Robotics I : Final Project Report
Launchpad McQuack
Team 22

Lucas Vanslette, Khurram Malik, Patrick Donelan

December 18, 2020

Contents

1 Executive Summary

This project applies lane following and object detection on a mobile robot, MIT's Duckiebot. This report outlines the methodology used to implement the features mentioned. Discussion about the chosen ROS model is included along with the project procedure and the sub goals achieved that led up to lane following and object detection. The results of combining techniques learned from Robotics 1 along with heuristic algorithms such as line-fitting and PID control are presented at the end of the report. Lastly, future goals and a comprehensive conclusion suggest possible improvements to design and opportunities to enhance the Duckiebot's capabilities.

1.1 Project Video and Links

Project Video Link :

`https://drive.google.com/file/d/189H3lRajXp_I8dLTLV1eNRTq0zB9ItbY/view`

Project GitHub Link :

`https://github.com/kmalik97/Launchpad-McDuck`

2 Introduction

Launchpad McQuack is Scrooge McDuck's personal driver/pilot in the kids cartoon Duck Tales; the aim of this project was to create an automated vehicle that would make Scrooge and Launchpad proud. As the use of autonomous vehicles increases throughout the world, it is important to understand the fundamental techniques used when building autonomous robots. The Duckiebot is a small-scale implementation of an autonomous mobile robot. Many of the problems encountered while creating the Duckiebot can be translated directly to large-scale autonomous vehicles such as color-based object detection and motor control.

Figure 1: The Duckiebot vs a Tesla at a stop sign.

Autonomous vehicles have a multitude of applications beside self-driving cars. Ease of package delivery with UAVs, increasing construction safety with autonomous cranes, and wheeled package-carrying robots would be used to operate in factories for more efficient labor. The Duckiebot is a stepping stone to understand a complex autonomous system.

3 Technical Content

3.1 Objective

The base goal was to achieve lane tracking using the camera module mounted on the robot. While Launchpad is known for crashing, the target goal was to implement path planning for obstacle avoidance. Lastly, the reach goal of the project was to perform different obstacle avoidance techniques based on obstacle color.

3.2 Plan

	Task
1	Duckietown Gym Setup
2	ROS integration
3	Camera output
4	User controlled movements
5	Line following
6	PID lane following
7	Object detection
8	React to object detection
9	Simple color-based actions
10	Color-based obstacle avoidance

Table 1: Key Project Tasks

The above table depicts the crucial steps involved with the project. The first established task was to get the simulated Duckietown environment running. This task was high priority due to the nature of the COVID-19 pandemic. It was desirable to have an environment which allowed for remote testing without having to schedule in person meetings in order to limit contact. The first few weeks were spent attempting to get the simulation to work but to no avail. The base simulation was functional, but there was no success connecting the simulation with ROS. Eventually, there was no longer time to spare on trying to get the simulation to work, so Task 1 was abandoned. Task 2 involved getting the skeleton of

the ROS nodes setup as well as basic communication between nodes. Once the skeleton was established, Tasks 3 and Tasks 4 could be developed concurrently. The goal of Task 3 was to successfully interface with the PiCamera module and display the image for the user to ensure proper image capturing. The goal of Task 4 was to use keyboard input from the user to remotely control the Duckiebot in order to verify the motors could be controlled properly. Once both tasks were completed, the next logical step was simple line following. Line following was considered the next logical step as it required less algorithmic development than lane following. Lane following using PID control was then completed afterwards. Object detection development began while also working on Task 6 as it would not cause any conflicts. Task 7 ended up tying in with Task 9 as the focus was on red objects. Once red objects were detected, development towards treating red obstacles as a stop sign was completed. Due to time constraints, obstacle avoidance was not implemented, but progress made in obstacle detection was sufficient.

3.3 ROS Architecture

The ROS client-service model was implemented over the alternative approach of the publish-subscribe model. This decision was due to the nature of the solution that was chosen, which included requesting an image rather than waiting for the a camera capture. The upside of the client-service model is a new camera image was received only at the moment requested. In the latter method there could be timing issues where the publisher and subscriber are out of sync and the image being processed would not be the most recent camera capture. This client-service model was also progressive as it formed a function call stack, which was easy to follow and debug.

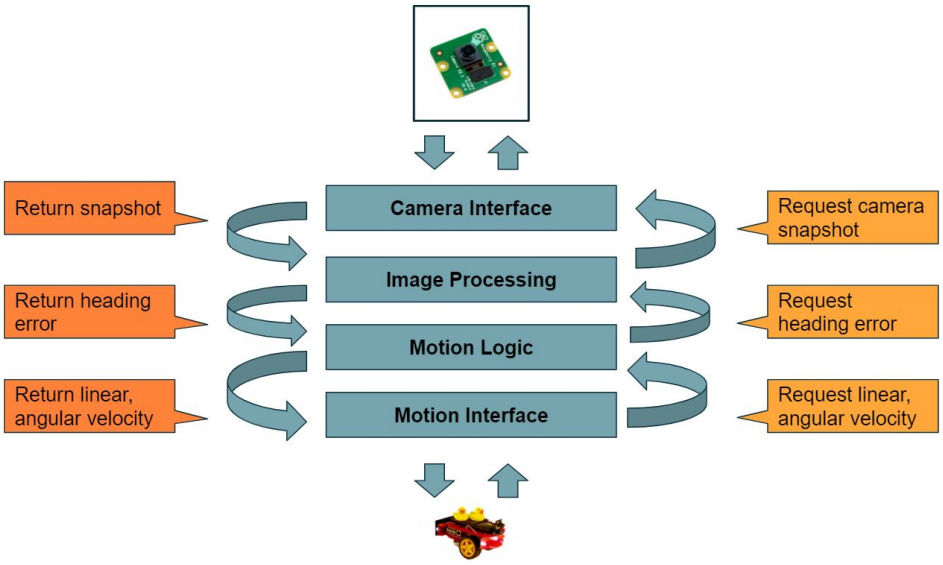

Figure 2: Flow diagram depicting the ROS node implementation strategy.

The first node that starts the ROS client-service model process depicted in Figure 2 is the Motion Interface node, which requests a linear and angular velocity from the Motion Logic node at a rate of 10 ms. One of the main functions of the Motion Interace node is converting the linear and angular velocities to 0-255 PWM values to send to the Duckiebot motors for movement. The Motion Logic node determines the velocities based on the error between the target path and the current trajectory of the Duckiebot. PID control is implemented for calculating angular velocity in the Motion Logic node, and linear velocity is held constant. The error is given to the Motion Logic node as a return value from the Image Processing node. Polyfit is used in the Image Processing node to get the line of best fit based on a color-filtered camera image. The image matrix that the Image Processing node uses to polyfit the data is returned from the Camera Interface node. Using the PiCamera video stream to take a camera capture and convert it into a 320x240 matrix of RGB pixel tuples, the Camera Interface node sends the array of pixels to the Image Processing node.

