Turtlebot 2 Mini Project

EE 144

Department of Electrical Engineering, UC Riverside

Project Team Member(s)	Alberto Arriaga Felix Kai Wen
Final Revision Date	3/ 09/2019
Revision	Version 1.1
Permanent Emails of all team members	albertoarriagafelix@gmail.com (Alberto) kwen003@gmail.com (Kai)

Revisions

Version	Description of Version	Author(s)	Date Completed
1.0	First Version	Aberto Arriaga Felix, Kai Wen	6/09/2017
1.1	Last Version	Aberto Arriaga Felix	3/10/2019

Table of Contents

1 Summary of Project

In this project we are working with a real robot called Turtlebot 2 as shown in figure 1.1. The objective is to make the robot kick an orange ball into a goal represented by two green cylinders. The ball is initially placed a preselected distance away from the two green cylinders. The robot is initially placed somewhere in the gray area as shown in figure 2. The robot should be able to detect the goal and the ball and navigate to a position where it can push the ball in a successful trajectory into the goal.

Figure 1.1 – Turtlebot 2.

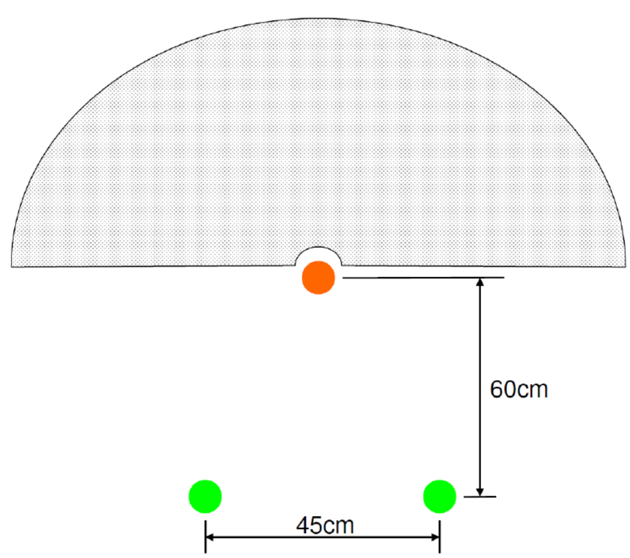


Figure 1.2 – The green cylinders are goal, and orange ball is above.

2 Robot Frame, Movement and Navigation

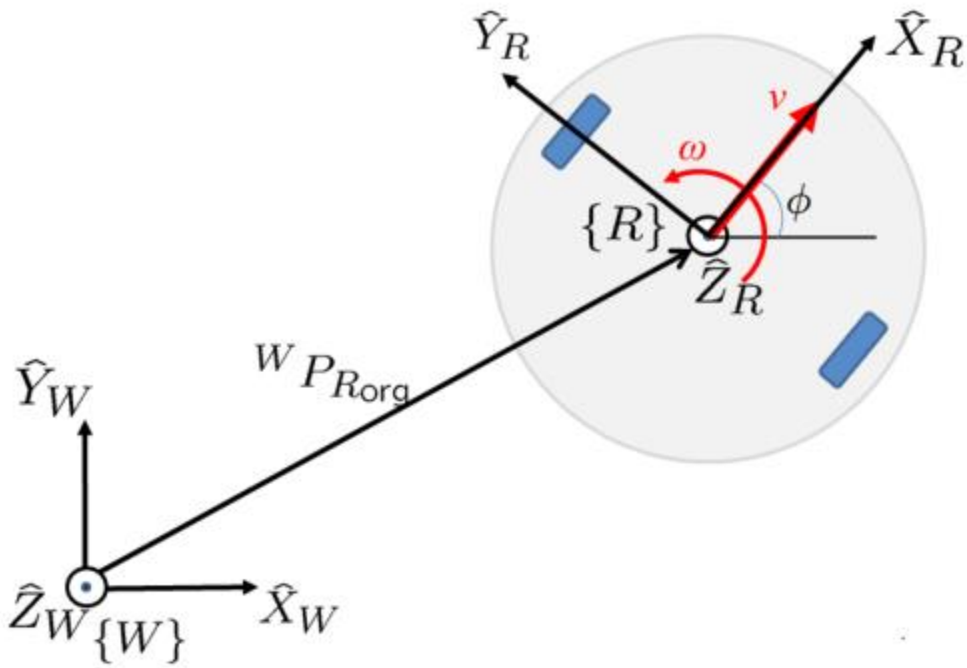


Figure 2.1 The definition of the robot and world coordinate frames

2.1 Mathematical Basis for Robot

The Turtlebot 2 robot is a modular robot, on which additional sensors can be placed. The robot uses differential drive for locomotion, it has two powered wheels (shown in blue in Figure 2.1.), located symmetrically about its center. By commanding the velocity of each of these wheels individually, we can obtain the desired linear (v) and rotational (ω) velocities for the robot.

To describe the position and orientation of the robot, we attach a "robot coordinate frame", {R}, to it. The origin of this coordinate frame is centered between its powered wheels. The X axis of this frame is pointing forward (along the direction of the linear velocity v), the Y axis is pointing to the "left", and the Z axis is pointing up (see Fig. 1).

To track the position and orientation of the robot, we generally define a "world reference frame" {W}, in the same plane in which the robot moves (i.e., the floor plane). With this frame assignment, the robot's position is constrained to lie in the $x - y$ plane of frame {W}. Moreover, any rotation between the robot and the world frames will be a rotation about the Z axis. Therefore, the position of the robot with respect to the world reference frame will have the form:

$$ {}^{W}P_{R_{\text{org}}} = \begin{bmatrix} x \\ y \\ 0 \end{bmatrix} $$

while the rotation matrix expressing the orientation of the robot frame with respect to {W} will be of the form:

$$ {}^{W}_{R}R = \begin{bmatrix} \cos(\phi) & -\sin(\phi) & 0 \\ \sin(\phi) & \cos(\phi) & 0 \\ 0 & 0 & 1 \end{bmatrix} $$

2.2 MATLAB Simulation of Robot

Odometry refers to the basic process by which the robot keeps track of its position and orientation as it moves in the environment. Specifically, the robot has encoders attached to each of its powered wheels.

These encoders allow it to keep track of the wheels' rotation (and in turn, of the wheels' displacement). Additionally, the robot has a gyroscope, which measures its rotational velocity about the Z axis. By combining this information, the robot can track its pose (position and orientation) as it moves in the plane.

In general, odometry is accurate over short time periods, but it becomes quite inaccurate over time.

The following scripts make use of MATLAB timer objects. These allow us to run a given function at regular intervals, effectively creating a multi-threaded environment. We are using the following timers:

- **odometry timer**: this timer runs every 100 msec, and reads the current pose estimate from the robot's odometry. The current estimate is returned in the variable curr pose, which is a 3×1 vector, containing the robot's x and y position, and its orientation, φ in the world frame (see previous section for the definition of these). The units for x and y are meters, and φ is in rad. Additionally, the pose is stored in the variable robot poses, which contains the entire history of robot poses. Each column of this variable contains the x, y, φ of the robot pose, as well as the time, t, at which this pose estimate was recorded.
- **plotting timer**: this timer runs every 5 sec and plots the entire history of robot poses obtained from odometry. Two plots are generated: one showing the trajectory on the x − y plane, and one showing the variables x, y, φ over time.
- **velocity command timer**: this timer runs every 100 msec and sends velocity commands to the robot.

2.2.1 Moving in a square

Open loop approach:

The code will command the robot to move continuously in a square of size 5 × 5 m, at a speed of 0.5 m/sec. This is an "open-loop" control approach. Specifically, the robot to moves straight at 0.5 m/sec for 10 seconds, then turns on the spot by 90 degrees, and repeats indefinitely.

Code:

```
clear all
close all
% initialize
rosshutdown % to 'close' any previous sessions
rosinit('192.168.111.129'); % initialize Matlab ROS node
tbot = turtlebot  % the data structure that allows access to the turtlebot and its
sensors
% this command sets the current robot pose estimate to zero (it does not
% move the robot). This is equivalent to setting the ``world coordinate
% frame'' to coincide with the current robot frame.
resetOdometry(tbot)
% the variable robot_poses stores the history of robot pose received from odometry. It
% is declared as global for computational efficiency
global robot_poses
% we initialize robot_poses as an empty matrix with 10000 columns.
% This can store up to 1000 seconds of odometry, received at 10Hz.
robot_poses = zeros(4,10000);
% create a Matlab "timer" for receiving the odometry. This will effectively create a
new thread
% that executes 'curr_pose = get_pose_from_tbot_odometry(tbot);' every 100 msec.
% this will return the current robot pose in the variable curr_pose, and
% will also store the current pose, together with its timestamp, in robot_poses
odometry_timer = timer('TimerFcn','curr_pose = get_pose_from_tbot_odome-
try(tbot);','Period',0.1,'ExecutionMode','fixedRate');
% start the timer. This will keep running until we stop it.
start(odometry_timer)
% these are the variables that are used to define the robot velocity
lin_vel = 0;  % meters per second
rot_vel = 0;  % rad/second
% create a Matlab "timer" for continuoulsy sending velocity commands to the robot.
% This will create a new thread that runs the command setVeloc-
ity(tbot,lin_vel,rot_vel) every 100msec
velocity_command_timer = timer('TimerFcn','setVelocity(tbot,lin_vel,rot_vel)','Peri-
od',0.1,'ExecutionMode','fixedSpacing');
% start the timer. This will keep running until we stop it.
start(velocity_command_timer)
% create a Matlab timer for plotting the trajectory
plotting_timer = timer('TimerFcn','plot_trajectory(robot_poses)','Period',5,'Execu-
tionMode','fixedSpacing');
% start the timer. This will keep running until we stop it.
start(plotting_timer)
% make the robot move in a circle continuously
% since the commands to the robot are sent by velocity_command_timer,
% this way of commanding the robot's velocity is non-blocking.
% We are free to execute other Matlab commands here, and
% therefore, in general (but perhaps not always), this approach to
% commanding the robot's velocity is preferrable to using blocking calls
while 1
lin_vel = 0.5;
rot_vel = 0.0;
pause(10);
lin_vel = 0.0;
```

```
rot_vel = 0.5;
pause(3.2);
end
```

Closed loop approach:

We will implement the same square-motion as above but using a very simple “closed- loop” control approach. Specifically, note that in the implementation of the previous step we are not using the robot’s odometry to control the robot motion. Therefore, even if the robot’s odometry shows that it is not moving in the correct trajectory, we ignore that information completely. In this step, we will use the robot’s odometry to try to correct the errors in the trajectory.

We begin by observing that to have the robot move in a square of size 5 × 5 m, we can implement the following approach:

1. Initialization: set the robot’s desired orientation as φ ‘ = 0.
2. Define P 0 to be the current position of the robot.
3. Keep moving forward, while maintaining the robot’s orientation as close to φ ‘ as possible, until we reach a location that is at a distance of 5 meters to P 0 .
4. Set φ ’= φ ‘ + π/2.
5. Rotate in place, until the robot’s orientation equals φ ‘ (within some small tolerance threshold), then repeat from Step 2.

In order to keep the robot’s orientation close to the desired one in Step 3, we can use a simple proportional controller (P-controller). Specifically, if we define the error between the robot’s curent orientation, φ, and the robot’s desired orientation φ ‘, as Δφ = φ − φ ? , then we can choose the robot’s rotational velocity as:

$$\omega = -k\,p\,\Delta\varphi \quad (1)$$

By choosing the robot’s rotational velocity in this way, we seek to rotate the robot in a way that reduces the error between the desired and the current (estimated) orientation of the robot.

When computing the orientation errors, we should always make sure that the result is expressed within the interval [−π,π], to avoid unjustified large corrections (for example, if the desired orientation is 0, and the orientation estimate is 2π, the computed value of Δφ is 2π. This will result in a large rotational velocity computed by equation (1), even though the orientation is in fact perfect). To make sure the error is within the desired bounds, we can use the MATLAB function wrapToPi.

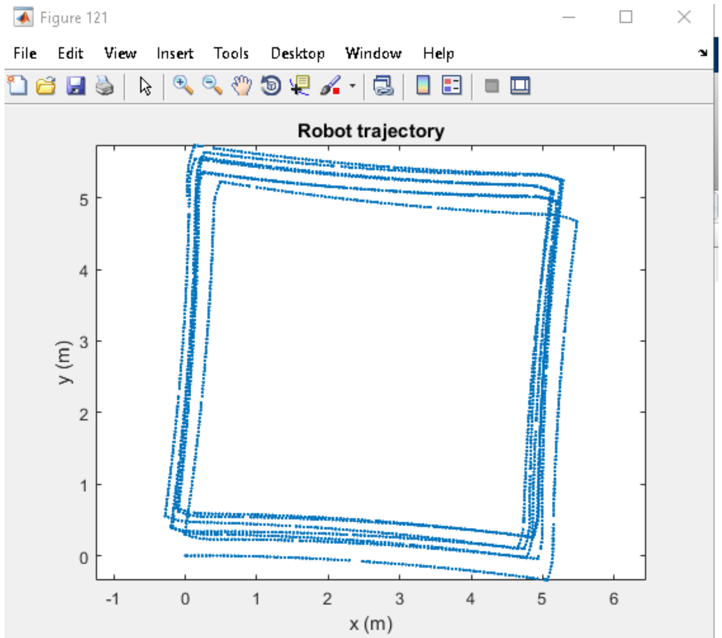
Code:

