the interruption.

3.2.4 Test Files

Although disabled in the submitted contest code, test functions were created to assess the functionality of different aspects of the program during development. These tests are defined in *tests.cpp* and *tests.h*, and selectively enabled by *#define* macros in *tests.h* and recompiling. These tests were designed to be run once in the main loop and then terminate the program, therefore only one test could be executed at a time.

There are three tests that were developed for the systems subsystems: file output, navigation / path planning, and vision processing. They all begin with a warning to the terminal that the test is going to be performed and the robot code will terminate afterwards before executing the actual test.

The file output test has the robot output a file with some preset results. These results ensure that the robot can successfully output the file in the right location, with the right data (such as correct duplicate marks). This allowed the team to iterate through different

To test the path finding and navigation systems the robot is sent to visit each box in the order they are loaded into the program (not the optimized loop) before returning to the start. At each box the robot was to wait a few seconds so the end of one motion and the beginning of the next motion were distinct to an observer, as well as outputting a message to the terminal informing the user of its progress.

Vision testing was the most utilised of the three tests. The vision testing uses saved test images of the different tags as they appeared in the simulation that the team collected and labelled manually and feeds the images into the image scanner as though they are the current scene before the robot and records the results.

The vision test function iterates through all the test files present in the *testpics* folder, or a subset which have the search string (passed into the test function when called) present in their filenames. This search functionality allowed the team to quickly iterate the image processing code to test different cases and focus on problematic tags.

A notable tag that caused issues was *tag_12*, which initially was very difficult for the team's code to identify. Using the testing code and focusing on the *tag_12* test images the team was able to quickly iterate through candidate solutions until the best solution was found (resizing tag reference images to a 5:4 aspect ratio before running the image matching code) and implemented.

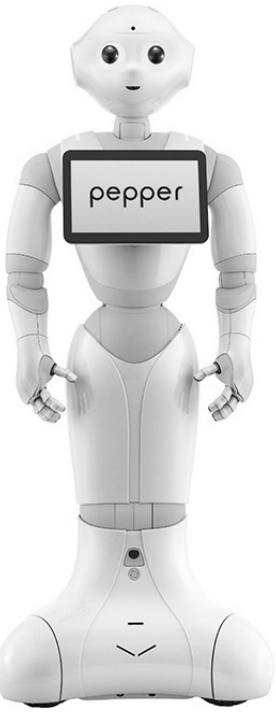


Figure 10: tag_12 reference image as provided to the team/robot

Appendix D contains some examples of the notable tests the vision system code had to analyze and the successful results. There were between five and fifteen test images prepared for each tag to test the system from various angles and distances relative to the box. Some tests also introduced additional challenges that include but are not limited to: multiple potential boxes, rotated tags, and only a portion of the tag being visible.

3.2.5 Other Supporting Code Used

The team was provided with code to hold odometry data and store it in *robot_pose.cpp* and *robot_pose.h*. The *RobotPose* class is defined in these which holds the robot's current x, y position and orientation as *phi*. A callback function is also defined so this class may subscribe to a ROS topic to be updated regularly. These files were not altered from their original state.

The team was also provided with code to load in the box and tag data and store it for use in the team's code in *boxes.cpp* and *boxes.h*. These files were also not altered from how they were provided. The coordinates of each box is stored as a vector in a vector containing all boxes *coords*, the reference images for each tag are stored in a vector of OpenCV Mat structures called *templates*.

4.0 Future Recommendations

One particular issue the team would have liked more time to solve involves the navigation near and around locations in the map where two or more objects create a narrow corridor, as shown in the practice environment in Figure 11 below. In this situation, the robot is required to navigate between the objects in order to properly scan the image (tag) face of the top box; however, there is very little space for the robot to do so. The objects are close enough that using the AMCL and navigation algorithms to move the robot proves difficult in this region due to the dead zones around objects that the robot cannot navigate to (i.e., the locations are deemed unreachable by the program). Thus, the robot struggles to get close to the object and to capture a good enough image for the vision system. While testing in the practice environment, this portion of the navigation required the most time, approximately 30 seconds to 1 minute. The team attempted to resolve the issue by changing parameters such as the offset of the robot from the face of objects, but they still experienced the same issue. With more time, the team would have looked into alternative solutions to make this section more efficient. If this type of situation occurs in the trials for the contest, the robot may struggle during navigation, costing the team valuable time.

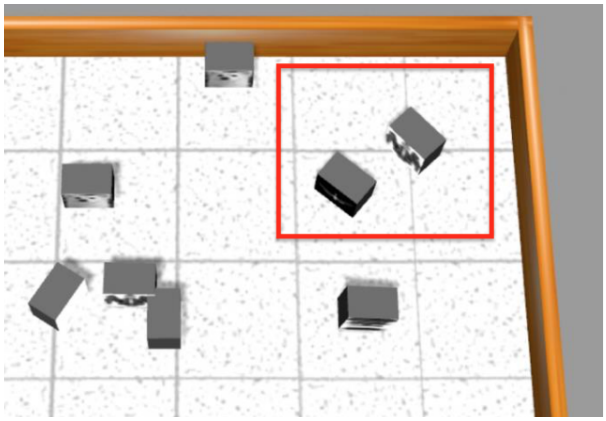


Figure 11: Region (red box) causing navigational issues in practice map

Another aspect that should be considered for future iterations of this project is in regards to how the team solved the traveling salesman problem. The team implemented a brute force algorithm to find all the possible Hamilton circuits of the 10 objects and then to select the most optimal path. Although this technique is guaranteed to find an optimal solution, the disadvantage is that it is inefficient due to the computation time required to solve it, especially as the number of objects increases. With 10 objects, the program takes 3-5 seconds to complete the computation; this is minimal in comparison with the maximum time limit of 8 minutes. However, if one more object was added (11 objects in total), the computation time would be 30-50 seconds, requiring a significant portion of the time limit. If a second object was added (12 objects in total), the computation time could vary from 6 to 10 minutes. At this point, the brute force method would be too time consuming. If the team's algorithm were to be used in a scenario that included more than 10 objects, the code would no longer be feasible. Instead, another method would have to be implemented in that scenario, such as the Nearest Neighbour Method. This technique sacrifices efficiency, in terms of path planning, with computation speed as it only needs to check the optimal path locally.

Lastly, the image processing system employed in the program could be modified for future considerations. The current algorithm utilizes the fact that the system is in a simulated and relatively unchanging environment (meaning that the walls, ground, sky, etc. are not modified from the practice world to the trial world). The team's image processing works by removing certain aspects of the captured image (walls, sky, etc.) to make identification easier and quicker. If the environment were to be changed, either another simulated environment or the real world, these aspects of the pre-processing phase would not work as they are specifically designed for the contest's environment. Thus, the vision system code would have to be modified significantly for different environments, including those that may slightly shift the colour of the walls/sky.

5.0 Contribution Table

Contributions noted by value: 1 - small amount, 3 - majority, blank for none.

Table 1: Contribution Table

Section	**Savo Bajic**	**Maximilian Glidden**	**Catherine Kucaba**
1.0			3
2.0	2		3
3.1		3	
3.2	3	2	
4.0			3
Robot Code	3	2	1

6.0 Appendices

Appendix A: Tag Preprocessing

The tag is preprocessed using the following steps, with an example of the intermediate result after each step shown as a figure below.

1. Load in original reference image
2. Resize to 5:4 aspect ratio (500 x 400 pixels is used in code)
3. Apply Gaussian blur
4. Store result for future reference

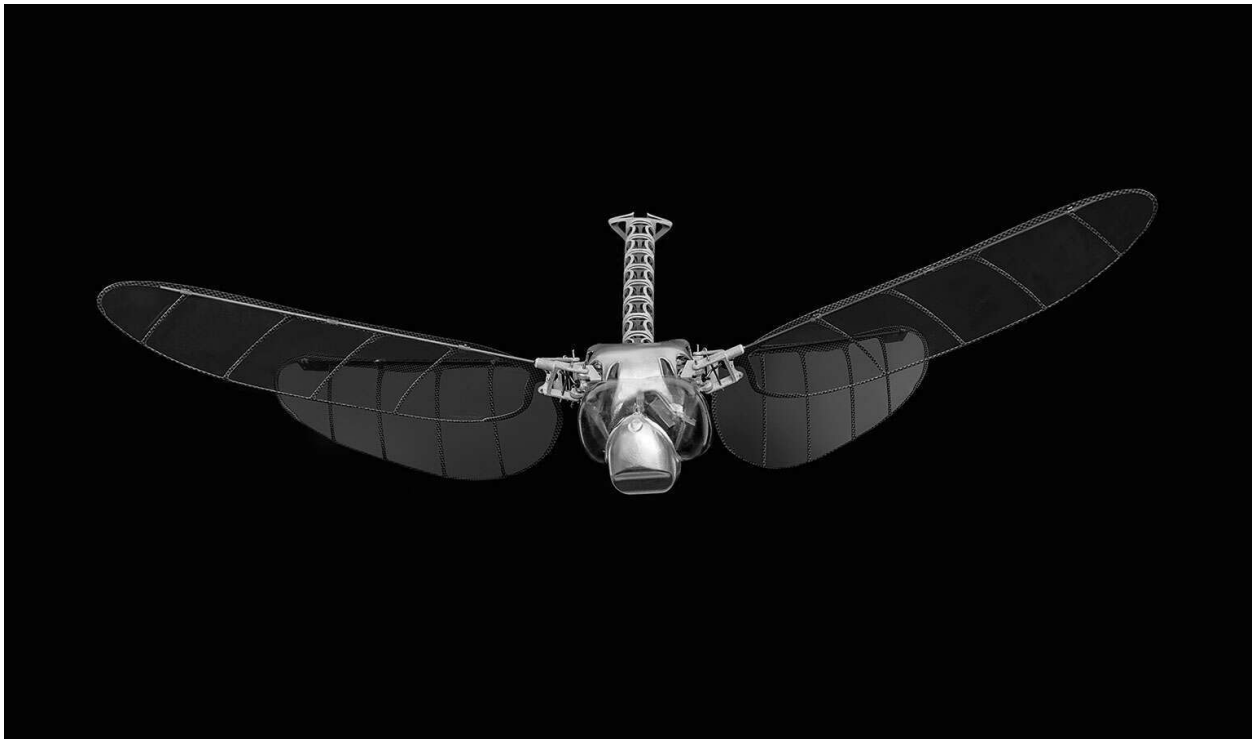


Figure 12: An original tag reference image provided to robot

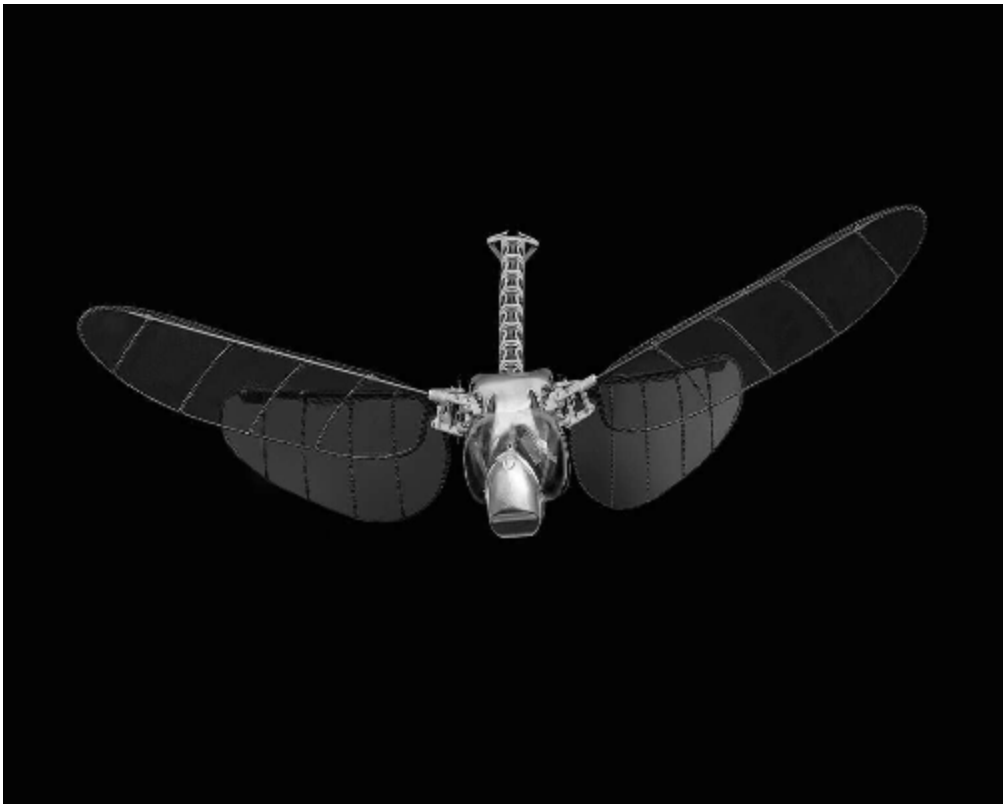


Figure 13: Image resized to 5:4 aspect ratio (500 by 400 pixels)

