GO FETCH!

SIMULTANEOUS LOCALIZATION AND MAPPING AND OBJECT RECOGNITION USING A ROOMBA ROBOT AND WEBCAM

08 December 2014
ME 4451 – A
(Fall ’14)

Prepared By:
Aida Yoguely Cortés-Peña
Doug Horgen
Phuc Le
Jeffrey Seiden

Prepared For:
Dr. Lipkin and Dr. Sadegh

TABLE OF CONTENTS

1. INTRODUCTION

The objective of this project was to create an autonomous system to search for a desired object in a foreign environment while displaying, in real-time, a two-dimensional map of the system's current and past positions, the terrain of the environment, and the object's location.

In order to accomplish this, a programmable robot, the iRobot Create Roomba, was used in conjunction with a TRENDnet Wireless Camera. The camera was rigidly mounted to the Roomba using an 80/20 aluminum extrusion structure and powered using a battery placed atop the Roomba. As this robot navigates the environment, images of the environment are captured and analyzed in using two different methods: (i) blob analysis is used to compare features in the image to the unique aspects of the image of the desired object, and (ii) edge detection and Hough transform are used to identify terrain features near the robot's path. Based on results of these methods, the map of the robot's current location and surroundings is updated. The robot will continue to map the environment until at least one of three conditions is met: (i) the object is found, (ii) the entire room is considered mapped, or (iii) the allotted mapping time is exceeded.

The most notable challenges included implementing a robust algorithm for efficiently navigating the environment with obstacles, distinguishing orientation of the obstacles, accurately reporting the robot's motion relative to these obstacles, and locating the desired object within the environment.

2. DESIGN AND ANALYSIS

2.1 Environment

A controlled environment was created to provide proof-of-concept for the tasks of mapping the area and finding an object. The environment consisted of an open classroom floor space bounded by blue tape to simulate the edge between two orthogonal surfaces, such as a floor and a wall; this can be seen in Figure 1. The tape segments are of arbitrary length, but must allow the robot to fit inside and always intersect at right angles for simplicity and compatibility with the edge detection algorithm (see 2.3.2). Similarly, obstacles can be of arbitrary size, provided that they will allow the robot sufficient space to navigate the environment and have orthogonal edges. Finally, a black and white image of a key silhouette, serving as the object of interest, is placed within the enclosure. Alternatively, the key can be removed, which allows the entire space to be mapped.

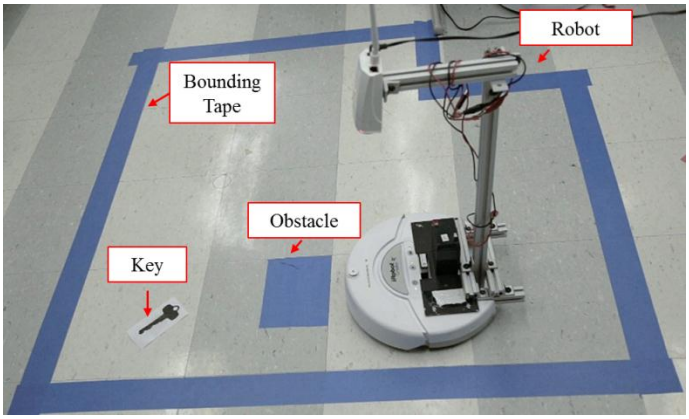

Figure 1. Environment used for proof of concept.

2.2 Hardware

This project was initially approached with the WowWee Rovio robot. Early testing revealed that the Rovio robot outputted its odometry in the form of "odometer ticks" in hexadecimal format. It was not clearly understood how often the odometry was sampled nor how many ticks represented a full revolution of the wheel encoder. The Rovio also did not allow for command of individual wheel velocities, which complicated the control. Additionally, use of the allowable motion commands did not yield consistent translations and rotations. For these reasons, the Roomba was selected for its simpler odometry system.

The final Roomba-camera system is shown in Figure 2. In this view, the camera is aimed almost directly downward. It is fixed above the Roomba unit using an 80/20 structure; this is secured to a mounting plate that was created in a previous semester by some other group. The 80/20 allowed for easy initial positioning and future adjustment of the camera relative to the Roomba. The camera was powered by a battery so that the system would not need to be tethered to a wall outlet.

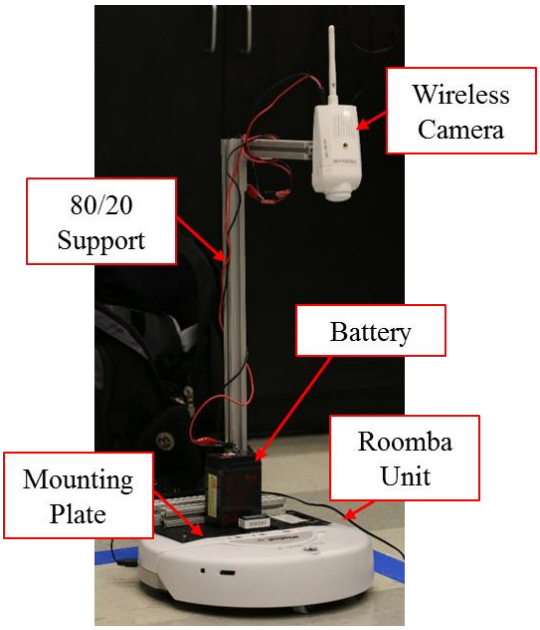

Figure 2. The Roomba-camera system.

Initially, the camera was mounted directly to the mounting plate, such that it pointed parallel to the Roomba. However, this led to issues with field of vision. Specifically, if an obstacle occluded part of a wall, the edge detection function would not be able register a sufficient edge length. This is illustrated in Figure 3, where (a) shows a line along the floor-wall edge while (b) does not. This problem was alleviated by completely changing the viewing angle; another solution could have filtered specifically for these lines, which would have allowed for much shorter line lengths to be detected. Because it was desired to perform all of the edge detection in one step in the interest of processing efficiency, this solution was not pursued.

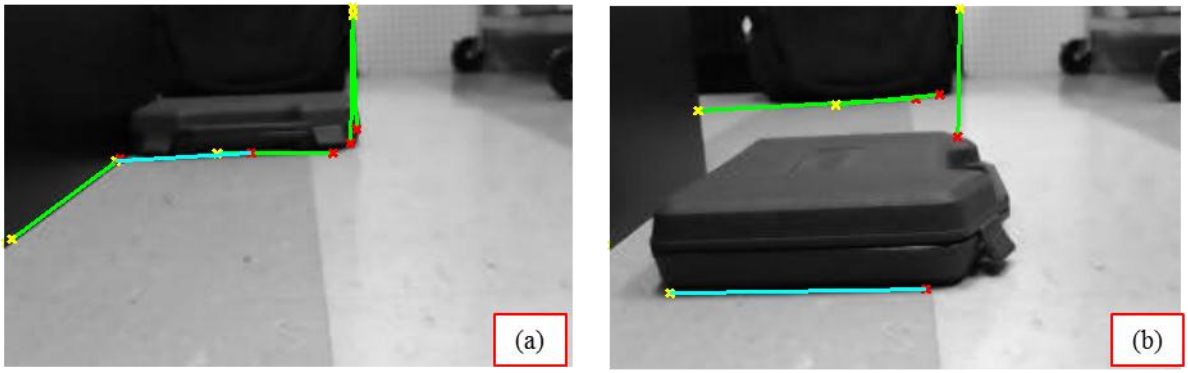

Figure 3. Camera image showing (a) sufficient and (b) insufficient left wall length to register.