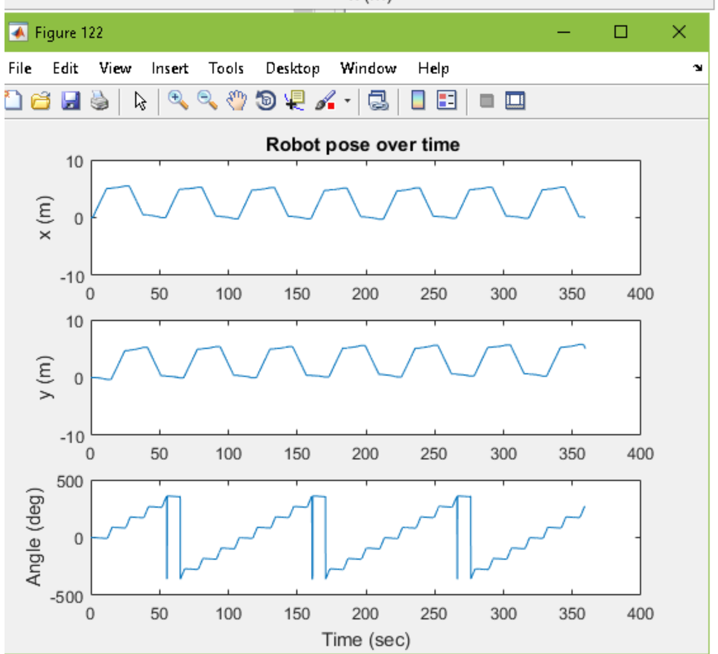
```
clear all
close all
% initialize
rosshutdown % to 'close' any previous sessions
rosinit('192.168.11.129'); % initialize Matlab ROS node
tbot = turtlebot  % the data structure that allows access to the turtlebot and its
sensors
% this command sets the current robot pose estimate to zero (it does not
% move the robot). This is equivalent to setting the ``world coordinate
% frame'' to coincide with the current robot frame.
resetOdometry(tbot)
% the variable robot_poses stores the history of robot pose received from odometry. It
% is declared as global for computational efficiency
global robot_poses
% we initialize robot_poses as an empty matrix with 10000 columns.
% This can store up to 1000 seconds of odometry, received at 10Hz.
robot_poses = zeros(4,10000);
% create a Matlab "timer" for receiving the odometry. This will effectively create a
new thread
% that executes 'curr_pose = get_pose_from_tbot_odometry(tbot);' every 100 msec.
% this will return the current robot pose in the variable curr_pose, and
% will also store the current pose, together with its timestamp, in robot_poses
odometry_timer = timer('TimerFcn','curr_pose = get_pose_from_tbot_odome-
try(tbot);','Period',0.1,'ExecutionMode','fixedRate');
% start the timer. This will keep running until we stop it.
start(odometry_timer)
% these are the variables that are used to define the robot velocity
lin_vel = 0;  % meters per second
rot_vel = 0;  % rad/second
% create a Matlab "timer" for continuoulsy sending velocity commands to the robot.
% This will create a new thread that runs the command setVeloc-
ity(tbot,lin_vel,rot_vel) every 100msec
velocity_command_timer = timer('TimerFcn','setVelocity(tbot,lin_vel,rot_vel)','Peri-
od',0.1,'ExecutionMode','fixedSpacing');
% start the timer. This will keep running until we stop it.
start(velocity_command_timer)
% create a Matlab timer cfor plotting the trajectory
plotting_timer = timer('TimerFcn','plot_trajectory(robot_poses)','Period',5,'Execu-
tionMode','fixedSpacing');
% start the timer. This will keep running until we stop it.
start(plotting_timer)
while 1
    %tic
    lin_vel = 0.4;
    rot_vel = 0.0;
    x_orig = curr_pose(1,1);
    y_orig = curr_pose(2,1);
    orien_orig = 180/pi*curr_pose(3,1);
    travel = 1;
    while (travel)
        x_curr = curr_pose(1,1);
        y_curr = curr_pose(2,1);
        dist =  abs(sqrt((x_curr-x_orig)^2-(y_curr-y_orig)^2))
        if (dist > 5)
            travel = 0;
```

```
        dist = 0;
    else
        travel = 1;
    end
    orien_curr = 180/pi*curr_pose(3,1)-90;
    delta_phi = abs(orien_curr - orien_orig);
    if(delta_phi > 1)
        if (orien_curr > orien_orig) %current orientation > orientation wanted
            turn = 1;
            lin_vel = 0.4;
            rot_vel = -0.1;
            while (turn)
                orien_curr = 180/pi*curr_pose(3,1);
                delta_phi = abs(orien_curr - orien_orig);
                if (delta_phi < 0.5)
                    turn = 0;
                else
                    turn = 1;
                end
            end
        else                 %current orientation < orientation wanted
            turn = 1;
            lin_vel = 0.4;
            rot_vel = 0.1;
            while (turn)
                orien_curr = 180/pi*curr_pose(3,1);
                delta_phi = abs(orien_curr - orien_orig);
                if (delta_phi < 0.5)
                    turn = 0;
                else
                    turn = 1;
                end
            end
        end
        lin_vel = 0.4;
        rot_vel = 0.0;
    end
end
lin_vel = 0.0;
rot_vel = 0.0;
delta_phi = 0;
turn_corner = 1;
while (turn_corner)
    orien_curr = 180/pi*curr_pose(3,1);
    delta_phi = abs(orien_orig - orien_curr)
    if ((delta_phi > 80) && (delta_phi < 100))
        if ((delta_phi > 85) && (delta_phi < 90))
            if ((delta_phi > 89.0) && (delta_phi < 89.90))
                turn_corner = 0;
                delta_phi = 0;
            else
                if (delta_phi < 89.95) %current orientation < orientation wanted
                    lin_vel = 0.0;
                    rot_vel = 0.05;
                else %current orientation > orientation wanted
                    lin_vel = 0.0;
                    rot_vel = -0.05;
                end
                turn_corner = 1;
            end
        else
            if (delta_phi < 90) %current orientation < orientation wanted
```

```
                        lin_vel = 0.0;
                        rot_vel = 0.1;
                else %current orientation > orientation wanted
                        lin_vel = 0.0;
                        rot_vel = -0.1;
                end
                turn_corner = 1;
            end
        else
            if (delta_phi < 100) %current orientation < orientation wanted
                    lin_vel = 0.0;
                    rot_vel = 1.0;
            else %current orientation > orientation wanted
                    lin_vel = 0.0;
                    rot_vel = -1.0;
            end
            turn_corner = 1;
        end
    end

    %toc
end
```

Results of tests:





3 Color Detection and Tracking

The goal is to understand color-based tracking and controlling the robot to move according to visual input.

3.1 Color Detection

The code HSV_color_detection.m will create a few colored balls in the simulation environment and will display the images that the robot's camera is recording in this environment. The goal is to write code that will detect the green ball in the images using thresholding in the HSV color space, and make the robot turn in place so that the green ball appears in the middle of the image.

One way to achieve this is by using the following steps for each image:

• use thresholds on the Hue of each pixel, to create a black-and-white (also called binary) image, where all pixels deemed to belong to the green ball are set to one, and all other pixels to zero.

• use the function regionprops to find the centroid of the image region that is identified as belonging to the green ball.

• define the rotational velocity of the robot using a "P-controller" whose input is the difference between the centroid column and the middle column of the image.

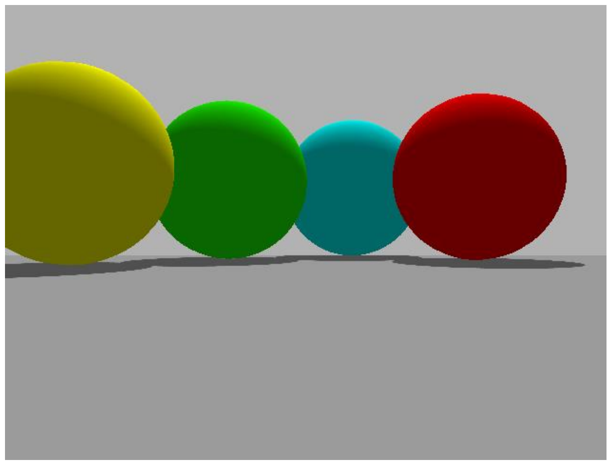
HSV_color_detection.m

```
% create a few colored balls in the environment
ball = ExampleHelperGazeboModel('Ball')
spherelink = addLink(ball,'sphere',1,'color',[0.1 1 0 1])
spawnModel(gazebo,ball,[6.5,1,1]);
ball2 = ExampleHelperGazeboModel('Ball')
spherelink = addLink(ball2,'sphere',1,'color',[1 1 0 1])
spawnModel(gazebo,ball2,[5.5,2,1]);
ball3 = ExampleHelperGazeboModel('Ball')
spherelink = addLink(ball3,'sphere',1,'color',[0 1 1 1])
spawnModel(gazebo,ball3,[7.5,-1,1]);
ball4 = ExampleHelperGazeboModel('Ball')
spherelink = addLink(ball4,'sphere',1,'color',[1 0 0 1])
spawnModel(gazebo,ball4,[6.5,-2,1]);
```

The goal is to detect the green ball in the image using thresholding in the HSV color space, and make the robot rotate so the green ball appears in the center of the image.

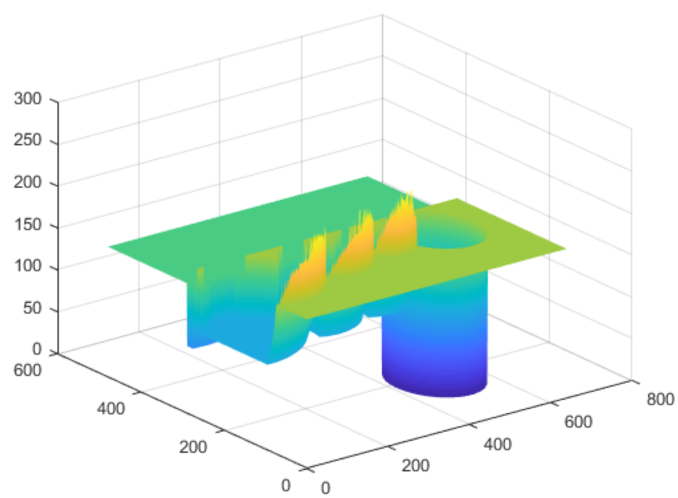
I used the following procedure in my code:

1. Take RGB image of what is in front of the bot.



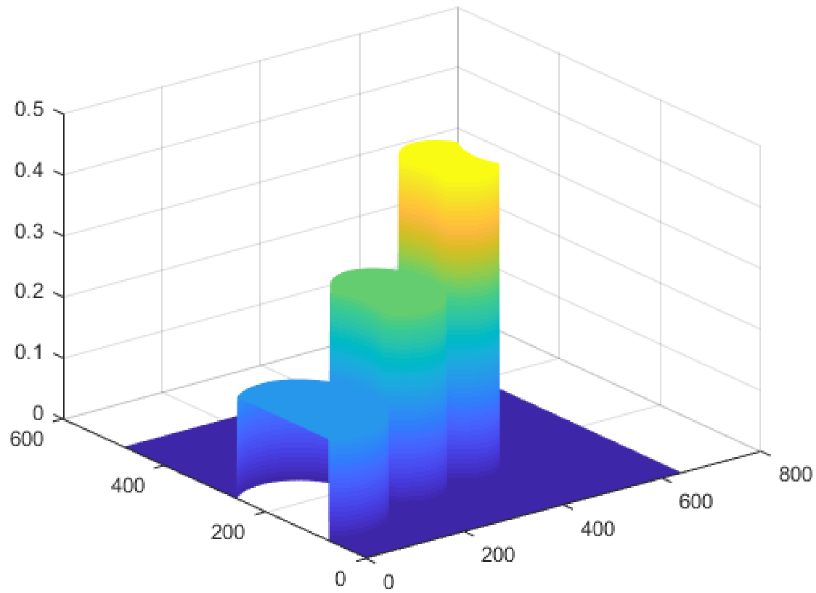
Original Image from bot.

2. Find the RGB space of this image.



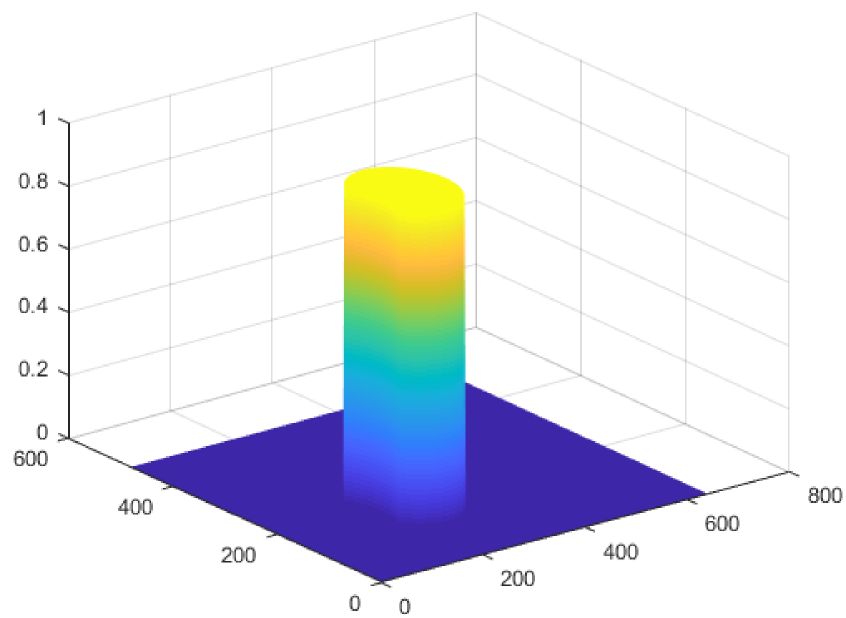
RGB space

3. Take the green part of the RGB space and convert it to HSV color space.



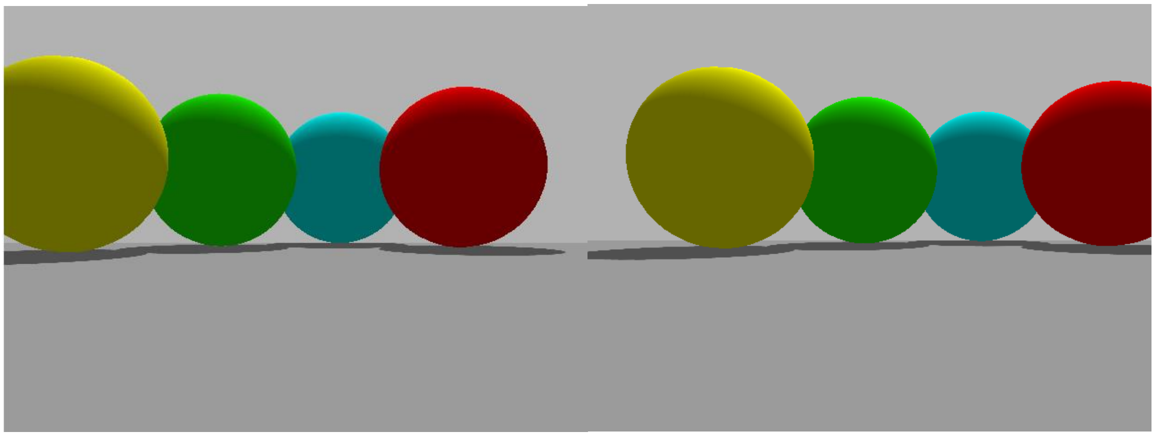
Green color HSV space, the center column is the green region we want to detect, from ~0.3 to ~0.4 in hue value.

4. Using thresholding of hue greater than 0.3 and lower than 0.4, find the parts in the image that are the green ball. Set the parts that are the green ball to 1 and all other parts to 0.

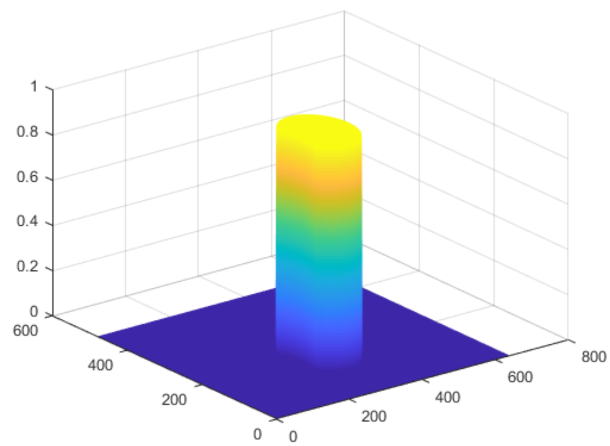


The regionprops result after using thresholding, the column represents the green ball in the image.

5. Start loop.
    a. If green ball is in the image.
        i. Find the center of the green ball using regionprops function.
        ii. Test to see if center of camera coincides with center of green ball.
            1. If it coincides, don't rotate bot.
            2. If it doesn't coincide, rotate bot towards the center of the green ball using the rotational proportional controller.
    b. If no green ball is found in the frame of the camera, rotate the bot until the green ball is in the image.
6. Repeat steps 1 to 5.



Original Image on left and Centered Image on the right.



