MIE 443 - Mechatronics Systems: Design & Integration

Contest 2 Report

Team Members:
Danyal Rehman
Parthkumar Parmar
Patrick Mirek
Xiangyu Luo
Yehia Mezen

Date: March 23, 2017

Table of Contents

1.Problem Definition

The team was tasked with the problem of programming the turtlebot to efficiently navigate to 5 different locations in a known environment, identify the image tag placed on the object and then return back to its starting position when finished. It was essential to optimize the robot's travel path to ensure the completion of the task within the time limit (*5 minutes*). In order to complete this task, the following information was provided to the team:

- The map of the environment
- The coordinates and orientation of the objects (5 in total)
- The image tags (3 in total)

For object identification purposes, each object had an image tag placed on it. There were three objects in the environment with unique tags, one object with a duplicate tag and one object without a tag. Once the turtlebot completed the task, it needed to output (i.e. display) all the tags it found with their respective locations.

2.Objectives and Constraints

In order to solve the problem outlined in the previous section, the following objectives have been identified:

2.1 Primary Objective

- Utilize the turtlebot to find and identify 5 objects (*using their image tags*) in a known environment

2.2 Secondary Objectives

- Should perform the task in the shortest amount of time by navigating the environment efficiently and minimizing the time spent identifying objects image tags
- Should have algorithms in place to avoid falsely identifying an image or the environment (*i.e. false positives*)
- Should have algorithms which accommodate for physical limitations of the turtlebot including but not limited to image recognition limits (*i.e. distance and angle*), speed of data acquisition/processing and inaccuracy in odometry readings

2.3 Constraints

The following constraints must be obeyed in order to successfully solve the problem identified:

- Shall use OpenCV to perform image recognition
- Shall use the provided navigation library to perform navigation and object avoidance

- Shall start from a random location in the map
- Shall return to the starting location upon completion of the task
- Shall complete the task in 5 minutes
- Shall output to the terminal all image tags found and their respective locations
- Shall utilize the environment map, image tags and object coordinates/orientations provided

3.Detailed Robot Design and Implementation

3.1 Sensory Design

This section gives an overview of the sensors used as well as the reasonings behind their use for the given contest. The overall sensory layout is shown in Figure 1. The RGBD sensor on the kinect is first used to localize the turtlebot in the known environment. Odometry data is updated once the localization is complete. The odometry information is compared to the destination information through the navigation algorithm, after which the turtlebot travels to the desired destination. Using the environment map and depth sensor, the turtlebot can avoid obstacles and walls as it travels. Once at the destination, the RGB camera is turned on and the video frames are compared with image tags to detect matches or blank images.

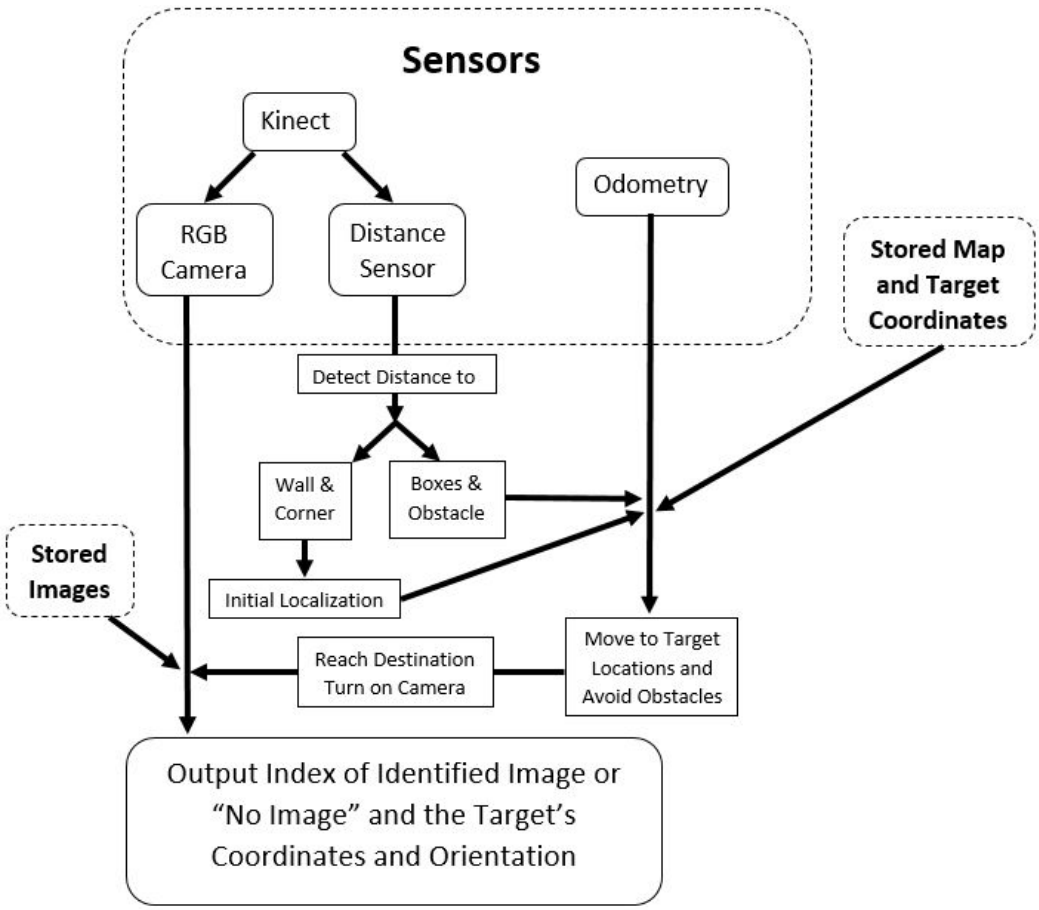

Figure 1: Overall sensory layout

3.1.1 Microsoft Kinect (RGDB Sensor)

To begin the task of navigating to objects, the robot needs to localize itself in the given environment (i.e. determine its (x,y, θ) coordinates). The initial localization requires depth information from the RGDB sensor so that the robot can apply particle filtering and estimate the most probable location in a known map starting from a 2D position estimate inputted by the user. This is illustrated in the figure below.