2.3 Programming

The movement, mapping, and object detection program is consolidated within a single script, *roombaDriveMap_withPath.m*, which is included in the Appendix. This script calls three separate functions that perform object recognition, obstacle detection, and path detection. The movement logic and map generation occurs within the script, based on outputs from the three functions.

2.3.1 Object Recognition

A core function of the robot is to routinely scan and search for the desired object. To perform this task, blob analysis is utilized. Before each increment in its motion through the room, the robot takes a snapshot of the floor with its camera and converts it to grayscale and then to binary. A very high threshold, 190 on a scale of 0 to 255, was used for the grayscale to binary process to filter undesired elements, such as scuff marks and portions of the bounding tape. The corresponding grayscale image is seen in Figure 4 (a) (after being processing according to code discussed in the subsequent section), and the blob analysis results are seen in Figure 4 (b). Analysis is performed on this binary image. Two vision toolbox MATLAB functions are used: bwboundaries identifies the blobs in the binary image, and regionprops extracts their geometric information – namely area, centroid, and perimeter. The program then determines the ratio of area to the squared perimeter for each object found to create a scale-invariant, non-dimensional number. However, the object' area must also be within a certain range, and the object's centroid must be in the region of the image corresponding to floor and not the Roomba itself. This multi-tiered process was deemed necessary during testing to eliminate false positives. If a blob fits these criteria, within a tolerance, the robot "declares" that it has found this object, highlights it, and stops all operations. The object recognition script is called in the main script preceding every robot motion increment. This function, *tempProcess.m*, is included in the Appendix.

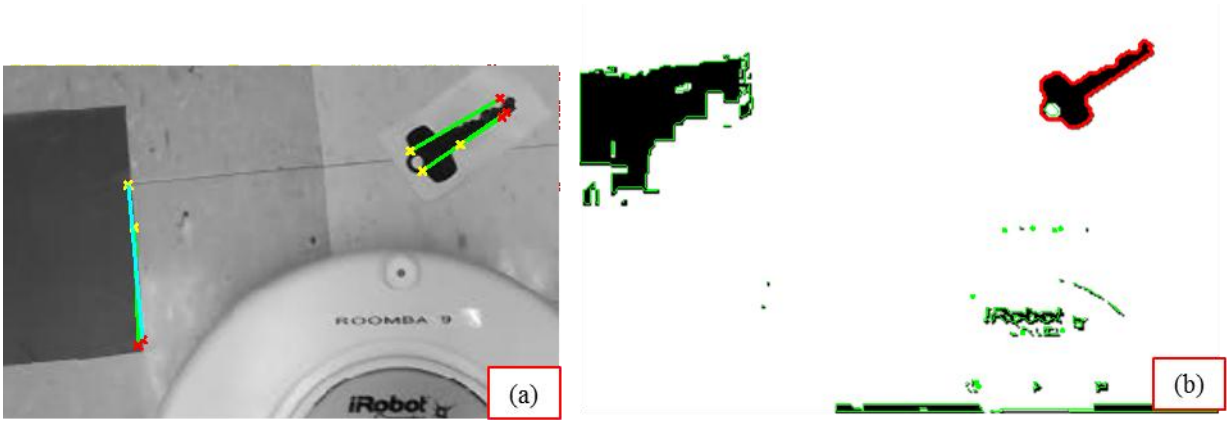

Figure 4. (a) Grayscale (processed) image and (b) binary analysis.

2.3.2 Obstacle Detection

The function *cameraProcess_2.m*, included in the Appendix, takes an image from the camera and outputs a vector of Booleans representing obstacles on the left, front, and right side of the Roomba. The image is first converted to grayscale and cropped to remove part of the Roomba from the image. A 10x10 average filter smoothens the image, and then a Canny edge detection function is run to find the edges in the image. A Hough transform is run on this image to determine the associated lines. The line segments corresponding to the edges are computed and plotted using the MATLAB functions houghpeaks and houghlines. Theta values are used to distinguish between horizontal and vertical lines; values of 77 to 90 degrees correspond to horizontal lines and values between 0 and 20 degrees correspond to vertical lines. Then, depending on the magnitude of the rho values associated with each line, obstacles are determined to be on the front, left, or right of the robot. If both of the endpoints of the lines lie on the Roomba itself, then the lines are excluded from further processing. An image from the camera illustrating detection of only a wall on the left side of the Robot is shown in Figure 5, and the detection of a wall on both the front and left is shown in Figure 6.


Figure 5. Processed image from the camera detecting the horizontal lines on the left wall and excluding points on the Roomba.

Figure 6. Processed image from the camera detecting the horizontal lines on the front and left front wall.

One issue encountered with the obstacle detection was the treatment of corners. If the robot is facing a corner, both horizontal and vertical lines will be detected. Thus, the lines can potentially be interpreted as an obstacle on the front and left simultaneously. In some instances this is a valid treatment, as the corner may actually represent an obstacle in front of the robot. However, the robot must also understand when it has cleared the corner and can continue following the wall. In order to overcome this issue, the endpoints output from the Hough lines function are tested to see if they are outside of the boundary of the robot. This was achieved through logic to test whether the column coordinate of the endpoint is less than a certain value where the boundary of the Roomba begins. If the endpoint is before this boundary, the line is discarded and motion can continue in the forward direction.

2.3.3 Real-Time Mapping

As the robot navigates the environment, the wheel encoders are polled in order to compute the distance travelled over each movement. This distance is added to the sum of the previous distances and appended to a vector of all of the positions that the robot has visited. Because rotations must be taken into account, each point of this vector is first multiplied by a rotation matrix. The rotation matrix is updated after each turn. Because the robot only rotates about the z-axis, premultiplying the previous rotation matrix by the individual rotation matrix associated with each turn yields the correct global result. The last coordinate pair in this vector is plotted after each movement as a blue circle. Turns are plotted as red ‘X’s. The walls and obstacles are indiscriminately plotted as black ‘X’s according to the results of the previous vision algorithms, and the desired object is plotted as a magenta diamond, if found. An example plot containing these items is seen in Figure 7. The robot’s initial position is at (x,y)=(0,0).