Regionprogs result of centered image.

Code:

```
clear all
close all
% initialize
rosshutdown % to 'close' any previous sessions
rosinit('192.168.111.129'); % initialize Matlab ROS node
tbot = turtlebot  % the data structure that allows access to the turtlebot and its
sensors
% these are the variables that are used to define the robot velocity
lin_vel = 0;  % meters per second
rot_vel = 0;  % rad/second
% create a Matlab "timer" for continuoulsy sending velocity commands to the robot.
% This will create a new thread that runs the command setVeloc-
ity(tbot,lin_vel,rot_vel) every 100msec
velocity_command_timer = timer('TimerFcn','setVelocity(tbot,lin_vel,rot_vel)','Peri-
od',0.1,'ExecutionMode','fixedSpacing');
% start the timer. This will keep running until we stop it.
start(velocity_command_timer)
% handle to the simulator
gazebo = ExampleHelperGazeboCommunicator();
% create a few colored balls in the environment
ball = ExampleHelperGazeboModel('Ball')
spherelink = addLink(ball,'sphere',1,'color',[0.1 1 0 1])
spawnModel(gazebo,ball,[6.5,1,1]);
ball2 = ExampleHelperGazeboModel('Ball')
spherelink = addLink(ball2,'sphere',1,'color',[1 1 0 1])
spawnModel(gazebo,ball2,[5.5,2,1]);
ball3 = ExampleHelperGazeboModel('Ball')
spherelink = addLink(ball3,'sphere',1,'color',[0 1 1 1])
spawnModel(gazebo,ball3,[7.5,-1,1]);
ball4 = ExampleHelperGazeboModel('Ball')
spherelink = addLink(ball4,'sphere',1,'color',[1 0 0 1])
spawnModel(gazebo,ball4,[6.5,-2,1]);
% pause so that everything has time to be initialized
pause(2);

rgbImg = getColorImage(tbot); %get image
figure(1); imshow(rgbImg);%show image
imwrite(rgbImg,['imagetest.bmp']); %write image to file
green_rgb_image = rgbImg(:,:,2); %get green part of rgb image
figure(2);surf(green_rgb_image);shading interp; %display green part
hsv_image = rgb2hsv(rgbImg); %conver rgb to hsv
hue = hsv_image(:,:,1); %get hue part of hsv
figure(3); surf(hue);shading interp; %display hue of hsv
% saturation = hsv_im(:,:,2); %get saturation part of hsv
% figure(4); surf(hue);shading interp; %display saturatin of hsv
% value = hsv_im(:,:,3); %get value part of hsv
% figure(5); surf(value);shading interp; %display value of hsv
green_part_hue = (hue > 0.3).*(hue < 0.4); %find green part of hue between selected
values
figure(6);surf(green_part_hue);shading interp; %display new hue image with just green
part
green_ball_center = regionprops(green_part_hue,'centroid') %find center of green part
x_image_midpoint = 320; %define default x mid point of image
Kp_Rotational = 0.001; % set kp rotational p controller
while 1
    tic
    rgbImg = getColorImage(tbot); %get image
    figure(1); imshow(rgbImg);%show image
    hsv_image = rgb2hsv(rgbImg); %convert RGB to HSV
    hue = hsv_image(:,:,1); % find the hue values of the image
```

```
    green_part_hue = (hue > 0.3).*(hue < 0.4); % if green color is between 0.3 to 0.4
    if (any(green_part_hue(:) > 0)) %green ball is in frame
%           figure(6);surf(green_part_hue);shading interp; %update green region graph
        green_ball_center = regionprops(green_part_hue,'centroid') % find the center
of the green cylinder
        x_new_center = green_ball_center.Centroid(1); %center of green region
        dist_x = x_new_center - x_image_midpoint; % find the horizontal distance be-
tween the green ball and the center of image columns
        absolute_dist = abs(dist_x) %absolute distance to center of green ball
        if (absolute_dist < 1)%it's not at the center, rotate the robot and to make it
at center
            rot_vel = 0.0; %if center of green ball found stop rotating
        else
            rot_vel = -Kp_Rotational*dist_x % %if not at center of green ball keep ro-
tating
        end
    else %otherwise if green ball not in frame
        rot_vel = 0.3; %find the green ball
    end
    pause(0.1);
    toc
end
```

3.2 Tracking

MATLAB script robot following.m. generates a second robot in the environment, and our goal is to have the Turtlebot track it as it moves in space. The code initializes the simulation environment, and uses the function find robot, to detect the pixels in an image that belong to the robot. This function is tailor-made for the specific characteristics of the simulation. It assumes that the color of the background is perfectly known, and any pixel that does not have that color is assumed to belong to the robot. Once these pixels are identified, the function returns the column coordinate of the robot's centroid, and the area of the image region that belongs to the robot.

following.m

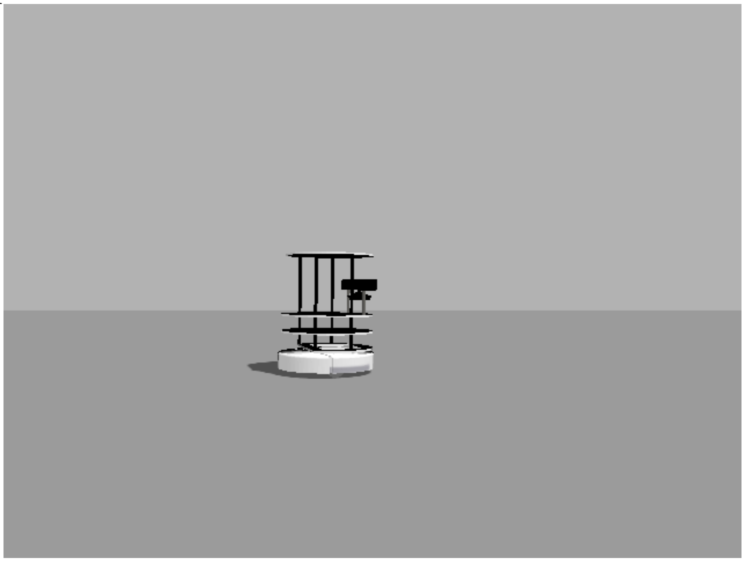
```
function [mean_col, area] = find_robot(rgbImg);

%% function to find the mean column and the width of the bounding box
% of the image area where the robot appears in the image
% this is a specialized function, will only work for tracking
% in the Empty Gazebo world simulator
% find non-background pixels
Im_top = rgbImg(1:265,:,1)~=178;
Im_bot = (rgbImg(266:480,:,1)~=155).*(rgbImg(266:480,:,1)~=80);
% form the image that contains the non-background pixels
Im_bw = [Im_top;Im_bot];
figure(20)
imshow(Im_bw)
% find the nonzero elements
[rows,cols] = find(Im_bw);
% the function returns empty variables if the robot is not found
if isempty(cols)
       mean_col = [];
       area = [];
else
       mean_col = mean(cols);
       area = length(cols);
end
```

We would like the robot to appear as close to the image center as possible, and have an area in the image as close as possible to 4000 pixels. These objectives can again be accomplished by using Proportional controllers for the rotational velocity.

The code should also be able to handle the case where the robot does not appear in the image One way to address this is to have the robot rotate in place, with a constant rotational velocity, until it re-detects the robot.

In this part, we have a second robot on the world that is moving around, I will refer to this as robot2. We want to detect the position of robot2 with the camera on robot1. We then center robot2 in the camera frame of robot1. After that we move robot1 within a selected distance of robot2, by using the area that the outline of robot 2 takes on the image of robot1.

1. First take a picture of what is in front of robot1.



RGB image from robot1.

2. Use the find_robot function to detect the pixels in the image that belong to robot2. From this function, we get the area to robot2 and center position of robot2 relative to the frame camera of robot1.

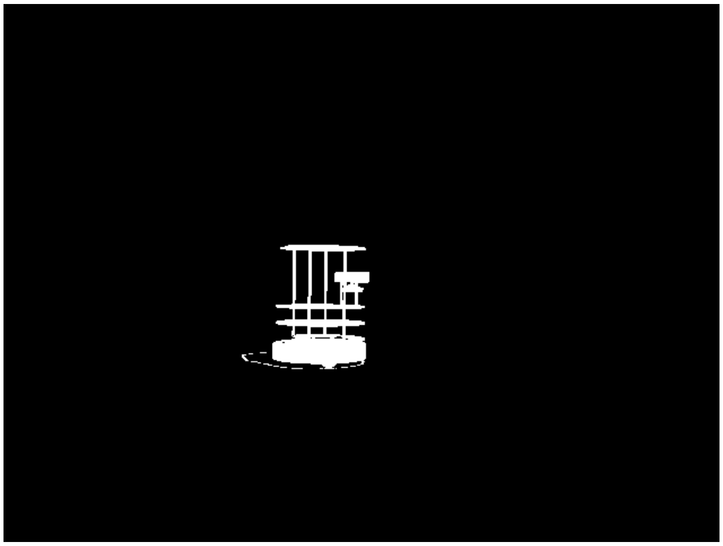
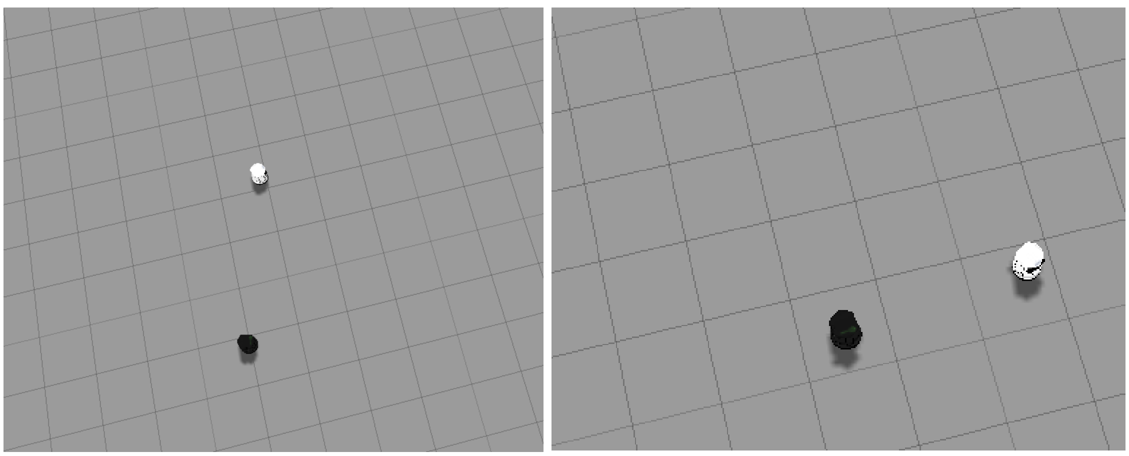


Image from find_robot function, that eliminates the background and gives just the outline of robot2 in the frame of the camera of robot1.

3. Start loop.
    a. See if robot2 is in the frame of the camera of robot1.
        i. If not, then rotate the robot1 until it finds robot2 in the frame of the camera.
        ii. If robot 2 is the frame of the camera of robot1.
            1. Calculate the middle point of robot2 with respect to the frame of the camera of robot1.
            2. Rotate robot1 with the rotational proportional controller until the center of robot2 coincides with the center of the camera of robot1.
            3. Calculate the distance from robot1 to robot2 by taking the area of outline of robot2 in the frame of the camera of robot1.
            4. If the distance is greater than the desired distance, move robot1 to robot2 until the area of robot2 satisfies the required value.



Robot1 in black following robot2.

Code:

```
clear all
close all

% initialize
rosshutdown % to 'close' any previous sessions
rosinit('192.168.111.130'); % initialize Matlab ROS node
tbot = turtlebot  % the data structure that allows access to the turtlebot and its
sensors

% these are the variables that are used to define the robot velocity
lin_vel = 0;  % meters per second
rot_vel = 0;  % rad/second

% create a Matlab "timer" for continuoulsy sending velocity commands to the robot.
% This will create a new thread that runs the command setVeloc-
ity(tbot,lin_vel,rot_vel) every 100msec
velocity_command_timer = timer('TimerFcn','setVelocity(tbot,lin_vel,rot_vel)','Peri-
od',0.1,'ExecutionMode','fixedSpacing');
% start the timer. This will keep running until we stop it.
start(velocity_command_timer)

% create a second robot in the Gazebo simulator
% handle to the simulator
gazebo = ExampleHelperGazeboCommunicator();
% the robot model
botmodel = ExampleHelperGazeboModel('turtlebot','gazeboDB')
% spawn the model in gazebo
bot = spawnModel(gazebo,botmodel,[3,0,0])
% set the initial orientation of the robot
setState(bot,'orientation',[0 0 pi/3])
% get the handle to the robot's joints (we'll need this to make the robot
% move)
[botlinks, botjoints] = getComponents(bot)

% pause so that everything has time to be initialized
pause(5)

% the number of images we have processed
image_count=0;

% create a Matlab timer for moving the second robot
robot_move_timer = timer('TimerFcn','move_robot(image_count,bot,botjoints)','Peri-
od',10,'ExecutionMode','fixedSpacing');
% start the timer. This will keep running until we stop it.
start(robot_move_timer)

x_image_midpoint = 320;
Kp_rotational = 0.002;
Kp_linear = 0.000001;
distance_to_robot = 4000;
while 1
        tic
        % get an image from the camera
        rgbImg = getColorImage(tbot); %get image and show image
        figure(10)
        imshow(rgbImg)

        % find the robot in the image. This function returns the column where
        % the middle of the robot will appear in the image, and the number of
        % pixels that belong to the robot in the image
```

```
        [mean_col, area] = find_robot(rgbImg);

    if(isempty(area)) %if it doesn't see robot in camera
        rot_vel = 0.4; %find robot
        lin_vel = 0.0;
    else %if robot found in camera
        x_dist = mean_col - x_image_midpoint;
        rot_vel = -Kp_rotational*x_dist;
        a_dist = area - distance_to_robot;
        lin_vel = -Kp_linear*a_dist;
    end

    pause(0.5);
       toc

       % the number of images we processed
       image_count=image_count+1;

end
```

4 Using the Physical Robot

Now we revisit the previous steps and make them work with the real robot.

4.1 Waypoint Navigation

Waypoint navigation will command the robot to move continuously on a square.

This procedure follows the code part1_waypoint_nav.m.

We tried different values of Kp and Dthresh and found the best values that worked for the real robot were Kp = 0.9 and Dthresh = 0.09.

These values gave the best performance, which can be seen in figure 4.1. The robot follows the path of the square relatively close.