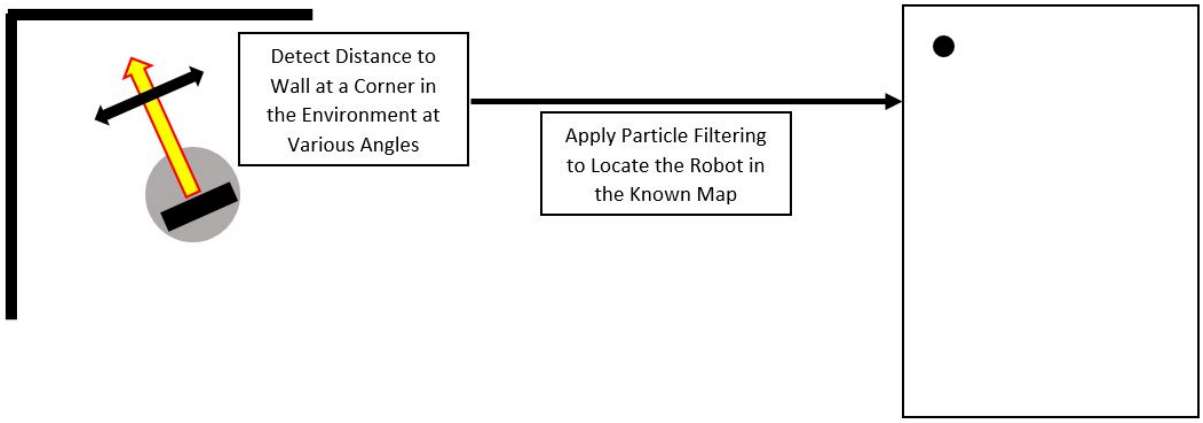

Figure 2: Localization of turtlebot in known environment

The provided map of the environment does not contain any objects, only the boundaries. In order to navigate through the environment, the turtlebot scans for the presence of obstacles using the depth information. With this information, the "moveToGoal" function can then adapt the path of the turtlebot to prevent collision with objects (*see figure below*). In addition, the depth information is also used to determine if it is possible to reach a predetermined destination. Using the depth information, the turtlebot can determine if an object is located on top of its target destination. If there is, the turtlebot returns "failed to reach the destination". Additionally, it is important to note, the turtlebot also returns the same message if the target location is outside the given map.

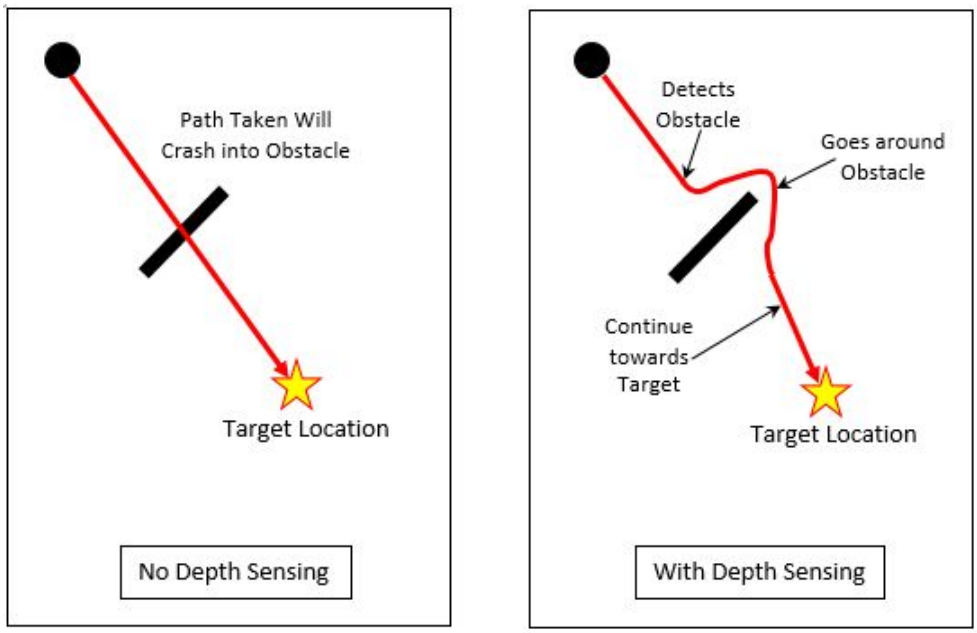

Figure 3: Path planning and object avoidance

3.2.2 Odometry

The position of the objects and home location are inputted to the turtlebot in the form of x, y and θ (yaw). This information can be used through odometry. During the task, the turtlebot uses the (x,y, θ) information from the odometry sensor coupled with the depth information to determine the optimal travel path to the desired location both before and during its travel.

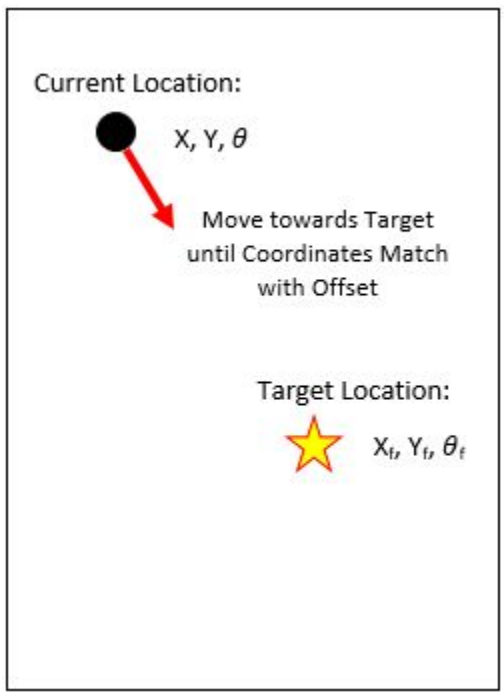

Figure 3: Path planning using current location based on odometry and object positions

3.1.3 Microsoft Kinect (RGB sensor)

The RGB camera is capable of capturing a video feed, enabling the turtlebot to store the video frames for processing. Once the robot is at its destination, the camera turns on feeding live video to the robot. The video feed is then processed by the robot which enables image identification.This sampling occurs at the same rate as the frame rate.