3.4 Camera Calibration

Initially, MATLAB and Python scripts were used for camera calibration. However, the distortion coefficients gathered by both resulted in poor image undistortion and the returned intrinsic matrices did not result in reasonable measurements. One of the greatest challenges with these scripts was the rejection of images deemed unusable. For strong calibration, many images are required. Thus, rejected images had to be replaced, but these new images themselves were not always accepted. The result was a very lengthy process of trying to calibrate with enough images. The following figure depicts a subset of the many images taken. Shown are 16 images in Figure 3 that were taken of which only 6 images were deemed good enough for calibration.

Figure 3: Image Calibration Attempts

The alternative solution to this problem was to use the camera calibration tool in ROS. This tool was incredibly convenient as it uses a video stream to obtain calibration images versus the process of taking photos, uploading them, and replacing rejected images. Furthermore, the calibrator visually displays if enough data has been obtained for X, Y, Size, and Skew estimation; this ensures diversity for calibration which we could not ensure with the image upload-and-check technique. The camera calibration tool requires the images to be sent through a raw image topic, so a publisher node was created using much of the same code as in *camera_interface.py*.

The resulting distortion coefficients and camera intrinsic matrix are illustrated below.

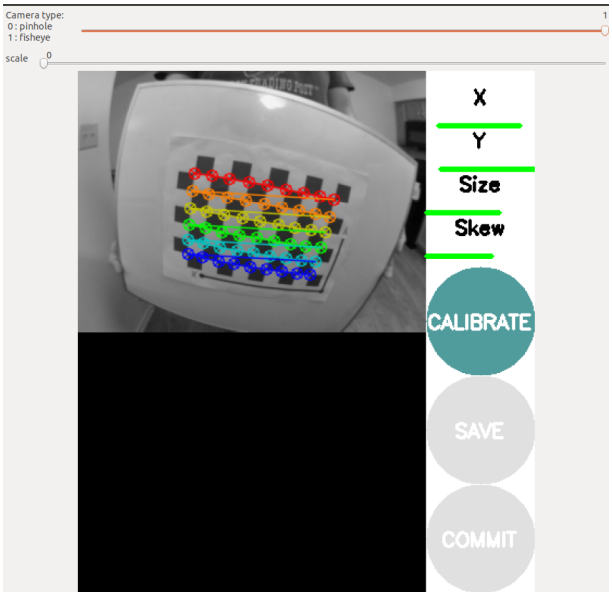

Figure 4: ROS Calibration Tool

$$D = \begin{bmatrix} -0.266205 & 0.045222 & -0.001402 & -0.000906 \end{bmatrix}$$

$$K = \begin{bmatrix} 160.023219 & 0 & 162.971810 \\ 0 & 160.263543 & 123.423868 \\ 0 & 0 & 1 \end{bmatrix}$$

Figure 5: Distortion Coefficients and Intrinsic Matrix

Using these values for K and D, a new undistorted intrinsic camera matrix, K_{new}, was obtained using Python's *cv2.getOptimalNewCameraMatrix* function. K_{new} was the intrinsic matrix used to calculate the homography matrix.

$$K_{new} = \begin{bmatrix} 97.19328308 & 0 & 161.60555983 \\ 0 & 97.18055725 & 122.03446744 \\ 0 & 0 & 1 \end{bmatrix}$$

Figure 6: Undistorted Intrinsic Matrix

The new intrinsic matrix states the optical center in the image plane is at $\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} 161.60555983 \\ 122.03446744 \end{bmatrix}$

As illustrated in Figure 7, undistortion was successful since the original image (left) depicted curved lines on the checkerboard whereas the undistorted image (right) portrays straight lines.

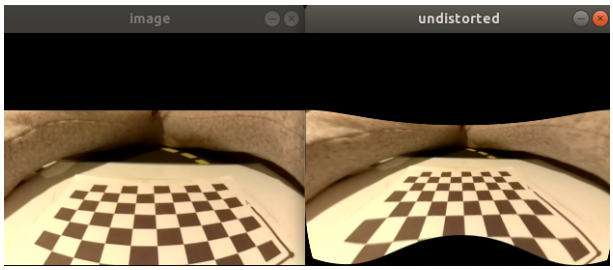

Figure 7: Undistortion

Once all of the intrinsic parameters were obtained, the extrinsic parameters had to be estimated. First, the pose of the camera in the world frame was estimated using a ruler and protractor. The camera was approximately 100mm above the ground and angled 110 degrees about the x-axis. Using MATLAB's *cameraPoseToExtrinsics* function, the extrinsic rotation matrix, $\boldsymbol{R}$, and translation vector, $\boldsymbol{t}$, were estimated. More accurate measurements could have been made for the camera pose, but was deemed unnecessary as any error would be accommodated for by the PID controller.

$$R = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -0.3420 & -0.9397 \\ 0 & 0.9397 & -0.3420 \end{bmatrix}$$

$$t = \begin{bmatrix} 0 \\ 93.9693 \\ 34.2020 \end{bmatrix}$$

Figure 8: Extrinsic Rotation Matrix and Translation Vector

With this camera pose, the scenario can be visualized as below.