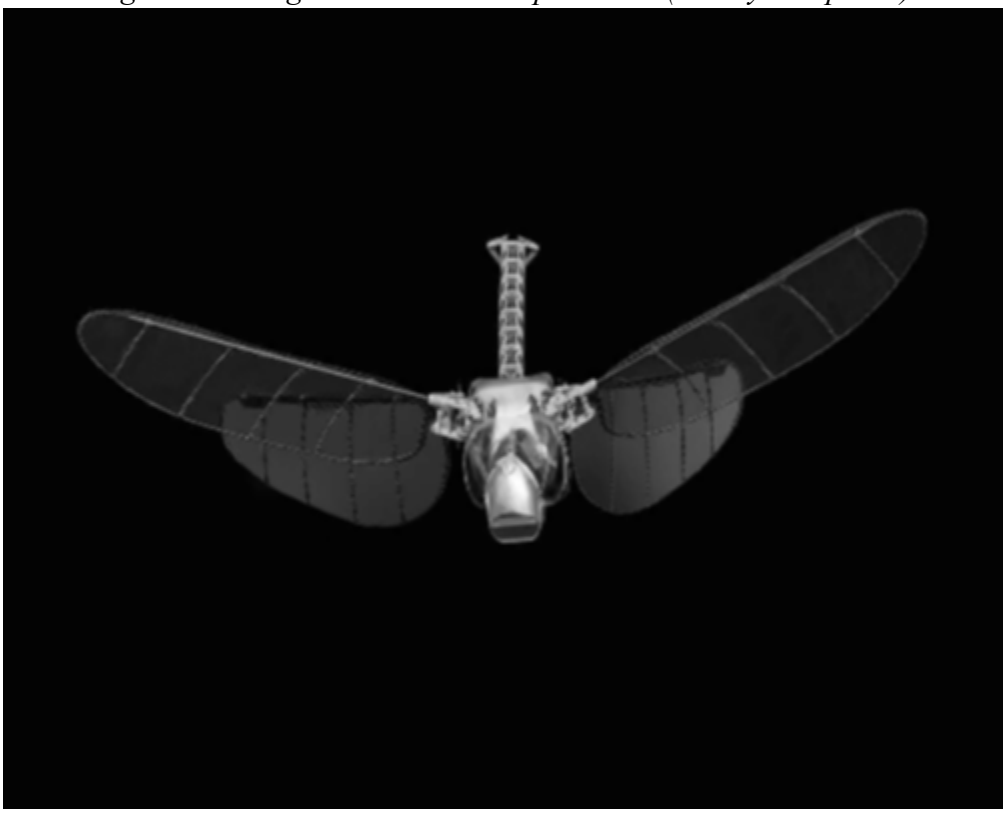


Figure 14: Gaussian blur applied. Image is stored.

Appendix B: Scene Processing

The scene is preprocessed using the following steps, with an example of the intermediate result after each step shown as a figure below.

1. Load in image from scene (640 x 480 pixels)
2. Crop image to remove bottom portion (lowest 60 rows of pixels)
3. Remove (blank) colour pixels
4. Convert to grayscale (omitted from figures due to it appearing the same as the previous step's result)
5. Remove (blank) the sky

Once processed the image has its features numerically determined for matching to the stored tags. The result of a successful match is included. This shows the bounding box for the dominant tag spotted in the scene as well as an overlay of the matching features detected and connected between the reference and scene images.

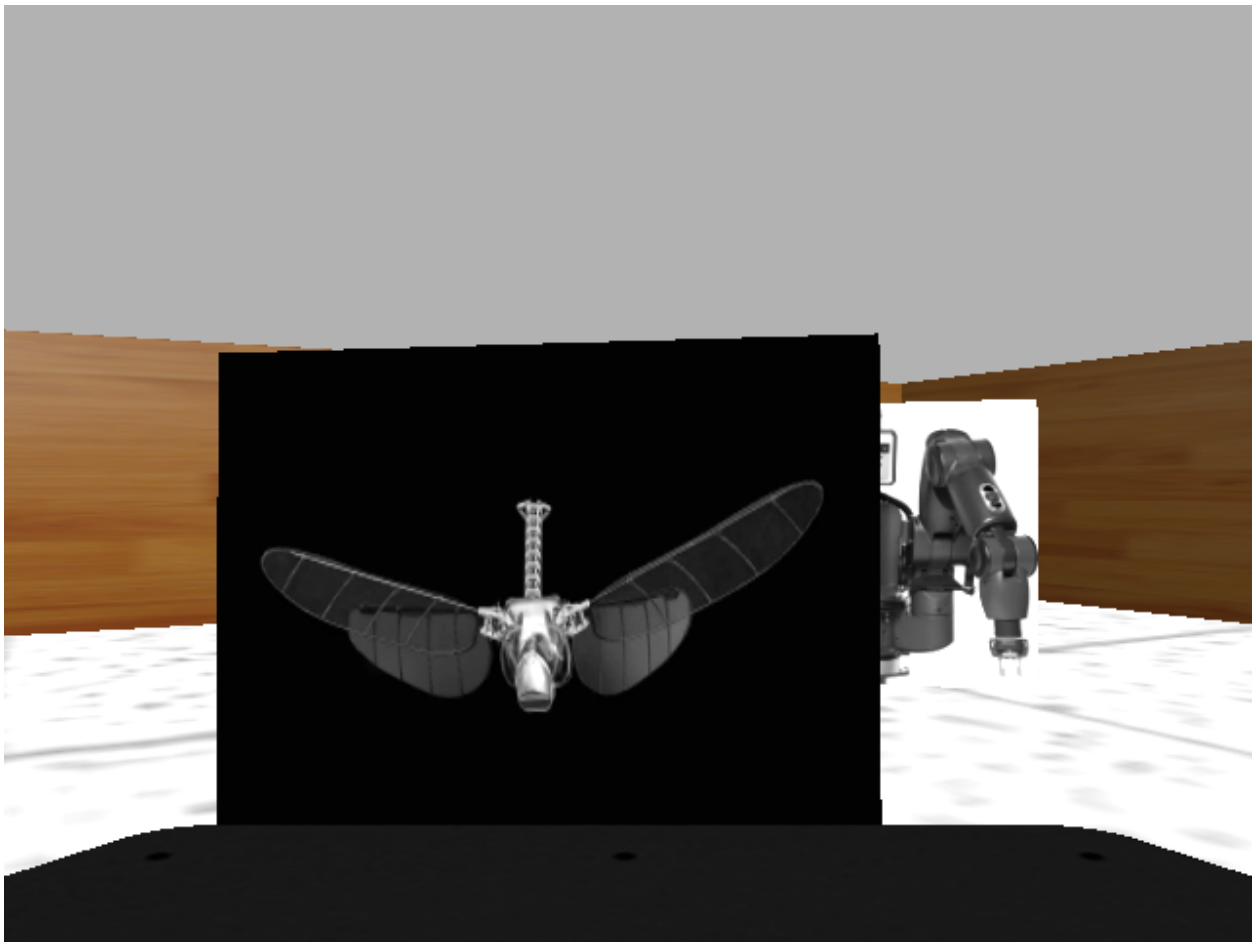


Figure 15: Original image of scene from Kinect sensor (640 pixels wide by 480 tall)

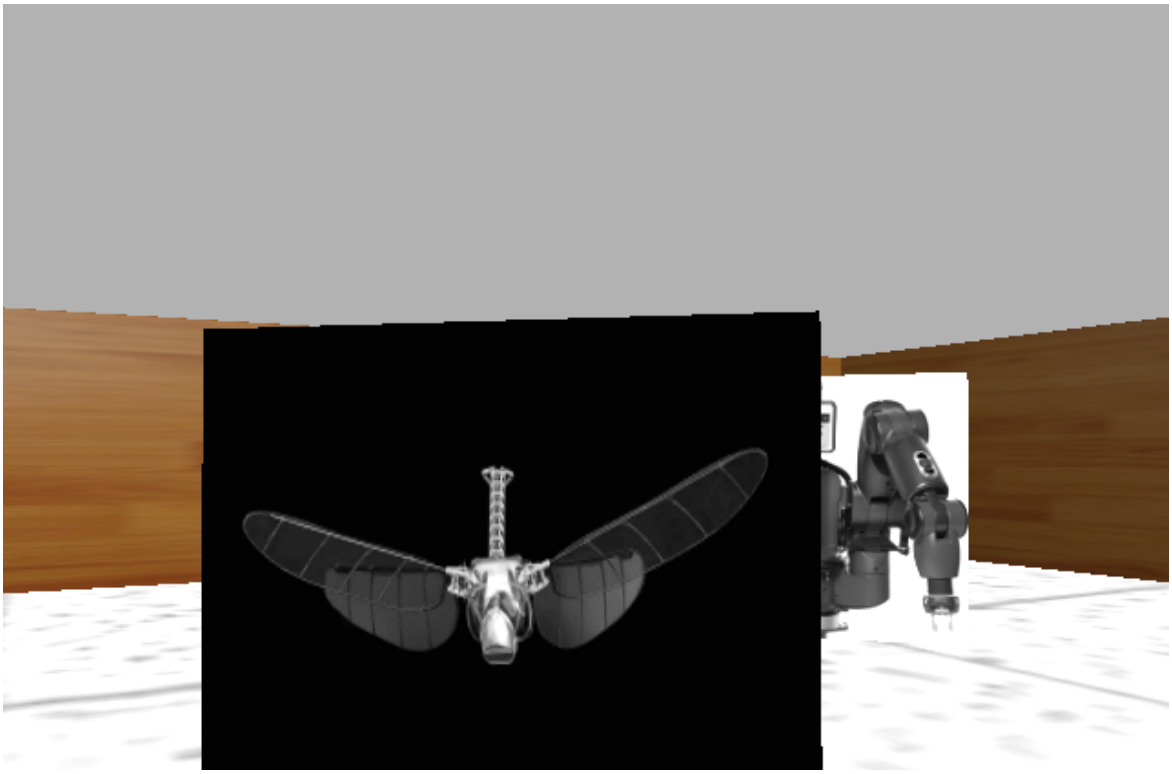


Figure 16: Cropped image to remove bottom portion (640 x 420 pixels)