3.2 Controller Design

The design of the controller for a turtlebot is linked to its ability to sense, think and act. The successful completion of this contest required the turtlebot to use all of those abilities. This type of control is deliberative control. These components were implemented as follows:

- **Sense**: Kinect depth sensor to detect nearby objects/obstacles, odometry sensors to determine location in map
- **Think**: Path planning between map locations
- **Act**: Turtlebot movement, image recognition

The table below summarizes the different situations the turtlebot encounters in its environment and the resulting actions it would take.

Situation	Action
Turtlebot enters known environment	Localization: Turtlebot determines its location (i.e. home position) in the environment in terms of (x, y, θ)
Turtlebot at home position (after entering environment) or destination (after outputting objects image tag/blank image index and coordinates)	Calculate the distances between the current position and the unvisited objects; determine closest object (i.e. nearest neighbor)
Turtlebot has determined closest object	Travel to (x, y, θ) location facing and offset from closest object using move_base (described below)
Turtlebot encounters an obstacle in its travel path	Recalculate travel path determined by move_base to navigate around the obstacle
Turtlebot arrives at object	Image recognition: Initialize Kinect camera video feed, compare video feed frames with stored image tags using image recognition algorithm (described below), determine object's image tag index or blank image
Turtlebot completes image recognition	Output objects image tax index or blank image index (-1) and coordinates

Turtlebot arrives at home position (after visiting all 5 objects)	Remains stationary

Table 1 : Situations the turtlebot will encounter and its resulting actions

3.3 Program Algorithms

In order for the turtlebot to achieve outlined objectives, it was identified that an algorithm was required for both: path planning and image recognition. The details of both algorithms are given below.

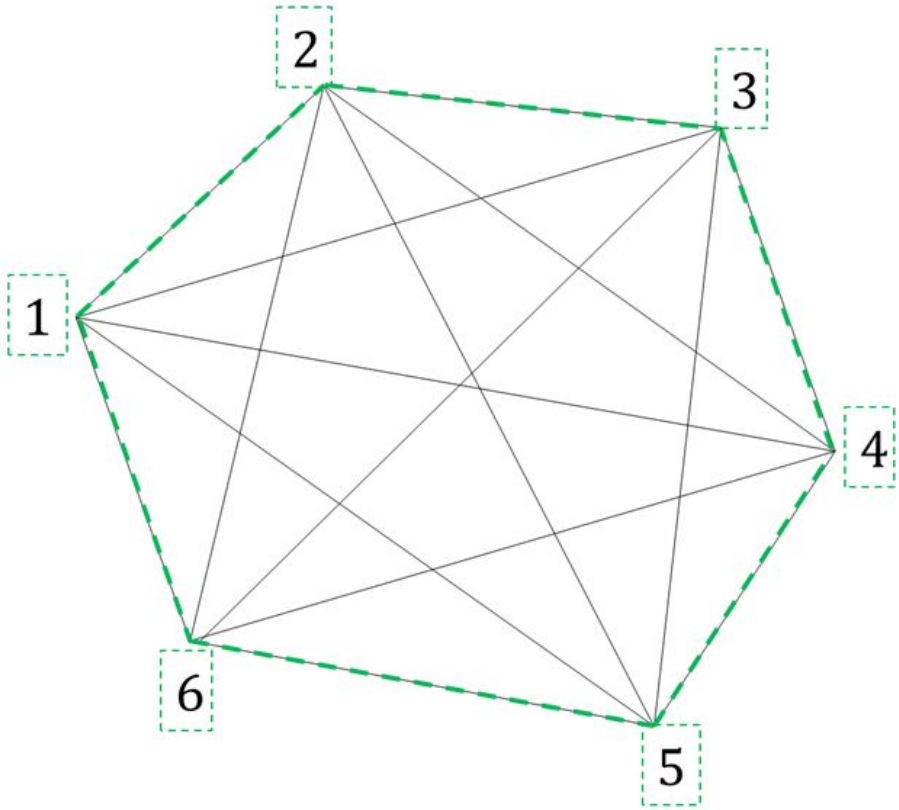

Figure 4: All possible paths and nearest neighbour path (highlighted in green) for the robot to take from its initial position (*Hamilton Circuit*)

3.3.1. Path Planning Algorithm

High level control

The path planning algorithm that the team selected is the nearest-neighbor algorithm. In the nearest-neighbour algorithm, at each location of the turtlebots tour (*i.e. object location or home position*), the turtlebot will determine the closest object which it has not yet visited and travel to it. It will do this until all objects have been visited, at which point it will return to its home position. The steps to complete this are the following:

1. Having the turtlebot determine its current (*i.e. start*) location by using the Kinect's RGBD sensor to identify nearby map features, which it would then correlate to the provided map

to locate (*i.e. localize*) itself in terms of position and orientation. Performed by running ros::spinOnce, until all localization variables (i.e. x, y, θ) are non zero.

2. Calculate the distances between the current location and the locations of all 5 objects using Euclidean distance. ($d = \sqrt{(x_{current} - x_{target})^2 + (y_{current} - y_{target})^2}$)

   Note: *object x,y coordinates and θ orientations are stored in separate vectors. Coord contains coordinates while orient contains yaws.*
3. Determine which object is closest and navigate to it using moveToGoal function. Have the turtlebot position itself to face the object's image tag (*i.e. by orienting itself 180° from the orientation of the object, as illustrated in the figure below*) with a predefined offset (*based on the distance needed to identify the objects image tag*).
4. Once the goal has been reached, perform image recognition (*see next section for details*).
5. Remove the visited objects x,y coordinates and θ orientation from the vectors to avoid revisitation. *Note: It was also possible that the turtlebot would fail to reach the destination. In these scenarios, the turtlebot would be guided to an arbitrary nearby point and attempt to perform the task again before erasing the visited object from the vector.*
6. Repeat step 2 for the 5 remaining objects until the vector is empty.
7. Return to the starting location and indicate that the task is finished by printing the image tags that correspond to each visited coordinate.