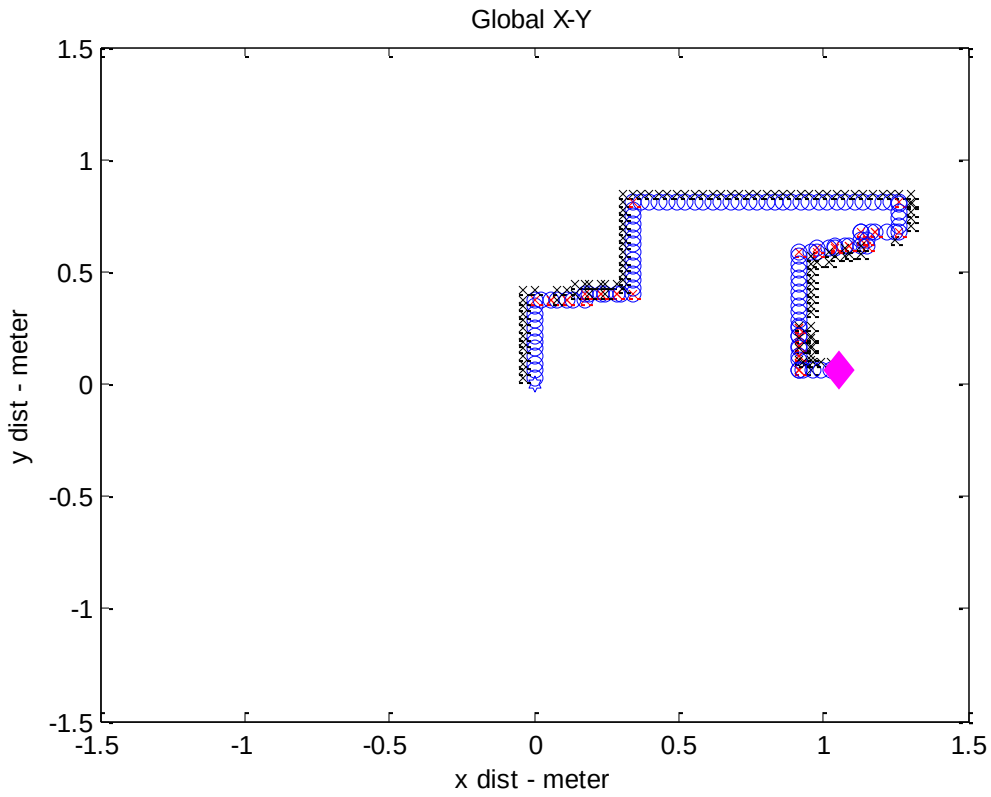

Figure 7. Global X-Y plot illustrating the Robot's path, walls, locations of rotation, and the location of the object.

2.3.4 Movement Logic

The movement logic was implemented such that the robot will follow the walls of an enclosed space until reaching the starting point, after which it will proceed asymptotically towards the center. The robot moves in a clockwise direction and has to begin parallel to a wall on its left side to guarantee room coverage. The robot will move straight as long as there is an obstacle on the left and no obstacle on the front. As soon as the robot does not observe a wall on the left, it will turn left. The robot will only turn to the right if there is an obstacle on both the front and left side. In this manner, the robot should be able to cover a complete, closed-loop path. However, in order to proceed asymptotically, a method was developed to determine when it has reached a point already visited.

2.3.5 Path Detection

In order to solve the path recognition problem, a function, *pathProcess2.m*, was written; this is included in the Appendix. The previous path points, namely the XY vector discussed previously, of the robot are given as inputs. Then, using a distance formula and atan2 functions, the cylindrical coordinates (rho and theta) from the robot's current position are computed to each of the robot's previous positions. Points within 0.25 meters and $\pm 31^{o}$ relative to the absolute left, front, and right of the robot are seen as having been visited and are avoided with the same logic as obstacles and walls.

3. IMPLEMENTATION

Each of the algorithms mentioned above had to be adapted to the unique programming of the Roomba robot. Because of the autonomous nature of the design, the integrity of the vision aspect of the robot is paramount. As such, the robot must move only in small increments to ensure that nothing is missed and that it does not accidentally run into an obstacle. A side-effect of this requirement is that vibrations from constant acceleration and deceleration can affect the camera. In order to counteract the vibrations, a small pause must be observed after each movement to give the vibrations time to dissipate. This slows down the process but ensures that the image taken will be valid and useful.

In the overall flow of the program, the main objective is to find the desired object. As such, the first thing the robot does in each iteration of the program is check the path in front of it for the object. It then crops the image used to check for the object and performs the obstacle detection algorithm on it. Finally, it uses the logic of the obstacle detection to determine where to go next, always showing preference to a forward motion, followed by a left turn, then a right turn, depending on the path it sees or whether it has been to a position already. It then maps the new position and any objects it saw based on the odometry output of the last movement.

The obstacle detection algorithm was also tailored to the robot. Depending on lighting conditions, the algorithm may detect lines on or around the Roomba as being valid lines. This can cause obvious problems as these lines are not true real-world obstacles, and treating them as such can cause the robot to lose its path and ultimately fail to map the room or find the object. To counteract this, the endpoints of each of the lines found are considered. If the endpoints are found to be within a certain region of the image, in this case the bottom-right corner where the body of the Roomba can be seen as in Figure 8, the line is discarded.

Following obstacle detection, the robot checks each of the points it has been to previously to determine if it is going to intersect that path. This functionality prevents the robot from running infinitely and ensures that it does map the entire room. In order to do this, the distance and orientation angle to each of its previously visited points is used. If the obstacle detection algorithm tells the robot to move in a certain direction, the path-checking algorithm then determines if that point it is about to hit has been previously visited, and, if so, tells the robot to turn in some direction. One of the ending conditions for the program is if the path-checking algorithm has determined that the robot has been to each of the positions on the robot's left, right, and front. It assumes the robot has already been to the position immediately behind it and that it has now mapped the entire space without finding the object.

4. RESULTS AND DISCUSSION

In general, the developed robotic system was successful. However, the occasional failures were due to several recurring causes. One of these was simply operating on a dirty floor, which led to features on the floor being detected as a line. Before the project demonstration, the floor was cleaned to avoid this, but random detections caused the robot to deviate from the "proper" course. This led to an incomplete mapping of the room. Next, this and other effects, as well as various mapping cases are presented and analyzed, and the findings are discussed below.

The blue tape laid out as walls worked well, as the high contrast aided edge detection. The angle threshold for detecting vertical and horizontal lines required a nontrivial amount of experimentation for properly detecting wall edges. Incorrect values would lead to the robot turning

too early, running into the wall, or too late, causing the robot to be too far away from a wall to detect it in the new orientation.

It was noted that if the robot was placed slightly unparalleled to the wall, the error would accumulate in the long run, resulting in slanted images and hence angled wall edges. This could have been better addressed by calibrating the robot's orientation to the angles of the detected edges. The obstacle detection code worked for a range of angles, making the robot very robust. The robot's movement speed was quite slow due to the heavy image processing and analysis to determine its next movement.

