# MIE443H1S: Contest 2

## Finding Objects of Interest in an Environment

### 1.1 Objective:

The goal of this contest is for the simulated TurtleBot to navigate an environment to find and identify ten objects placed at different locations in the environment and return back to its starting position when the robot is finished. For identification purposes, each object will have a tag placed on it. The map of the environment will be provided to your team in advance. Your team will be in charge of programming the simulated TurtleBot to navigate itself to locations in the environment where it can view and identify each object, and then return to its starting location, all within a specified time limit.

### 1.2 Requirements:

The contest requirements are:

- The contest environment will be 6x6 $m^2$ simulated environment. For simplicity, there will be no additional objects in the environment other than the ten objects to be examined as shown in Figure 1 (as an example scene). Each object is represented by a box of dimensions 50x32x40 $cm^3$ ($l$ x $w$ x $h$). A test set of object coordinates will be provided for these objects, measured with respect to the world coordinate frame of the map for the provided practice scene in Figure 1. This can be used for development purposes during the lab sessions. A new set of objects will be generated by the Instructor/TAs for Contest 2 to test your code, you will not know them in advance.

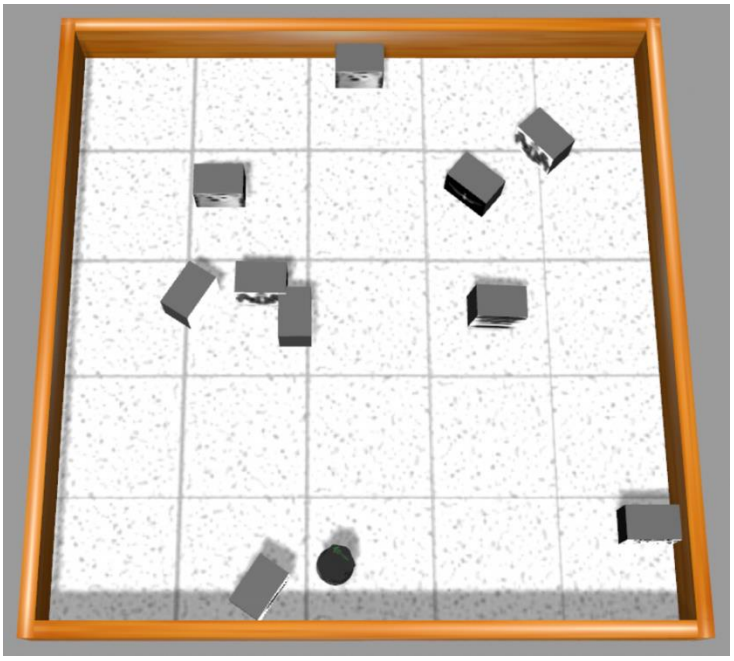

Figure 1: Example Environment Layout with 10 Objects

- Your team will be provided with: 1) a 2D map of the contest environment (without the objects) generated with gmapping, 2) test locations for the ten objects within that environment, and 3) a dataset of potential image tags that can be used during the contest.
- An object's location in the environment map is defined by the coordinates of its centre and its orientation ($x,y,\varphi$), where $\varphi$ is about the $z$-axis, with respect to the origin of the given 2D map. The origin of the world coordinate frame is determined by the TurtleBot's starting location when the map was created, see Figure 2. The origin is a native property of the map and all distances within the map coordinate frame are measured with respect to it. For more information on the world coordinate frame for the map please refer to the reference material in section A.1.3 in the Appendix.
- The simulated TurtleBot does not have to start at the origin of the world coordinate frame when the map is loaded; however, its pose within the map, just like the objects, will be measured with respect to this origin.