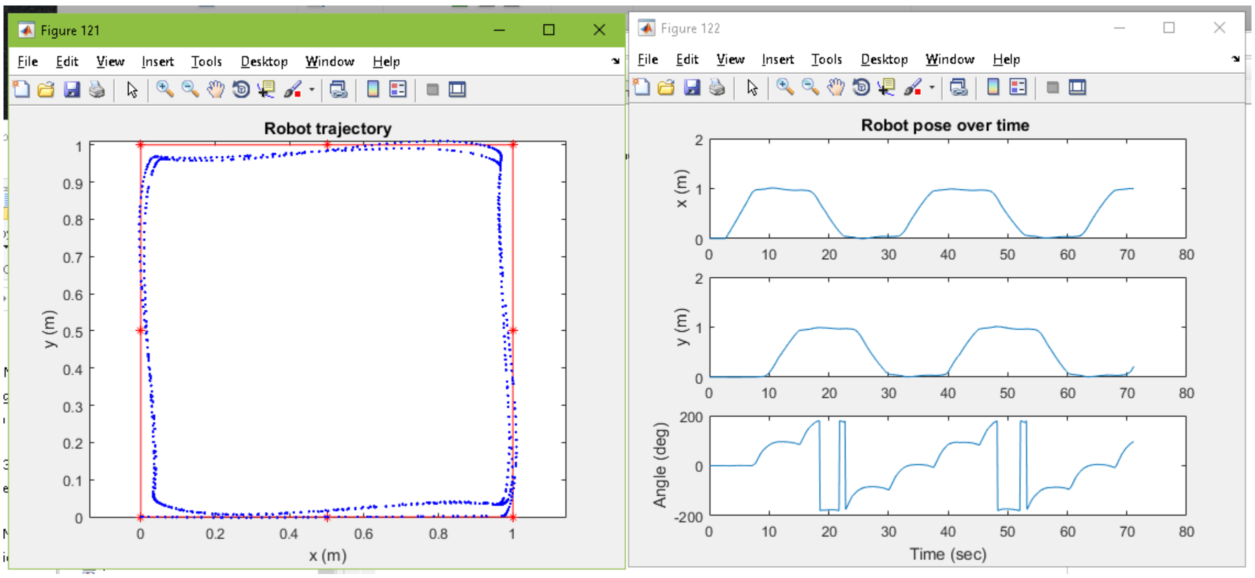


Figure 4.1 - Robot path with Kp = 0.9 and Dthresh = 0.09

We tried other values for the parameters and we got the following results.

First varied the Kp parameter. We tried Kp = 0.3 and kept Dthresh =0.09 as in figure 4.2. The robot can stay on the waypoint path, but when it makes a mistake it, the turning is not fast enough, and it has to turn back to the waypoint it missed.

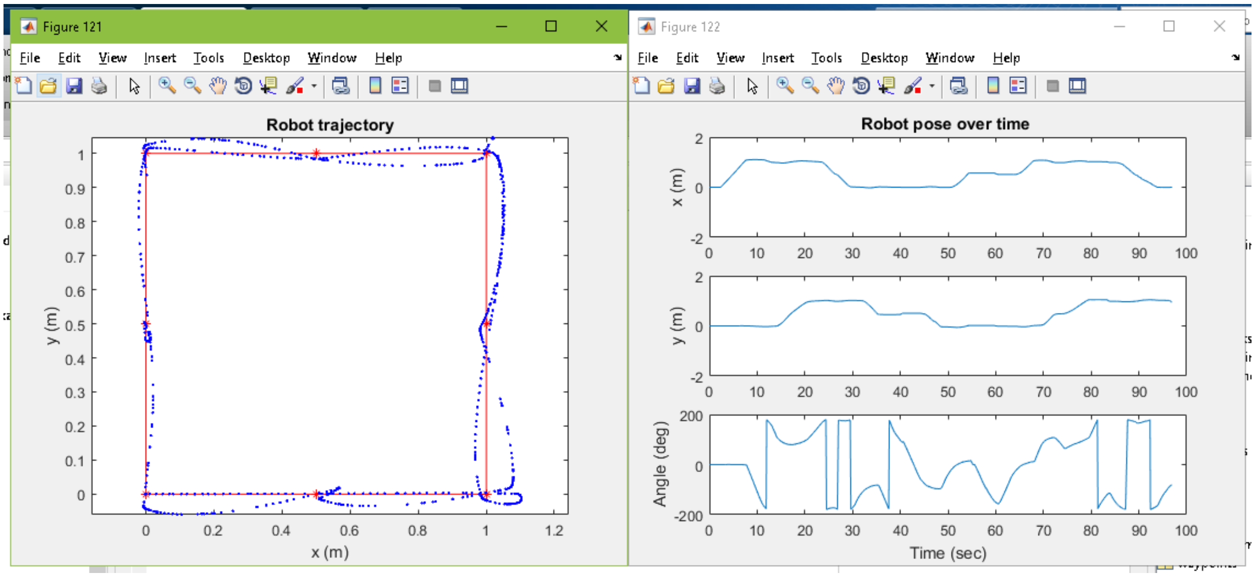


Figure 4.2 - Robot path with Kp = 0.4 and Dthresh = 0.09.

We tried Kp = 0.75 and kept Dthresh = 0.09 as in figure 4.3. This much better, but the robot doesn't quiet reach the corner waypoints.

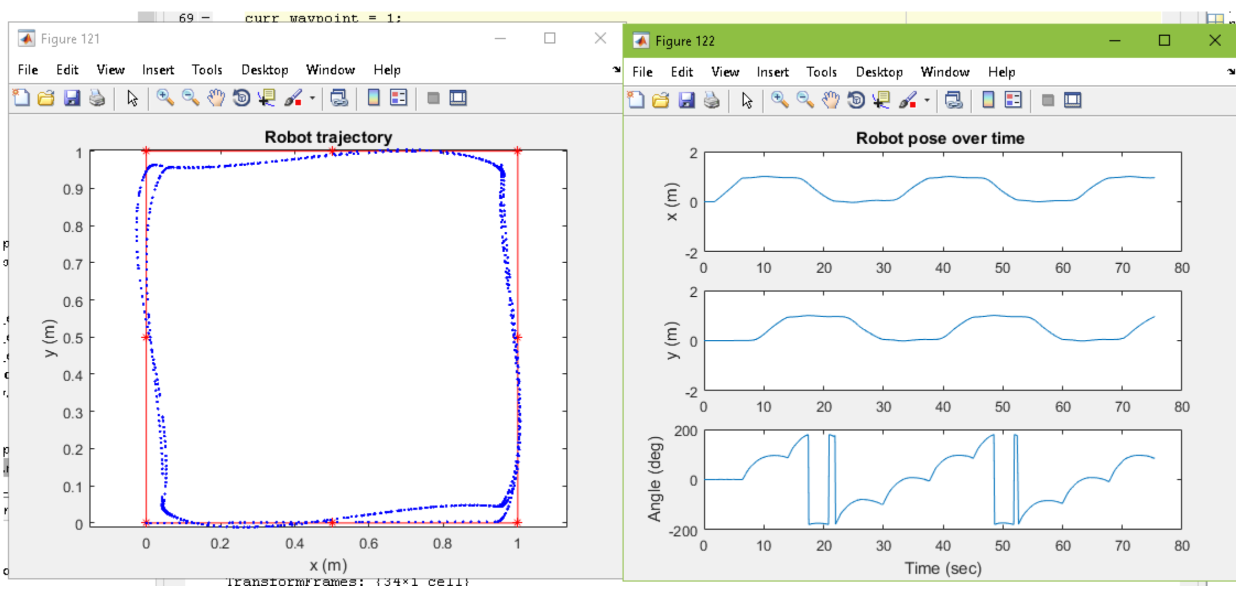


Figure 4.3 - Robot path with Kp = 0.75 and Dthresh = 0.09.

Next, we varied the Dthresh parameter. We tried Dthresh = 0.02 and kept Kp = 0.9 as in figure 4.4. Since Dthresh is so low, the robot misses the waypoints and rotates to come back for a closer pass. This wastes time as the robot has to make several turns some times to get close enough to a waypoint.

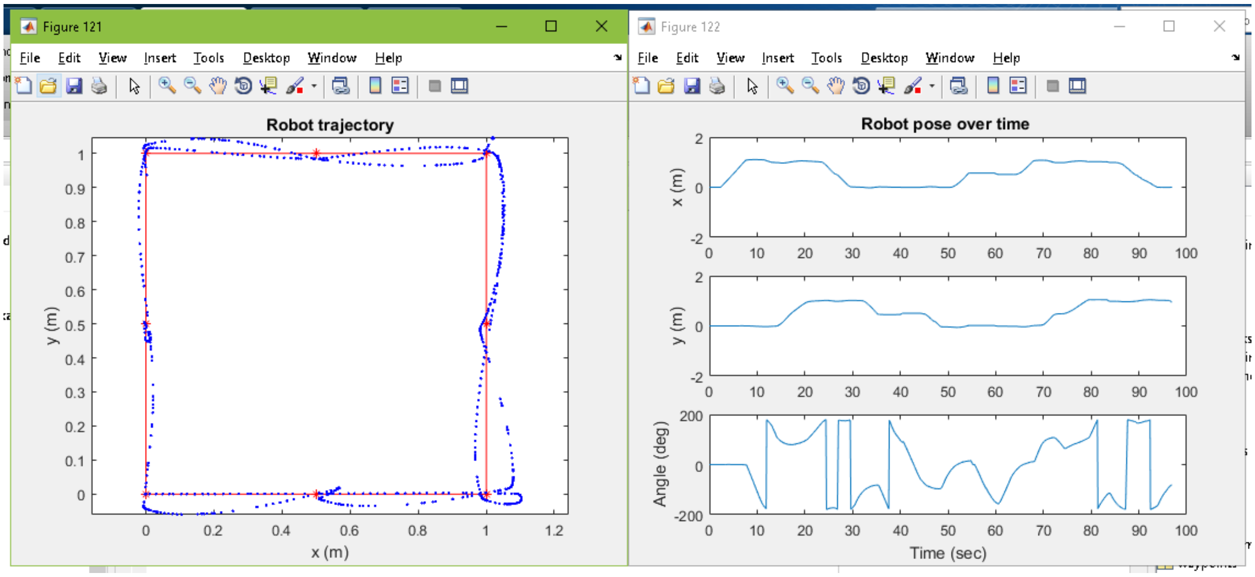


Figure 4.4 - Robot path with Kp = 0.9 and Dthresh = 0.02.

We tried Dthresh = 0.3 and kept Kp = 0.9 as in figure 4.5. With these parameters the robot is making more of a circle than a square, as the margins are too big and the robot doesn't have to reach the corner waypoints.

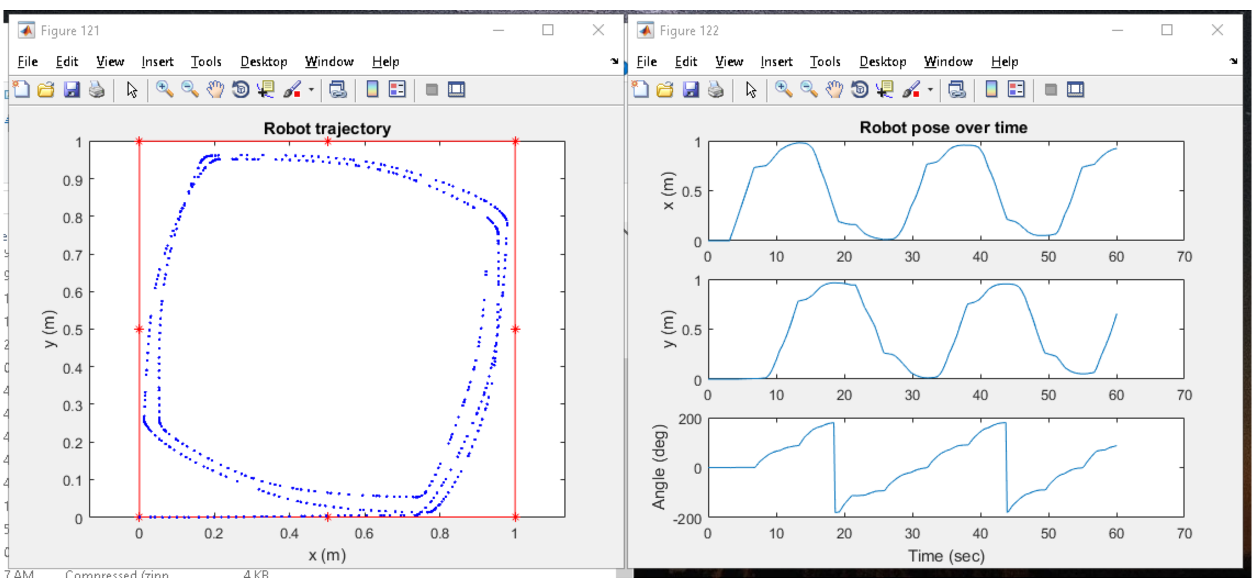


Figure 4.5 - Robot path with Kp = 0.9 and Dthresh = 0.3.

After trying out all the different parameters, and comparing the MATLAB plots to the actual trajectory. We determined the robot doesn't really follow the same actual path as in the MATLAB plot. This is maybe due to the slippery floor and the wheels not having perfect traction. Sometimes the distance covered is not the same, and the actual path covered is less than the MATLAB plot. The video shows the actual path taken by the robot with the best parameters we found.

Video: https://youtu.be/N_UOAWCw3ew

Code:

```
clear all
close all
% initialize
rosshutdown % to 'close' any previous sessions
rosinit('192.168.0.110'); % initialize Matlab ROS node
tbot = turtlebot  % the data structure that allows access to the turtlebot and its
sensors
% this command sets the current robot pose estimate to zero (it does not
% move the robot). This is equivalent to setting the ``world coordinate
% frame'' to coincide with the current robot frame.
resetOdometry(tbot)
% the variable robot_poses stores the history of robot pose received from odometry. It
% is declared as global for computational efficiency
global robot_poses
% we initialize robot_poses as an empty matrix with 10000 columns.
% This can store up to 1000 seconds of odometry, received at 10Hz.
robot_poses = zeros(4,10000);
% create a Matlab "timer" for receiving the odometry. This will effectively create a
new thread
% that executes 'curr_pose = get_pose_from_tbot_odometry(tbot);' every 100 msec.
% this will return the current robot pose in the variable curr_pose, and
% will also store the current pose, together with its timestamp, in robot_poses
odometry_timer = timer('TimerFcn','curr_pose = get_pose_from_tbot_odome-
try(tbot);','Period',0.1,'ExecutionMode','fixedRate');
% start the timer. This will keep running until we stop it.
start(odometry_timer)
% these are the variables that are used to define the robot velocity
lin_vel = 0;  % meters per second
rot_vel = 0;  % rad/second
% create a Matlab "timer" for continuoulsy sending velocity commands to the robot.
% This will create a new thread that runs the command setVeloc-
ity(tbot,lin_vel,rot_vel) every 50msec
velocity_command_timer = timer('TimerFcn','setVelocity(tbot,lin_vel,rot_vel)','Peri-
od',0.05,'ExecutionMode','fixedSpacing');
% start the timer. This will keep running until we stop it.
start(velocity_command_timer)
%% The waypoints for moving on a square:
% we make the robot move on a square
% the size of the square
square_size = 1;
% generate the waypoints
dp = .5; % spacing between waypoints
pp = [0:0.5:square_size];
waypoints = [pp;zeros(size(pp))];
waypoints = [waypoints [square_size*ones(size(pp));pp]];
waypoints = [waypoints [pp(end:-1:1);square_size*ones(size(pp))]];
waypoints = [waypoints [zeros(size(pp));pp(end:-1:1)]];
% create a Matlab timer for plotting the trajectory
plotting_timer = timer('TimerFcn','plot_trajectory(robot_poses, waypoints)','Peri-
od',5,'ExecutionMode','fixedSpacing');
% start the timer. This will keep running until we stop it.
start(plotting_timer)
% the variable "waypoints" is a 2xN matrix, which contains the x-y
% coordinates of the N waypoints that the robot needs to go to.
% these coordinates are in the "world" coordinate frame.
% the number of waypoints
N_waypoints = size(waypoints,2);
% the index to the current goal waypoint
curr_waypoint = 2;
% distance threshold to decide when we've reached a waypoint
```

```
d_thresh = 0.09;
% controller gain
k_p = .9;
while 1

        % keep doing this while the distance to the current goal waypoint is
        % larger than d_thresh
        while norm(curr_pose(1:2)-waypoints(:,curr_waypoint))>d_thresh

                % find waypoint's position in the local frame:
                % this is the rotation matrix expressing the robot's orientation in
                % the world coordinate frame.
                R = [cos(curr_pose(3)) -sin(curr_pose(3)) 0 ;
                      sin(curr_pose(3)) cos(curr_pose(3))  0
                         0                       0             1];

            R_p  = R'*([waypoints(:,curr_waypoint)-curr_pose(1:2) ; 0]);

                % find the angle theta
                theta = atan2(R_p(2),R_p(1));

                % define the velocities
                % make sure the robot does not turn too fast... we don't want the
                % netbook to fly off
                rot_vel = sign(theta)* min(abs(k_p* theta), 1);
                % the linear velocity is limited to 0.25 m/sec. When the rotational
                % velocity is large (i.e., theta is large), we reduce the linear
                % velocity, to make sure there is enough time to turn towards the next
                % waypoint.
                lin_vel = min(0.25, max(0,0.25-0.5*abs(rot_vel)));
                pause(0.05)

        end

        % increase the waypoint index
        curr_waypoint = curr_waypoint+1;
        if curr_waypoint == N_waypoints+1;
                curr_waypoint=1;
        end

end
```