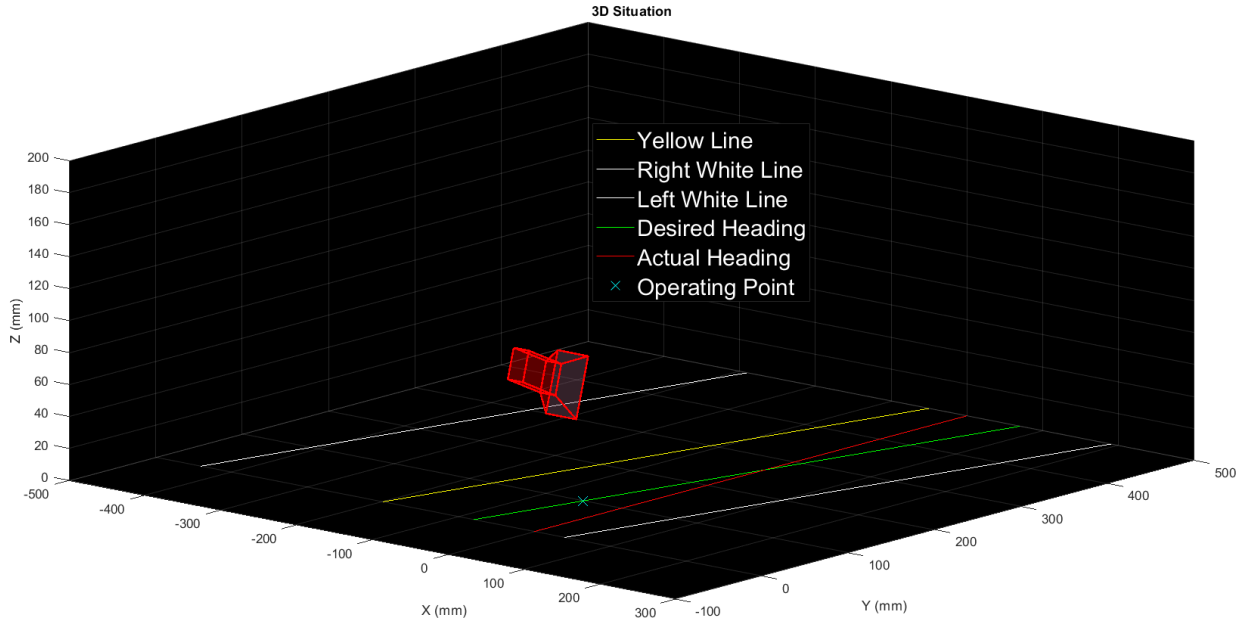

Figure 9: 3D Scenario View

Lastly, the homography matrix for mapping from image plane to world plane and vice versa was computed.

$$H = K_{new} * [R^{(1)} R^{(2)} t] = 1e{+}04 \begin{bmatrix} 0.0097 & 0.0152 & 0.5527 \\ 0 & 0.0081 & 1.3306 \\ 0 & 0.0001 & 0.0034 \end{bmatrix}$$

Figure 10: Homography Matrix

3.5 Lane Following

This section will cover the lane following algorithm used by Launchpad McQuack in order to follow the path between a dotted yellow and solid white line. The following figure depicts the general outline of the algorithm used for lane following.

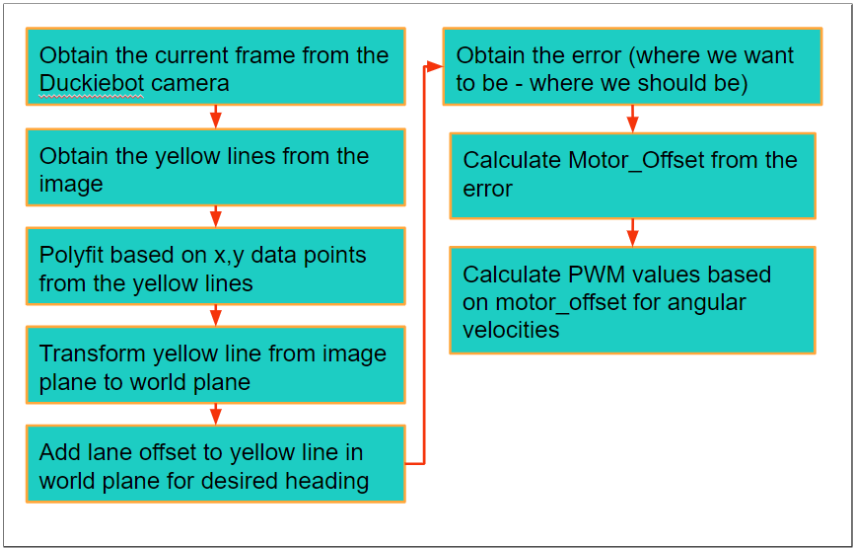

Figure 11: Lane Following Algorithm Flowchart

3.5.1 Image Manipulation

The lane following algorithm begins by first obtaining a frame from the Duckiebot's camera during each iteration of *image_processing.py*. Once the image is the received, the top **1/3** of the image is blacked. This is done to remove the walls from the image. The top 1/3 of the image isn't useful for analysis; noisy data would often result since the objects farther away will be harder to detect and undesirable objects may be mistakenly detected. The following Figure 12 shows the results of this image manipulation. Note that the cropped image is only used for lane detecting. Other aspects such as object detecting do now use a cropped image.

Figure 12: Before and after of image removal

3.5.2 Obtaining Yellow Lines

Once the top 1/3 of the image is removed, the image is then transformed from the distorted form to its undistorted form using the intrinsic camera matrix and the distortion coefficients that were obtained. The opencv function *undistort* is used to accomplish this. Once the undistorted image is obtained, the yellow lines are found by masking out any yellow colors using an upper and lower bound on the HSV values found in the image. In order to assist in determining the required upper and lower bounds, the code shown in Appendix A was used. The HSV values needed to be updated frequently as different lighting conditions would result in different acceptable HSV values. From the resulting images of the yellow lines, the pixel coordinate values that are yellow are extracted. These data points are then fit with a 1 degree polynomial fit to obtain a path of the yellow line.

During testing it was determined that a 2 degree polynomial fit served better in some cases, namely curves, however there were some issues where the polynomial fit would over fit the line and cause the robot to veer off in an uncontrollable state. Once a polynomial fit was obtained, the line was then converted from image coordinates to world coordinates. In other words, from pixel values into real world mm values. This was done by using the homography matrix and the calculation is described in the following equation.

$$Yellow_{points\_mm} = H^{-1} \cdot Yellow_{points\_px} \tag{1}$$

Once the yellow points were transformed from image coordinates to world coordinates, the line was offset by a real world distance of 100 mm to be in the center of the lane. The robot is essentially following points offset from the yellow. The following Figure 13 depicts how the results would look like, where the yellow line is the polynomial fit of the yellow dashed lines. The green line is the offset from the yellow line or the robot's desired heading and the red line is illustrated as the robot's actual heading.