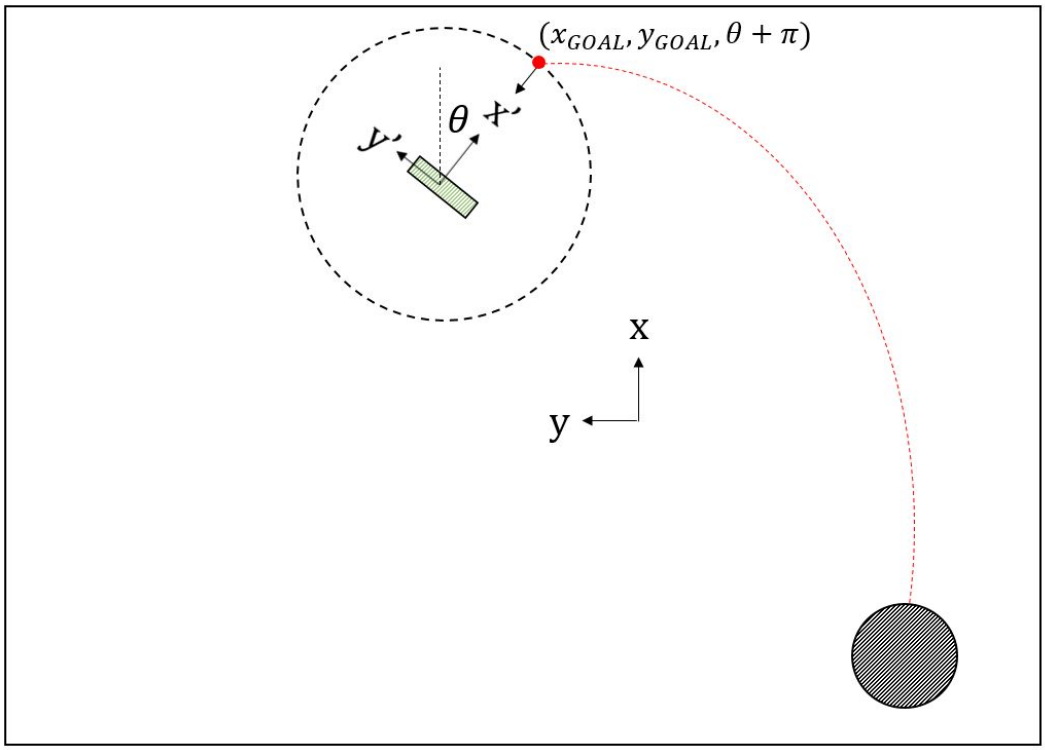

Figure 5) Goal Position and Orientation relative to Desired Location

Note: x, y and the global coordinate frame. x' and y' are the box's local coordinate frame. Θ is the yaw of the box relative to the global coordinate frame. Lastly, θ + *pi,* is the desired orientation.

Low level control

The low level control of the turtlebots navigation is achieved through the "*moveToGoal*" function. This function sends the Turtlebot to the selected destination while avoiding obstacles along the way. The turtlebot does this by comparing its current (x, y, θ) coordinates with those of the destination and tries to move towards the destination so that its current (x, y, θ) coordinates match with those of the destination. Also exploiting the depth sensor, it avoids obstacles. The steps involved in this function are the following:

1. Calculate the difference between the (x, y, θ) coordinates of the Turtlebot's current location to those of the destination.
2. The robot tries to move towards the destination so that its current (x, y, θ) coordinates match with those of the destination with a user-defined offset. Amcl_pose are recursively updated with changing position, allowing the turtlebot to know when it has reached its destination.
3. During the process, distances of the turtlebot to any walls or obstacles are monitored using the RGBD sensor and kept above a threshold value to avoid collision.

3.3.2 Image Recognition Algorithm

High level control

The main purpose of the image recognition algorithm is to match the image tags on the objects (*i.e. images pasted on the boxes*) with the images stored in the Turtlebot's database and identify the object which has no image tag. This algorithm was implemented using OpenCV. The steps involved in this algorithm are the following [1]:

1. Upon reaching the destination (*i.e. facing the object*), the RGB camera on the kinect sensor will begin a live video feed. Each frame of the video feed will be stored for comparison to the image tags
2. The video frame will be compared to starting from the first stored image:
3. The video frame is converted to grayscale since the image tag is in greyscale.
4. Using the SURF feature detector, key features such as corners, blobs and junctions are identified in the video frame and image tag. 400 key points are detected from each.
5. Using the SURF descriptor extractor, descriptors are calculated in the video frame and image tag. Descriptors describe the points neighboring a key point [2].
6. Using the FLANN matcher, descriptors are matched between the video frame and image tag. Matching involves calculating the distances between key points
7. The maximum and minimum distances between the video frame and image tag are determined.
8. The "good matches" of key features are identified and stored. In this algorithm, a good match is one with a distance less than 3 times the minimum distance calculated in the previous step.
9. The "good matches" are drawn, meaning that there are lines drawn between them for illustrative purposes (*see figure below*).

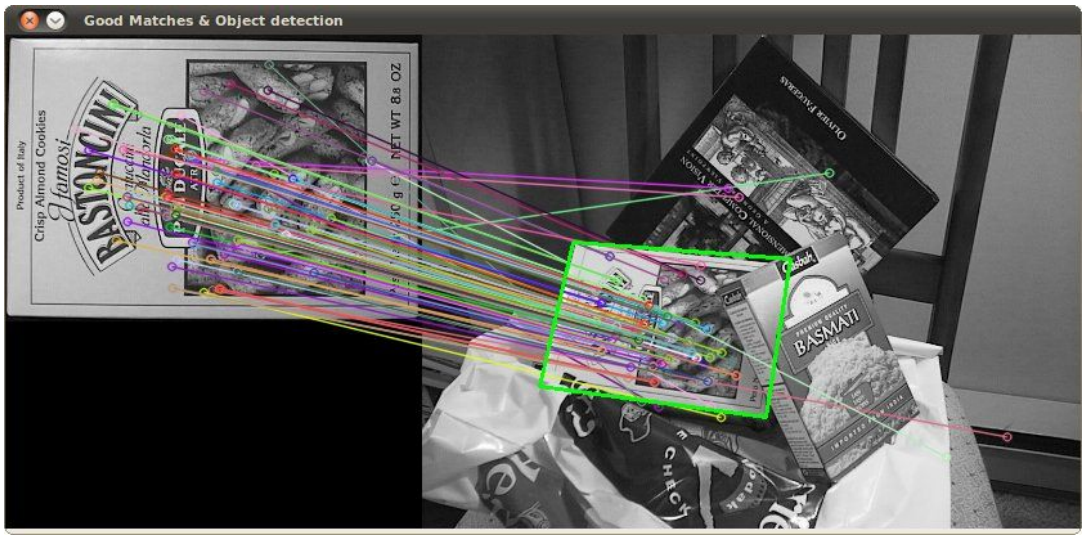

Figure 6: Lines drawn between good matches of image tag (*left*) and video frame (*right*).
Image tag in video frame is rotated with respect to stored image tag
Green lines drawn between 4 corners of image tag within the video frame [1]