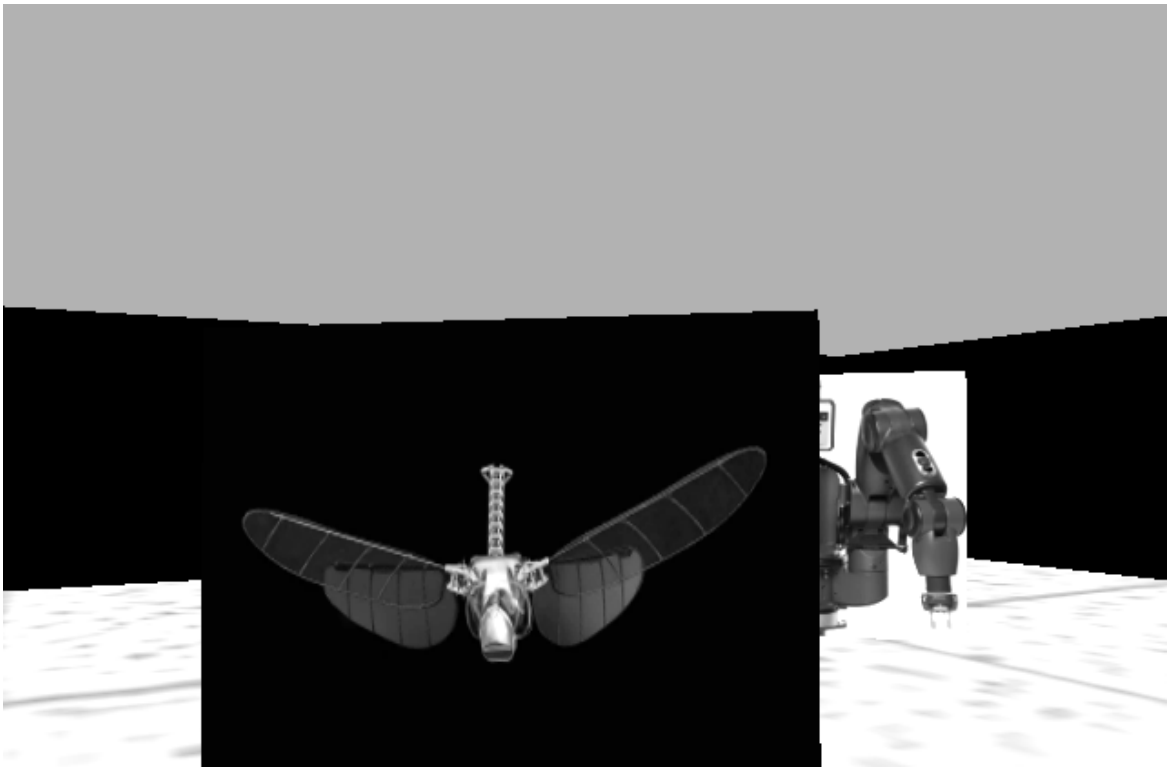


Figure 17: Coloured pixels are blanked (removed) by setting them to pure black. The image is functionally grayscaled at this point before converting to grayscale data formatting.

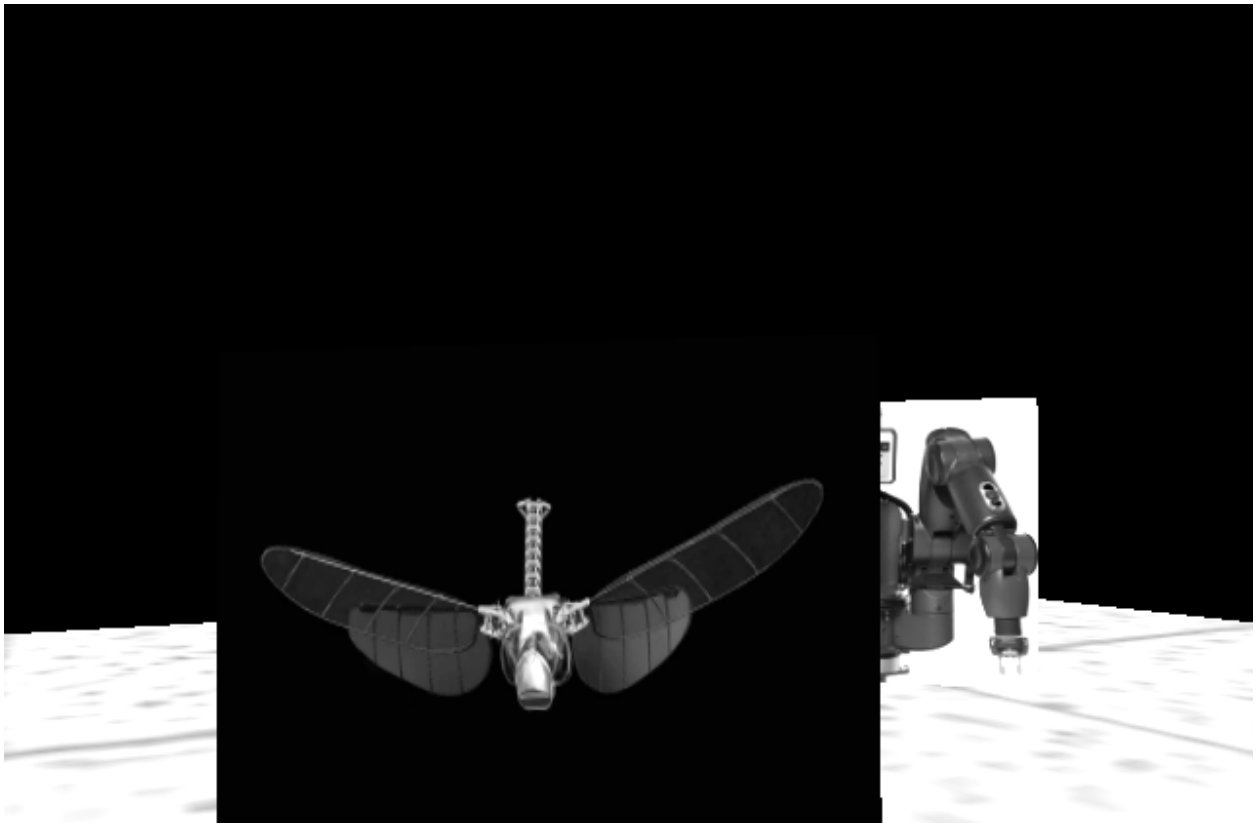


Figure 18: Sky is blanked (removed) from the image

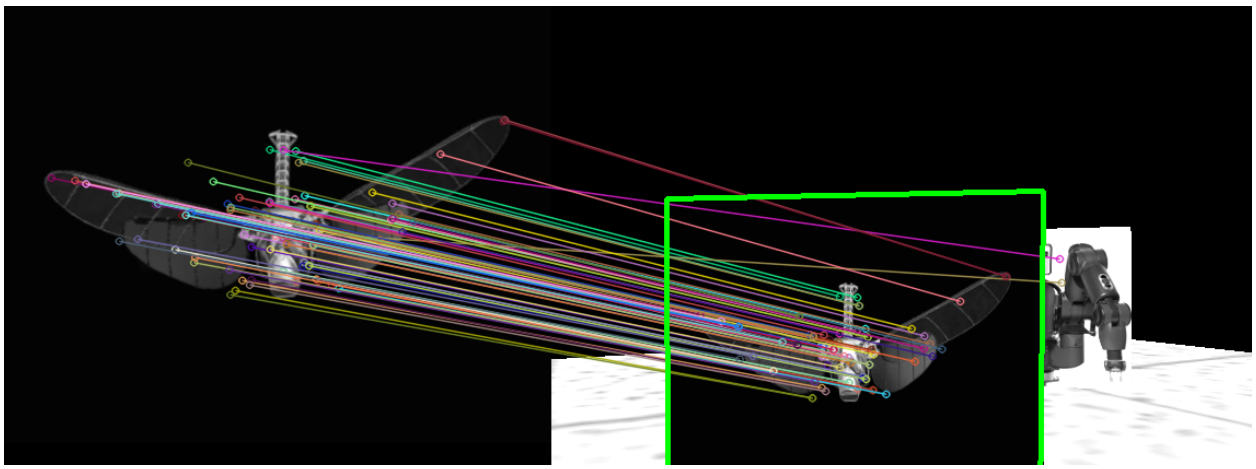


Figure 19: Demonstration of a match. Reference image on the left, processed scene on the right. Green bounding box is put to outline the dominant match, matching features between the two images are linked by thin coloured lines.

Appendix C: Example Output File

Below are the contents of an example output file from the robot. These results are from a successful run through the practice map for this contest.

```
Tag ID (0 for no ID) - Coordinates (x (m), y (m), yaw (rad)) - new or
duplicate (dup)

Tag 14 - ( 0.18,  0.59, -3.033) - new
Tag 15 - (-2.00, -2.14,  2.347) - new
Tag  8 - ( 0.38, -2.74,  2.331) - new
Tag 11 - ( 1.31, -4.41,  1.617) - new
Tag 10 - ( 1.53, -3.57,  1.636) - new
Tag  5 - ( 3.19, -3.76,  2.330) - new
Tag  5 - ( 3.35, -1.85,  2.397) - dup
Tag  0 - ( 2.75, -0.90,  3.060) - new
Tag  2 - ( 1.94, -1.41,  0.788) - new
Tag 11 - ( 2.34, -1.38,  2.347) - dup
```

Appendix D: Notable vision tests

Below are some examples of notable test files used to verify the vision system's capabilities and the results of using them.

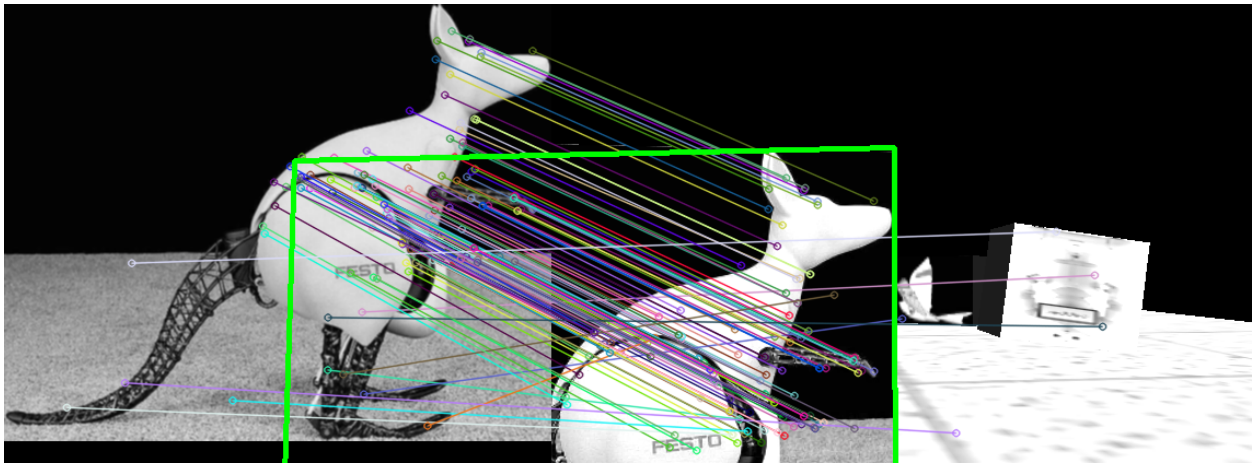


Figure 20: A tag that is partially visible and has other tags visible in their entirety in the background (the bounding box is extrapolated out of the scene image and thus overlaps the reference image on the left)