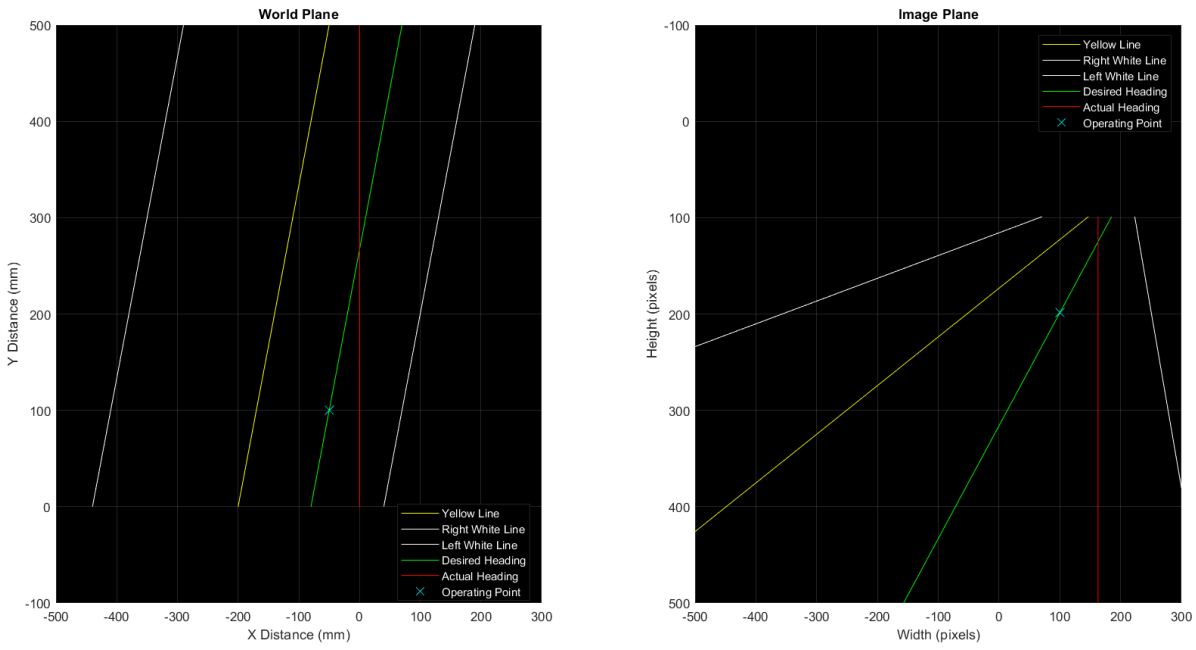

Figure 13: Lane offset Diagram from matlab

Now that the desired heading of the robot has been calculated, as well as the actual heading of the robot, by taking the difference at an operating point. This number is

defined as the error. The error is the difference between where the robot's actual heading is and where the robot's heading should be. The following Figure 14 illustrates how the error is calculated.

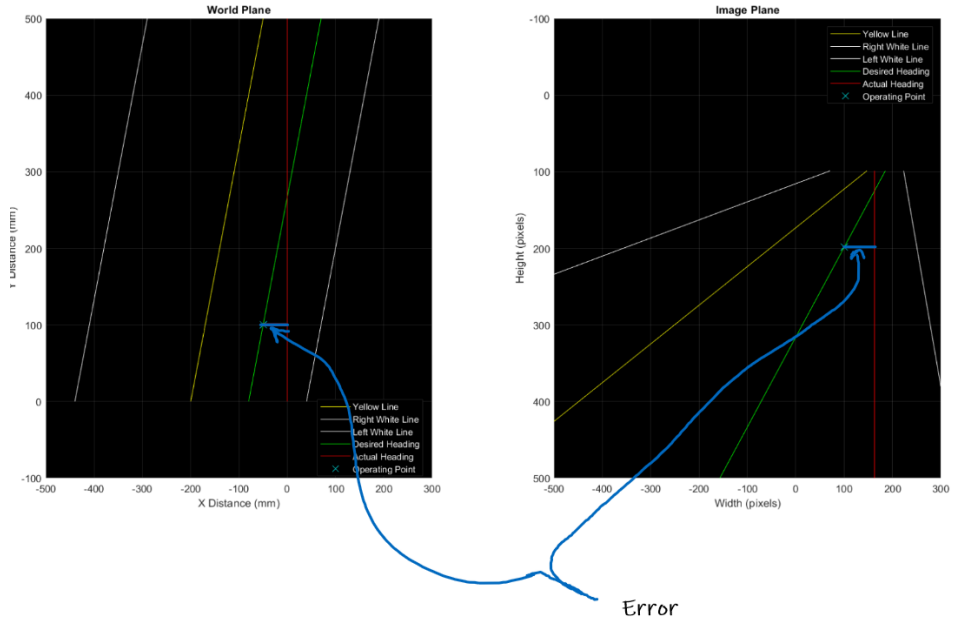

Figure 14: Error shown as the difference between the green line and the red line

3.5.3 Motor Offset Calculation

After the error is obtained, the motor offset must be calculated. The motor offset is a function of the error that was calculated. The motor offset is used to calculate how much angular velocity to apply to each motor. The following graph in Figure 15 demonstrates the general concept on how the error that was calculated relates to the motor offset that is calculated.

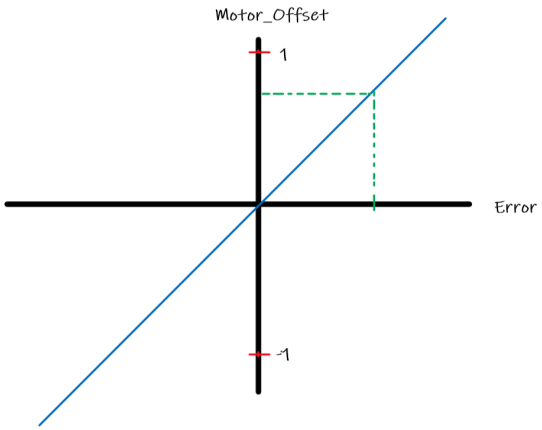

Figure 15: Motor Offset Graph

The motor_offset ranges from -1 to 1. This maps to a full right or a full left turn respectively, and any value in between -1 and 1 is some combination of a left and right turn. According the to the graph, if the error is negative, that means the robot has veered to the left and needs to turn right to get back on the desired path. Similarly, if the robot has a positive error, it will require left turn to return to the desired position. The motor offset is modeled as a PID equation as shown below.

$$Motor\_Offset = K_p \times error_{current} + K_i \times error_{running} + K_d \times \Delta error \tag{2}$$

The K_p, K_i and K_d parameters are referenced to as the proportional, integral and derivative terms. Essentially, the K_p term is proportional to the error, or the slope of the line in the graph shown in Figure 15. The K_i parameter takes into account the cumulative error. As the error grows larger, the integral term will result in a larger component attempting to "pull" the robot back. The K_d parameter takes a look at the change in error. If the

error changes are significant, the derivative term will result in a proportionally large term to pull the error back. The following table shows the values PID values that were used.

K_p	0.006
K_i	0.001
K_d	0.01

Table 2: PID Parameters Used

The following snippet of code is shown from *motion_logic.py* on how the motor offset equation was implemented.

```
# running error is the error accumulation over time
# if there is a sign change or its close to zero, otherwise keep adding the error
if not(np.sign(x_error) + np.sign(self.prev_error)) or (abs(x_error) < 0.01):
    self.running_error = 0
else:
    self.running_error += x_error
# error difference between this function call and the previous function call
delta_error = x_error - self.prev_error
motor_offset = x_error * Kp + Ki * self.running_error + Kd * delta_error
```

3.5.4 PWM Value Calculation

Once linear, v, and angular, ω, velocities are obtained, they need to be converted into PWM values. Each motor has a designated minimum PWM and maximum PWM, and the motor's resolution is defined as the difference between the maximum and minimum PWM. Two PWM values are calculated for each motor - a linear PWM and angular PWM.

$$PWM_{L|v} = -sign(v) * \lfloor |v| * res_L + min_L \rfloor \quad (3)$$
$$PWM_{R|v} = sign(v) * \lfloor |v| * res_R + min_R \rfloor \quad (4)$$

Figure 16: Linear PWM Equations

The only difference between the equation for the left motor and the right motor is that the PWM for the left motor is multiplied by -1 since the left motor has opposite polarity.