10. Using the findHomography function, we create a matrix which maps the key features in the image tag to those in the video frame. This localizes the image tag within the video frame. This matrix accounts for any rotation of the image tag within the video frame (see figure above) [3].
11. Using the perspective transform function, locate the 4 corners of the image tag within the video frame [4].
12. Using the line function, draw lines between the 4 corners of the image tag within the image frame (*see figure above*). The lines effectively draw out the edges of the image tag that the program believes matches with the stored image. The coordinates of the ends of the lines are used in the next step.
13. The quality of match between the image tag in the frame from the video feed and the stored image are determined as follows:
    - 13.1. The angle between the top and bottom edges of the matched image tag is determined (*i.e. top and bottom green lines in the figure above*)
    - 13.2. The angle between the left and right edges of the matched image tag is determined (*i.e. left and right green lines in the figure above*)
    - 13.3. The angle between the top and left edges of the matched image tag is determined
    - 13.4. As the turtlebot programmed to face the object directly, the tag edges should appear square. There must be some deviation allowed for inaccuracy in sensors like odometry (*i.e. slip*). Hence, when a video frame matches with an image tag, the following conditions should be true:
        - 13.4.1. Top and bottom edges of the tag are almost parallel.
        - 13.4.2. Left and right edges of the tag are almost parallel.
        - 13.4.3. Top and left edges of the tag are almost perpendicular.
    - 13.5. When the above conditions are met, the program determines that the video frame and image tag are matched. The index of the image tag is stored. If any of the conditions above is not met, an index of -1 is stored.
    - 13.6. The next video frame is loaded for comparison

13.7. Steps 2-13.6 are repeated a total of 20 times. Once complete, the average of the stored indexes will be calculated and rounded to the nearest whole number. The result is the object's image tag index or a no image found index (-1). This value will be returned to the main code and outputted along with the current object's coordinates. The turtlebot will resume its navigation algorithm.
Note 1: The robot will be stationary during this process, so the video frames will be identical.
Note 2: By taking the average of 20 indexes/values, we are protected from false positives. Since it is fairly rare to get a false positive (*i.e. 1/10 video frames or less*), they will not have much impact on the calculated average.

Additional notes:
Throughout the code there are checks in place to ensure that images (i.e. Mats) are not empty, as using an empty Mat file would cause a segmentation fault

4.Strategy

Navigation

In order to meet our objective of programing the turtlebot to find and identify 5 objects in a known environment in the shortest amount of time, we implemented the nearest-neighbor algorithm and image box detection. In the nearest-neighbour algorithm, at each location of the turtlebots tour (i.e. object location or home position), the turtlebot will determine the closest object which it has not yet visited and travel to it. It will do this until all objects have been visited, at which point it will return to its home position. We believe that this is the winning strategy because it is simple to code and test, leading to less time spent troubleshooting. Additionally, although it is not always the most efficient (*i.e. does not always result in the lowest weight hamilton circuit*), it is capable of producing a path well under the 5 minute time limit (*determined through testing*) [5].

Image Recognition & Matching

The image recognition and matching involves intensive calculation. Hence, it is not desirable to have the algorithm running for the entirety of the contest. As the Turtlebot is sent to predetermined destinations, the condition of reaching destination already exists in the algorithm for navigation. Making use of the condition, the RGB camera on Kinect is only turned on and image recognition is only executed only when the robot is facing the image tag. As the robot is not constantly obtaining images from the environment, this also reduces the chance of false positive. That is the robot recognizes and matches an image tag that is not at the desired destination but happen to "see" while it is moving to its next destination.

When the RGB camera on Kinect is turned on, the first frame of image is always corrupted. During image recognition, there may also be cases where image tags certain in frames can not be matched or are matched to the wrong image. Hence, to reduce the chances of false negative or positive, that is the chances of determining the tag is empty based on corrupted image frames or occasional mismatch of images, each image tag is matched 20 times. The image indexes coming out of the 20 matches are summed and averaged to give the final judgment. In this way, the algorithm is more robust.

Lastly, there are two main ways to determine if the scanned image tag is matched with a stored image. One way is to setting a threshold for the “min_dist” value. If the “min_dist” values of between all matched key features. It is easy to code. But this method is very sensitive to lighting condition and is device dependent. That is, the threshold working on web cam in lab will very likely not work for Kinect in actual environment. The team uses the more robust alternative of determining if the scanned image tag has a close-to-rectangular outline. Using geometry features of the tag minimizes the influences from the lighting and use of a specific device.

5.Future Recommendations

A future recommendation for the navigation algorithm is to use the brute-force algorithm. This algorithm has the advantage that it always results in the most optimal travel path (i.e. path with the lowest weight hamilton circuit), resulting in achieving our objective of performing the tasks in the shortest amount of time. The disadvantage of this algorithm in its time spent computing the hamilton circuits is not an issue in this contest since we only have N = 5 locations to travel to, which would result in a total of 24 hamilton circuits, (N-1)!, an instantaneous calculation for a computer.

A future recommendation for the image recognition algorithm is better couple the algorithm with the navigation algorithm. Knowing that the first frame of the Kinect camera when it is turned on is always corrupted, the robot program can turn it on seconds before the robot reaches its destination. And the first frame should be disregarded. In this way, the robot can start image recognition immediately. Also, images can be matched even at a slight angle and at various distances. Hence, the image matching process can even start when the robot is approximately facing the image tag. In this way, the algorithm can start looping through the 20 cycles of image matching while the robot is moving. The robot will spend less time being stationary. The task can then be completed in a shorter period of time.