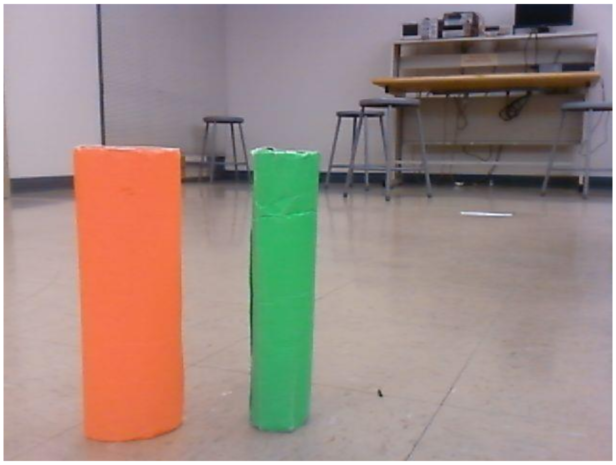
4.2 Color Based Tracking

This procedure follows the code (part2_camera_detection_v2.m).

For this part we get the robot camera to detect the green cylinder in the frame of the camera and rotate the robot so that the center of the camera coincides with the center of the green cylinder.

Our code follows the following procedure:

1. Take RGB image of what is in front of the bot.



Original Image from bot.

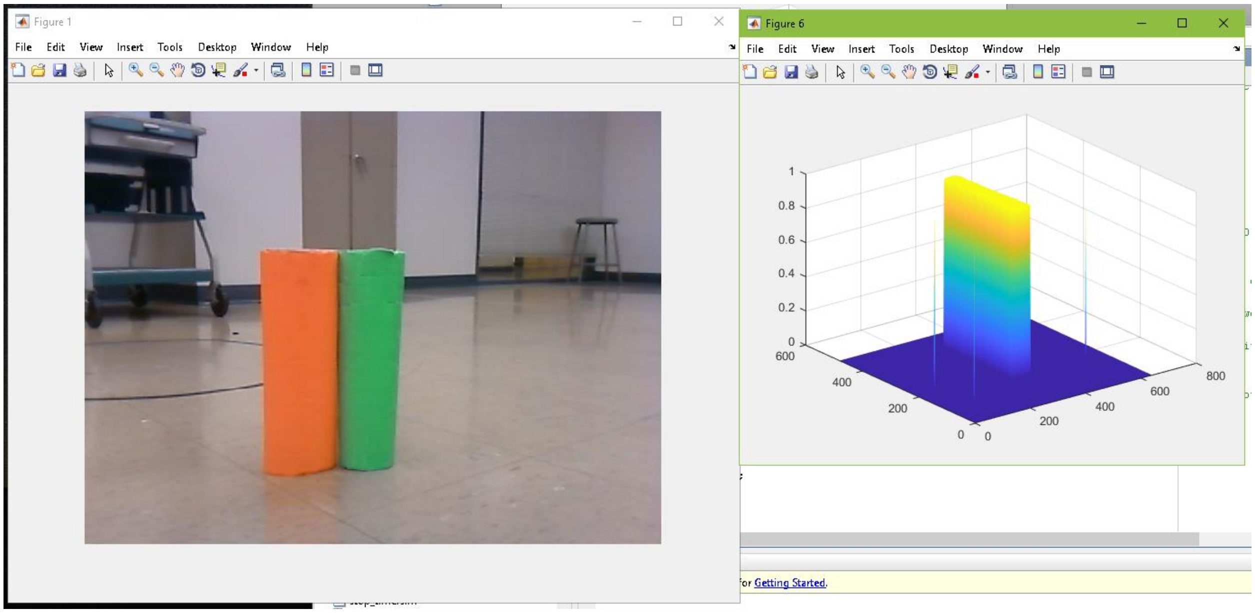
2. Find the RGB space of this image.
3. Take the green part of the RGB space and convert it to HSV color space.
    a. We chose hue values from 0.38 to 0.41 to represent the green cylinder.

4. Using thresholding of hue greater than 0.38 and lower than 0.41, we find the parts in the image that are the green cylinder. Set the parts that are the green cylinder to 1 and all other parts to 0.

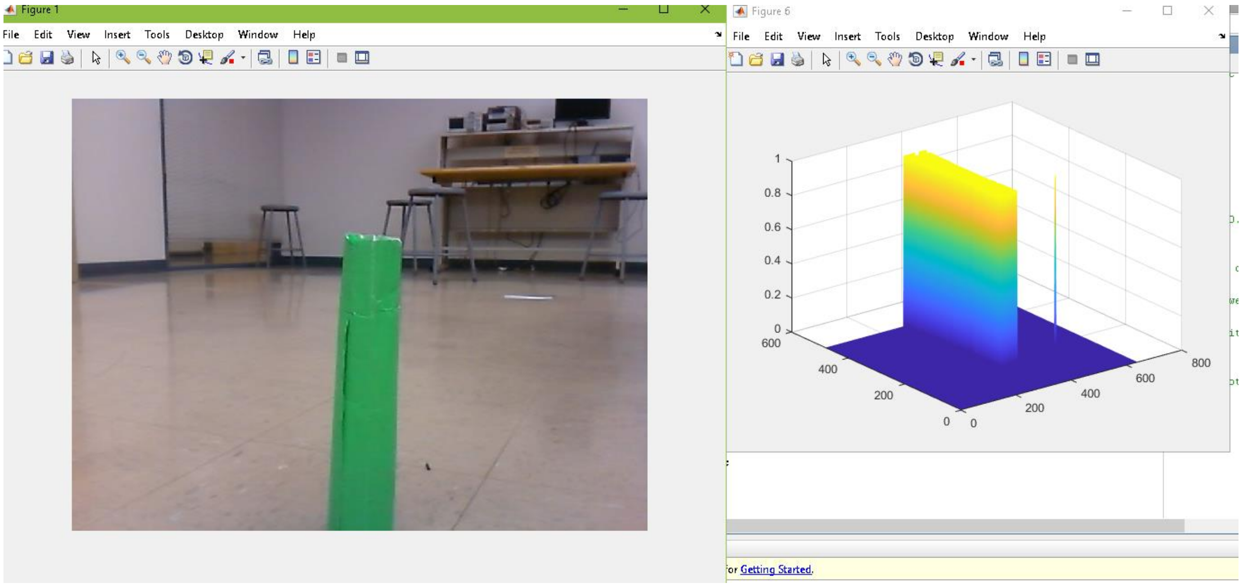
The thresholding result, the white parts represent a 1, meaning the green parts in the image, and 0 the non-green parts in the image according to our thresholding numbers.

We tried setting the cylinder at different distances from the camera and different orientations. Then tested if the camera could center on the green cylinder. With the parameters for hue we set above, the robot was successful in finding the green cylinder and centering the camera on it at all distances below.

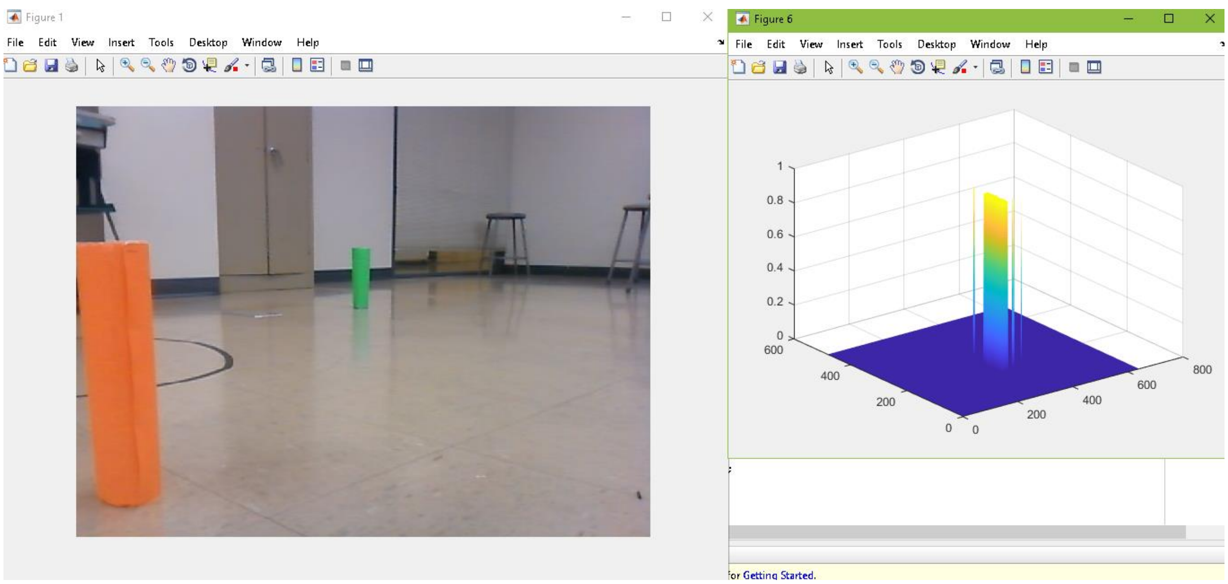
The green cylinder at close range, with thresholding graph of the green part at the right.



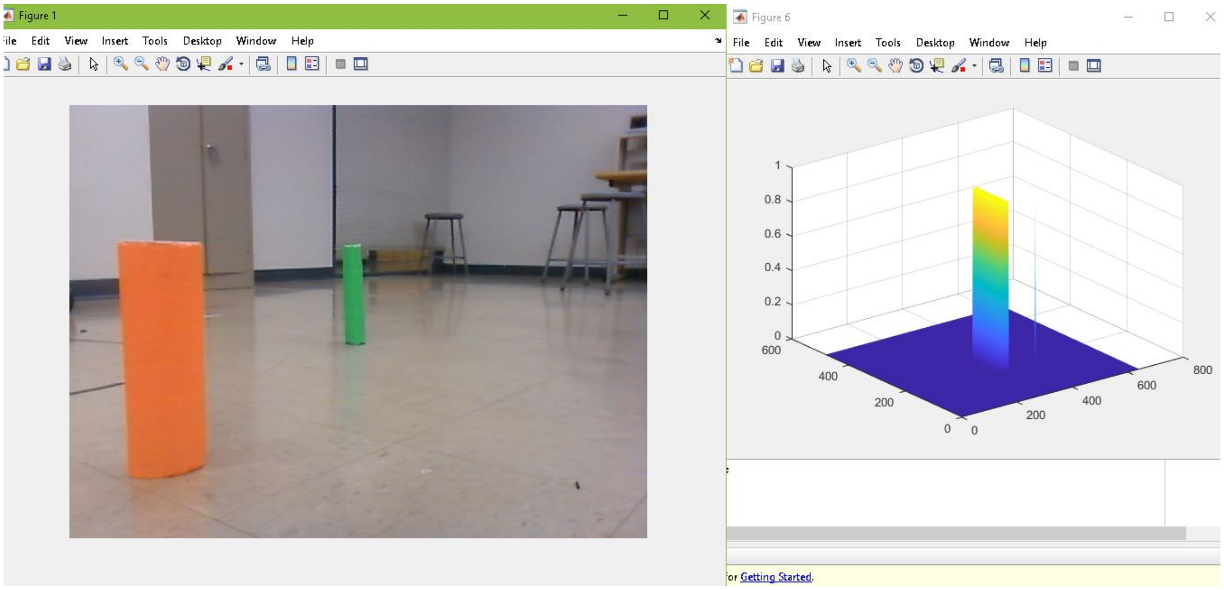
The green cylinder at close range but sideways, with thresholding graph of the green part at the right.



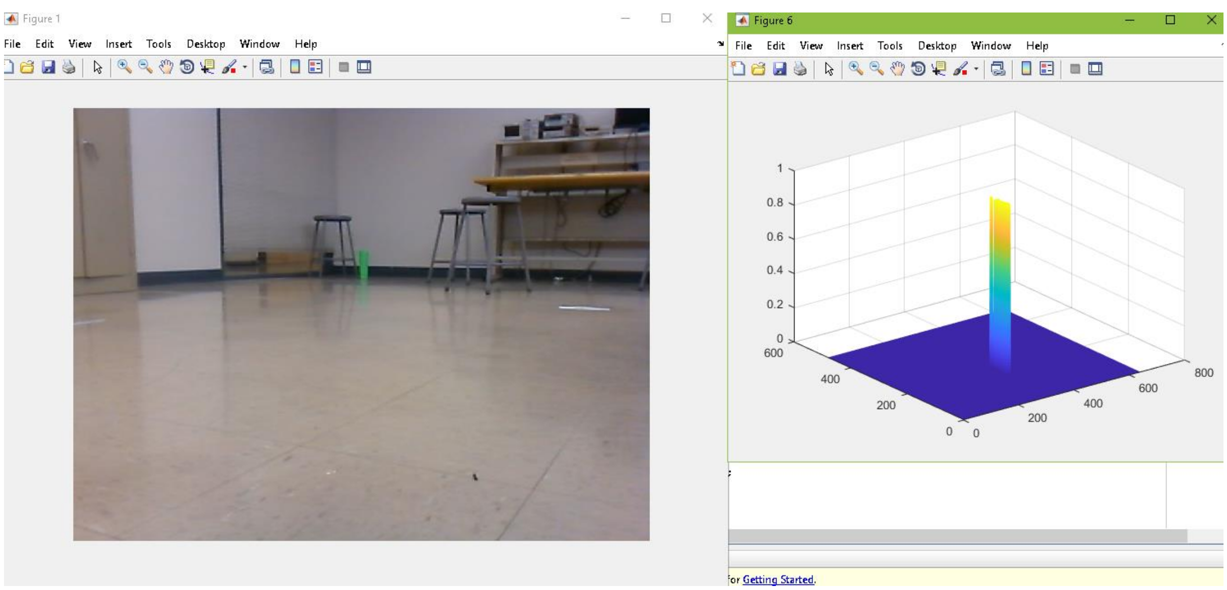
The green cylinder at medium range, with thresholding graph of the green part at the right.