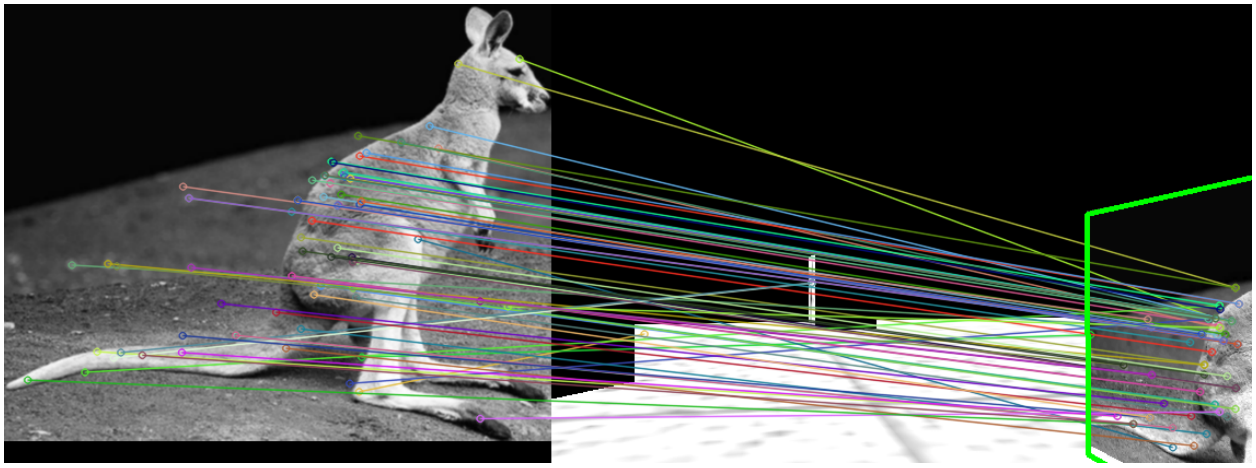


Figure 21: A tag that is not facing the robot and is only partially visible still manages to be properly identified

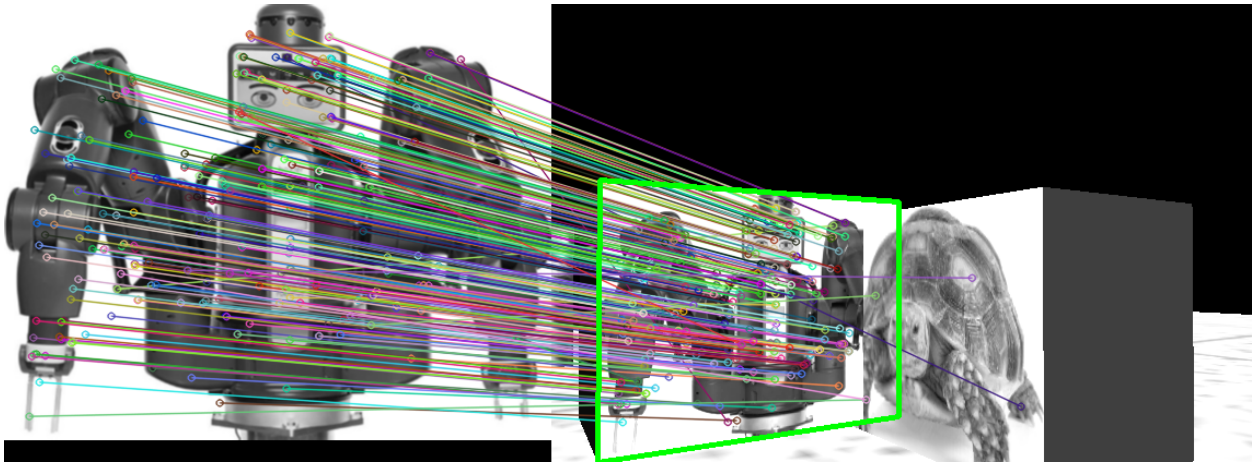


Figure 22: Two tags are adjacent and occupy a similar amount of visual area.

This figure above was primarily used as a test by the team to ensure the robot would only mark down a match if it was clearly dominant (largest one, therefore closest) in the image. Since there are two similarly dominant tags in this image, the system should have not determined a "winning" match. However, since the system decided that there was a tag present, this means the system failed this case in this instance.

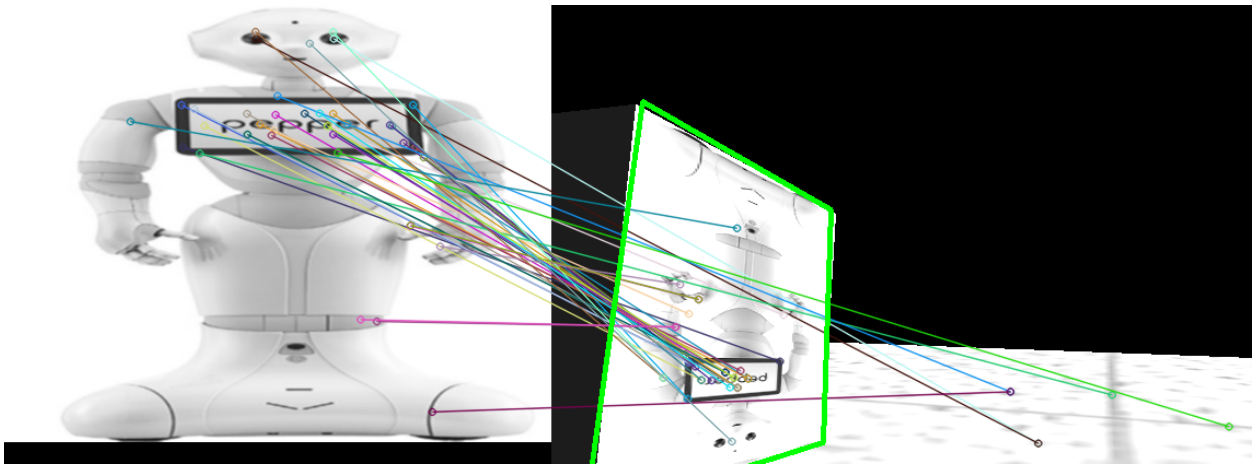


Figure 23: A test to see how the robot handles rotated tags in the scene. This test was not originally planned, however this image was recorded when a simulation had a glitch that rotated this box and was kept going forward.

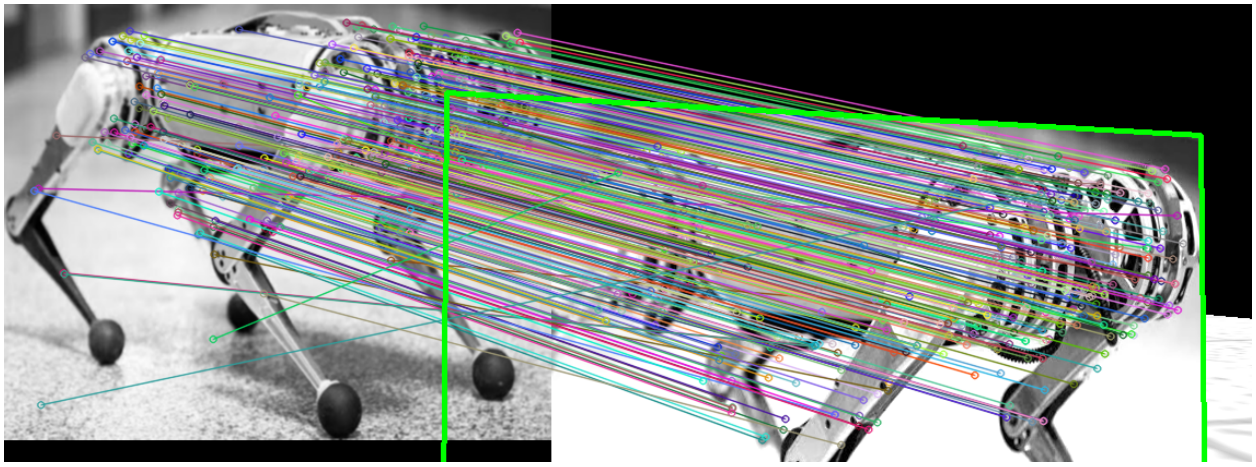


Figure 24: Testing a feature rich tag to a feature rich scene. This was a stress test used to ensure the vision processing code would not potentially run out of memory in such cases where there is a lot of data present.

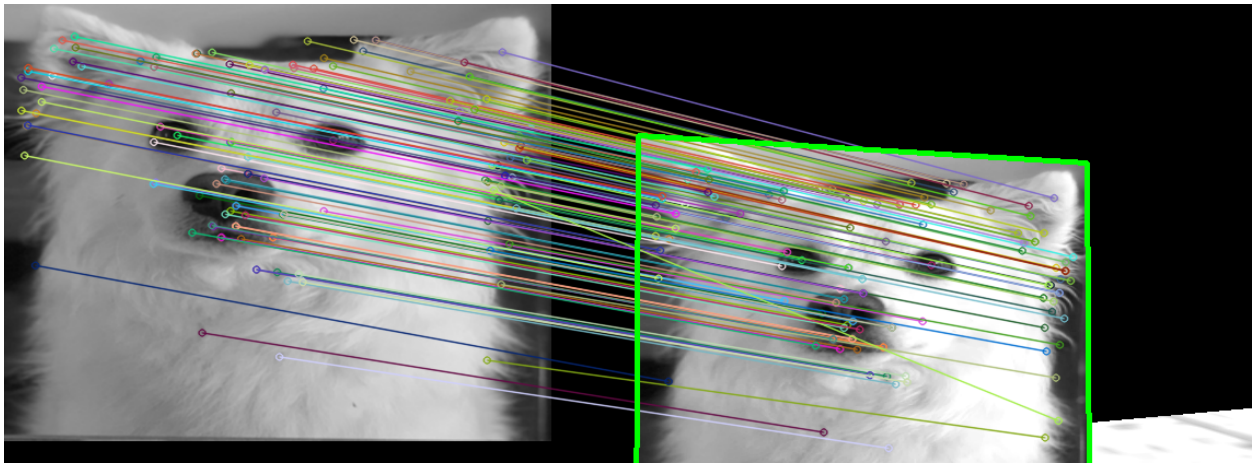


Figure 25: A basic test with a feature poor tag to a feature poor scene. Used to ensure the system did not favour tags that simply had more features that although had a smaller portion of their features match, resulting in a larger absolute count of matches.

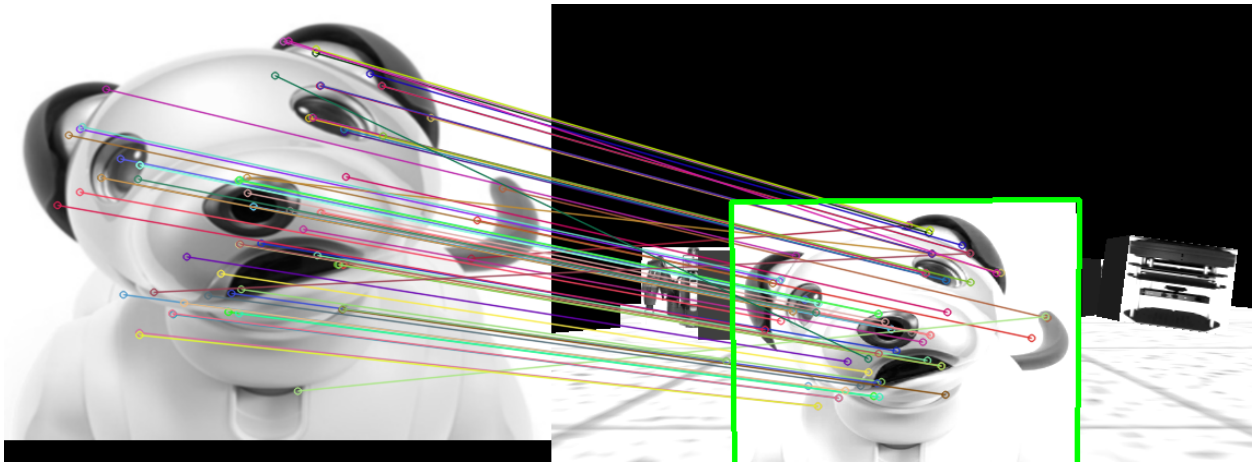


Figure 26: Testing to ensure a feature poor tag is selected over other tags present in the background.