6.Conclusion

The primary goal of the this contest was to construct an appropriate control architecture by which the turtlebot can perform the function of path planning, obstacle avoidance and image recognition. In order to achieve the objectives, the design team primarily made use of laser/kinect sensors for obstacle avoidance and image recognition, bumper sensors for collision recovery and odometry for path planning.

To successfully develop algorithms using the aforementioned sensors, the team extensively made use of topics within ROS introduced during the tutorial sessions/lectures (i.e controller design and strategy formation). Lastly,robust logic was implemented as highlighted in the strategy section to ensure the system could successfully achieve the task within the time limit.

7. References

[1] Opencv Dev Team, "Features2D + Homography to find a known object," OpenCV, 22 March 2017. [Online]. Available: http://docs.opencv.org/2.4/doc/tutorials/features2d/feature_homography/feature_homography.html. [Accessed 22 March 2017].

[2] C. Bupe and Anonymous, "What is feature detector, descriptor, descriptor extractor and matcher in computer vision?," Quora, 31 July 2015. [Online]. Available: https://www.quora.com/What-is-feature-detector-descriptor-descriptor-extractor-and-matcher-in-computer-vision. [Accessed 22 March 2017].

[3] S. Mallick, "Homography Examples using OpenCV (Python / C ++)," Learn OpenCV, 3 January 2016. [Online]. Available: https://www.learnopencv.com/homography-examples-using-opencv-python-c/. [Accessed 22 March 2017].

[4] Anonymous, "Perspective Transform," Learn OpenCV By Examples, 2015. [Online]. Available: http://opencvexamples.blogspot.com/2014/01/perspective-transform.html. [Accessed 22 March 2017].

[5] G. Nejat, Writer, *Intelligent Control Lecture 3.* [Lecture]. University of Toronto, 2017.

8.Appendices

8.1 C++ Code

8.1.1 contest2.cpp

```
#include <move_base_msgs/MoveBaseAction.h>
#include <actionlib/client/simple_action_client.h>
#include <geometry_msgs/PoseWithCovarianceStamped.h>
#include "nav_header.h"
#include <eStop.h>

typedef actionlib::SimpleActionClient<move_base_msgs::MoveBaseAction> MoveBaseClient;
int navigation_algorithm(float x, float y,float globalx,float globaly,float globalangle,
vector<vector<float> > temp_coord,  vector<vector<float> > orient);

float x=0;
float y=0;
```

```
float phi=0;

bool path_defined = false;
bool jouney_over = false;

void 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;
}

int main(int argc, char** argv)
{
        cv::initModule_nonfree();
        ros::init(argc, argv, "map_navigation_node");
        ros::NodeHandle n;
        ros::spinOnce();
  teleController eStop;

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

        vector<vector<float> > coord;
        vector<vector<float> > orient;
        std::vector<cv::Mat> imgs_track;
        init(coord, orient, imgs_track);

        // added code to create a global variable for tampering with in the navigation algorithm
        vector<vector<float> > temp_coord;
        vector<vector<float> > temp_orient;

        cout << "temp_coord.empty() : " << temp_coord.empty() << endl;

        temp_coord = coord;
        temp_orient = orient;

        cout << "temp_coord.empty()  : " << temp_coord.empty() << endl;
        cout << "!temp_coord.empty() : " << !temp_coord.empty() << endl;

        // vectors to store boxes in the order visted and the corresponding pictures detected
        vector<int> output_box_index;
        vector<int> output_img_index;

        float globalx, globaly, globalangle;

        while (x == 1 && y == 0 && phi == 0) // change x == 1 to x == 0
```

```
        {
                cout << "stuck in while loop" << endl;
                ros::spinOnce();
        }

        globalx = x;
        globaly = y;
        globalangle = phi;
        int visited_node;
        int img_read_index;

  while(ros::ok())
  {
    //.....**E-STOP DO NOT TOUCH**.......
    eStop.block();
    //.....................................

        findPic(n, imgs_track);

    //fill with your code
    /*if(!temp_coord.empty())
    {
                        visited_node = navigation_algorithm(x, y, globalx, globaly, globalangle,
temp_coord, temp_orient);
                        output_box_index.push_back(visited_node);

                        img_read_index = findPic(n, imgs_track);
                        output_img_index.push_back(img_read_index);

                        temp_coord.erase(temp_coord.begin()+visited_node);
                        temp_orient.erase(temp_orient.begin()+visited_node);
    }

    ros::spinOnce();

    // if all targets visited, display the coordinates in order, and the corresponding images
detected
    if(temp_coord.empty())
    {
        navigation_algorithm(x, y, globalx, globaly, globalangle, temp_coord, temp_orient);
                for(int i = 0; i < coord.size(); i++)
                        {
                                cout << "coordinates of box " << i+1 << " : " <<
coord[output_box_index[i]][0] << ", " << coord[output_box_index[i]][1] << endl;
                                cout << "image found at box " << i+1 << " : " << output_img_index[i]
<< endl;
```

```
                        }
                        return 0;
    }*/


  }
        return 0;
}

//-------------------------move robot function----------------
bool moveToGoal(float xGoal, float yGoal, float phiGoal)
{
        //define a client for to send goal requests to the move_base server through a
SimpleActionClient
        actionlib::SimpleActionClient<move_base_msgs::MoveBaseAction> ac("move_base",
true);

        //wait for the action server to come up
        while(!ac.waitForServer(ros::Duration(5.0))){
                ROS_INFO("Waiting for the move_base action server to come up");
        }

        move_base_msgs::MoveBaseGoal goal;

        //set up the frame parameters
        goal.target_pose.header.frame_id = "map";
        goal.target_pose.header.stamp = ros::Time::now();

        /* moving towards the goal*/
  geometry_msgs::Quaternion phi = tf::createQuaternionMsgFromYaw(phiGoal);

        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 ...");
        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;
        }
}

int navigation_algorithm(float x, float y,float globalx,float globaly,float globalangle, \
                                                vector<vector<float> > temp_coord,
vector<vector<float> > temp_orient){

        // set global variables for starting position as globalx and globaly

        int i;
        float d = 0.0, finalpoint = 0.0;
        float visitx, visity, visitphi;
        float current_posx = x;
        float current_posy = y;
        float tempx, tempy;
        int tempi;
        float deviation = 0.5;
        float degrees;
        bool reached=false;

        if (temp_coord.empty())
        {
                // if there is no element left in the vector, navigate back home.
                moveToGoal(globalx, globaly, globalangle);
                return 0;
        }
        else
        {
                for(i = 0; i < temp_coord.size(); i++)
                {
                        // determines the distance from the current position to any of the points
within the vector, coord
                        d = pow(pow(current_posx - temp_coord.at(i).at(0),2) + pow(current_posy
- temp_coord.at(i).at(1),2), 0.5);
                        ROS_INFO("Distance to point %d is: %f", i, d);

                        // if it is the first iteration of the loop, simply add the distance to the first
element of the vector into
                        // the variable finalpoint.
                        if(i == 0)
                        {
```

```
                                finalpoint = d;
                                tempx = temp_coord.at(i).at(0);
                                tempy = temp_coord.at(i).at(1);
                                tempi = i;
                        }
                        else
                        {
                                // else, check to see if the distance in the ith iteration is less than
the previous distance
                                // if true, then let the variable finalpoint = d
                                if(d < finalpoint)
                                {
                                        finalpoint = d;
                                        tempi = i;
                                        tempx = temp_coord.at(i).at(0);
                                        tempy = temp_coord.at(i).at(1);
                                }
                        }
                }

                // prints the coordinates of the closest point to be visited
                if (temp_orient.at(tempi).at(0) < 0)
                {
                        visitphi = temp_orient.at(tempi).at(0) + 3.141592;
                }
                else
                {
                        visitphi = temp_orient.at(tempi).at(0) - 3.141592;
                }

                visitx = tempx + deviation*cos(temp_orient.at(tempi).at(0));
                visity = tempy + deviation*sin(temp_orient.at(tempi).at(0));
                degrees = visitphi*180/3.141592;
                ROS_INFO("Current Location: (%f, %f)\n", x, y);
                ROS_INFO("Target Location: (%f, %f), at Angle: %f degrees \n", visitx, visity,
degrees);
                moveToGoal(visitx, visity, visitphi);

                return tempi;
        }
}
```