As a simplification only for the purpose of in-lab demonstration, a three-dimensional object was not used for detection. Instead, a silhouette of a key was printed on paper and taped within the constrained test area discussed above. The reason for the choice of a key silhouette was to test the robot's ability to detect objects whose shapes are more complex than rectangular and circular blocks tested in the previous machine vision lab. In this environment, the basic blob analysis proved to be very robust, and the robot detected the key most of the time. However, to prevent the robot from mistaking the boundary between the leading edge of the paper on which the key is printed and the floor as an obstacle, the key has to be placed about 45 degrees off the desired path. In addition, as blob analysis is known to be somewhat sensitive to large variation in scales and orientation, very accurate geometric properties of the key silhouette, like its area and centroid as viewed from the same camera placement, had to be known in advance. If the camera were to be elevated or lowered from its original position, these properties would have to be re-evaluated.

Obviously, the constrained environment and objective like in the demonstration could hardly be the case in real life situation. The lost items can be at any orientation as well as elevation from the camera, and the fact that the user has to know all about the item geometrically is unacceptable. The novel goal of this project is that images of daily objects can be stored in a database, e.g. a folder on the user's desktop computer. During the search, the robot can compare and match surrounding objects with these reference images. One such method is explored below.

Prior to the implementation of basic blob analysis, a more advanced and practical object detection method was researched and developed. It uses the Speeded-up Robust Features algorithm, or SURF. The algorithm extracts unique features of the object, giving each of them a "description". It is performed again with the snapshot from the robot's camera. If the "descriptions" of the reference object matches those of an object the robot "sees" on its way, that object is declared found. Multiple objects can be found at the same time and the found items are simply crossed off the list. Iterative methods such as Random Sample Consensus (or RANSAC) can be used to eliminate noises and match items more accurately.

Figure 8 (a) shows a possible flow chart of the main script for the image detection and movement procedure that would utilize this method. Figure 8 (b) shows a flow chart of the function *ObjectScan.m* that detects, compares and refines SURF features; this code is included in the Appendix. It is observed that a user can instruct the robot to find one, several, or all objects stored in the database. The function needs only the reference image and the snapshot for comparison. Using this script the robot will continue to search until all items are found.

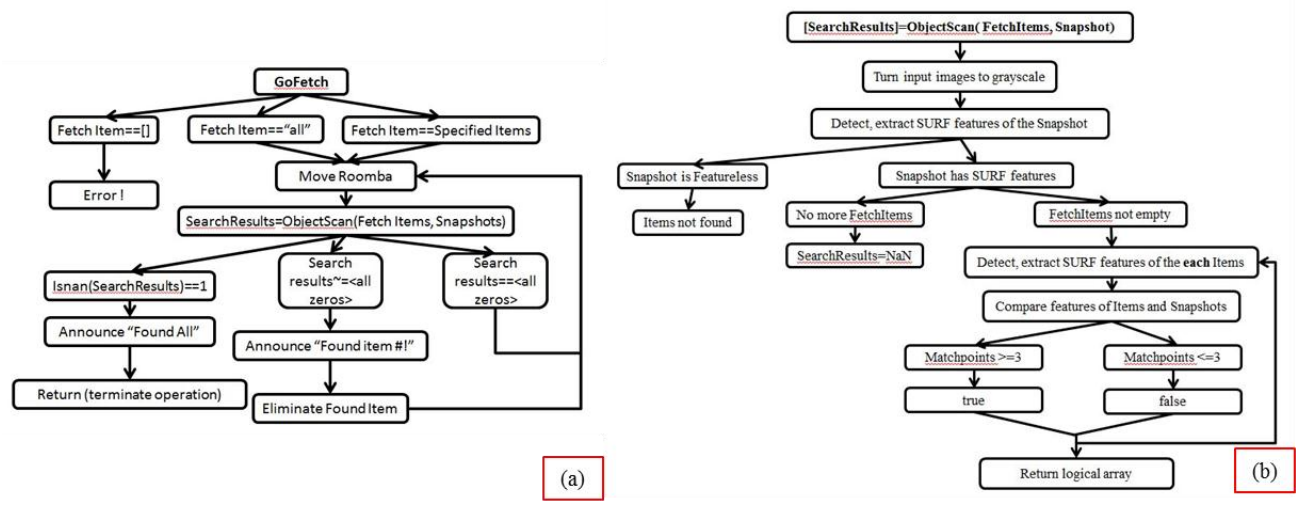

Figure 8. Alternate overall function (a) and object detection (b) pseudo-code

Unfortunately, it was not discovered prior to the project demonstration that the reference image and the environment image needed to be of the same quality and resolution for proper comparison. This is demonstrated below, in Figure 9. The high quality reference image (left) and camera snapshot (right) are shown with their strongest features highlighted in green circles. No feature matches can be found because of the different image quality. The difference is too large to be overcome by adjusting thresholds and input arguments.

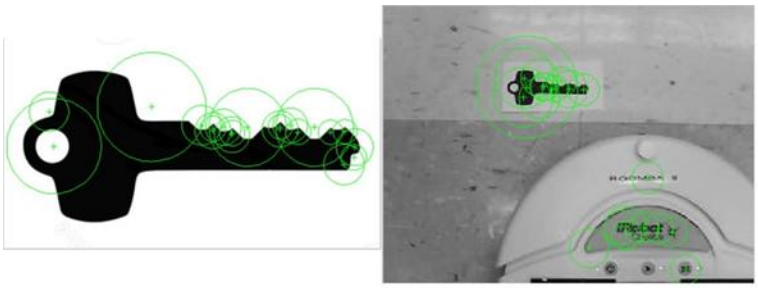

Figure 9. Comparing high quality (left) and low quality (right) images yield no feature matches.

In Figure 10 (a), however, when comparing images with similar quality, the object can be matched significantly better, even though the objects are in different orientations. Furthermore, even when the object is partially obstructed by the robot, a sufficient number of feature matches can be obtained. Limited testing of these cases suggests that the object can be identified if at least three feature matches are found between the reference image and the snapshot. While the basic blob analysis requires many constraints regarding the environment and robot setup, the SURF features detection and comparison method only requires the quality between the images in comparison be of similar quality, which can be easily achieved with proper preparation.

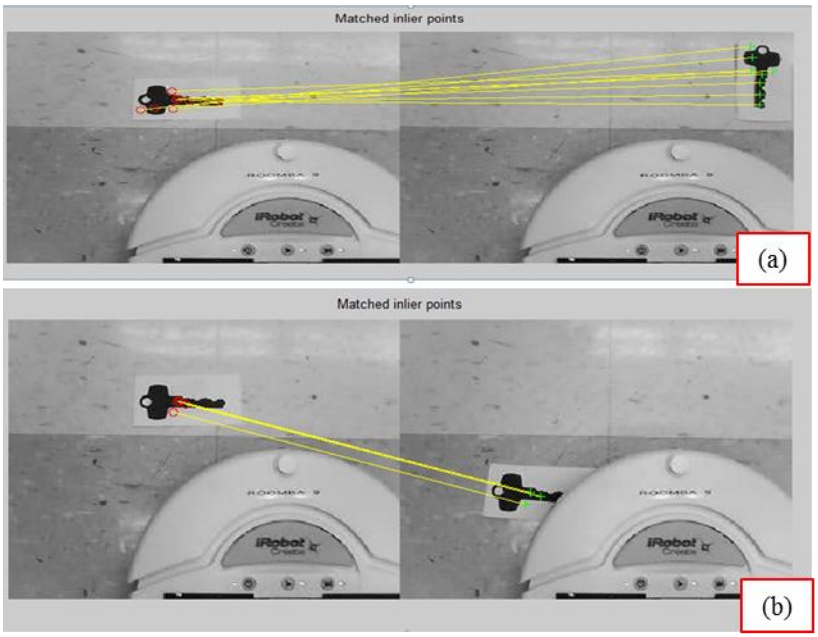

Figure 10. Successful matching of images without (a) and with (b) partial obstruction.