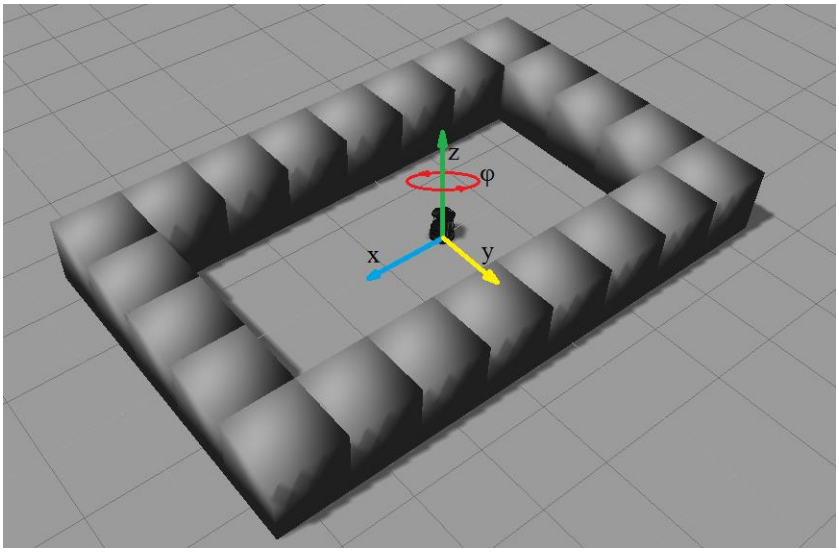

Figure 2: Example World Coordinate Frame

- The object locations are defined by two vectors: 1) a coordinate vector which defines each object's location in $x$ and $y$, and 2) an orientation vector which contains the object rotation about the $z$ axis. These locations are measured from the object's local frame (Figure 3) with respect to the world coordinate frame (Figure 2) at the origin of the map.
- The image tags are high contrast images with many unique features. An example is shown in Figure 4. These tags will be placed in the centre of one of the long faces of the objects, Figure 3.
- You will be given a dataset of all potential image tags that can be used for this contest.

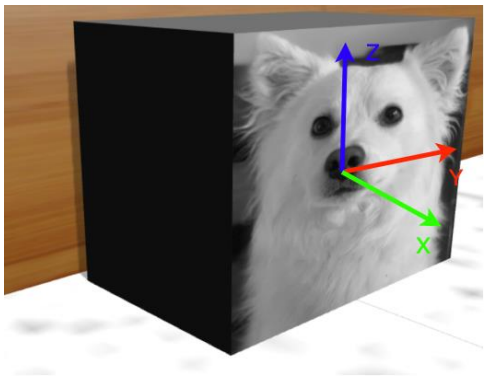


Figure 3. Object Coordinate Frame

Figure 4. Image Tag Example

- For Contest 2, your team will receive a new set of object locations to utilize for the contest and test your code. Your team must ensure that your code is robust enough to handle such dynamic changes to the locations of the objects. There will be no additional objects in the environment.
- During the contest, the simulated TurtleBot will start at a location in the map of the Instructor/TAs choosing. The robot will have a maximum time limit of **8 minutes** to traverse to and identify all the objects provided, before returning to its starting location and indicating that it is done.
- Your simulated TurtleBot must utilize the provided navigation library to drive the robot base safely and use the RGB camera on the Kinect sensor to perform SURF feature detection in order to find and identify the image tag on each object. Please become familiar with how this feature detection method works.
- For this contest, there will be ten objects in the environment, of these one or two objects will have a duplicate tag, and one or two objects will not have a tag, and the rest will have unique tags. As previously mentioned, the image tags in the environment will be taken from the data set of images provided to you.
- Once the simulated TurtleBot has traversed to all the objects and returned to its starting location, it must output to a file (http://www.cplusplus.com/doc/tutorial/files/) all the tags it has found and at which object location (i.e., tag+location) in the exact order that they were found. This information will be used for scoring. Please make sure you output the tag name given to the object to ensure proper labeling of tags.
- In order to reduce run time, your team **must** optimize the robot's path to find the objects.