Important to note is $sign(0) = 0$, so no linear velocity corresponds to no linear PWM. For velocities not equal to zero, they are offset by PWM_{min}. Without these minimum offsets, the Duckiebot gets stuck; there is not enough power to drive a motor forward. The values are kept as integers using the floor function.

$$PWM_{L|\omega} = sign(\omega) * \lfloor |\omega| * max_L \rfloor \quad (5)$$
$$PWM_{R|\omega} = sign(\omega) * \lfloor |\omega| * max_R \rfloor \quad (6)$$

Figure 17: Angular PWM Equations

The angular PWM equations are identical for the two motors. These equations do not have a PWM_{min} offset because getting stuck is already accommodated for by the linear PWM equations. If the Duckiebot was allowed to turn in place, which it isn't, then the PWM_{min} offset would be required. Another important difference is the lack of sign flipping for the left motor to account for the reversed polarity. This is due to the fact that a negative angular velocity accounts for a right turn, meaning the left wheel should rotate forwards faster than the right wheel. Thus, for a forward right turn the linear PWM for the left motor is negative and the angular PWM is negative, so the sum of the two is a PWM value with a larger magnitude. For the right motor, the linear PWM is positive and the angular PWM is negative, resulting in a PWM with a smaller magnitude. A left turn is analogous with the right motor getting a larger PWM value than the left motor. This illustrates the basic idea of turning - whatever is added to one motor is subtracted from the other motor.

The two PWM components cannot simply be added together, however, as resolution is limited. Thus, the components must have a reserved fraction of the resolution. Reserving 20% of the resolution for the angular PWM proved to be effective. Lower values resulted in turning too slow while higher values resulted in oscillatory behavior. Thus, 80% of the resolution was used for the linear PWM. Through testing, it was discovered that the robot was traveling too fast on turns, so the solution was to add damping to the linear resolution, Thus, the linear resolution fraction was calculated as

$$res_v = 1 - res_\omega - 0.25 * |\omega|$$

Figure 18: Linear Resolution

This equation illustrates that the harder the turn (larger $|\omega|$), the lower the linear resolution, thus leaving up to 25% of the total motor resolution untouched. This method allowed for slowing the robot without the trade-off of increasing its angular resolution.

$$PWM_L = \lfloor res_v * PWM_{L|v} + res_\omega * PWM_{L|\omega} \rfloor \quad (7)$$
$$PWM_R = \lfloor res_v * PWM_{R|v} + res_\omega * PWM_{R|\omega} \rfloor \quad (8)$$

Figure 19: PWM Equations

Bound checks were applied to ensure the final PWM values did not extend beyond the possible range. Once the final PWM values were obtained, the motor direction and speed were set using the Adafruit motor hat library provided by Honglu He[1].

3.6 Object Detection

This section will cover LaunchPad McQuack's obstacle avoidance algorithms. The following Figure 20 illustrates the general steps taken during object detection.

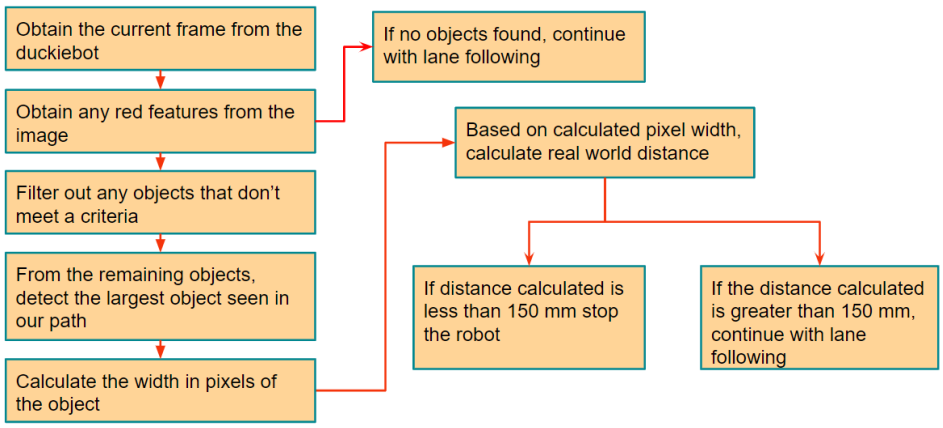

Figure 20: Object Detection Flowchart

3.6.1 Image Manipulation

During each iteration of the *image_processing.py* a frame from the Duckiebot in analyzed. Once the image is obtained, the image is then transformed into its undistorted form using the intrinsic and distortion coefficients, similar to the Lane following algorithm. Once the image is undistorted, a red mask is applied using the hsv values for a red object. This will result in an image appearing like Figure 21.

Figure 21: Red image masking before and after

From the above figure the contours are then found. Contours can be explained simply as a curve joining all the continuous points along a boundary having the same color or intensity [3]. Using the result from the contour, the largest red object is then taken by find the contour is the largest area.

3.6.2 Distance Calculation

The objective of obstacle avoidance at this time was to stop at a red object that was in the path of the Duckiebot and less than some distance away. In order to calculate the distance to the object, the focal length of the camera must be used. There are two methods of calculating the focal length.

1. Using the real world dimensions and pixel values to calculate distance using the following formula.

$$FOCAL\_LENGTH = \frac{(Width_{px} \times Distance_{real})}{Width_{real}} \tag{9}$$

2. Using the intrinsic parameters of the camera from the homography matrix which are the f_x or f_y values to determine the focal length.

During the testing of both methods, it was determined that both methods resulted in similar results with only a less than 4% relative difference between the two values at close ranges. For consistency purposes, the focal length was also calculated using the f_x values from the camera matrix, but both methods would have yielded similar results at close ranges. In order to calculate the distance to an object the following equation was applied.

$$distance = \frac{(width_{known\_real} * FOCAL\_LENGTH)}{width_{px\_seen}} \tag{10}$$

It was important to measure the real word width of the red object that was begin detected in order to transform a pixel reading into a distance reading. Although both methods resulted in similar distance calculations as they were close to an object, discrepancies started to appear on objects that were further away. In order to determine which method would result in the most accurate result a test was conducted comparing the actual real world distance with the calculate distances for both methods. Through this test, it was found that using the homography matrix resulted in a lower mean squared error as compared to using the focal length calculation. This meant that the distance calculations using the homography matrix were more accurate as compared to the distance calculations

using the calculated focal length. The results of this test are shown in the following Figure 22

Test Point	Actual Distance Calc	Using H	
1	165.1	166.07	0.97
2	209.55	155.26	54.29
3	285.75	293.6	7.85
4	298.45	264.95	33.5
5	412.75	399.9	12.85
6	942.9	868.21	74.69
			184.15
		MSE [mm]	30.69166667
Test Point	Actual Distance Calc	Using Focal Length Calc	
1	165.1	212.23	47.13
2	209.55	176	33.55
3	285.75	337.5	51.75
4	298.45	308	9.55
5	412.75	454.78	42.03
6	942.9	738.32	204.58
			388.59
		MSE [mm]	64.765

Figure 22: Distance Measured via two methods. All units in mm

When the data shown above is plotted, the following figure is the result.