For reference, the following figures are included to highlight successful operation. Figure 11 shows the successful mapping of a room containing no object. In this case, path is clearly clockwise asymptotic, as desired. Figure 12 shows mapping in a room without obstacles, and Figure 13 shows mapping in a room with an obstacle. In the bottom right of each figure, the corresponding object recognition, edge detection, and the global X-Y plot are shown in that order.

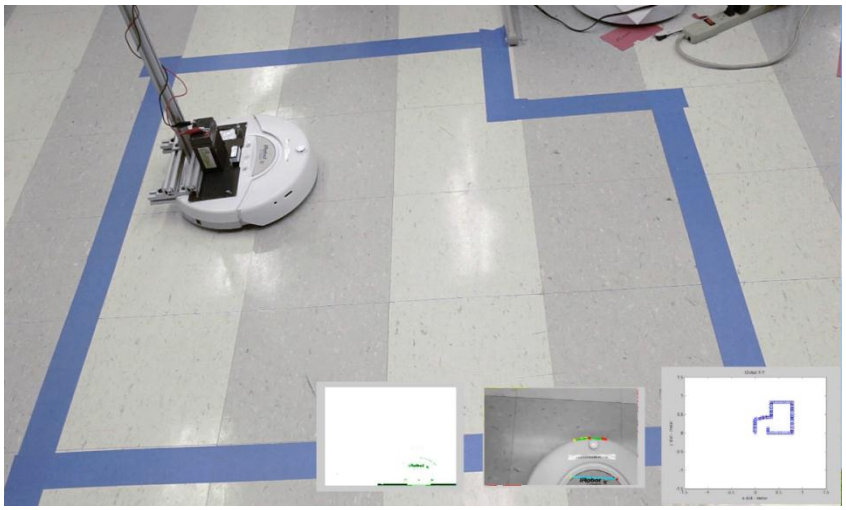

Figure 11. Scenario for mapping without object.

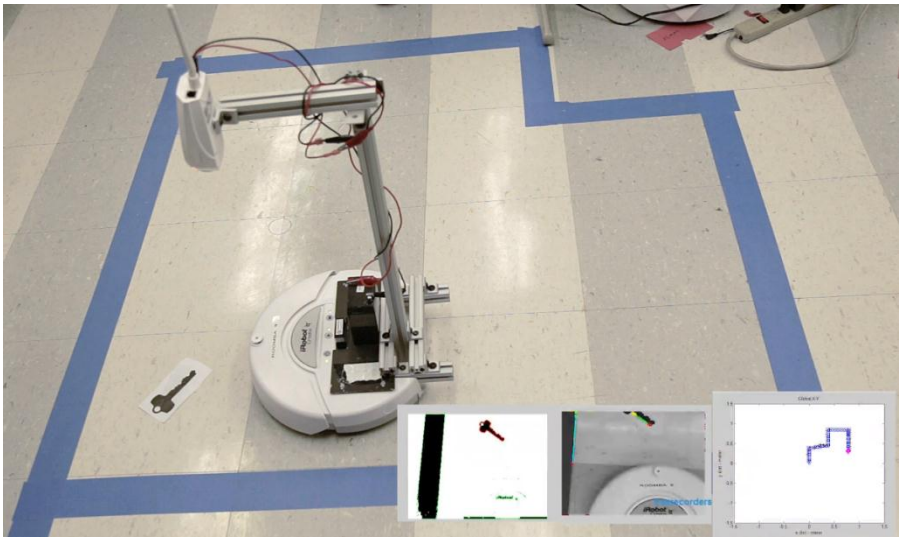

Figure 12. Scenario for mapping with object.

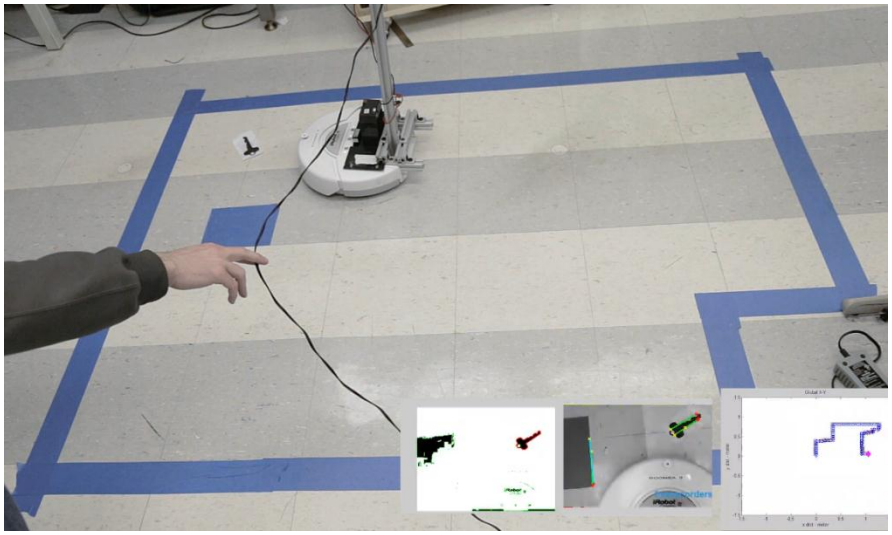

Figure 13. Scenario for mapping with object and obstacle.

5. LEARNING EXPERIENCES

While working on this project, some aspects worked successfully with only minor adjustments. Other aspects, however, proved more difficult to develop. These more difficult aspects led to positive learning experiences and an improved product.

One such difficulty was the relationship between the Roomba's accuracy of turning and battery strength. Variations in battery strength made some movements more unpredictable, such as turning. The Roomba is turned using a command telling it how many degrees to rotate. However, with varying battery strength comes varying motor power, so while the robot will get close to the desired angle, the discrepancies needed to be addressed to ensure accuracy.

In working with the vision system, it became apparent that changes in light can greatly affect the outcome of the algorithm. This was discovered when variations in lighting and shadows

caused the algorithm to detect lines on the Roomba itself and cause unpredictable movements. However, with these errors came the necessary improvements in the code and knowledge of how to implement it correctly the first time in the future.