### 1.3 Scoring (15%):

- Your team will be given a total of 2 trials. The best trial will be counted towards your final score.
- The scores for each trial are as follows. One mark will be provided for the navigation to each object location (10 marks in total). One mark will be given if your TurtleBot is able to correctly identify the image tag on an object at each location (including determining duplicate tags) or determine that there is no tag on the object (10 marks in total).

### 1.4 Procedure:

Code Development - The following steps should be followed to complete the work needed for this contest:

- Download the mie443_contest2.zip from Quercus. Then extract the contest 2 package (right click > Extract Here) and then move the mie443_contest2 folder to the catkin_ws/src folder located in your home directory.
- In terminal, change your current directory to ~/catkin_ws using the cd command. Then run catkin_make to compile the code to make sure everything works properly. ****Please note if you do not run this command in the correct directory a new executable of your program will not be created.****
- It is recommended that robot navigation for this contest be handled by the ROS package called *move_base*. This library takes coordinates and orientations in the global map of the environment and drives the robot to a specified location while handling robot localization and obstacle avoidance. For more information please refer to the reference material in section A.1.3 in the Appendix.
- For tag feature detection, it is recommended to use the OpenCV feature finding framework. Please refer to the reference material in section A.1.2 in the Appendix for more information and tutorials with regards to this library. Please keep in mind that the OpenCV feature finding tutorial is only programmed to handle a single image at a time, and therefore, it is important to understand how it works so that it can be implemented for the purpose of this contest with as few repetitive variable declarations as possible.
- Refer to the contest2.cpp and imagePipeline.cpp file located in the mie443_contest2/src/ folder. For readability and ease of troubleshooting it is important that your team develops the navigation code and the image processing code in these respective files. Contest2.cpp will handle the robot navigation code and then will call a function from imagePipeline.cpp that handles the image processing.
- **DO NOT** make changes to any other files.

Compiling

- Every time the code is changed you must compile it from terminal in the catkin_ws directory using the following command. If you do not do this, a new executable will not be created when you run it.

```
catkin_make
```

Robot Gazebo Simulation- To simulate the TurtleBot and laser scan information the following steps should be followed:

- Move_base and OpenCV in Gazebo
    - Begin by launching the simulated world through the following commands (you can change the world name if other world files are available in ../worlds folder):

```
roslaunch mie443_contest2 turtlebot_world.launch world:=practice
```

- To run the amcl localization and obstacle avoidance algorithm in simulation use the following command (make sure the map number corresponds to the world number) and provide a location estimate in RVIZ:

```
roslaunch turtlebot_gazebo amcl_demo.launch
map_file:=/home/turtlebot/catkin_ws/src/mie443_contest2/maps/map_practice.yaml
```

- This command opens a program called RVIZ that visualizes data published on the ROS network. It is useful for troubleshooting the navigation algorithm by visualizing the location of the robot. RVIZ also provides an interface for the user to: 1) publish navigation goals for the robot to move to, and 2) provide an initial pose estimate to initialize AMCL.

```
roslaunch turtlebot_rviz_launchers view_navigation.launch
```

- All other commands to view and save the map are the same as in the amcl tutorial.
    - Before running your code, make sure the information contained in ../src/coords.xml is for the map which you launched in the previous step. If not, copy the correct information to ../src/ coords.xml. For Gazebo, xml files corresponding to the maps are located in ../maps folder. Run the following command to start running your code in simulation:

```
rosrun mie443_contest2 contest2
```

- OpenCV testing
    - You can create unit tests to verify the functionality of your algorithm.
    - Create a stand-alone program and calculate the precision and recall of your detector on various images taken within the environment.
    - You can obtain these images by saving the images returned by the camera topic
    - Gather images from multiple perspectives of each tag to get a robust idea of your performance.
- If your team chooses to test in a different area than the provided environment, then the default map **will not work** and a new map must be made for the robot. To do this, please complete the following steps:
    - Follow the mapping tutorial in the reference material section A.1.3 in the Appendix.
    - Copy the resulting my_map.yaml, and my_map.png files to the maps folder in your code package at the address: ../mie443_contest2/maps.
    - Change test destination coordinates in your coords.xml file to reflect the changes in the map.
- If all is working correctly, your robot will begin navigating through the simulated environment to find the objects at their locations, while identifying the image tags on each object.
- After all objects have been examined and the simulated TurtleBot has returned to its starting location it ****MUST**** output what tags have been found (tag names) as well as their respective coordinate index in the order that they have been found for evaluation.
- Cancel all processes by pressing Ctrl-C in their respective terminals when you are done.