8.1.2 img_proc.cpp

```
#include "nav_header.h"
#include <stdio.h>
```

```
#include <iostream>
#include "opencv2/core/core.hpp"
#include "opencv2/features2d/features2d.hpp"
#include "opencv2/highgui/highgui.hpp"
#include "opencv2/calib3d/calib3d.hpp"
#include "opencv2/nonfree/nonfree.hpp"


using namespace cv;

int a = 0;
int b = 0;
int sum_a = 0;

float cos12;
float cos34;
float cos13;

Mat img;

double max_dist = 0; double min_dist = 100;

//cout << "1" << endl; - use this type of line for troubleshooting

//callback function recieves images from kinect and saves it to a global variable (img)
void imageCallback(const sensor_msgs::ImageConstPtr& msg){
 try{
        img = cv_bridge::toCvShare(msg, "bgr8")->image;

 }catch (cv_bridge::Exception& e){
   ROS_ERROR("Could not convert from '%s' to 'bgr8'.", msg->encoding.c_str());
   //img.release();
 }
}

//findPic function takes images from the video feed (from imageCallback function) and compares
them to the image tags (imgs_track)
int findPic(ros::NodeHandle nh, vector<cv::Mat> imgs_track)
{
        //image window declaration
        namedWindow("view");
        startWindowThread();

        //subscribing to camera sources
        image_transport::ImageTransport it(nh);
```

```
	image_transport::Subscriber sub = it.subscribe("camera/image/", 1, imageCallback);
//--for the webcam

	ros::spinOnce();
	//image_transport::Subscriber sub = it.subscribe("camera/rgb/image_raw", 1,
imageCallback); //--for the kinect

	int foundPic = 0;

	Mat video;
	Mat grey;
	Mat imgs_search;
	Mat descriptors_object, descriptors_scene;
	Mat img_matches;
	Mat H;

	//only looking at image tag 1
	//Mat imgs_search = imgs_track[a].clone();

/*
	// leave this out for now. If we need to initialize outside of the loop add this in, as well as
	a set of instructions to reset the vectors/mats at the beginning of each while loop
	//initialize vectors outside of for loop to converve memory and avoid "core dumped" error
	int minHessian = 400;
	std::vector<KeyPoint> keypoints_
	std::vector< DMatch > matches;
*/

  while(!img.data){ros::spinOnce();}

	while(ros::ok())
	{
		ros::spinOnce();

		ros::Duration(0.25).sleep(); // sleep for 0.25 seconds

		imgs_search = imgs_track[a].clone();

		//if there is an image saved from the kinect, copy it to a local variable (video)
		if(!img.empty())
		{
			img.copyTo(video);
			img.release();

			int minHessian = 400;
			//convert it to greyscale, save it as grey
```

```
cvtColor(video, grey, CV_BGR2GRAY);

//-- Step 1: Detect the keypoints using SURF Detector
SurfFeatureDetector detector(minHessian);

std::vector<KeyPoint> keypoints_object, keypoints_scene;


if (!grey.empty())
{
	//cout << "here" << endl;

	detector.detect(imgs_search, keypoints_object);
	detector.detect(grey, keypoints_scene);


	//-- Step 2: Calculate descriptors (feature vectors)
	SurfDescriptorExtractor extractor;

	//cout << imgs_search.size() << " " << keypoints_object.size() <<
endl;

	extractor.compute(imgs_search, keypoints_object,
descriptors_object);
	extractor.compute(grey, keypoints_scene, descriptors_scene);

	//-- Step 3: Matching descriptor vectors using FLANN matcher

	FlannBasedMatcher matcher;

	std::vector< DMatch > matches;

	if(!descriptors_object.empty() && !descriptors_scene.empty())
	{
	matcher.match( descriptors_object, descriptors_scene, matches );
	//cout << "now here" << endl;

	max_dist = 0; min_dist = 100;

	//-- Quick calculation of max and min distances between keypoints

		//cout << descriptors_object.rows << " " << matches.size()
<< endl;

		for(int j = 0; j < descriptors_object.rows; j++)
		{
```

```
                                        double dist = matches[j].distance;
                                        if(dist < min_dist) min_dist = dist;
                                        if(dist > max_dist) max_dist = dist;
                                }

        //////////////////////////////////////////////////////////////////////////////////

        // We can comment out from here to the next comment block for the contest
        // It is only used for illustrative purposes to display the matches, but our object
recognition algorithm only requires min_dist

        //////////////////////////////////////////////////////////////////////////////////

                                //-- Draw only "good" matches (i.e. whose distance is less
than 3*min_dist )
                                std::vector<DMatch> good_matches;

                                for(int j = 0; j < descriptors_object.rows; j++)
                                {
                                        if(matches[j].distance < 3*min_dist)
good_matches.push_back(matches[j]);
                                }

                                drawMatches(imgs_search, keypoints_object, grey,
keypoints_scene,
                                                good_matches, img_matches,
Scalar::all(-1), Scalar::all(-1),
                                                vector<char>(),
DrawMatchesFlags::NOT_DRAW_SINGLE_POINTS);

                                //-- Localize the object

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

                                for(int j = 0; j < good_matches.size(); j++)
                                {

obj.push_back(keypoints_object[good_matches[j].queryIdx].pt);

scene.push_back(keypoints_scene[good_matches[j].trainIdx].pt);
                                }

                                if (obj.size() > 3) {
                                H = findHomography( obj, scene, CV_RANSAC );
```