Appendix E: contest2.cpp

```
#include <boxes.h>
#include <navigation.h>
#include <robot_pose.h>
#include <imagePipeline.h>
#include <cmath>
#include <algorithm>
#include "fileWrite.h"
#include <time.h>
#include "tests.h"
#include "pathPlanning.h"

const int timeLimit = 8 * 60; // Time limit in seconds

int main(int argc, char** argv) {
   // Monitor time elapsed
   time_t startTime = time(NULL);
   float secondsElapsed = 0;

   // Initialize box coordinates and templates
   Boxes boxes;
   if(!boxes.load_coords() || !boxes.load_templates()) {
       ROS_FATAL("Could not load coords or templates");
       return -1;
   }
   else {
       ROS_INFO("Box coordinates loaded successfully:");
       std::cout << "Box #\tx (m)\ty (m)\tyaw (rad)\n"; // Header

       // Output data for each box nicely
       for(int i = 0; i < boxes.coords.size(); ++i) {
           char buffer[100];
           sprintf(buffer, "%3d\t%5.2f\t%5.2f\t%6.3f\n", i,
               boxes.coords[i][0], boxes.coords[i][1],
boxes.coords[i][2]);
           std::cout << buffer;
       }
   }
```

```
    // 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 image object and subscriber.
    ImagePipeline imagePipeline(n, boxes);

    // Spin a couple times to sync properly
    for (int i = 0; i < 3; i++) {
        ros::Duration(0.1).sleep();
        ros::spinOnce();
    }

    // Record starting position
    std::vector<float> startPosition(3);
    startPosition[0] = robotPose.x;
    startPosition[1] = robotPose.y;
    startPosition[2] = robotPose.phi;
    ROS_INFO("Starting position:\n\tx: %5.2f\ty: %5.2f\tyaw: %5.2f",
startPosition[0], startPosition[1], startPosition[3]);

    // Initialize path planner
    pathPlanning pathPlanned(n, boxes, startPosition, true); // Path data

    // Variable to record identification of boxes
    std::vector<int> boxIDs(boxes.coords.size()); // Recoding IDs of each
box
    int currentStop = 0;

    while(ros::ok() && (secondsElapsed < timeLimit)) {
        ros::spinOnce();

        // ====================================================
        // Tests for features, these will only be executed once
        // Configured in "tests.h"
```

```
#ifdef FILE_WRITE_TEST
	fileWriteTest(boxes, bestRoute, false);
	return 0;
#endif
#ifdef VISION_SAMPLES_TEST
	// Leave search term for vision as "" for all test cases
	visionSystemTest("", boxes, imagePipeline, true);
	return 0;
#endif
#ifdef MOTION_TEST
	navigationSystemTest(pathPlanned);
	return 0;
#endif

	// ========================================================
	// Actual loop code
	int boxNumber = pathPlanned.idealOrder[currentStop]; // Stores
current box index

	// ========================================================
	// Motion code
	// Start by checking if it is time to return to start and terminate

	if (currentStop == pathPlanned.idealOrder.size()) {
	    ROS_WARN("Done all stops, returning to start.");
	    bool atEnd = pathPlanned.goToCoords(pathPlanned.startCoord);

	    // Check if it has successfully returned to the start
	    if (atEnd) ROS_WARN("Reached end in %.1f seconds\n",
secondsElapsed);
	    else ROS_ERROR("FAILED TO REACH END! (%.1f seconds
elapsed).\n\tTerminating anyways.\n", secondsElapsed);

	    break; // Break out of while loop (to terminate)
	}

	// Inform us what step is started and target
	ROS_INFO("Starting step %d, target is box %d.", currentStop + 1,
boxNumber);
```

```
        // Do normal motion otherwise
        bool atSpotToScan = pathPlanned.goToStop(currentStop);

        if (atSpotToScan) ROS_INFO("Reached stop %d (box %d)", currentStop
+ 1, boxNumber);
        else ROS_ERROR("Failed to reach stop %d (box %d)", currentStop + 1,
boxNumber);

        if (atSpotToScan) {
            // Perhaps some code to try and align bot with target even if
unreachable
        }

        // ========================================================
        // Vision code
        // Scan regardless of if we succcessfully reached destination or
not
        // since we might be lucky and be able to see it from where we
stand

        ros::spinOnce(); // Update video feed after moving

        // Check if there is something present
        int ID = imagePipeline.getTemplateID(boxes, false);

        if (ID >= 0) boxIDs[currentStop] = ID; // Good scan (error-free)
        else {
            // Handle error... or don't
        }

        // ========================================================
        // Record results
        // Every step, so in the event of interruption our progress isn't
lost
        writeLog(boxes, pathPlanned.idealOrder, boxIDs);

        // ========================================================
        // Increment to next step
        currentStop++;
        std::cout << "\n\n"; // Make a break in terminal between stops
```

```
        // Sleep and record elapsed time
        ros::Duration(0.1).sleep();
        secondsElapsed = time(NULL) - startTime;
    }

    // ========================================================
    // Handle proper closure

    if (secondsElapsed >= timeLimit) {
        // Time's up handle proper closure
        ROS_WARN("TIME'S UP! (%.1f of %d seconds)\n RECORDING OUTPUT AND
TERMINATING.\n", secondsElapsed, timeLimit);
    }

    writeLog(boxes, pathPlanned.idealOrder, boxIDs, true); // Write results
before closing (with terminal output)

    ROS_FATAL("Ending now. Goodbye.\n");
    return 0;
}
```

Appendix F: boxes.h

```
#pragma once

#include <opencv2/highgui/highgui.hpp>
#include <tf/transform_datatypes.h>
#include <tf/transform_broadcaster.h>
#include <vector>

class Boxes {
   public:
       std::vector<cv::Mat> templates;
       std::vector<std::vector<float> > coords;
   public:
       bool load_coords();
       bool load_templates();
};
```

Appendix G: boxes.cpp

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

bool Boxes::load_coords() {
   std::string filePath = ros::package::getPath("mie443_contest2") +
                          std::string("/boxes_database/coords.xml");
   cv::FileStorage fs(filePath, cv::FileStorage::READ);
   if(fs.isOpened()) {
       cv::FileNode node;
       cv::FileNodeIterator it, end;
       std::vector<float> coordVec;
       std::string coords_xml[10] = {"coordinate1", "coordinate2",
"coordinate3", "coordinate4",
                                     "coordinate5", "coordinate6",
"coordinate7", "coordinate8",
                                     "coordinate9", "coordinate10"};
       for(int i = 0; i < 10; ++i) {
           node = fs[coords_xml[i]];
```

```
            if(node.type() != cv::FileNode::SEQ) {
                std::cout << "XML ERROR: Data in " << coords_xml[i]
                             << " is improperly formatted - check input.xml"
<< std::endl;
            } else {
                it = node.begin();
                end = node.end();
                coordVec = std::vector<float>();
                for(int j = 0; it != end; ++it, ++j) {
                    coordVec.push_back((float)*it);
                }
                tf::Quaternion q(coordVec[3], coordVec[4], coordVec[5],
coordVec[6]);
                tf::Matrix3x3 m(q);
                double roll, pitch, yaw;
                m.getRPY(roll, pitch, yaw);
                auto boxCoords = std::vector<float>();
                boxCoords.push_back(coordVec[0]);
                boxCoords.push_back(coordVec[1]);
                boxCoords.push_back(yaw);
                if(coordVec.size() == 7) {

                    coords.push_back(boxCoords);
                } else {
                    std::cout << "XML ERROR: Data in " << coords_xml[i]
                                 << " is improperly formatted - check
input.xml" << std::endl;
                }
            }
        }
        if(coords.size() == 0) {
            std::cout << "XML ERROR: Coordinate data is improperly
formatted - check input.xml"
                         << std::endl;
            return false;
        }
    } else {
        std::cout << "Could not open XML - check FilePath in " << filePath
<< std::endl;
        return false;
```

```
    }
    return true;
}

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

Appendix H: fileWrite.h

```
#pragma once

#include <fstream>
#include "boxes.h"
#include <vector>

const std::string logfile = "/Documents/team22results.txt"; // Where to
record (relative to home)

// Function to write results to a file
void writeLog(Boxes boxList, std::vector<int> movePlan, std::vector<int>
boxIDs, bool printInfo = false);
```

Appendix I: fileWrite.cpp

```
#include "fileWrite.h"

void writeLog(Boxes boxList, std::vector<int> movePlan, std::vector<int>
boxIDs, bool printInfo) {

   // Opens/creates log file (clears the contents if there was one prior
to this)
   char * homeDir = std::getenv("HOME"); // Get home directory for user
   std::string fileLocation = homeDir + logfile; // Combines with the
prefered file path

   std::ofstream outputFile;
   outputFile.open(fileLocation.c_str(), std::ios_base::out |
std::ios_base::trunc);

   if (outputFile.is_open()) {

       std::string headerText = "Tag ID (0 for no ID) - Coordinates (x
(m), y (m), yaw (rad)) - new or duplicate (dup)\n\n";
       outputFile << headerText;
       if (printInfo) {
           ROS_INFO("Results of our run.\n");
```

```
            std::cout << headerText;
        }

        // ================================================
        // Read box IDs and their coordinates into the file

        // Records if a template has already appeared (plus one for blank
(at index 0))
        bool alreadyTagged[boxList.templates.size() + 1];
        for (int i = 0; i < (boxList.templates.size() + 1); i++)
alreadyTagged[i] = false;

        // Go through each entry and record things
        for (int i = 0; i < boxIDs.size(); i++) {

            // Record tag ID for that stop
            char tagText[10];
            sprintf(tagText,"Tag %2d", boxIDs[i]);

            // Record duplicate status
            char dupText[7];
            if (alreadyTagged[boxIDs[i]] == true) {
                sprintf(dupText, "dup");
            }
            else {
                sprintf(dupText, "new");
                alreadyTagged[boxIDs[i]] = true; // Mark down it has
already been listed
            }

            // Record coordinates for the stop from move list
            std::vector<float> curCoords = boxList.coords[movePlan[i]];
            char coordText[25];
            sprintf(coordText, "(%5.2f, %5.2f, %6.3f)", curCoords[0],
curCoords[1], curCoords[2]);

            // Output the entry
            char outputBuffer[150];
            sprintf(outputBuffer, "%s - %s - %s\n", tagText, coordText,
dupText);
```

```
            outputFile <<  outputBuffer; // Write to file
            if (printInfo) std::cout << outputBuffer;
        }
        std::cout << std::endl; // Add space after results

        outputFile.close(); // Must close file once complete

        ROS_INFO_COND(printInfo, "File with results written to:\n\n%s\n",
fileLocation.c_str());
    }
    else {
        // File failed to open
        ROS_FATAL("Unable to prepare output file at:\n\n%s\n",
fileLocation.c_str());
    }
}
```