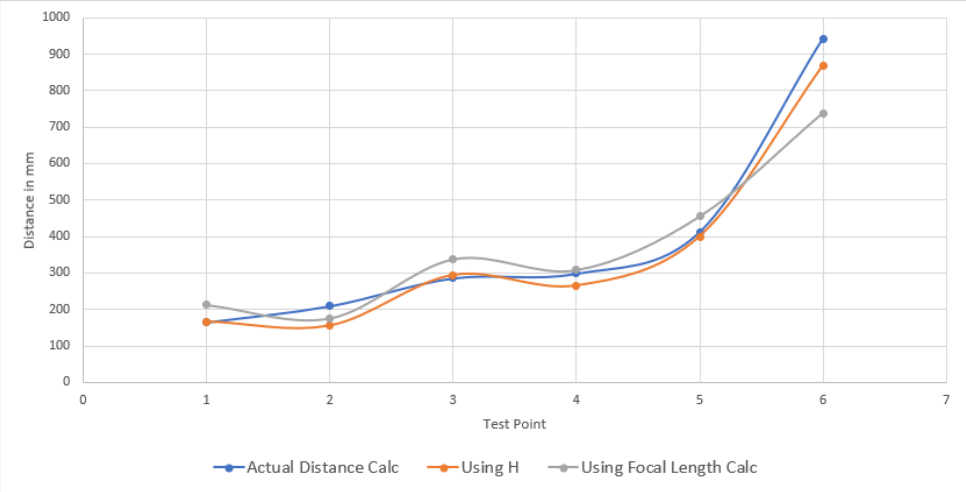

Figure 23: Distance Measured via two methods graph

From the data gathered, the measurements, from both methods were more accurate the closer the Duckiebot was to the object. Notice on the graphs that the error increases as the actual distance increased. At the end it was decided to that using the homography matrix yielded better and more accurate results.

Once the distance to the object was detected, a simple condition was checked in the object was less than 150 mm away from the Duckiebot. If so, stop the Duckiebot by setting both the angular and linear velocities to 0. If not, continue with the lane following algorithm.

3.7 Object Detection Improved

Once the object detection was working consistently, the algorithm was improved to have the Duckiebot wait for a set period of time and the continue with its lane following. The flow chart is shown in the following diagram with the changes highlighted in a different color. Much of the algorithm is still the same as described in the previous section.

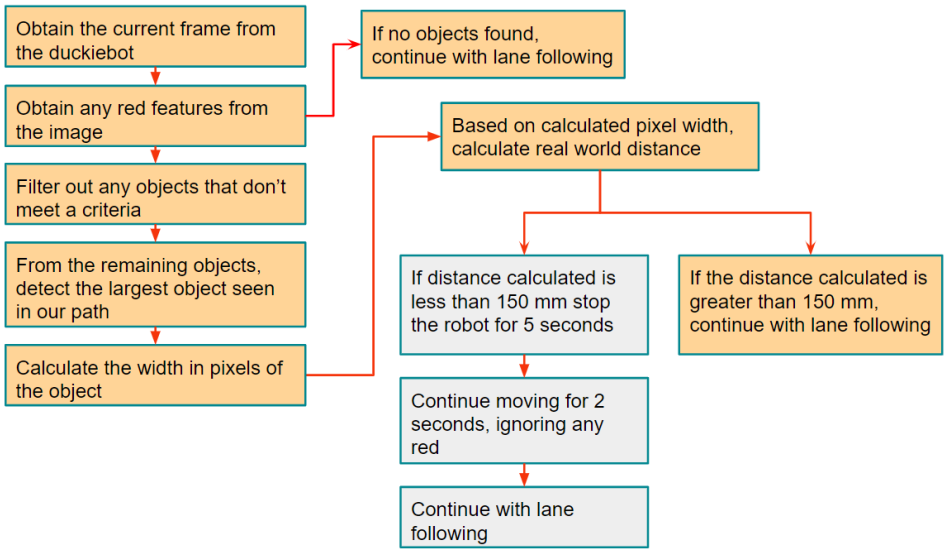

Figure 24: Object Detection Flowchart

Other changes to the object detection also included a new object that the Duckiebot was detecting. Instead of detecting a large red container, the Duckiebot is now detecting a relatively small rectangular stop sign. An image of this stop sign is shown in Figure 25

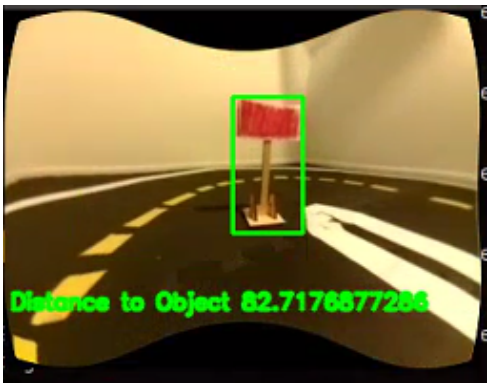

Figure 25: Duckiebot Stop Sign

Similar to the original algorithm, the Duckiebot masks all the red objects and finds the object that meets its criteria and measures its distance as discussed before. The change

in the algorithm is that instead of waiting indefinitely the Duckiebot will only wait for 5 seconds until continuing with its lane following program. During implementation of this new algorithm a bug was discovered when the 5 seconds were up, the Duckiebot would still have the stop sign within its stopping criteria and would endlessly loop and thus would become stuck. In order to combat this issue, after a stop sign is seen and 5 seconds have passed, the duckiebot will overwrite it's red object stopping criteria for 3 seconds and continue on its path. This would give the Duckiebot enough time to move pass the red stop sign and not trigger an endless loop. A snippet of the code that was used to accomplish this is shown below.

```
1             # Get the current time
2             current_time = time.time()
3
4             # stop at stop sign
5             if red_obj_det and not self.just_stopped:
6                 linear_vel = 0.0
7                 angular_vel = 0.0
8                 self.just_stopped = True
9                 self.stop_time = time.time()
10            elif self.just_stopped and current_time - self.stop_time <= 5:
11                linear_vel = 0.0
12                angular_vel = 0.0
13            elif self.just_stopped and current_time - self.stop_time > 5 and
               ↪  current_time - self.stop_time < 8:
14                linear_vel = v
15            else:
16                self.just_stopped = False
```

3.8 Results

In order to have accurate lane following, accurate x-distance measurement was needed. Thus, tests were conducted and the results are shown in Figure 26. The measurements proved to be both extremely accurate and precise when the error was less than about

70mm. After 70mm, the measurements continued to be precise but lost accuracy. This loss of accuracy was tolerable as the PID control would still pull the Duckiebot closer to the desired position, where the measurements would be more accurate. Another important note is that the measurement accuracy and precision is consistent for both positive and negative errors.