Due to the addition of the camera support structure atop the Roomba, the end of each movement step was followed by a small oscillation of the camera-Roomba system about the wheel axle. This phenomenon resulted in occasional poorer quality images. This was accounted for by pausing after every stop command to allow time for the vibrations to dissipate. However, in a future iteration of the design, a more continuous approach to motion could be used. With the use of parallel processing or a threaded program, the vision algorithms which were a part of the overall program could be run on a continuous basis that constantly updates its output to be visible to the overall program. This was not feasible in the initial prototype because the image took an appreciable amount of time to reach the computer and MATLAB is a singly-threaded program by nature. With this improvement the robot could be told to move continuously in a single direction while constantly checking the vision conditions and then use the existing logic to determine where to go.

Another potential improvement to the design would be to use the focal parameters of the camera to transform the camera coordinates into real-world coordinates. This process could be used to determine where the obstacle lines and endpoints are in relation to the Roomba and could be used in the motion algorithm. With this knowledge the program could simply command the robot to move forward a certain distance that would bring it to the boundary of the wall, rather than constantly checking the surroundings with small motion intervals in between each check. This could also be used to help the robot clear corners more efficiently and plot its previous path visually on an image to help the user know exactly where it has been.

Overall, this project was a learning experience for all involved. Through a great deal of testing and debugging, the finished product functioned well. It also became apparent that there is never only one way to accomplish a task, exemplified by the various options considered above. Robotics is, by nature, a broad subject that encapsulates many disciplines, and this project helped to expand the way of thinking for all involved.

6. APPENDIX

6.1 roombaDriveMap_withPath.m

```
% roombaDriveMap_withPath.m
% run this script to operate the roomba mapping and object recognition

clear; clc;
close all;
figure(1) % global map
figure(3) % blob analysis
figure(8) % edge detection
pause % position where you want them to be, then continue

%object locating
areaThresh=830;
ratThresh = 0.0244;
turnRightCount = 0;
turnLeftCount = 0;

% this SCRIPT currently moves roomba via obstacle vector or until max time
% reached. plotting occurs fairly close to real time, it basically plots
% current position right before it moves.

roombaClose()
[serPort] = roombaOpen('COM5');


% turning calibration
% [-Angle,+ Angle] both are 0.1 m/s turn
angleCAL=[(90-87.25) -(90-87.25)]; %RECALIBRATED ON 12/3

start=[0 0]'; %starting [X Y]
XY=start;

maxMappingTime=60*20; % sec % note, 5 min is max demo time...
maptime=[];


% WALL LEFT OFFSET
O_L=(1.5)*0.0254; % inch to meter
O_R=(10000000+1.5)*0.0254; % not actually used
O_F=(1.5)*0.0254;


%% OUTPUT PLOT OF ROOM
figure(1) %DO NOT CLOSE
plot(XY(1),XY(2),'bh')
drawnow
title('Global X-Y')
```

```
% map size
axis([-1.5 1.5 -1.5 1.5]) % (meter)
axis manual
xlabel('x dist - meter')
ylabel('y dist - meter')
hold on

% rotation matrix anonymous function
RZ=@(TH)round([cos(TH) sin(TH); -sin(TH) cos(TH)]); %% ROUNDED
RR=RZ(0); % initially zero angle
tic % start timer

T=start; %initialize relative x y
n=1; % initialize #data point (once used in earlier test code)
preTurn=start;

[object gray ap area perim]=...
    tempProcess('http://192.168.7.202/image.jpg',areaThresh,ratThresh);

if object
    disp('game over')
    figure(1)
    plot(XY(1,end),XY(2,end),'md','linewidth',5)
    return
end

obstacle=cameraProcess_2('http://192.168.7.202/image.jpg')
obPlot(:,1)=RR'*(T(:,end)+[-O_L 0]')+preTurn(:,end)
plot(obPlot(1,end),obPlot(2,end),'kx')
drawnow
pathVec=[0 0 0];

while toc<maxMappingTime % X sec total operation time
    %% Drive Straight
    % Obstacle on left and no obstacle in front
    if (obstacle(1) && ~obstacle(2))  || (pathVec(1) && ~pathVec(2))
        roombaDrive(serPort, 0.1, inf) %drive straight
        pause(.4)

        roombaDrive(serPort, 0, 0) %stop

        [object gray ap area perim]=...

tempProcess('http://192.168.7.202/image.jpg',areaThresh,ratThresh);
        if object
            disp('game over')
            figure(1)
            plot(XY(1,end),XY(2,end),'md','linewidth',5)
            return
        end



        obstacle=cameraProcess_2('http://192.168.7.202/image.jpg')
```

```
        % get distance travelled
        [Distance] = sensorDistance(serPort);

        T(:,end+1)=T(:,end)+[0 Distance]';
        XY(:,end+1)=RR'*T(:,end)+preTurn(:,end); %global coordinates

        figure(1)
        plot(XY(1,end),XY(2,end),'bo') % plot move points as blue o
        drawnow

        % only plot if obstacles, not prev paths
        if (obstacle(1) && ~obstacle(2))
            obPlot(:,end+1)=RR'*(T(:,end)+[-O_L 0]')+preTurn(:,end);
            plot(obPlot(1,end),obPlot(2,end),'kx')
            % plot obstacles points as black lines -
            drawnow
        end

        pathVec=pathProcess2(XY)
        n=n+1;
    end
    %% Turn left
    % No obstacle on left --> absolutely NOTHING on left
    if (~obstacle(1) ) && (~pathVec(1) )
        angle=90; %deg
        % at one point, turns other than +90 degrees were considered here.
        if angle<0
            angleTrue=angle+angleCAL(1)*abs(angle/90);
        elseif angle>0
            angleTrue=angle+angleCAL(2)*abs(angle/90);
        end


        % drive forward to clear the obstacle, if clear to do so
        if ~obstacle(2) && ~pathVec(2)
            roombaDrive(serPort, 0.1, inf) %drive straight
            pause(.5)
            roombaDrive(serPort, 0, 0)
            [object gray ap area perim]=...

tempProcess('http://192.168.7.202/image.jpg',areaThresh,ratThresh);

            if object
                disp('game over')
                figure(1)
                plot(XY(1,end),XY(2,end),'md','linewidth',5)
            return
            end

            obstacle=cameraProcess_2('http://192.168.7.202/image.jpg');
            [Distance] = sensorDistance(serPort);

            T(:,end+1)=T(:,end)+[0 Distance]';
```

```
            XY(:,end+1)=RR'*T(:,end)+preTurn(:,end); %global coordinates
            figure(1)
            plot(XY(1,end),XY(2,end),'bo')% plot move points as blue o
            drawnow
        end


        % turn in place
        roombaTurnAngle(serPort, 0.15, angleTrue)

        %counter set to compensate for non-perfect turns
        turnLeftCount = turnLeftCount + 1;
        if (turnLeftCount >= 3)
            pause(0.25)
            roombaTurnAngle(serPort, 0.15, +.33)
            turnLeftCount = 0;
        end
        pause(1)

        % update rotation matrix
        RR=RZ(angle*pi/180)*RR;

        [object gray ap area perim]=...

tempProcess('http://192.168.7.202/image.jpg',areaThresh,ratThresh);

        if object
            disp('game over')
            figure(1)
            plot(XY(1,end),XY(2,end),'md','linewidth',5)
            return
        end

        obstacle = cameraProcess_2('http://192.168.7.202/image.jpg')

        figure(1)
        plot(XY(1,end),XY(2,end),'rx') %plot turn points as red x
        drawnow

        T(:,2:end)=[]; %reset rel moves here
        preTurn=XY(:,end);
        pathVec=[1 0 0] % force a forward motion here
        n=n+1;
        end

    %% Turn Right
    % Obstacle/path on front & no path on right
    if (obstacle(2) || (pathVec(2) && ~pathVec(3)))

        angle=-90; %deg

        if angle<0
            angleTrue=angle+angleCAL(1)*abs(angle/90)
```

```
        elseif angle>0
            angleTrue=angle+angleCAL(2)*abs(angle/90)
        end

        % turn in place
        roombaTurnAngle(serPort, 0.15, angleTrue) %calibration
        turnRightCount = turnRightCount + 1;
        if (turnRightCount >= 3)
            pause(0.25)
            roombaTurnAngle(serPort, 0.15, -.33)
            turnRightCount = 0;
        end
        pause(1)

        % only plot if obstacles, not prev paths
        if (obstacle(2) && ~obstacle(3))

            % plot corner points
            obPlot(:,end+1)=RR'*(T(:,end)+[-O_L 0]')+preTurn(:,end);
            obPlot(:,end+1)=RR'*(T(:,end)+[-O_L O_F]')+preTurn(:,end);
            obPlot(:,end+1)=RR'*(T(:,end)+[0 O_F]')+preTurn(:,end);
            plot(obPlot(1,end-2:end),obPlot(2,end-2:end),'kx')
                % plot obstacles points as black lines -
            drawnow
        end

        % update rotation matrix
        RR=RZ(angle*pi/180)*RR;


        [object gray ap area perim]=...

tempProcess('http://192.168.7.202/image.jpg',areaThresh,ratThresh);
        if object
            disp('game over')
            figure(1)
            plot(XY(1,end),XY(2,end),'md','linewidth',5)
            return
        end

        obstacle=cameraProcess_2('http://192.168.7.202/image.jpg')

        figure(1)
        plot(XY(1,end),XY(2,end),'rx') %plot turn points as red x
        drawnow

        T(:,2:end)=[]; %reset rel moves here
        preTurn=XY(:,end); % var might not req. history indexing (:,end+1)
        pathVec=[1 0 0]
        n=n+1;
    end

    %% Mapping ends before object found. If time expires before mapping is
complete, this will not run.
```

```
    if toc<maxMappingTime && all(pathVec)
        maptime=toc;
        toc=maxMappingTime;
        roombaClose();
    end

end

if isempty(maptime)
    error(['max mapping time of ' num2str(maxMappingTime) 'sec reached'])
    error('mapping incomplete')
else
    disp(['mapping time was ' num2str(maptime) 'sec'])
    disp('mapping complete')
    disp('object not found')
end

roombaClose();
```