Appendix J: imagePipeline.h

```
#pragma once

#include <image_transport/image_transport.h>
#include <std_msgs/String.h>
#include <opencv2/core.hpp>
#include <cv.h>
#include <cv_bridge/cv_bridge.h>
#include <boxes.h>
#include "opencv2/features2d.hpp"
#include "opencv2/xfeatures2d.hpp"
#include "opencv2/highgui.hpp"
#include <stdio.h>
#include <iostream>
#include "opencv2/calib3d.hpp"
#include "opencv2/imgproc.hpp"

class ImagePipeline {
   private:
       cv::Mat img;
       bool isValid;
       image_transport::Subscriber sub;

       uint8_t skyVal = 178; // RGB value of the skybox
       const uint8_t removeVal = 0; // Value used to overwrite pixels we
don't care for

       // Checks if the lines plotted by the four corners don't intersect
       bool checkTangledBox(std::vector<cv::Point2f> corners);

       // See if the test point lies above the line between points A and B
       bool checkAbove(cv::Point2f test, cv::Point2f a, cv::Point2f b);

       // Used to produce an output of the reference image in the scene
based on existing matches and points
       cv::Mat drawSceneMatches(cv::Mat &scene, cv::Mat &refImage,
std::vector<cv::DMatch> &matches,
           std::vector<cv::KeyPoint> &keyPointsRef,
std::vector<cv::KeyPoint> &keyPointsScene);
```

```
        // Searches for an image in a scene
        void searchInScene(cv::Mat &refImage, cv::Mat &descriptorsScene,
std::vector<cv::KeyPoint> &keypointsObject,
            std::vector<cv::DMatch> &goodMatches,
cv::Ptr<cv::xfeatures2d::SURF> &detector);

        void tagPreprocess(cv::Mat &tag); // Preprocess image for tagging
(result saved to image)
    public:
        float reqConfRatio   = 1.20; // Ratio between max and second to
make a conclusion
        float reqConfMinimum = 0.07; // Minimum threshold to be considered
conclusive

        float reqMinArea = 0.0; // Minimum area needed to be considered
        float areaConfidenceFactor = 0.00005; // Factor used to multiply
area by before multiplying that to confidence

        // Confidence = (matching features / reference's features) * area *
areaFactor

        int minHessian = 350;

        ImagePipeline(ros::NodeHandle& n, Boxes& boxes);
        void imageCallback(const sensor_msgs::ImageConstPtr& msg);
        int getTemplateID(Boxes& boxes, bool showInternals = true);
        void loadImage(std::string fileLocation, bool printMessage = true);
// Used for testing by loading in our own images
};
```

Appendix K: imagePipeline.cpp

```
#include <imagePipeline.h>

#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, Boxes &boxes) {
   image_transport::ImageTransport it(n);
   sub = it.subscribe(IMAGE_TOPIC, 1, &ImagePipeline::imageCallback,
this);
   isValid = false;

   // Preprocress all tags and store in boxes so this only has to happen
once
   for (int i = 0; i < boxes.templates.size(); i++) {
       tagPreprocess(boxes.templates[i]);
   }
}

void ImagePipeline::tagPreprocess(cv::Mat &tag) {
   cv::resize(tag,tag, cv::Size(500,400));               // Resize to
roughly match aspect ratio on boxes
   cv::GaussianBlur(tag, tag, cv::Size( 3, 3), 0, 0);    // Add blur to
aid feature matching
   //cv::imshow("Tag as used", tagImage); // Show image used in search
}

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) {
       ROS_ERROR("Could not convert from %s to %s!",
msg->encoding.c_str(),  IMAGE_TYPE.c_str());
       isValid = false;
   }
```

```
}

int ImagePipeline::getTemplateID(Boxes& boxes, bool showInternals) {
   int determinedId = -1; // Default to error

   if(!isValid) {
       ROS_ERROR("INVALID IMAGE!");
       return determinedId;
   } else if(img.empty() || img.rows <= 0 || img.cols <= 0) {
       ROS_ERROR("VALID IMAGE, BUT STILL A PROBLEM EXISTS!");
       std::cout << "\timg.empty():" << img.empty() << std::endl;
       std::cout << "\timg.rows:" << img.rows << std::endl;
       std::cout << "\timg.cols:" << img.cols << std::endl;
       return determinedId;
   }

   // ============================================================
   // Preprocesing of incoming image
   img = img(cv::Rect(0,0,640,420)); // Crop out the constant lip of the
rover at the bottom

   // Black out coloured pixels (currently only walls)
   uint8_t* pixelPtr = (uint8_t*)img.data;
   int cn = img.channels();
   uint8_t bgrPixel[3];

   for(int i = 0; i < img.rows; i++) {
       for(int j = 0; j < img.cols; j++) {
           bgrPixel[0] = pixelPtr[i*img.cols*cn + j*cn + 0]; // B
           bgrPixel[1] = pixelPtr[i*img.cols*cn + j*cn + 1]; // G
           bgrPixel[2] = pixelPtr[i*img.cols*cn + j*cn + 2]; // R

           // Check if its greyscale (R=B=G), blank them if they aren't
           if ((bgrPixel[0] == bgrPixel[1]) && (bgrPixel[0] ==
bgrPixel[2])) continue;
           else {
               pixelPtr[i*img.cols*cn + j*cn + 0] = removeVal;
               pixelPtr[i*img.cols*cn + j*cn + 1] = removeVal;
               pixelPtr[i*img.cols*cn + j*cn + 2] = removeVal;
           }
```

```
        }
    }

    // Convert image from RGB to greyscale space
    // This should reduce memory usage and computation time
    cv::cvtColor(img, img, cv::COLOR_RGBA2GRAY, 0);

    // Remove sky
    // The sky is a uniform colour (in our sim 178, 178, 178) and always
starts from the top until it is
    // interrupted by an object, none of which have pixels of that value
along the top

    // skyVal is defined in the header, but it can be dynamically set
before this is desired.
    pixelPtr = (uint8_t*)img.data; // Update the pointer before this loop
    for (int j = 0; j < img.cols; j++) {
        for (int i = 0; i < img.rows; i++) {
            // Go column by column from top to bottom until no longer in
the sky

            // Check if its greyscale (R=B=G), blank them if they aren't
            if (pixelPtr[i*img.cols + j] != skyVal) break; // Hit an
object, go to next column
            else {
                pixelPtr[i*img.cols + j] = removeVal;
            }
        }
    }

    //cv::imshow("Processed view. Press any key to continue.", img); //
Show result of preprocessing

    // ============================================================
    // Setup scan of image
    using namespace cv;
    using namespace cv::xfeatures2d;

    // Setup the SURF detector for features in images and associated data
    Ptr<SURF> detector = SURF::create( minHessian ); // Defined in header
```

```
   std::vector<KeyPoint> keyPointsObject, keyPointsScene;
   Mat descriptorsScene;

   // Determine features for the scene before looping through options
   detector->detectAndCompute( img, noArray(), keyPointsScene,
descriptorsScene );
   if (showInternals) {
       printf("\tScene has %d keypoints\n", (int) keyPointsScene.size());
   }

   // Find the ID and confidence levels of the two highest rated
candidates
   float maxConfidence = 0.0, secondConfidence = 0.0;
   uint8_t maxID = 0;
   Mat bestTag; // Stores the best matched reference tag for display
purposes

   // ============================================================
   // Loop through all possible tags
   for (int tagID = 0; tagID < boxes.templates.size(); tagID++) {

       // Load reference to tag from boxes (just to help simplfy the code
that follows)
       Mat &tagImage = boxes.templates[tagID];

       // See what portion of features from the reference are matched in
the scene
       std::vector<DMatch> goodMatches;
       searchInScene(tagImage, descriptorsScene, keyPointsObject,
goodMatches, detector);
       float confidence = (float)goodMatches.size() /
(float)keyPointsObject.size();

       // ============================================================
       // Investigate futher if initial confidence is good
       float area = 0; // Area object takes up in scene (pixels)
       if (confidence > reqConfMinimum) {
           // Localize the object
           std::vector<Point2f> refPoints;
           std::vector<Point2f> scenePoints;
```

```
            for(int i = 0; i < goodMatches.size(); i++) {
                // Get the keypoints from the good matches

refPoints.push_back(keyPointsObject[goodMatches[i].queryIdx].pt);

scenePoints.push_back(keyPointsScene[goodMatches[i].trainIdx].pt);
            }

            // Determine transformation matrix of reference to scene pixels
            Mat H = findHomography(refPoints, scenePoints, RANSAC);

            // Check if there is a possible transform
            if (H.empty()) {
                // Failed to find a transform from reference to scene
                ROS_WARN("Unable to transform perspective using reference
%d.", tagID + 1);
                confidence = 0; // It's a bad match
            }
            else {
                // Tranform from refence image to scene is possible

                // Get the corners from the reference image
                std::vector<Point2f> cornersInReference(4);
                cornersInReference[0] = Point2f(0, 0);
                cornersInReference[1] = Point2f((float)tagImage.cols, 0 );
                cornersInReference[2] = Point2f((float)tagImage.cols,
(float)tagImage.rows );
                cornersInReference[3] = Point2f(0, (float)tagImage.rows );
                std::vector<Point2f> corInScene(4);

                // Apply transform to reference corners to transform into
scene bounds
                cv::perspectiveTransform( cornersInReference, corInScene,
H);

                // Check if the corners are not "tangled" before
calculating area (forming a "bow" shape (invalid))
                if (checkTangledBox(corInScene) == false) {
                    // Calculate area of the region
```

```
                    // (1/2) * [(x1y2 + x2y3 + x3y4 + x4y1) - (x2y1 + x3y2
+ x4y3 + x1y4)]
                    area = corInScene[0].x * corInScene[1].y +
corInScene[1].x * corInScene[2].y +
                            corInScene[2].x * corInScene[3].y + corInScene[3].x
* corInScene[0].y;
                    area = area - (corInScene[1].x * corInScene[0].y +
corInScene[2].x * corInScene[1].y +
                            corInScene[3].x * corInScene[2].y + corInScene[0].x
* corInScene[3].y);
                    area = area / 2;
                }
            }
        }

        // ===========================================================
        // Look to record this match if it's worthy

        if (area <= reqMinArea) confidence = 0; // Nullify confidence if it
isn't present
        else confidence = confidence * area * areaConfidenceFactor; // Add
area to confidence
        // Area is added to prefer objects that are closer to rover but
might have some of their features out of
        // frame, resulting in a lower "confidence" than a fully visible,
but futher object in the background

        if (showInternals) {
            printf("Template %2d - Confidence %6.2f%% - KP %4d / %4d - Area
%6.0f\n",
                tagID + 1, confidence * 100.0, (int)goodMatches.size(),
(int) keyPointsObject.size(), area);
        }

        // See how this compares to previous cases
        if (confidence > maxConfidence) {
            // New best
            secondConfidence = maxConfidence;
            maxConfidence = confidence;
```

```
            // Record values needed outside the loop
            maxID = tagID;
            bestTag = tagImage.clone();
        }
        else if (confidence > secondConfidence) {
            // Record second place confidence for ratio comparison later
            secondConfidence = confidence;
        }
    }

    // ============================================================
    // Process the results of the scan
    determinedId = 0; // Default (inconclusive scan, but at least no
error!)

    if (maxConfidence < reqConfMinimum) {
        // If there is no satisfactory option
        ROS_INFO("Failed to find a match.");
    }
    else if ((maxConfidence > reqConfMinimum) && ((maxConfidence /
secondConfidence) > reqConfRatio)) {
        determinedId = maxID + 1; // Add one to match file names and to
allow 0 to be used as a fail code

        ROS_INFO("Image contains %d, %.2f%% (%.2f) confidence",
determinedId,
            maxConfidence * 100.0, (maxConfidence / secondConfidence));

        if (showInternals) {
            // Redo winning search
            std::vector<DMatch> goodMatches;
            searchInScene(bestTag, descriptorsScene, keyPointsObject,
goodMatches, detector);

            // Show resulting matches
            Mat imgOfMatches = ImagePipeline::drawSceneMatches(img,
bestTag, goodMatches, keyPointsObject, keyPointsScene);
            imshow("Selected match", imgOfMatches);
```

```
            cv::waitKey(250); // Wait until any key is pressed or 250ms
pass
        }
    }

    return determinedId;
}

void ImagePipeline::searchInScene(cv::Mat &tagImage, cv::Mat
&descriptorsScene, std::vector<cv::KeyPoint> &keyPointsObject,
        std::vector<cv::DMatch> &goodMatches,
cv::Ptr<cv::xfeatures2d::SURF> &detector) {

    using namespace cv;
    Mat descriptors_object;

    // Detect markers for the reference to find in the scene
    detector->detectAndCompute( tagImage, noArray(), keyPointsObject,
descriptors_object );

    // Matching descriptor vectors with a FLANN based matcher
    // Since SURF is a floating-point descriptor NORM_L2 is used
    Ptr<DescriptorMatcher> matcher =
DescriptorMatcher::create(DescriptorMatcher::FLANNBASED);
    std::vector< std::vector<DMatch> > knn_matches;
    matcher->knnMatch( descriptors_object, descriptorsScene, knn_matches, 2
);

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

void ImagePipeline::loadImage(std::string fileLocation, bool printMessage)
{
```

```
    // Replace image in pipeline with something else
    img = cv::imread(fileLocation, 1);
    isValid = true;

    if (printMessage) ROS_INFO("Image loaded from into video
feed.\n\t\"%s\"", fileLocation.c_str());
}

cv::Mat ImagePipeline::drawSceneMatches(cv::Mat &scene, cv::Mat &tagImage,
std::vector<cv::DMatch> &matches,
    std::vector<cv::KeyPoint> &keyPointsRef, std::vector<cv::KeyPoint>
&keyPointsScene){

    using namespace cv;

    // Draw matches
    Mat imageOfMatches; // Image with matches illustrated
    drawMatches(tagImage, keyPointsRef, scene, keyPointsScene, matches,
imageOfMatches, Scalar::all(-1),
                    Scalar::all(-1), std::vector<char>(),
DrawMatchesFlags::NOT_DRAW_SINGLE_POINTS );
    // Localize the object
    std::vector<Point2f> refPoints;
    std::vector<Point2f> scenePoints;
    for( size_t i = 0; i < matches.size(); i++ ) {
        // Get the keypoints from the good matches
        refPoints.push_back( keyPointsRef[ matches[i].queryIdx ].pt );
        scenePoints.push_back( keyPointsScene[ matches[i].trainIdx ].pt );
    }

    Mat H = findHomography(refPoints, scenePoints, RANSAC );

    if (!H.empty()) {
        // Can preform transform from reference to scene

        float refImageCol = (float)tagImage.cols;

        // Get the corners from the reference image (the object to be
"detected")
        std::vector<Point2f> cornersInReference(4);
```

```
        cornersInReference[0] = Point2f(0, 0);
        cornersInReference[1] = Point2f( refImageCol, 0 );
        cornersInReference[2] = Point2f( refImageCol, (float)tagImage.rows
);
        cornersInReference[3] = Point2f( 0, (float)tagImage.rows );
        std::vector<Point2f> corInScene(4);

        cv::perspectiveTransform( cornersInReference, corInScene, H);

        // Draw lines between the corners of the spotted object (reference)
in the scene
        cv::line( imageOfMatches, corInScene[0] + Point2f(refImageCol, 0),
                corInScene[1] + Point2f(refImageCol, 0), Scalar(0, 255, 0),
4 );
        cv::line( imageOfMatches, corInScene[1] + Point2f(refImageCol, 0),
                corInScene[2] + Point2f(refImageCol, 0), Scalar( 0, 255,
0), 4 );
        cv::line( imageOfMatches, corInScene[2] + Point2f(refImageCol, 0),
                corInScene[3] + Point2f(refImageCol, 0), Scalar( 0, 255,
0), 4 );
        cv::line( imageOfMatches, corInScene[3] + Point2f(refImageCol, 0),
                corInScene[0] + Point2f(refImageCol, 0), Scalar( 0, 255,
0), 4 );
    }
    else {
        ROS_WARN("Can't draw matches. Corners cannot be transformed.");
    }

    return imageOfMatches;
}

bool ImagePipeline::checkTangledBox(std::vector<cv::Point2f> corners) {

    // Check that line between 1 and 4 does not cross 23
    bool tempA = checkAbove(corners[0],corners[1],corners[2]);
    bool tempB = checkAbove(corners[3],corners[1],corners[2]);
    bool result = tempA == tempB; // Store if both points fall on the same
side (not tangled)

    // Check that line 12 does not cross 34
```

```
    tempA = checkAbove(corners[0],corners[3],corners[2]);
    tempB = checkAbove(corners[1],corners[3],corners[2]);
    result = result && (tempA == tempB); // Update result to ensure no sets
of lines intersect

    return !result; // Return true if the system IS tangled
}

bool ImagePipeline::checkAbove(cv::Point2f test, cv::Point2f a,
cv::Point2f b) {
    // Define line between a and b
    float gradient = (a.y - b.y) / (a.x - b.x);

    // Linearly extrapolate line between a and b to test point
    float dx = test.x - a.x;
    float estimateY = a.y + dx * gradient;

    // Return if the point lies above the line or not
    return test.y > estimateY;
}
```