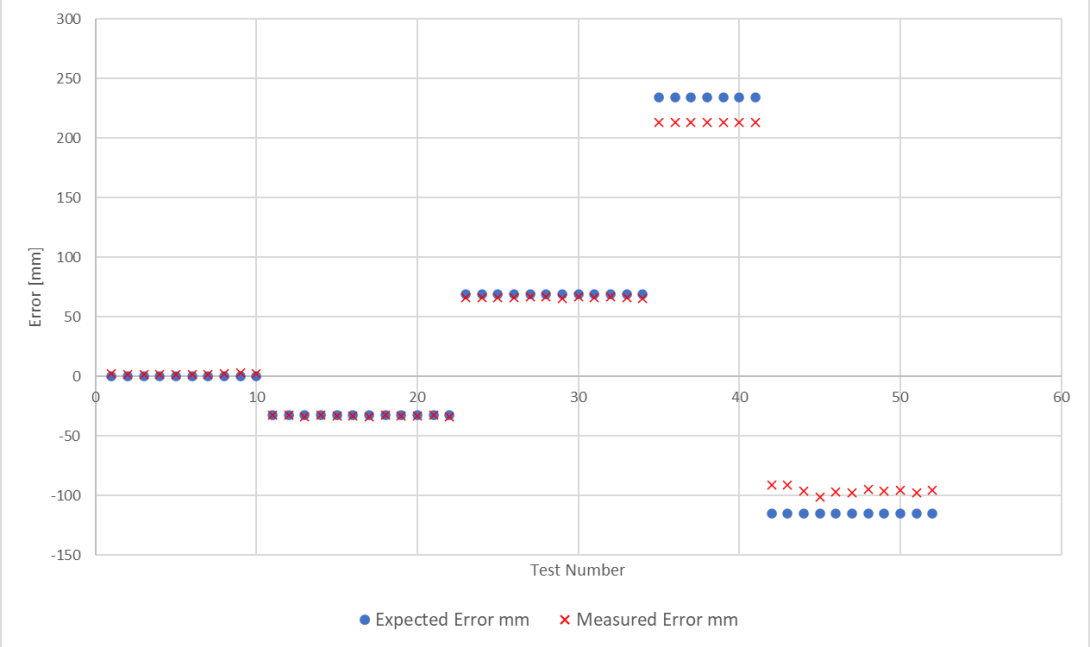

Figure 26: X Measurement Error. Note: The closer the dots are together, the more accurate the measurements were.

The following graph illustrates the error over time for two runs on a racetrack loop*. Both runs exhibit oscillatory behavior and generally transition from larger magnitude errors to lower magnitude errors overtime. Interesting to note is the steady-state error of about -60mm in run02 starting around the 58 second mark. The integral control should have accounted for this error and pulled the Duckiebot to the right, but perhaps the integral control gain was not large enough to accomplish this. This steady-state error is the cause of run02 having a larger mean error (-26.72mm) than run01 (-6.44mm). Despite not achieving roughly zero error, the Duckiebot is able to reasonably stay within a lane while completing at least one loop without veering off the track.

* : The second run shown in the graph is the run shown in the video.

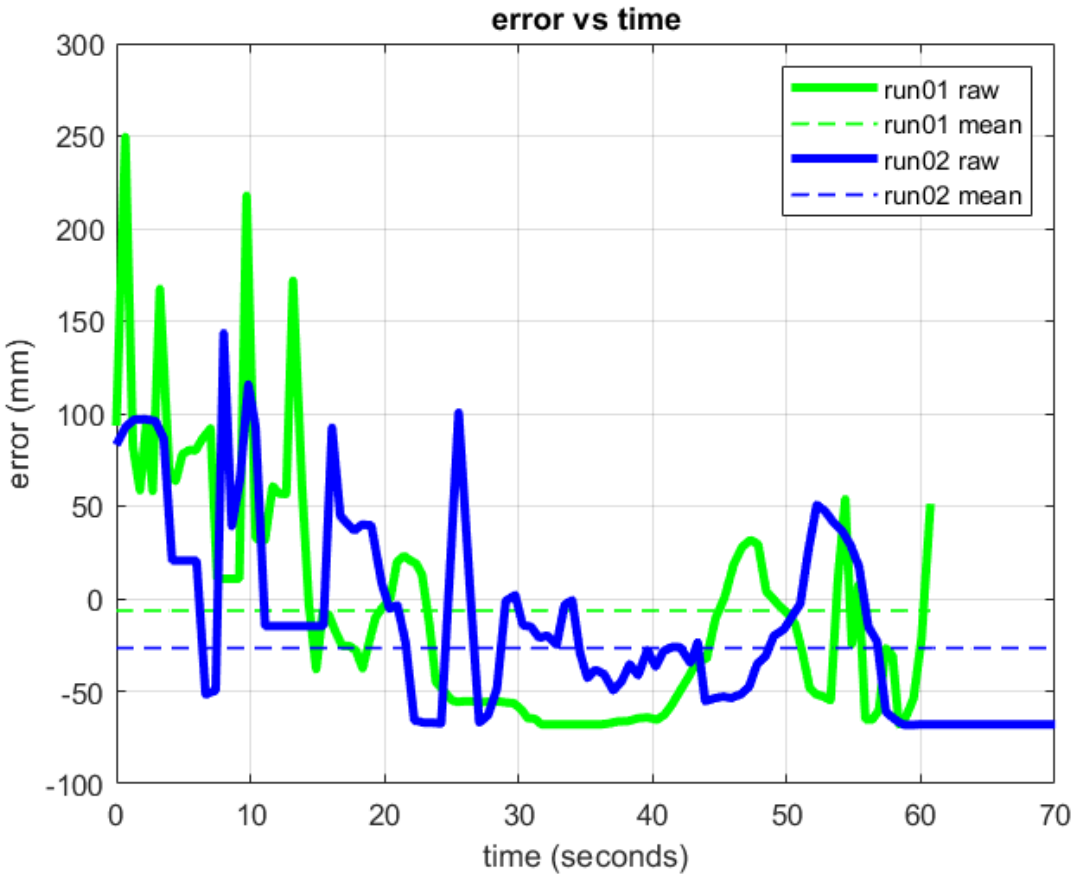

Figure 27: Error vs Time

For object detection, the Duckiebot was successfully able to locate and measure the distance to a red obstacle. Then, the Duckiebot would continue driving until the stopping distance was met, where it paused for a couple seconds before resuming driving. This performed exactly as desired. Unfortunately, there was not enough time to implement object avoidance. Had there been enough time, local path planning algorithms, such as potential field method, would have been explored.