The green cylinder at medium range but sideways, with thresholding graph of the green part at the right.



The green cylinder at far range, with thresholding graph of the green part at the right.



Part 2

This procedure follows the code (part2_camera_detection_v2_orange.m)

We tried to detect the orange cylinder just using the Hue values, but it is not possible because the Hue values for the orange part are too low as shown in figure 4.6. We added thresholding with the Saturation and Value parameters. This way we were successful in detecting the orange cylinder.

We used thresholding of Saturation greater than 0.65 and less than 0.71, Value greater than 0.9 and less than 0.95. The thresholding result is shown in figure 4.6 top right graph. These are the distinctive values that represent the orange cylinder as shown in figure 4.6.

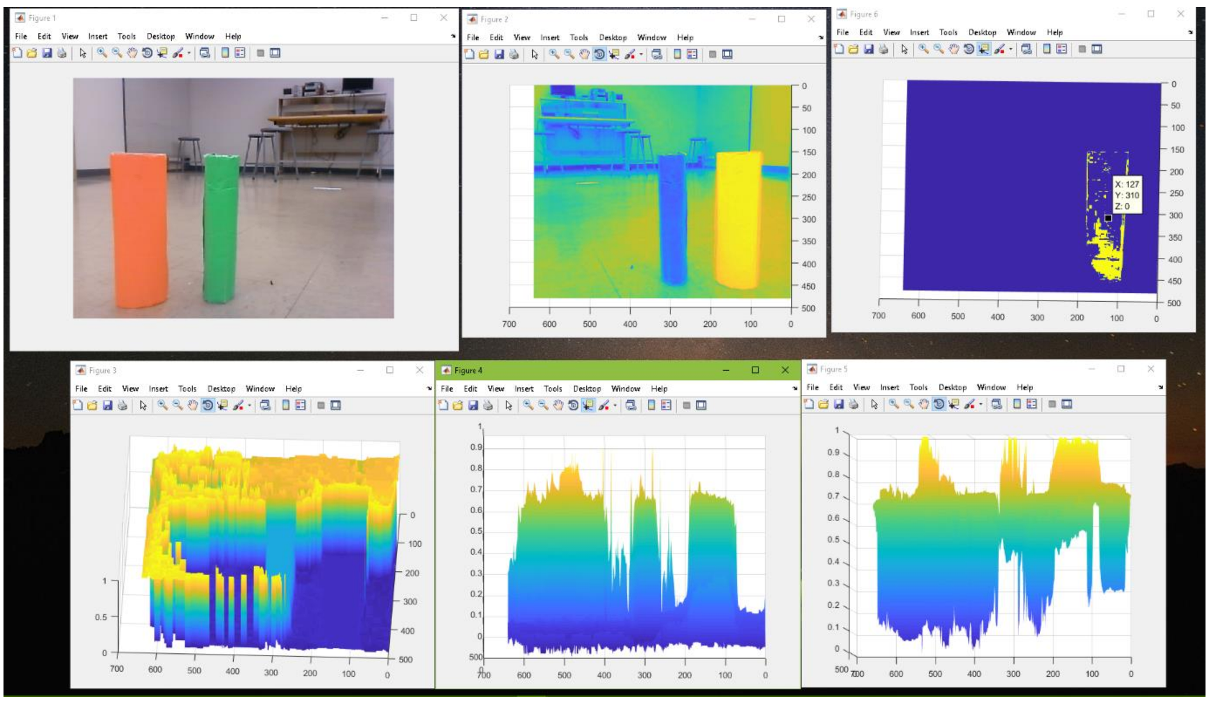


Figure 4.6 – (Top Left) Image from bot, (Top Middle) Red Values part of RGB image, (Top Right) Thresholding Result, (Bottom Left) Hue values, (Bottom Middle) Saturation Values, (Bottom Right) Value values.

After adding the additional thresholding for the orange cylinder, the camera was able to detect and follow the orange cylinder effectively, it did increase performance. For the green cylinder performance remained the same.

Code:

```
clear all
close all

% initialize
rosshutdown % to 'close' any previous sessions
rosinit('192.168.0.101'); % initialize Matlab ROS node
tbot = turtlebot  % the data structure that allows access to the turtlebot and its
sensors
tbot.ColorImage.TopicName = '/usb_cam/image_raw/compressed'; % set netbook webcam as
the camera for tbot

% these are the variables that are used to define the robot velocity
lin_vel = 0;  % meters per second
rot_vel = 0;  % rad/second

% create a Matlab "timer" for continuoulsy sending velocity commands to the robot.
% This will create a new thread that runs the command setVeloc-
ity(tbot,lin_vel,rot_vel) every 50msec
velocity_command_timer = timer('TimerFcn','setVelocity(tbot,lin_vel,rot_vel)','Peri-
od',0.05,'ExecutionMode','fixedSpacing');
% start the timer. This will keep running until we stop it.
start(velocity_command_timer)

% the following loop reads and stores 20 images from the camera
% for i = 1:20
%
%      rgbImg = getColorImage(tbot);
%      imwrite(rgbImg,['image',num2str(i),'.bmp']);
%
% end
%
% % the following loop reads the stored images, and displays them
% for i = 1:20
%
%      rgbImg = imread(['image',num2str(i),'.bmp']);
%      figure(10)
%      subplot(4,5,i)
%      imshow(rgbImg)
%
% end

% pause so that everything has time to be initialized
pause(2);

rgbImg = getColorImage(tbot); %get image
figure(1); imshow(rgbImg);%show image
imwrite(rgbImg,['imagetest.bmp']); %write image to file
green_rgb_image = rgbImg(:,:,2); %get green part of rgb image
figure(2);surf(green_rgb_image);shading interp; %display green part
hsv_image = rgb2hsv(rgbImg); %conver rgb to hsv
hue = hsv_image(:,:,1); %get hue part of hsv
figure(3); surf(hue);shading interp; %display hue of hsv
% saturation = hsv_im(:,:,2); %get saturation part of hsv
% figure(4); surf(hue);shading interp; %display saturatin of hsv
% value = hsv_im(:,:,3); %get value part of hsv
% figure(5); surf(value);shading interp; %display value of hsv
green_part_hue = (hue > 0.29).*(hue < 0.32); %find green part of hue between selected
values
figure(6);surf(green_part_hue);shading interp; %display new hue image with just green
part
```

```
green_ball_center = regionprops(green_part_hue,'centroid') %find center of green part
x_image_midpoint = 320; %define default x mid point of image
Kp_Rotational = 0.002; % set kp rotational p controller
while 1
    tic
    rgbImg = getColorImage(tbot); %get image
    figure(1); imshow(rgbImg);%show image
    hsv_image = rgb2hsv(rgbImg); %convert RGB to HSV
    hue = hsv_image(:,:,1); % find the hue values of the image
    green_part_hue = (hue > 0.29).*(hue < 0.32); % if green color is between 0.3 to
0.4
    if (any(green_part_hue(:) > 0)) %green ball is in frame
        figure(6);surf(green_part_hue);shading interp; %update green region graph
        green_ball_center = regionprops(green_part_hue,'centroid') % find the center
of the green cylinder
        x_new_center = green_ball_center.Centroid(1); %center of green region
        dist_x = x_new_center - x_image_midpoint; % find the horizontal distance be-
tween the green ball and the center of image columns
        absolute_dist = abs(dist_x) %absolute distance to center of green ball
        if (absolute_dist < 1)%it's not at the center, rotate the robot and to make it
at center
            rot_vel = 0.0; %if center of green ball found stop rotating
        else
            rot_vel = -Kp_Rotational*dist_x % %if not at center of green ball keep ro-
tating
        end
    else %otherwise if green ball not in frame
        rot_vel = 0.3; %find the green ball
    end
    pause(0.1);
    toc
end
```

Part 3

To make the robot follows the green cylinder as it moved around, or look for the green cylinder in its surrounding, we made the following loop in our code (part2_camera_detection_v2.m and part2_camera_detection_v2_orange.m)

1. Start loop.
    a. If green/orange cylinder is in the image.
        i. Find the center of the green/orange cylinder using thresholding and regionprops function.
        ii. Test to see if center of camera coincides with center of green ball.
            1. If it coincides, don’t rotate bot.
            2. If it doesn’t coincide, rotate bot towards the center of the green ball using the rotational proportional controller.
    b. If no green/orange cylinder is found in the frame of the camera, rotate the bot until the green/orange cylinder is in the image.
2. End Loop

With this code we were successful in tracking the green or orange cylinder around with the camera in the bot. The following videos demonstrates the tracking of the cylinders.

Video for Orange Cylinder: https://youtu.be/rhyof6Sk5WI

Video for Green Cylinder: https://youtu.be/Q2owsbqLHnM

Code:

```
clear all
close all

% initialize
rosshutdown % to 'close' any previous sessions
rosinit('192.168.1.80'); % initialize Matlab ROS node
tbot = turtlebot  % the data structure that allows access to the turtlebot and its
sensors
tbot.ColorImage.TopicName = '/usb_cam/image_raw/compressed'; % set netbook webcam as
the camera for tbot

% these are the variables that are used to define the robot velocity
lin_vel = 0;  % meters per second
rot_vel = 0;  % rad/second

% create a Matlab "timer" for continuoulsy sending velocity commands to the robot.
% This will create a new thread that runs the command setVeloc-
ity(tbot,lin_vel,rot_vel) every 50msec
velocity_command_timer = timer('TimerFcn','setVelocity(tbot,lin_vel,rot_vel)','Peri-
od',0.05,'ExecutionMode','fixedSpacing');
% start the timer. This will keep running until we stop it.
start(velocity_command_timer)

% the following loop reads and stores 20 images from the camera
% for i = 1:20
%
%       rgbImg = getColorImage(tbot);
%       imwrite(rgbImg,['image',num2str(i),'.bmp']);
%
% end
%
% % the following loop reads the stored images, and displays them
% for i = 1:20
%
%       rgbImg = imread(['image',num2str(i),'.bmp']);
%       figure(10)
%       subplot(4,5,i)
%       imshow(rgbImg)
%
% end

% pause so that everything has time to be initialized
pause(2);

rgbImg = getColorImage(tbot); %get image
figure(1); imshow(rgbImg);%show image
imwrite(rgbImg,['imagetestred.bmp']); %write image to file
red_rgb_image = rgbImg(:,:,1); %get green part of rgb image
figure(2);surf(red_rgb_image);shading interp; %display green part
hsv_image = rgb2hsv(rgbImg); %conver rgb to hsv
hue = hsv_image(:,:,1); %get hue part of hsv
figure(3); surf(hue);shading interp; %display hue of hsv
saturation = hsv_image(:,:,2); %get saturation part of hsv
figure(4); surf(saturation);shading interp; %display saturatin of hsv
value = hsv_image(:,:,3); %get value part of hsv
figure(5); surf(value);shading interp; %display value of hsv
red_part = ((saturation > 0.65).*(saturation < 0.71)).*((value > 0.9).*(value <
0.95)); %find green part of hue between selected values
figure(6);surf(red_part);shading interp; %display new hue image with just green part
red_ball_center = regionprops(red_part,'centroid') %find center of green part
```

```
x_image_midpoint = 320; %define default x mid point of image
Kp_Rotational = 0.002; % set kp rotational p controller
while 1
    tic
    rgbImg = getColorImage(tbot); %get image
    figure(1); imshow(rgbImg);%show image
    hsv_image = rgb2hsv(rgbImg); %convert RGB to HSV
    saturation = hsv_image(:,:,2); %get saturation part of hsv
    value = hsv_image(:,:,3); %get value part of hsv
    hue = hsv_image(:,:,1); % find the hue values of the image
    red_part = ((saturation > 0.65).*(saturation < 0.71)).*((value > 0.9).*(value < 0.95)); % if green color is between 0.3 to 0.4
    if (any(red_part(:) > 0)) %green ball is in frame
        figure(6);surf(red_part);shading interp; %display new hue image with just green part
        red_ball_center = regionprops(red_part,'centroid') % find the center of the green cylinder
        x_new_center = red_ball_center.Centroid(1); %center of green region
        dist_x = x_new_center - x_image_midpoint; % find the horizontal distance between the green ball and the center of image columns
        absolute_dist = abs(dist_x) %absolute distance to center of green ball
        if (absolute_dist < 1)%it's not at the center, rotate the robot and to make it at center
            rot_vel = 0.0; %if center of green ball found stop rotating
        else
            rot_vel = -Kp_Rotational*dist_x % %if not at center of green ball keep rotating
        end
    else %otherwise if green ball not in frame
        rot_vel = 0.3; %find the green ball
    end
    pause(0.1);
    toc
end
```

5 Advanced Navigation

Now we will attempt to make the turtlebot dock at a pre-specified location. The bot can start from any position around the dock, and then the robot must navigate towards the dock and be on top of the position within an error of 2 cmm.

To solve the problem of docking the robot at a prespecified location, the following approach was taken. We decided to use only a single green cylinder to represent the dock. This green cylinder is suspended from a string at a height right above the robot, but still in the camera's view.

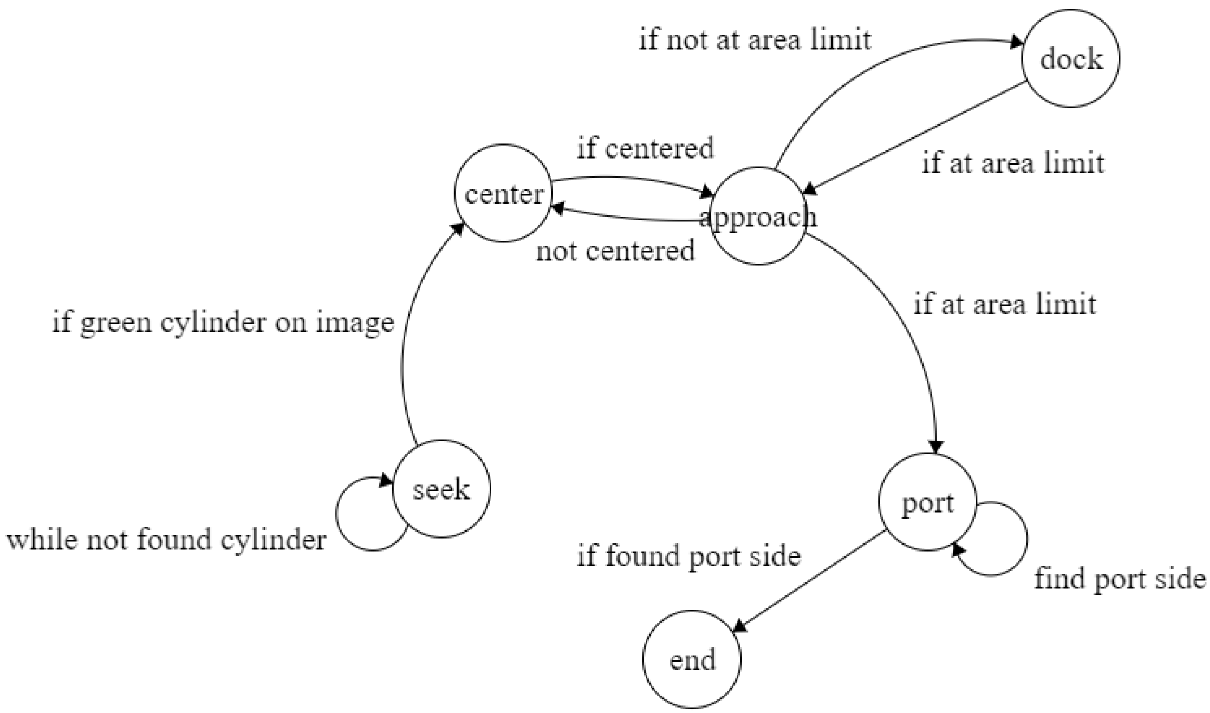


Figure 5.1 – The state machine for docking procedure.