Appendix L: navigation.h

```
#pragma once
#include <vector>

class Navigation {
   public:
       static bool moveToGoal(float xGoal, float yGoal, float phiGoal);
       static bool moveToGoal(std::vector<float> goal); // Overloaded to
simplify our code which uses vectors to store coordinates
};
```

Appendix M: navigation.cpp

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

bool Navigation::moveToGoal(float xGoal, float yGoal, float phiGoal){
   // Set up and wait for actionClient.
   actionlib::SimpleActionClient<move_base_msgs::MoveBaseAction>
ac("move_base", true);
   while(!ac.waitForServer(ros::Duration(5.0))){
       ROS_INFO("Waiting for the move_base action server to come up");
   }
   // Set goal.
   geometry_msgs::Quaternion phi =
tf::createQuaternionMsgFromYaw(phiGoal);
   move_base_msgs::MoveBaseGoal goal;
   goal.target_pose.header.frame_id = "map";
   goal.target_pose.header.stamp = ros::Time::now();
   goal.target_pose.pose.position.x =  xGoal;
   goal.target_pose.pose.position.y =  yGoal;
   goal.target_pose.pose.position.z =  0.0;
   goal.target_pose.pose.orientation.x = 0.0;
   goal.target_pose.pose.orientation.y = 0.0;
   goal.target_pose.pose.orientation.z = phi.z;
   goal.target_pose.pose.orientation.w = phi.w;
```

```
    ROS_INFO("Sending goal location ...");
    // Send goal and wait for response.
    ac.sendGoal(goal);
    ac.waitForResult();
    if(ac.getState() == actionlib::SimpleClientGoalState::SUCCEEDED){
        ROS_INFO("You have reached the destination");
        return true;
    } else {
        ROS_INFO("The robot failed to reach the destination");
        return false;
    }
}
bool Navigation::moveToGoal(std::vector<float> goal) {
    // Overload to simplify our commands
    return Navigation::moveToGoal(goal[0], goal[1], goal[2]);
}
```

Appendix N: pathPlanning.h

```
#pragma once

#include <vector>
#include "boxes.h"
#include <robot_pose.h>
#include <ros/ros.h>
#include <tf/transform_datatypes.h>
#include <nav_msgs/GetPlan.h>
#include <std_srvs/Empty.h>
#include "navigation.h"

// define marcos used solely for constants
#define RAD2DEG(rad) ((rad) * 180. / M_PI)
#define DEG2RAD(deg) ((deg) * M_PI / 180.)

class pathPlanning {
   private:
       float deg2rad(float angle);
       float rad2deg(float angle);

       // Setting up points constants when approaching boxes
       const float offsetAngleLimit = DEG2RAD(50.0);
       const float offsetAngleStep = DEG2RAD(10.0);

       const float offsetDistStart = 0.35; // Any less the 0.35and the
rover gets stuck
       const float offsetDistStep = 0.05;
       const float offsetDistLimit = 0.5;

       // Internal copies/references so they don't need to contantly be
passed in
       ros::NodeHandle nh;
       std::vector<std::vector<float> > boxCoordList;

       double loopCost(double **adjMatrix, std::vector<int> movePlan);
       std::vector<int> findOptimalPath(bool printResult); // Returns best
route as box IDs to take
```

```
   public:
        std::vector<float> startCoord;
        std::vector<int> idealOrder; // Holds the order to take through the
world
        std::vector<std::vector<float> > stopCoords; // Stores the
coordinates of each stop corrisponding to a box

        pathPlanning(ros::NodeHandle& n, Boxes boxesIn, std::vector<float>
startPosition, bool printStuff = false);
        bool clearCostMap();
        std::vector<float> faceBoxPoint(int boxIndex);
        bool checkPossible(std::vector<float> goalCoord, bool printStuff =
false);

        bool goToCoords(std::vector<float> target);
        bool goToStop(int index);
};
```

Appendix O: pathPlanning.cpp

```
#include "pathPlanning.h"

float pathPlanning::deg2rad(float angle) {
   return (angle * M_PI) / 180.0;
}
float pathPlanning::rad2deg(float angle) {
   return (angle * 180.0) / M_PI;
}

bool pathPlanning::goToStop(int index) {
   return goToCoords(stopCoords[idealOrder[index]]);
}

bool pathPlanning::goToCoords(std::vector<float> target) {

   bool gotThere = Navigation::moveToGoal(target);

   // Handle initial failure
   if (!gotThere) {
       ROS_INFO("Initial attempt failed, clearing cost table.");

       clearCostMap(); // Clear cost map and reattempt motion
       gotThere = Navigation::moveToGoal(target);

       // Absolute failure
       if (!gotThere) ROS_ERROR("Failed to reach point (%5.2f, %5.2f,
%6.3f)", target[0], target[1], target[2]);
   }

   return gotThere;
}

pathPlanning::pathPlanning(ros::NodeHandle& n, Boxes boxesIn,
std::vector<float> startPosition, bool printStuff) {
   nh = n;
   startCoord = startPosition;
   boxCoordList = boxesIn.coords;
```

```
    //Initialize list
    for (int i = 0; i < boxCoordList.size(); i++) {
        stopCoords.push_back(faceBoxPoint(i));
    }

    idealOrder = findOptimalPath(printStuff);
}

std::vector<float> pathPlanning::faceBoxPoint(int boxIndex) {
    std::vector<float> output(3,0);

    std::vector<float> boxCoords = boxCoordList[boxIndex];

    // Generate points starting from middle and then going outwards up to a
limit
    // Also gradually increase distance as needed
    for (float offAngle = 0; offAngle <= offsetAngleLimit; offAngle =
offAngle + offsetAngleStep) {
    for (float offDist = offsetDistStart; offDist <= offsetDistLimit;
offDist = offDist + offsetDistStep) {


            // Adjust position coordinates based off box face
            output[0] = boxCoords[0] + offDist * cosf(boxCoords[2] +
offAngle);
            output[1] = boxCoords[1] + offDist * sinf(boxCoords[2] +
offAngle);

            // Set angle to face the point
            output[2] = boxCoords[2] + offAngle;
            if (output[2] > 0) output[2] = output[2] - M_PI;
            else output[2] = output[2] + M_PI;

            // Adjust increment for next iteration
            if (offAngle > 0) offAngle = 0.0 - offAngle;       // Flip from
positive to negative
            else offAngle = (0.0 - offAngle) + offsetAngleStep;   // Flip
and add increment (once positive again)

            // Check if the plotted point is valid
```

```
            bool validPoint = checkPossible(output);

            if (validPoint) return output; // Return first possible point
        }
    }

    // Alert user to failure
    ROS_ERROR("No valid location to offset to.\n\tPoint: (%5.2f, %5.2f.
%5.1f)\n\tOffset Dist.: %5.2f m\tOffset Angle: %5.2f",
        boxCoords[0], boxCoords[1], boxCoords[2], offsetDistLimit,
rad2deg(offsetAngleLimit));

    return boxCoords; // Return the input if failed to find a point
}

bool pathPlanning::clearCostMap() {
    // Clear cost map using the service
    std_srvs::Empty srv;
    ros::ServiceClient clear =
nh.serviceClient<std_srvs::Empty>("move_base/clear_costmaps");
    bool callExecuted = clear.call(srv);

    if (callExecuted) ROS_INFO("Cleared cost map");

    return callExecuted;
}

bool pathPlanning::checkPossible(std::vector<float> goalCoord, bool
printStuff) {

    bool callExecuted, validPlan;

    nav_msgs::GetPlan srv;