6.2 tempProcess.m

```
function [object gray ap area perim]=tempProcess(img,areaThresh,ratThresh)
% blob analysis function. outputs are useful for testing but otherwise,
% only 'object' (logical) is required. input is image location and desired
% object parameters (approx area and area/perim^2 ratio

% initialize output
object=false;
a=imread(img);

%grayscale
gray=rgb2gray(a);

%bw threshold
thresh=190;

gray2=imcomplement(gray);
bw=im2bw(gray2,thresh/255);

[B L N A]=bwboundaries(bw);

%green boundary around blobs
figure(3)
imshow(imcomplement(bw))
drawnow
hold on

for i=1:length(B)
        plot(B{i}(:,2),B{i}(:,1),'g')
end
```

```
%get properties of blobs
allProps=regionprops(L,'Area', 'Perimeter','Centroid');

err=0.0025;
err2=50;
ap=[];area=[];perim=[];ratio=[];
%track blobs
for j=1:length(B)
    ap=[ap allProps(j).Area/(allProps(j).Perimeter)^2];
    area = [area allProps(j).Area];
    perim = [perim allProps(j).Perimeter];
    ratio = [ratio allProps(j).Area/(allProps(j).Perimeter^2)];

    if (abs(ratio(end))>(ratThresh-err)) ...
            && (abs(ratio(end))<(ratThresh+err))

        % first failure
        if (allProps(j).Centroid(1)>170 && allProps(j).Centroid(2)>150)

        % second failure
        elseif ~((abs(allProps(j).Area )>(areaThresh-err2)) ...
                && (abs(allProps(j).Area )<(areaThresh+err2)))

        % otherwise, pass
        else
            plot(B{j}(:,2),B{j}(:,1),'r', 'LineWidth',2)
            drawnow
            disp('found')
            object=true;
        end
    end

end
hold off

if ~object
    disp('not found')
end

end
```

6.3 cameraProcess_2.m

```
function visVec = cameraProcess_2(img, varargin)
% this function takes in camera ip and outputs 3x1 vector of 'obstacles':
% visVec(1) = obstacle to left, visVec(2) = obstacle to front, visVec(3) =
% obstacle to right. note, visVec(3) no longer used

I0=imread(img);
I0=rgb2gray(I0);

OFFSET=10;
% cropping
I0=I0(2*OFFSET:end-4*OFFSET,2.5*OFFSET:end-OFFSET);

% smoothing
average=fspecial('average',10);
averaged=imfilter(I0,average,'replicate');%'replicate' to eliminate border


% initialize so if no lines found, these stay at zero
   WALL_L=0;
   OBS=0;
   WALL_R=0;

BW=edge(averaged,'canny',[.55 .65],2);

% the following mainly c/o mathworks, inc.
[H,T,R] = hough(BW);

P  = houghpeaks(H,5,'threshold',10);

if ~isempty(P)
    x = T(P(:,2));
    y = R(P(:,1));

    lines = houghlines(BW,T,R,P,'FillGap',45,'MinLength',25);
    figure(8), imshow(I0), hold on
    max_len = 0;

    if ~isempty(fieldnames(lines)) % if lines are found

        for k = 1:length(lines)
            xy = [lines(k).point1; lines(k).point2];
            plot(xy(:,1),xy(:,2),'LineWidth',2,'Color','green');

            % plot beginnings and ends of lines
            plot(xy(1,1),xy(1,2),'x','LineWidth',2,'Color','yellow');
            plot(xy(2,1),xy(2,2),'x','LineWidth',2,'Color','red');

            % determine the endpoints of the longest line segment
            len = norm(lines(k).point1 - lines(k).point2);
```

```
            if ( len > max_len)
                max_len = len;
                xy_long = xy;
            end
        end

        % highlight the longest line segment
        plot(xy_long(:,1),xy_long(:,2),'LineWidth',2,'Color','cyan');

        for n=1:length(lines)
            THETA(n)=lines(n).theta;
            RHO(n)=lines(n).rho;
        end

        % line criteria
        horiz=(abs(THETA)<=90 & abs(THETA)>=77);

        for i = 1:length(lines)
            if lines(i).point2(1) < 70 | ((lines(i).point1(1) > 83 ...
                & lines(i).point1(2) > 90) & (lines(i).point2(1) > 83 ...
                & lines(i).point2(2) > 90))& horiz(i)

                horiz(i) = false;
            end
        end
        vert=(abs(THETA)<=20);

        % proximities

        % obstacles
        if any(horiz) & any(abs(RHO(horiz))>=46 & abs(RHO(horiz))<=100)
            OBS=1;
        end

        if any(vert) & any(abs(RHO(vert))<120)
            WALL_L=1;
        end

        if any(sum(vert)) & any(abs(RHO(vert))>200)
            WALL_R=1;
        end

    else
            disp('no lines')
    end
else
    disp('no lines')
end

visVec=[WALL_L OBS WALL_R];
drawnow

end
```