4 Future Work

There are nearly limitless expansions to this project. Obtaining the reach goal of color-based obstacle avoidance is the most obvious expansion. Different local obstacle-avoiding path planning algorithms, such as potential field, could be implemented for obstacles of

different colors. Not only could obstacle avoidance be implemented for stationary objects, but also algorithms could be developed to account for moving obstacles, such as another Duckiebot driving in the same or opposite lane. Global path planning algorithms could be implemented to essentially give the Duckiebot a ”GPS route” to follow through a Duckietown setup, while continuing to use lane following and local path planning for obstacle avoidance. Additionally, machine learning could be implemented so the Duckiebot learns more and more about the routes as it drives, resulting in better performance on the tracks. Perhaps one of the most practical expansions that could be implemented is some sort of adaptive color thresholding. Isolating certain colors was very difficult throughout this project as it is extremely sensitive to lighting conditions, which are hard to keep constant. Thus, adaptive thresholding would allow for running the Duckiebot in different environments without re-calibrating color threshold values.

5 Conclusion

The goal of this project was to implement lane following as well as object detection. The goals specified were achieved by the end of the project timeline. Launchpad McQuack attempts to find a target line based on the middle yellow line. If the target line is lost, the Duckiebot continues with its previous trajectory. This is to account for points at which the yellow line is not easily seen on the camera such as taking a tight turn. When testing, Launchpad McQuack successfully made multiple loops around the test track as well as stopped at stop signs for a short amount of time before continuing lane following. Issues arose when a camera capture did not find enough points from the middle yellow line, resulting in a small amount of data given to the polyfit function and thus a bad estimate for the target line. PID control helped with mitigating the noisy camera images as well as any other unaccounted for biases.

Additional functionality of the Duckiebot can be devised such as object avoidance and variable threshold values to increase color detection accuracy. However, the starting goal of lane following and object detection was successfully met. Further objectives can be

feasibly reached; there is a high ceiling in regards to the possibilities for Launchpad McQuack.

6 References

References

[1] hehonglu123. *Duckiebot_Survey_ROS_RR*. URL: https://github.com/hehonglu123/Duckiebot_Survey_ROS_RR.

[2] n/a. *Build a line-following robot*. URL: https://projects.raspberrypi.org/en/projects/rpi-python-line-following/6.

[3] OpenCV. *OpenCv Camera Calibration Documentation*. URL: https://docs.opencv.org/master/dc/dbb/tutorial_py_calibration.html.

[4] Team. *Project Github*. URL: https://github.com/kmalik97/Launchpad-McDuck.

7 Appendix A : HSV Value Finder Code

```
import cv2
import sys
import numpy as np
# https://stackoverflow.com/questions/10948589
#/choosing-the-correct-upper-and-lower-hsv-boundaries-for-color-detection-withcv
def nothing(x):
    pass

useCamera=False

# Check if filename is passed
if (len(sys.argv) <= 1) :
    print("'Usage: python hsvThresholder.py <ImageFilePath>'
    to ignore camera and use a local image.")
    useCamera = True
# Create a window
cv2.namedWindow('image')

# create trackbars for color change
cv2.createTrackbar('HMin','image',0,179,nothing) # Hue is from 0-179 for Opencv
cv2.createTrackbar('SMin','image',0,255,nothing)
cv2.createTrackbar('VMin','image',0,255,nothing)
cv2.createTrackbar('HMax','image',0,179,nothing)
cv2.createTrackbar('SMax','image',0,255,nothing)
cv2.createTrackbar('VMax','image',0,255,nothing)

# Set default value for MAX HSV trackbars.
cv2.setTrackbarPos('HMax', 'image', 179)
cv2.setTrackbarPos('SMax', 'image', 255)
cv2.setTrackbarPos('VMax', 'image', 255)

# Initialize to check if HSV min/max value changes
```

```
hMin = sMin = vMin = hMax = sMax = vMax = 0
phMin = psMin = pvMin = phMax = psMax = pvMax = 0

# Output Image to display
if useCamera:
    cap = cv2.VideoCapture(0)
    # Wait longer to prevent freeze for videos.
    waitTime = 330
else:
    img = cv2.imread(sys.argv[1])
    output = img
    waitTime = 33

while(1):

    if useCamera:
        # Capture frame-by-frame
        ret, img = cap.read()
        output = img

    # get current positions of all trackbars
    hMin = cv2.getTrackbarPos('HMin','image')
    sMin = cv2.getTrackbarPos('SMin','image')
    vMin = cv2.getTrackbarPos('VMin','image')

    hMax = cv2.getTrackbarPos('HMax','image')
    sMax = cv2.getTrackbarPos('SMax','image')
    vMax = cv2.getTrackbarPos('VMax','image')

    # Set minimum and max HSV values to display
    lower = np.array([hMin, sMin, vMin])
    upper = np.array([hMax, sMax, vMax])

    # Create HSV Image and threshold into a range.
    hsv = cv2.cvtColor(img, cv2.COLOR_BGR2HSV)
```

```
68      mask = cv2.inRange(hsv, lower, upper)
69      output = cv2.bitwise_and(img,img, mask= mask)
70
71      # Print if there is a change in HSV value
72      if( (phMin != hMin) | (psMin != sMin) | (pvMin != vMin) | (phMax != hMax) | (psMax
        ↪  != sMax) | (pvMax != vMax) ):
73          print("(hMin = %d , sMin = %d, vMin = %d), (hMax = %d , sMax = %d, vMax = %d)"
            ↪  % (hMin , sMin , vMin, hMax, sMax , vMax))
74          phMin = hMin
75          psMin = sMin
76          pvMin = vMin
77          phMax = hMax
78          psMax = sMax
79          pvMax = vMax
80
81      # Display output image
82      cv2.resize(output,(1000 ,2000))
83      cv2.imshow('image',output)
84
85      # Wait longer to prevent freeze for videos.
86      if cv2.waitKey(waitTime) & 0xFF == ord('q'):
87          break
88
89  # Release resources
90  if useCamera:
91      cap.release()
92  cv2.destroyAllWindows()
```