Code Breakdown – contest2.cpp

- Sensor Implementation - Odometry
    - This is the callback function that is run whenever the simulated robot's pose ($x$, $y$, $\varphi$) in the map is published. The variable msg is an object that contains the data pertaining to the robot's current position and orientation.

```
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
}
```

    - Main block and ROS initialization functions.

```
int main(int argc, char** argv){
      ros::init(argc, argv, "map_navigation_node");
      ros::NodeHandle n;
      // Robot pose object + subscriber
      RobotPose robotPose(0,0,0);
```

    - Declaration of the subscriber amclSub that is used to return the global pose of the TurtleBot.

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

- Robot Navigation
    - Initialization of the vector containing the object coordinates ($x$, $y$ *and* $\varphi$), and the vector containing the image tags that the TurtleBot is searching for in the environment. If the function is not able to properly initialize the variables it will terminate the program and state what the problem was.

```
// Initialize box coordinates and templates
Boxes boxes;
if(!boxes.load_coords() || !boxes.load_templates()) {
      std::cout << "ERROR: could not load cords or templates" << std::endl;
      return -1;
}
```

"`boxes.coords`" is a two-dimensional vector, where the first index of the vector is used to specify which coordinate you would like to choose and the second index is used to select whether you would like to access the $x$ location, $y$ location or orientation of that coordinate, stored in that order. For example typing `boxes.coords[1][2]` would return the orientation of the second coordinate stored in the vector (vectors are zero-indexed data structures).

"`boxes.templates`" is a one-dimensional vector, containing the templates from the boxes_database.

- Sensor Implementation – RGB camera
    - The following block defines a class that is used to return an image from an image topic provided by the Kinect sensor.

```
ImagePipeline imagePipeline(n);
```

Code Breakdown – navigation.cpp

- The `Navigation::`*moveToGoal()* function is used to move the TurtleBot to a goal location. When destination coordinates ($x$, $y$ and $\varphi$) are passed to the function, it allows the simulated TurtleBot to navigate to the commanded location while performing obstacle avoidance. For more information on this function please refer to reference material in A.1.3 in the Appendix.

```
bool Navigation::moveToGoal(float xGoal, float yGoal, float phiGoal){
```

Code Breakdown – imagePipeline.cpp

- The *getTemplateID()* function takes in the boxes object containing the image templates. The algorithm should search the images that are returned from the image callback for the matching image tags from the vector. If it finds a tag in the images, it should return the index of the found tag to the main function.

```
int getTemplateID(Boxes &boxes){
      int template_id = -1;
```

- The below checks to see if an image has properly been set to the video variable and then displays it to the screen before releasing the memory and returning an integer.

```
if(!isValid){
      std::cout << "ERROR: INVALID IMAGE!" << std::endl;
} else if(img.empty() || img.rows <= 0 || img.cols <= 0) {
      std::cout << "ERROR: INVALID IMAGE BUST STILL SOME PROBLEM!" << std::endl;
      std::cout << "img.empty():" << img.empty() << std::endl;
      std::cout << "img.rows:" << img.rows << std::endl;
      std::cout << "img.cols:" << img.cols << std::endl;
} else {
      /***YOUR CODE HERE***/
      // Use: boxes.templates
      cv::imshow("view", img);
      cv::waitKey(10);
  }
  return template_id;
}
```

- Refer to the coords.xml file located in mie443_contest2/src/boxes_database folder. This file is used to store the coordinate locations that the robot will be traversing. During testing, these locations can be altered to test for robustness but **do not** make changes to the structure of the code for the contest here as a new coords.xml will be used for Contest 2 with a new set of object coordinates.