6.4 pathProcess2.m

```
function pathVec = pathProcess2(XY)
% takes in global path coordinates from movement and converts them to
% 'virtual wall' for use in roombaDriveMap_withPath

% initialize
pathVec=[false false false];

% cylindrical coord R-Theta criteria
dist=0.250; % meter
off=31; % deg --dont exceed 45degrees or there will be overlap


if length(XY(1,:))<2
    pathVec; % too few points to do this
else
    DISPL=XY(:,end)-XY(:,end-1); % determine type of movement
                                 % (fwd, rev, left, right (GLOBAL))



    % First, compute all radii and thetas relative to current position.
    for n=1:length(XY)-1
        R(n)=((XY(1,n)-XY(1,end))^2 + (XY(2,n)-XY(2,end))^2 )^(1/2);
        TH(n)=atan2((XY(2,n)-XY(2,end)),(XY(1,n)-XY(1,end)))*180/pi;
    end

    % Then, compare computed radii and thetas to path/wall criteria,
    % based on last displacement

    % if X MOVEMENT
    if abs(DISPL(2))<1e-10 && DISPL(1)~=0

        % +X GLOBAL
        if DISPL(1)>0
            front=((TH>=(0-off) & TH<=(0+off)) | (TH>=(360-off) ...
                & TH<=(360+off))) & (abs(R)<=dist);
            if any(front)
                pathVec(2)=true;
            end

            % WALLS (RELATIVE TO MOVEMENT)
            top=((TH>=(90-off) & TH<=(90+off)) | (TH>=(-270-off) ...
                & TH<=(-270+off))) & (abs(R)<=dist);
            if any(top)
                pathVec(1)=true;
            end
            bottom=((TH>=(-90-off) & TH<=(-90+off))|(TH>=(270-off) ...
                & TH<=(270+off))) & (abs(R)<=dist);
```

```
            if any(bottom)
                pathVec(3)=true;
            end

        % -X GLOBAL
        elseif DISPL(1)<0
            front=((TH>=(180-off) & TH<=(180+off))|(TH>=(-180-off) ...
                & TH<=(-180+off))) & (abs(R)<=dist);
            if any(front)
                pathVec(2)=true;
            end

            % WALLS (RELATIVE TO MOVEMENT)
            top=((TH>=(90-off) & TH<=(90+off))|(TH>=(-270-off) ...
                & TH<=(-270+off))) & (abs(R)<=dist);
            if any(top)
                pathVec(3)=true;
            end
            bottom=((TH>=(-90-off) & TH<=(-90+off))|(TH>=(270-off) ...
                & TH<=(270+off))) & (abs(R)<=dist);
            if any(bottom)
                pathVec(1)=true;
            end
        end

    % if Y MOVEMENT
    elseif abs(DISPL(1))<1e-10 && DISPL(2)~=0

        % +Y GLOBAL
        if DISPL(2)>0
            front=((TH>=(90-off) & TH<=(90+off))|(TH>=(-270-off) ...
                & TH<=(-270+off))) & (abs(R)<=dist);
            if any(front)
                pathVec(2)=true;
            end

            % WALLS (RELATIVE TO MOVEMENT)
            left=((TH>=(180-off) & TH<=(180+off))|(TH>=(-180-off) ...
                & TH<=(-180+off))) & (abs(R)<=dist);
            if any(left)
                pathVec(1)=true;
            end
            right=((TH>=(0-off) & TH<=(0+off))|(TH>=(360-off) ...
                & TH<=(360+off))) & (abs(R)<=dist);
            if any(right)
                pathVec(3)=true;
            end

        % -Y GLOBAL
        elseif DISPL(2)<0
            front=((TH>=(-90-off) & TH<=(-90+off))|(TH>=(270-off) ...
                & TH<=(270+off))) & (abs(R)<=dist);
            if any(front)
                pathVec(2)=true;
            end
```

```
            % WALLS (RELATIVE TO MOVEMENT)
            right=((TH>=(180-off) & TH<=(180+off))|(TH>=(-180-off) ...
                & TH<=(-180+off))) & (abs(R)<=dist);
            if any(right)
                pathVec(3)=true;
            end
            left=((TH>=(0-off) & TH<=(0+off))|(TH>=(360-off) ...
                & TH<=(360+off))) & (abs(R)<=dist);
            if any(left)
                pathVec(1)=true;
            end
        end

    % if no motion because both XY unchanged
    else
        disp('No Motion')
    end

end
end
```

6.5 ObjectScan.m

```
function [ SearchResult ] = ObjectScan( FetchItems, Snapshot )
results=zeros(length(FetchItems));
Snapshot_gray=rgb2gray(Snapshot);
Snapshot_pts=detectSURFFeatures(Snapshot_gray);
[f1, vpts1] = extractFeatures(Snapshot_gray, Snapshot_pts);
if isempty(FetchItems)==1
    result=nan;
elseif isempty(Snapshot_pts)==0
    for i=1:length(FetchItems)
         % Step 1: input ref image and cam snapshot
         I1 = rgb2gray(imread('Key5.jpg'));

         %Step 2: Dectect SURF Features
         points1 = detectSURFFeatures(I1);

         %Step 3: Extract Feature
         [f1, vpts1] = extractFeatures(I1, points1);

         %Step 4: Match features
         index_pairs = matchFeatures(f1, f2, 'MatchThreshold', 1);
         matched_pts1 = vpts1(index_pairs(:, 1));
         matched_pts2 = vpts2(index_pairs(:, 2));

         % Step 5: Clean up noises: RANSAC (estimate geomtrictranform)
         %[tform,inlierPtsOut,inlierPtsIn] =
estimateGeometricTransform(matched_pts2,matched_pts1,'affine');
         if min(length(matched_pts1),length(matched_pts2))>10
             results(i)=1;
```

```
            end
        end
    end
    %%
    SearchResult=result;
    end
```