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

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

                                        perspectiveTransform(obj_corners, scene_corners, H);

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

                                        //-- Show detected matches
                                        imshow("view", img_matches);

                                        float x1 = scene_corners[1].x - scene_corners[0].x;
                                        float y1 = scene_corners[1].y - scene_corners[0].y;
                                        float x2 = scene_corners[2].x - scene_corners[3].x;
                                        float y2 = scene_corners[2].y - scene_corners[3].y;

                                        cos12 = (x1*x2 + y1*y2) / (pow((x1*x1 + y1*y1),
0.5)*pow((x2*x2 + y2*y2), 0.5));

                                        float x3 = scene_corners[3].x - scene_corners[0].x;
                                        float y3 = scene_corners[3].y - scene_corners[0].y;
                                        float x4 = scene_corners[2].x - scene_corners[1].x;
                                        float y4 = scene_corners[2].y - scene_corners[1].y;

                                        cos34 = (x3*x4 + y3*y4) / (pow((x3*x3 + y3*y3),
0.5)*pow((x4*x4 + y4*y4), 0.5));

                                        cos13 = (x1*x3 + y1*y3) / (pow((x1*x1 + y1*y1),
0.5)*pow((x3*x3 + y3*y3), 0.5));
```

```
                                        ////////////////////////////////////

                                        H.release();
                                        }

                                        img_matches.release();


////////////////////////////////////////////////////////////////////////////////
                ////////////////////////////////// next comment block ///////////////////////////////////
                ////////////////////////////////////////////////////////////////////////////////////////
                                        }


                                        // if image detected (min_dist < 0.07), add the index to sum_a
                                        cout << "cos12 : " << cos12 << endl;
                                        cout << "cos34 : " << cos34 << endl;
                                        cout << "cos13 : " << cos13 << endl << endl;

                                        if(cos12 > 0.9 && cos34 > 0.9 && cos13 < 0.34 && cos13 > -0.34)
                                        {
                                                cout << "PICTURE DETECTED!" << endl << endl;
                                                sum_a += a;
                                        }

                                        // if no image detected (min_dist > 0.07), add -1 to sum_a

                                        a++;


                                        if(a == 3) a = 0;                                       // if a == 3, reset to
0

                                        b++;                                                                    //
increment b counter

                                        if(b == 20)                                                             // if
image checked 20 times
                                        {
                                                b = 0;
                                                foundPic = round(sum_a/20.0); // calculate the average of
sum_a, round to nearest integer, and asign to foundPic
                                                sum_a = 0;
                                                //cout << "foundPic is " << foundPic << endl;
                                                return foundPic;                                        // return
foundPic (index of image found, or -1 for no image found)
```

```
                                }

                                cout << "a : " << a << endl;
                                cout << "b : " << b << endl;

                                //cv::imshow("view", grey);

                                waitKey(5);
                        }
                        descriptors_object.release();
                        descriptors_scene.release();
                }
                imgs_search.release();

        }
        video.release();
        grey.release();

}
```

8.2 Attribution Table

Section No.	Danyal	Patrick	Yehia	Xiangyu	Parth
Introduction	RS, MR	RS	RS	RS	RS, RD
Background	RS	RS, RD	RS	RS	RS
Problem Definition	RS	RS, RD	RS	RS, RD, MR	RS
Detailed Robot Design	RS, MR	RD, MR	RS	RS	RS, RD
Strategy	RS, RD, MR	RS, RD, MR	RS	RS	RS
Recommendation	RS	RS, RD	RS	RS	RS
Code	RS, RD	RS, RD	RS, RD, MR	RS, RD	RS, RD
Complete Report	CM, FP	FP	FP	FP	FP

Table 3: Attribution Table

RS research

RD wrote first draft

MR major revision

ET edited for grammar and spelling

FP final read through of complete document for flow and consistency

CM responsible for compiling the elements into the complete document

OR other

If you put OR (other) in a cell put it in as OR1, OR2, etc. Explain briefly below the role referred to:

OR1:
By signing below, you verify that you have:

- Read the attribution table and agree that it accurately reflects your contribution to the associated document.
- Written the sections of the document attributed to you and that they are entirely original.
- Accurately cited and referenced any ideas or expressions of ideas taken from other sources according to an accepted standard.
- Read the University of Toronto Academic Integrity Handout and understand the definition of academic offence includes (*but is not limited to*) all forms of plagiarism.
- Additionally you understand that if you provide another student with any part of your own or your team's work, and the student having received the work uses it for the purposes of committing an academic offence, then you are considered an equal party in the offence and will be subject to academic sanctions.

Danyal Rehman

Patrick Mirek

Xiangyu Luo

Parthkumar Parmar

Yehia Mezen