The procedure for the program follows the state machine as shown in figure 5.1 and below.

1. First step is to define constants used in the program, this include; Kp_rotational, Kp_Linear, image midpoint, area max, area min, hue high base, hue low base.
2. Start the while 1 loop.
   A. Take a picture of what is in front of the robot, and analyze the following parameters as shown in figures 1-3.
      a) Convert RGB image to HSV.
      b) Extract the Hue from HSV. (Figure 5.2)
      c) Threshold the green parts of the image to detect the green cylinder. (Figure 5.3)

d) Eliminate green noise from the image using the im2bw function. (Figure 5.3)

B. Start the state machine with a switch statement. The state machine can have the following states.

a) Seek state

(1) The seek state looks to see if the green cylinder is in the camera of the robot.

(i) If the green cylinder is in the image go to **Center state**.

(ii) Otherwise, rotate robot until green cylinder is found.

b) Center state

(1) The center state will center the robot on the green cylinder if it is in the image.

(i) Use the regionprops function to detect the center of the cylinder, then use the Kp_rotational to center the camera on the green cylinder.

(a) If the camera is centered on the cylinder. Go to **Approach state**.

(b) Otherwise stay in **Center state**.

c) Approach state

(1) The approach state makes the robot move towards the green cylinder.

(i) If the green cylinder is not in the image, go to **Center state**.

(ii) If it is, then move towards the green cylinder using bot the Kp_rotational and Kp_linear parameters.

(a) If the area of the green cylinder is above the threshold area (robot is close to the docking station), then go to **Dock state**.

(b) Otherwise stay in **Approach state**.

d) Dock state

(1) The dock state makes sure the robot is centered on the dock position, and uses slower movements to achieve this.

(i) If robot position is within docking threshold, then go to **Port state**.

(ii) Otherwise stay in **Dock state**.

e) Port state

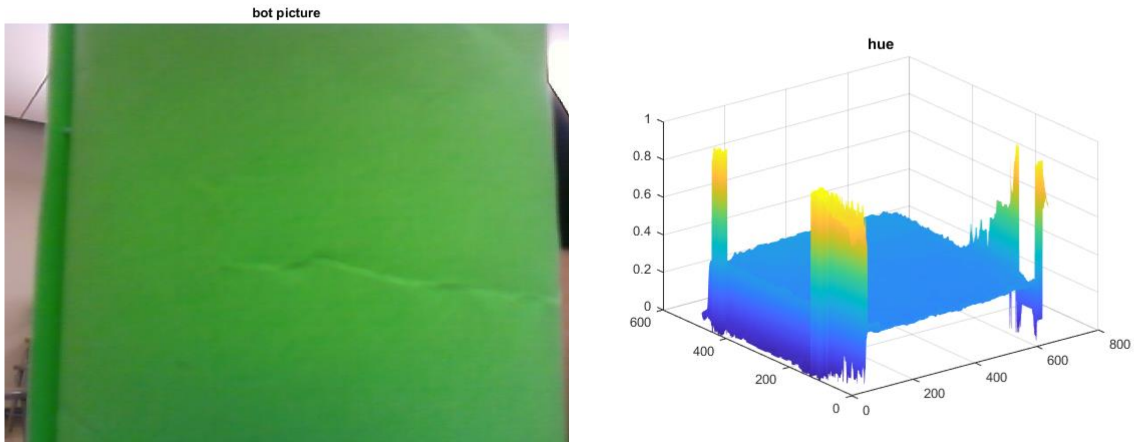
(1) The port state makes sure the orientation of the robot is correct according to a pre-defined port state that is determined by the orange cylinder.

(i) If the camera of the robot is facing the orange cylinder, go to **End state**.

(ii) Otherwise stay in **Port state**.

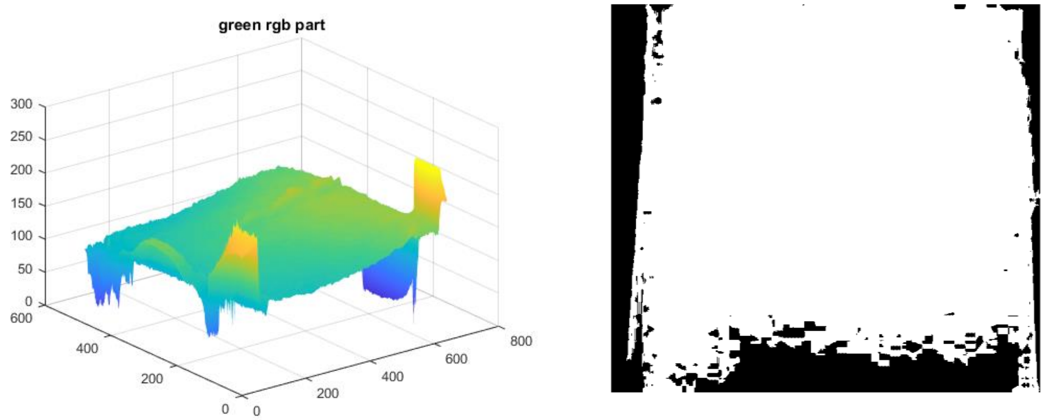
f) End state

(1) This is the final state, and the robot has docked at the predefined location.



–

Figure 5.2 – Image from robot, and hue values from hsv.



–

Figure 5.3 – Green parts from thresholding and BW image with noise removed.

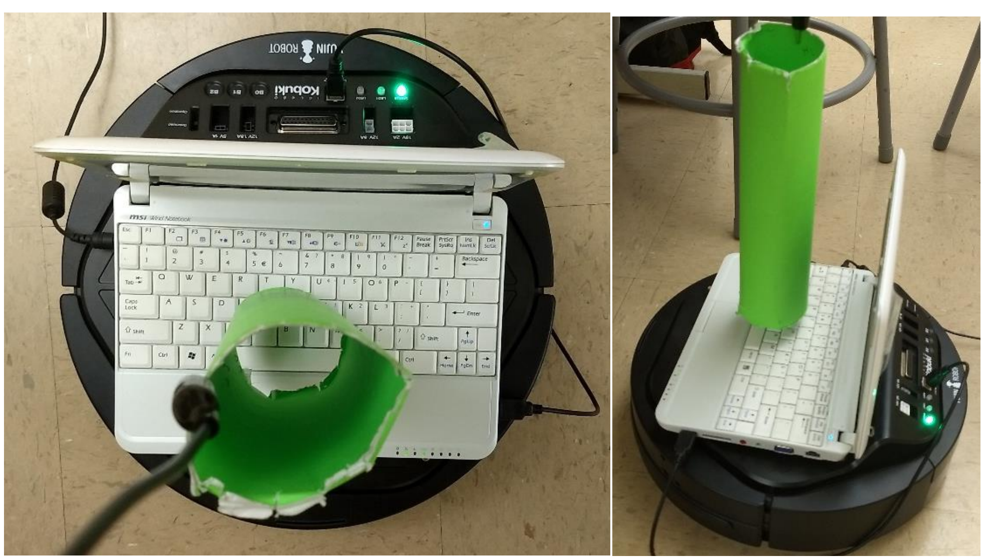


Figure 5.4 – Results from test, robot is <2cm from designated dock position.

Code:

```
clear all
close all
% initialize
rosshutdown % to 'close' any previous sessions
rosinit('192.168.0.101'); % initialize Matlab ROS node
tbot = turtlebot  % the data structure that allows access to the turtlebot and its
sensors
tbot.ColorImage.TopicName = '/usb_cam/image_raw/compressed'; % set netbook webcam as
the camera for tbot
% these are the variables that are used to define the robot velocity
lin_vel = 0;  % meters per second
rot_vel = 0;  % rad/second
% create a Matlab "timer" for continuoulsy sending velocity commands to the robot.
% This will create a new thread that runs the command setVeloc-
ity(tbot,lin_vel,rot_vel) every 50msec
velocity_command_timer = timer('TimerFcn','setVelocity(tbot,lin_vel,rot_vel)','Peri-
od',0.05,'ExecutionMode','fixedSpacing');
% start the timer. This will keep running until we stop it.
start(velocity_command_timer)
% % handle to the simulator
% gazebo = ExampleHelperGazeboCommunicator();
% % create a few colored balls in the environment
% ball = ExampleHelperGazeboModel('Ball')
% spherelink = addLink(ball,'sphere',1,'color',[0.1 1 0 1])
% spawnModel(gazebo,ball,[6.5,1,1]);
% pause so that everything has time to be initialized
pause(2);
x_image_midpoint = 320; %define default x mid point of image
Kp_Rotational = 0.0015; % set kp rotational p controller
Kp_Linear = 0.00000019;
seek_vel = 0.18;
center_error = 10;
area_initial = 2500;
area_dock = 95000;
area_max = 170000;
area_min = 140000;
hue_hb = 0.34;
hue_lb = 0.27;
state = {'seek' 'center' 'approach' 'dock' 'port'};
state = 'seek';
while 1
    tic
    rgbImg = getColorImage(tbot); %get image
    figure(1);imshow(rgbImg);title('bot picture');%show image
    hsv_image = rgb2hsv(rgbImg); %conver rgb to hsv
    hue = hsv_image(:,:,1); %get hue part of hsv
%     figure(2); surf(hue);shading interp; title('hue');%display hue of hsv
    green_part = (hue > hue_lb).*(hue < hue_hb); %find green part of hue between se-
lected values
%     figure(7); surf(green_part);shading interp; title('green cylinder');%display new
hue image with just green part
    bw=im2bw(green_part);
%     figure(10);imshow(bw)
    bw2 = bwareafilt(bw,1);
    figure(11); imshow(bw2);
    [rows,cols]= find(bw2);%finding the area of the cylinder in the image
    if isempty(cols)
        mean_col = [];
        area = [];
    else
```

```
        mean_col = mean(cols);
        area = length(cols)
    end
    switch state
        case 'seek'
            if ((any(green_part(:) > 0)) && (area > area_initial))
                state = 'center';
            else
                rot_vel = seek_vel; %find the cylinders
                state = 'seek';
            end
        case 'center'
            if (any(green_part(:) > 0))
                green_cylinder_center = regionprops(green_part,'centroid'); %find cen-
ter of green part
            end
            x_new_center = green_cylinder_center.Centroid(1); %center of green region
            dist_x = x_new_center - x_image_midpoint; % find the horizontal distance
between the green part and the center of image columns
            absolute_dist = abs(dist_x) %absolute distance to center of green part
            if (absolute_dist < center_error)%it's not at the center, rotate the robot
and to make it at center
                rot_vel = 0.0; %if center of green part found stop rotating
                state = 'approach';
            else
                rot_vel = -Kp_Rotational*dist_x % %if not at center of green cylunder
keep rotating
                state = 'seek';
            end
        case 'approach'
            if(isempty(area)) %if it doesn't see robot in camera
                lin_vel = 0.0;
                rot_vel = 0.0;
                state = 'center';
            else %if robot found in camera
                if(area > area_dock)
                    rot_vel = 0.0;
                    lin_vel = 0.0;
                    state = 'dock';
                else
                    if (any(green_part(:) > 0))
                        green_cylinder_center = regionprops(green_part,'cen-
troid'); %find center of green part
                    end
                    x_new_center = green_cylinder_center.Centroid(1); %center of green
region
                    dist_x = x_new_center - x_image_midpoint; % find the horizontal
distance between the green part and the center of image columns
                    absolute_dist = abs(dist_x);
                    rot_vel = -Kp_Rotational*dist_x;
                    a_dist = area - area_max;
                    lin_vel = -Kp_Linear*a_dist;
                    state = 'approach';
                end
            end
        case 'dock'
            if((area < area_max)&&(area > area_min))
                rot_vel = 0.0;
                lin_vel = 0.0;
                state = 'port';
            else
                if (any(green_part(:) > 0))
```

```
                    green_cylinder_center = regionprops(green_part,'centroid'); %find
center of green part
                end
                x_new_center = green_cylinder_center.Centroid(1); %center of green re-
gion
                dist_x = x_new_center - x_image_midpoint; % find the horizontal dis-
tance between the green part and the center of image columns
                absolute_dist = abs(dist_x);
                rot_vel = -Kp_Rotational*dist_x;
                a_dist = area - area_max
                lin_vel = -Kp_Linear*a_dist*0.5;
                state = 'dock';
            end
        case 'port'
            rot_vel = 0.0;
            lin_vel = 0.0;
            if (any(green_part(:) > 0))
                green_cylinder_center = regionprops(green_part,'centroid'); %find cen-
ter of green part
                x_new_center = green_cylinder_center.Centroid(1); %center of green re-
gion
                dist_x = x_new_center - x_image_midpoint; % find the horizontal dis-
tance between the green part and the center of image columns
                absolute_dist = abs(dist_x);
                rot_vel = -Kp_Rotational*dist_x;
                state = 'port';
            else
                state = 'seek';
            end
    end
    toc
end
```

6 Final Navigation and Kicking Algorithm.

Now we combine all the techniques of the previous steps from the simulation and real robot. We use color detection with thresholding, waypoint navigation, and additionally state machines. But this final step involves more complex image detection processes.

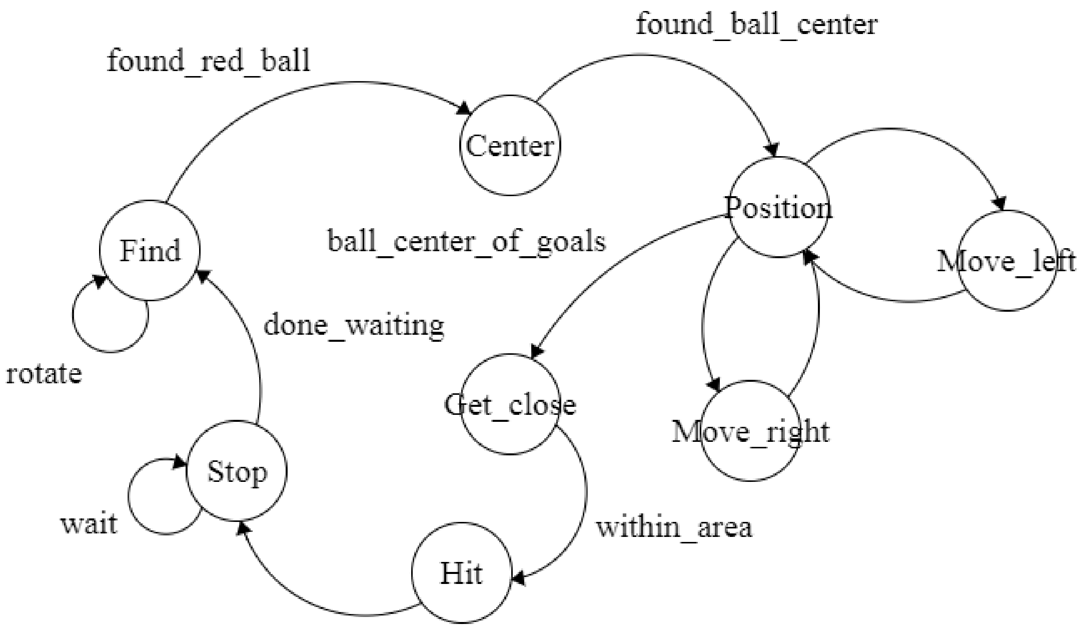


Figure 6.1 – The state machine for this assignment.