    // Set start position
    geometry_msgs::PoseStamped start;
    geometry_msgs::Quaternion phi =
tf::createQuaternionMsgFromYaw(startCoord[2]);
    start.header.frame_id = "map";
    start.pose.position.x = startCoord[0];
```

```
    start.pose.position.y = startCoord[1];
    start.pose.position.z = 0;
    start.pose.orientation.x = 0;
    start.pose.orientation.y = 0;
    start.pose.orientation.z = phi.z;
    start.pose.orientation.w = phi.w;

    // Set goal position
    geometry_msgs::PoseStamped goal;
    phi = tf::createQuaternionMsgFromYaw(goalCoord[2]);
    goal.header.frame_id = "map";
    goal.pose.position.x = goalCoord[0];
    goal.pose.position.y = goalCoord[1];
    goal.pose.position.z = 0;
    goal.pose.orientation.x = 0;
    goal.pose.orientation.y = 0;
    goal.pose.orientation.z = phi.z;
    goal.pose.orientation.w = phi.w;

    // Set up the service and call it
    ros::ServiceClient checkPath =
nh.serviceClient<nav_msgs::GetPlan>("move_base/NavfnROS/make_plan");

    srv.request.start = start;
    srv.request.goal = goal;
    srv.request.tolerance = 0.0;
    callExecuted = checkPath.call(srv);

    // Output print statments
    if(!callExecuted){
        ROS_ERROR("Call to check plan NOT sent");
    }

    if(srv.response.plan.poses.size() > 0){
        validPlan = true;
        ROS_INFO_COND(printStuff, "Successful plan.\n\tStart: (%5.2f,
%5.2f. %6.3f)\n\tGoal:  (%5.2f, %5.2f. %6.3f)",
            startCoord[0], startCoord[1], startCoord[2], goalCoord[0],
goalCoord[1], goalCoord[2]);
    }
```

```
    else{
        validPlan = false;
        ROS_INFO_COND(printStuff, "Unsuccessful plan.\n\tStart: (%5.2f,
%5.2f. %6.3f)\n\tGoal:  (%5.2f, %5.2f. %6.3f)",
            startCoord[0], startCoord[1], startCoord[2], goalCoord[0],
goalCoord[1], goalCoord[2]);
    }

    return validPlan;
}

double pathPlanning::loopCost(double **adjMatrix, std::vector<int>
movePlan) {
    // Note, adjMatrix has been passed in by reference so any changes to it
will
    // be reflected in the variable used when calling this
    double cost = 0;
    for(int i = 1; i < movePlan.size(); ++i) {
        cost += (adjMatrix[movePlan[i]])[movePlan[i-1]];
    }
    cost += adjMatrix[movePlan[movePlan.size() - 1]][movePlan[0]];
    return cost;
}

std::vector<int> pathPlanning::findOptimalPath(bool printResult) {
    ROS_INFO("Determining optimal path using brute force method.");

    // Create an adjacency matrix
    int tour_points = stopCoords.size() + 1; // Number of points to take
    double adjMatrix[tour_points][tour_points]; // Stores cost between any
two spots
    // Start is 0, 1 onwards are boxes 1 onwards

    for(int i = 0; i < tour_points; ++i) {
        for(int j = 0; j < tour_points; ++j) {
            if(i == j){
                adjMatrix[i][j] = 0;
            }
            else if(i == 0) {
                double dx = startCoord[0] - stopCoords[j-1][0];
```

```
                double dy = startCoord[1] - stopCoords[j-1][1];
                adjMatrix[i][j] = sqrt(dx * dx + dy * dy);
            }
            else if(j == 0) {
                double dx = startCoord[0] - stopCoords[i-1][0];
                double dy = startCoord[1] - stopCoords[i-1][1];
                adjMatrix[i][j] = sqrt(dx * dx + dy * dy);
            }
            else {
                double dx = stopCoords[i-1][0] - stopCoords[j-1][0];
                double dy = stopCoords[i-1][1] - stopCoords[j-1][1];
                adjMatrix[i][j] = sqrt(dx * dx + dy * dy);
            }
        }
    }

    // Initialize vector as a set of numbers from 0 to the number of tour
points
    std::vector<int> movePlan(tour_points);
    for(int i = 0; i < tour_points; ++i) {
        movePlan[i] = i;
    }

    // Prepare pointer to pass adjMatrix
    double *temp[tour_points];
    for(int i = 0; i < tour_points; ++i) temp[i] = adjMatrix[i];

    double bestScore = loopCost(temp, movePlan);
    std::vector<int> bestRoute = movePlan;

    while(std::next_permutation(movePlan.begin() + 1, movePlan.end())) {
        double s = loopCost(temp, movePlan);
        if(s < bestScore) {
            bestScore = s;
            bestRoute = movePlan;
        }
    }

    // Edit route to be better fed into navigation
    for (int i = 1; i <= tour_points; i++) {
```

```
        bestRoute[i] = bestRoute[i] - 1;     // Fix box indexing to start
from 0 and not 1
        bestRoute[i-1] = bestRoute[i];       // Move all points up one
(removing the start point from the list)
    }
    bestRoute.pop_back(); // Remove the redundant end point

    if(printResult) {
        ROS_INFO("Best path determined for %d given boxes.\nEstimated
travel of %.2f m with",(int) bestRoute.size(), bestScore);

        char buffer[60];

        for (int i = 0; i < bestRoute.size(); i++) {
            sprintf(buffer, "\tStop %2d - Box %2d\t(%5.2f, %5.2f,
%6.3f)\n", i + 1, bestRoute[i],
                stopCoords[bestRoute[i]][0], stopCoords[bestRoute[i]][1],
stopCoords[bestRoute[i]][2]);
            std::cout << buffer;
        }

        // Tack on reminder that we return to start
        sprintf(buffer, "\tReturn to start \t(%5.2f, %5.2f, %6.3f)\n\n",
startCoord[0], startCoord[1], startCoord[2]);
        std::cout << buffer;
    }

    return bestRoute;
}
```

Appendix P: robot_pose.h

```
#pragma once

#include <geometry_msgs/PoseWithCovarianceStamped.h>

class RobotPose {
   public:
       float x;
       float y;
       float phi;
   public:
       RobotPose(float x, float y, float phi);
       void poseCallback(const geometry_msgs::PoseWithCovarianceStamped&
msg);
};
```

Appendix Q: robot_pose.cpp

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

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

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

Appendix R: tests.h

```
#pragma once

#include "fileWrite.h"
#include "imagePipeline.h"
#include <dirent.h>         // Used for reading in the test files
#include "pathPlanning.h"
#include "navigation.h"

// "#define"s used to run tests. Comment out any unwanted tests
// These are used over standard "if"s since these will omit the code from
the compilation properly

//#define FILE_WRITE_TEST       // Execute file output test
//#define VISION_SAMPLES_TEST   // Run test code for vision (go through
test files)
//#define MOTION_TEST           // Test motion functions / systems

const std::string testPhotoFolder =
"/catkin_ws/src/mie443_contest2/testpics/"; // Relative to user home

void fileWriteTest(Boxes boxes, std::vector<int> movePlan, bool
printStuff);
void visionSystemTest(std::string searchTerm, Boxes boxes, ImagePipeline
&imagePipeline, bool printInnerWorks);
void navigationSystemTest(pathPlanning pathPlanners);
```

Appendix S: tests.cpp

```
#include "tests.h"

void navigationSystemTest(pathPlanning pathPlanner) {
   ROS_WARN("\n\nMOTION TEST \n(will terminate once complete)\n");

   std::vector<float> testPoint(3, 0); // Initialize with 0s

   /* Random point test
   srand(time(NULL)); // Seed the random number generator with the current
time

   // Generate a goal it can reach within the 6x6 maze
   do {
       testPoint[0] = -3.0 + (float)(rand() % 600) / 100.0;
       testPoint[1] = -3.0 + (float)(rand() % 600) / 100.0;
   } while(pathPlanner.checkPossible(testPoint) == false);

   Navigation::moveToGoal(testPoint);
   pathPlanner.clearCostMap();
   */

   // Go to all boxes
   for (int i = 0; i < pathPlanner.stopCoords.size(); i ++ ) {
       ROS_INFO("\n\tGOING TO STOP %d", i);
       testPoint = pathPlanner.stopCoords[i];

       bool gotThere = pathPlanner.goToCoords(testPoint);

       if (gotThere) ROS_INFO("Reached box %d", i);
       else ROS_ERROR("Failed to reach box %d", i);

       ros::Duration(5).sleep(); // Simulate scan and so we can register
it stopping
   }

   bool gotThere = pathPlanner.goToCoords(pathPlanner.startCoord);

   if (!gotThere) ROS_ERROR("Failed to return to starting point.");
```

```
}

void fileWriteTest(Boxes boxes, std::vector<int> movePlan, bool
printStuff) {
   ROS_WARN("\n\nRUNNING FILE OUTPUT TEST\n(will terminate once
complete)\n");

   std::vector<int> boxIDs(boxes.coords.size()); // Dummy box IDs

   for (int i = 0; i < boxes.coords.size(); i++) {
       boxIDs[i] = i / 2; // i/2 to get duplicates
   }

   writeLog(boxes, movePlan, boxIDs);
}

void visionSystemTest(std::string searchTerm, Boxes boxes, ImagePipeline
&imagePipeline, bool printInnerWorks) {
   ROS_WARN("\n\nRUNNING VISION TEST \nSearch term: \"%s\".\n(will
terminate once complete)\n", searchTerm.c_str());

   // Find folder on user
   char * homeDir = std::getenv("HOME"); // Get home directory for user
   const std::string folderLocation = homeDir + testPhotoFolder;

   ROS_INFO("Test photo location used:\n%s\n", folderLocation.c_str());

   // Load in test files
   std::vector<std::string> fileNames;

   DIR *dr;
   struct dirent *en;
   dr = opendir(folderLocation.c_str()); // Open directory
   if (dr) {
       while ((en = readdir(dr)) != NULL) {
           std::string temp = en->d_name; // Grab file names

           // Add files that end in PNG and contain search term
           if (temp.find(".png") != std::string::npos) {
```

```
                if (temp.find(searchTerm) != std::string::npos) {
                    fileNames.push_back(temp);
                }
            }
        }
        closedir(dr); // Close directory
    }

    // Check if empty
    if (fileNames.empty()) {
        ROS_FATAL("\n\nNO TEST FILES FOUND!\nCheck for valid search string
or folder location.\n");
        return;
    }

    std::sort(fileNames.begin(), fileNames.end()); // Sort files
alphabetically

    int result[fileNames.size()]; // Result array

    // Go through each test file to ID
    for (int i = 0; i < fileNames.size(); i++) {
        std::string testFile = folderLocation + fileNames[i];
        imagePipeline.loadImage(testFile);
        result[i] = imagePipeline.getTemplateID(boxes, printInnerWorks);
    }

    // Print result summary
    printf("\nResults of image analysis, ID# and file name.\n");
    for (int i = 0; i < fileNames.size(); i++) {
        printf("ID %2d: %s\n", result[i], fileNames[i].c_str());
    }
}
```

Appendix T: webcam_publisher.cpp

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

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

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

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

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

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

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

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

7.0 References

[1] *MIE443H1S: Contest 2: Finding Objects of Interest in an Environment*, University of Toronto, Toronto, ON, 2021, pp. 1-3.