- Refer to the templates.xml file located in mie443_contest2/src/boxes_database folder. This file is used to store the image tag names. The tags can be found in the same folder.

Code Breakdown – coords.xml

- XML is a coding syntax that allows data to be stored in a very efficient structure for retrieval in code. Also, because it is XML and not C++ code, it does not need to be complied every time you make a change.
    - The headings in the following block of code contain the *x*, *y*, *z* and quaternion $q_x, q_y, q_x, q_w$ information of each coordinate the robot will navigate to with respect to the world coordinate frame of the practice map. The data in each heading is listed as x, y, z, $q_x, q_y, q_x, q_w$ in descending order. The code reading this xml file automatically converts the quaternion to each box's orientation in the environment and removes the z coordinate. The pose of the box is, therefore, available within the code as *x*, *y*, $\varphi$.

```
<coordinate1>
1.92707419395 <!-- x -->
2.27079343796 <!-- y-->
0.0 <!-- z-->
0.0 <!-- quaternion x-->
0.0 <!-- quaternion y-->
-0.804490596157 <!-- quaternion z-->
0.593965386782 <!-- quaternion w-->
</coordinate1>
…
<coordinate10>
-0.49049949646
-2.51549220085
0.0
0.0
0.0
-0.473812811517
0.880625584254
</coordinate10>
```

Code Breakdown – templates.xml

- The following initializes the folder path to load the images to be identified during the contest. The folder paths to the pictures are defined with respect to the folder path of the ROS package location on your computer.

```
<templates>
      "tag1.jpg"
      "tag2.jpg"
      "tag3.jpg"
      …
      "tag14.jpg"
      "tag15.jpg"
</templates>
```

### 1.5 Code Submission:

- In the README.md, please include the following:
    - Group Number and all names of students in your group.
    - The list of commands that we need to run in order to execute your group's code as needed in consecutive order.
    - Any computer specific *file path string* that we need to modify for your code to run.
    - Where exactly the output file will be stored.

### 1.6 Report:

- The report for each contest is worth half the marks and should provide detailed development and implementation information to meet contest objectives (15 marks).
- Marking Scheme:
    - The problem definition/objective of the contest. (1 mark)
        - Requirements and constraints
    - Strategy used to motivate the choice of design and winning/completing the contest within the requirements given. (2 marks)
    - Detailed Robot Design and Implementation including:
        - Sensory Design (4 marks)
            - All Sensors used, motivation for using them, and how are they used to meet the requirements for the contest including functionality.
        - Controller Design, both low-level and high-level control (including architecture and all algorithms developed) (5 marks)
            - Architecture type and design of high-level controller used (adapted from concepts in lectures)
            - Low-level controller
            - All algorithms developed and/or used
            - Changes and additions to the existing code for functionality
    - Future Recommendations (1 mark)
        - What would you do if you had more time?
        - Would you use different methods or approaches based on the insight you now have?
    - Full C++ ROS code (in an appendix). Please do not put full code in the text of your report. (2 marks)
    - Contribution of each team member with respect to the tasks of the particular contest (robot design and report). (Towards Participation Marks)
    - The contest package folder containing all your code, README file, etc. that will also be submitted alongside the softcopy (PDF version) of your report. Check and make sure your code runs before submitting (Towards the Contest and Code Marks Above)

**1.7 Individual Report (One per group member):** Each group member will also submit an individual report answering a set of questions on Contest 1 development to be completed on your own. These questions will count towards your individual Participation Marks (10%).

**1.8 Reference Materials for the Contest**:

The following materials in Appendix A will be of use for Contest 2:

- A.1.1
- A.1.2
- A.1.3