The procedure of the program follows the state machine as shown in figure 6.1 and described below. The code file is named hit_ball_v2.m.

3. First step is to define constants used in the program that do not change during operation of the state machine. These parameters are defined depending on exterior conditions to the robot. kp_rotational, kp_linear, seek_vel, center_error, hue bounds, value bounds, saturation bounds, area of objects.
4. Start the while 1 loop.
   A. Take a picture of what is in front of the robot and analyze the following parameters.
      a) Convert RGB image to HSV.
      b) Extract the Hue, Saturation, and Value from HSV.
      c) Threshold the green parts of the image to detect the green cylinders of the goal.
      d) Threshold the orange parts to detect the ball.
      e) Eliminate noise from ball image using the im2bw function.
      f) Calculate the are the ball takes in the image to determine the distance of the robot from the ball.
   B. Start the state machine with a switch statement. The state machine can have the following states; ‘find’, center_on_ball’, ‘find_position’, ‘get_close’, ‘hit’, ‘stop’.
      a) ‘find’ state

The find state looks to see if the orange ball is in the camera of the robot.

(i) If the orange ball is in the image go to **'center_on_ball' state**.

(ii) Otherwise, rotate robot until orange ball is found.

b) 'center_on_ball' state

The center state will center the robot on the orange ball if it is in the image.

(i) Use the regionprops function to detect the center of the orange region, then use the Kp_rotational to center the camera on the orange ball.

   (a) If the camera is centered on the orange ball within the center_error. Then go to **'find_position' state**.

   (b) Otherwise stay in **'center_on_ball' state** until center is found.

c) 'find_position' state

The find position state determines the position of the robot with respect to the ball and goal. Then moves the robot to a position where it can hit the ball into goal.

(i) Find the position of the ball and the goal using the centroid function with the previously determined green and orange regions.

   (a) If the ball is within the margin of error of delta_center_max, and it's close to the center of the goal. Go to **'get_close' state**.

   (b) Otherwise correct the position of the robot one time and go to back to **'find' state.**

   - If the orange ball is to the right of the goal, move the robot to the right slightly.
   - If the orange ball is to the left of the goal, move the robot to the left slightly.

d) 'get_close' state

The get close state positions the robot to hit the ball. It uses the detected area_red_ball to determine the distance of the robot from the ball.

(i) If the robot is in the correct position to hit the ball, go to **'hit' state.**

(ii) Otherwise, if the robot is not in the correct position to hit the ball, remain in **'get_close' state.**

   (a) If the robot is too close to the ball, move the robot back.

   (b) If the robot is too far away from the ball, move the robot closer to the ball.

e) 'hit' state

The hit state makes the robot hit the ball

(i) If the ball is in the view of the camera, move forward, remain in **'hit' state.**

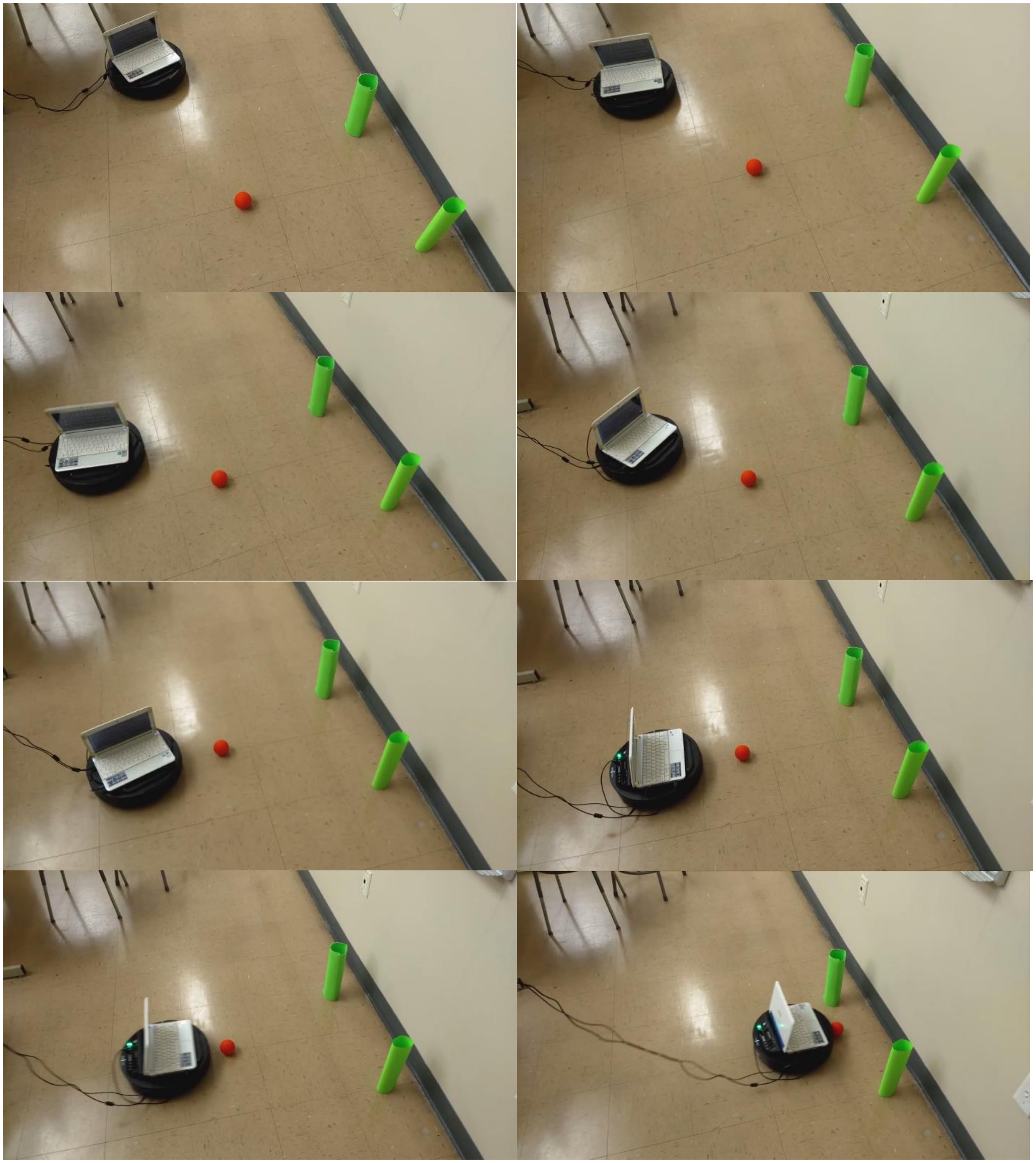
(ii) Otherwise, got to **'stop' state.**

f) 'stop' state

This is the final state. The robot waits for a certain time, and then resets back to **'find' state** to hit ball again.

Video of Robot in action: https://youtu.be/uuPvphMsW40

Results:



Final Code:

```
clear all
close all

% initialize
rosshutdown % to 'close' any previous sessions
rosinit('192.168.111.129'); % initialize Matlab ROS node
tbot = turtlebot  % the data structure that allows access to the turtlebot and its
sensors
% tbot.ColorImage.TopicName = '/usb_cam/image_raw/compressed'; % set netbook webcam as
the camera for tbot

resetOdometry(tbot);
global robot_poses;
robot_poses = zeros(4,10000);
odometry_timer = timer('TimerFcn','curr_pose = get_pose_from_tbot_odome-
try(tbot);','Period',0.1,'ExecutionMode','fixedRate');
start(odometry_timer);

lin_vel = 0;  % meters per second
rot_vel = 0;  % rad/second

velocity_command_timer = timer('TimerFcn','setVelocity(tbot,lin_vel,rot_vel)','Peri-
od',0.05,'ExecutionMode','fixedSpacing');
start(velocity_command_timer);

% handle to the simulator
gazebo = ExampleHelperGazeboCommunicator();
% create a red ball
ball = ExampleHelperGazeboModel('Ball');
spherelink = addLink(ball,'sphere',0.1,'color',[1 0 0 1]);
spawnModel(gazebo,ball,[4,0,1]);
cylinder1 = ExampleHelperGazeboModel('Ball')
spherelink = addLink(cylinder1,'cylinder',[0.5 0.075],'color',[0.1 1 0 1])
spawnModel(gazebo,cylinder1,[5.5,-0.5,1]);
cylinder2 = ExampleHelperGazeboModel('Ball')
spherelink = addLink(cylinder2,'cylinder',[0.5 0.075],'color',[0.1 1 0 1])
spawnModel(gazebo,cylinder2,[5.5,0.5,1]);

% pause so that everything has time to be initialized
pause(2);

x_image_midpoint = 320; %define default x mid point of image %real bot - 320
kp_rotational = 0.0012; % set kp rotational p controller %real bot - 0.0012
kp_linear = 0.00015; %real bot - 0.00015
seek_vel = 0.25; %real bot - 0.25
center_error = 18; %real bot - 18
hue_ub = 0.32; %real bot - 0.39
hue_lb = 0.30; %real bot - 0.30
val_ub = 1.0; %real bot - 1.0
val_lb = 0.35; %real bot - 0.60
sat_ub = 1.0; %real bot - 1.0
sat_lb = 0.70; %real bot - 0.70
area_detect_ball = 500; %real bot - 500 for far
area_close_to_ball = 2000; %real bot - 3000 for infront of bot
delta_center_max =50; %real bot - 50
seek_left = 1;

state = {'find' 'center_on_ball' 'find_position' 'get_close' 'hit' 'stop'};
state = 'find';
```

```
while 1
    tic

    rgbImg = getColorImage(tbot); %get image
    figure(1);imshow(rgbImg);title('bot picture');%show image
    hsv_image = rgb2hsv(rgbImg); %conver rgb to hsv
    hue = hsv_image(:,:,1); %get hue part of hsv
    saturation = hsv_image(:,:,2);
    value = hsv_image(:,:,3);
    green_part = (hue > hue_lb).*(hue < hue_ub);
    red_part = (((value >= val_lb).*(value <= val_ub)).*((saturation >= sat_lb).*(sat-
uration <= sat_ub))).*(hue < hue_lb);

    bw=im2bw(red_part);
%     figure(10);imshow(bw)
    red_bw = bwareafilt(bw,1);
    figure(11); imshow(red_bw);

    green_goal_center = regionprops(green_part,'centroid')
    red_ball_center = regionprops(red_bw,'centroid');

    [rows,cols]= find(red_bw);%finding the area of the object in the image
    if isempty(cols)
        mean_col = [];
        area_red_ball = [];
    else
        mean_col = mean(cols);
        area_red_ball = length(cols)
    end

    switch state
        case 'find'
            state = 'find'
            if ((any(red_bw(:) > 0)) && (area_red_ball > area_detect_ball))
                state = 'center_on_ball';
            else
                if (seek_left)
                    rot_vel = seek_vel; %find the ball
                    state = 'find';
                else
                    rot_vel = -seek_vel; %find the ball
                    state = 'find';
                end
            end
        case 'center_on_ball'
            state = 'center_on_ball'
            if ((any(red_bw(:) > 0)) && (area_red_ball > area_detect_ball))
                red_ball_center = regionprops(red_bw,'centroid');
                x_new_center = red_ball_center.Centroid(1);
            end
            dist_x = x_new_center - x_image_midpoint;
            absolute_dist = abs(dist_x)
            if (absolute_dist < center_error)
                lin_vel = 0.0;
                rot_vel = 0.0;
                state = 'find_position';
            else
                rot_vel = -kp_rotational*dist_x
                state = 'center_on_ball';
            end
```

```
        case 'find_position'
            state = 'find_position'
            if (((any(green_part(:) > 0)) && (any(red_bw(:) > 0))) && (area_red_ball >
area_detect_ball))
                red_ball_center = regionprops(red_bw,'centroid');
                green_goal_center = regionprops(green_part,'centroid')
                delta_center = abs(green_goal_center.Centroid(1)-red_ball_center.Cen-
troid(1));
            end
            if (delta_center < delta_center_max)
                lin_vel = 0.0;
                rot_vel = 0.0;
                state = 'get_close';
            else
                if (green_goal_center.Centroid(1) < red_ball_center.Centroid(1))
                    lin_vel = 0.0;
                    rot_vel = -0.5;
                    pause(2.6);
                    lin_vel = 0.3;
                    rot_vel = 0.0;
                    pause(1.0);
                    lin_vel = 0.0;
                    rot_vel = 0.0;
                    seek_left = 1;
                    state = 'find';
                else
                    lin_vel = 0.0;
                    rot_vel = 0.5;
                    pause(2.5);
                    lin_vel = 0.3;
                    rot_vel = 0.0;
                    pause(1.0);
                    lin_vel = 0.0;
                    rot_vel = 0.0;
                    seek_left = 0;
                    state = 'find';
                end
            end
```

```
        case 'get_close'
            state = 'get_close'
            if((any(red_bw(:) > 0)))%if it doesn't see ball in camera
                if (area_red_ball > area_close_to_ball)
                    lin_vel = 0.0;
                    rot_vel = 0.0;
                    pause(1);
                    state = 'hit';
                else %if ball found in camera
                    if ((any(red_bw(:) > 0)) && (area_red_ball > area_detect_ball))
                        red_ball_center = regionprops(red_bw,'centroid');
                        x_new_center = red_ball_center.Centroid(1);
                    end
                    dist_x = x_new_center - x_image_midpoint;
                    absolute_dist = abs(dist_x);
                    rot_vel = -kp_rotational*dist_x;
                    a_dist = area_red_ball - area_close_to_ball;
                    lin_vel = -kp_linear*a_dist;
                    state = 'get_close';
                end
            else
                lin_vel = 0.0;
                rot_vel = 0.0;
```

```
                state = 'find';
            end

        case 'hit'
            pause(1);
            if((isempty(area_red_ball))) %if it doesn't see ball in camera
                lin_vel = 0.0;
              rot_vel = 0.0;
                state = 'stop'
            else %if ball found in camera
                if ((any(red_bw(:) > 0)) && (area_red_ball > area_detect_ball))
                    red_ball_center = regionprops(red_bw,'centroid');
                    x_new_center = red_ball_center.Centroid(1);
                end
                dist_x = x_new_center - x_image_midpoint;
                absolute_dist = abs(dist_x);
                rot_vel = -kp_rotational*dist_x;
                lin_vel = 0.4;
                state = 'hit'
            end
        case 'stop'
            state = 'stop'
            lin_vel = 0.0;
            rot_vel = 0.0;
            pause(2);
            state = 'find';
    end
    toc
end
```

7 Appendices

Appendix A: Parts and Software List

- Matlab
- Matlab Robotics System Toolbox
- Turtlebot 2 Robot
- Router
- Laptop