CS5335 Robotic Science & Systems Final Project Report

Topic: Duckiebot Autonomous Mobile Robot

Team Name: Mobile-Vision

Team Member: Shaoshu Xu, Xingyu Lu

Demo Video Link: https://youtu.be/8fMeBfrNmNY

**** Thanks professor for providing us the Duckiebot and all the related parts. This is a rare experience!***

I. Problem

Autonomous vehicle (AV), also known as self-driving car, driverless car or robotic car, is a vehicle that is capable of sensing its environment and moving safely with little or no human input. It combines a variety of sensors to perceive their surroundings, such as radar, lidar, sonar, GPS, odometry and inertial measurement units. Advanced control systems interpret sensory information to identify appropriate navigation paths, as well as obstacles and relevant signage [1].

To achieve fully automated driving, the whole system must monitor the environment and make the appropriate response to the dynamic driving task. The overall problem is decomposed into several manageable subtasks. The following Figure 1 shows different parts and the subtasks of the system:

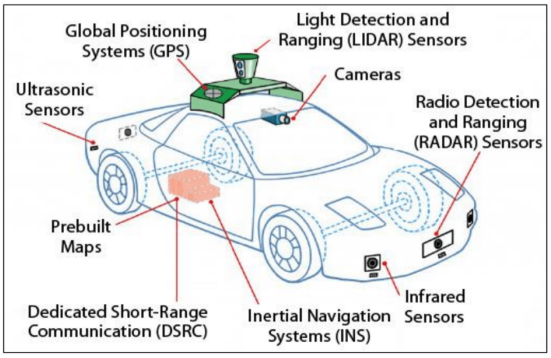

Figure 1. Autonomous Vehicle Components

Self-driving cars in general have four main components:

- Perception: Allows cars to perceive the environment and make sense of it using cameras, LiDAR, radar and ultrasonic sensors. Sensor fusion is used here to combine the measurements from all these sensors and augment the perception.
- Localization: Gives vehicle's location using GPS and implements algorithms to estimate vehicle's precise position on the map. By localizing itself, the vehicle can estimate its precise relationship to all the elements on the map.
- Planning: Given the localization and perception, finding an appropriate path to maneuver around the dynamic environment.

- Control: Given the plan, send signals to make the vehicle move. It is responsible for the path tracking and driving safety.

In our project, we address the problems that present in the real autonomous driving field by using one Duckiebot autonomous mobile robot and Duckietown environment, as shown in figure 2. More specifically, the components that we focus on are as follows:

- Perception (Lane Following and Object Detection): First, we use a monocular camera to detect lanes on the Duckietown tiles. The Duckiebot drives autonomously following the lane in Duckietown. Second, we train a YOLO object detector which receives the camera image and outputs each objects' classification and bounding box. To reduce the model size, we use tiny YOLO which is a variant version of the YOLO architecture that has fewer convolutional layers. The training dataset contains four classes of objects: Duckiebots, Duckies, Stop signs and Road signs. The dataset is from this GitHub repository [2] and contains 420 raw images collected from the Duckiebot camera.
- Sensor Fusion (Map 2D Laser Scan Points to 2.5D Point Clouds): Due to the limit information provided by 2D laser scanner, we proposed a method to map 2D laser scan points to 2.5D point clouds. Even though this is a pseudo-3d point cloud (this is also the reason why we call it 2.5D), it still provides us more information compared with 2D laser scan.
- Localization (Hector-SLAM): We use the SlamTec RPLiDAR A1 to implement Hector-SLAM method building a map of the environment. Rplidar is a low cost 2D LIDAR solution developed by RoboPeak Team, SlamTec company [3]. It can scan a 360° environment within 6meter radius. Hector-SLAM mapping is an approach that can be used without odometry as well as on platforms that exhibit roll/pitch motion. It leverages the high update rate of modern LiDAR systems and provides 2D pose estimates at scan rate of the sensors. It constructs 2D mapping in an unknown environment and capability to localize its own location based on landmarks detected [4].

Figure 2. Duckiebot and Duckietown environment

II. Process

The following subtasks of our whole system are introduced in chronological order and the time consuming are indicated. The whole process will be discussed as the following six parts:

A. Duckiebot & Duckietown Hardware Assemble

B. Duckiebot Preliminary Setting

C. Hector-SLAM

D. Map 2D Laser Scan Points to 2.5D Point Clouds

E. Lane Following

F. YOLO Object Detector

A. Duckiebot & Duckietown Hardware Assemble (Shaoshu Xu & Xingyu Lu, 7 hours)

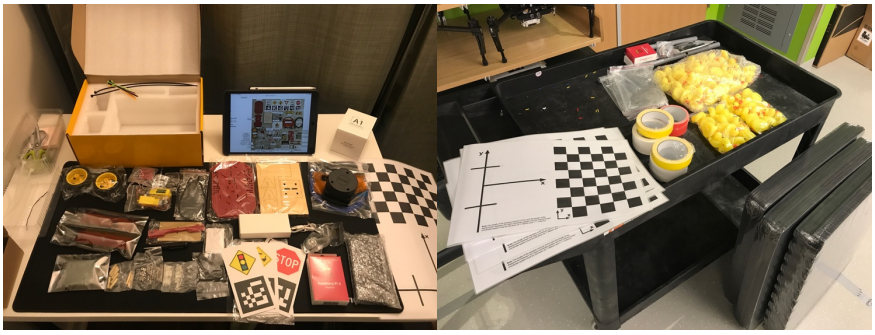

Figure 3. All the pieces needed to assemble Duckiebot and Duckietown

The first step of our project is building Duckiebot and Duckietown. All the pieces are shown above in figure 3. Next several steps and figures will show some detailed process to assemble the hardware:

A-1: Mount the motors and omni-directional wheel (shown in figure 4)

Figure 4. Motors and omni-directional wheel

A-2: Raspberry Pi, Duckiebot Hut and camera cable (shown in figure 5)

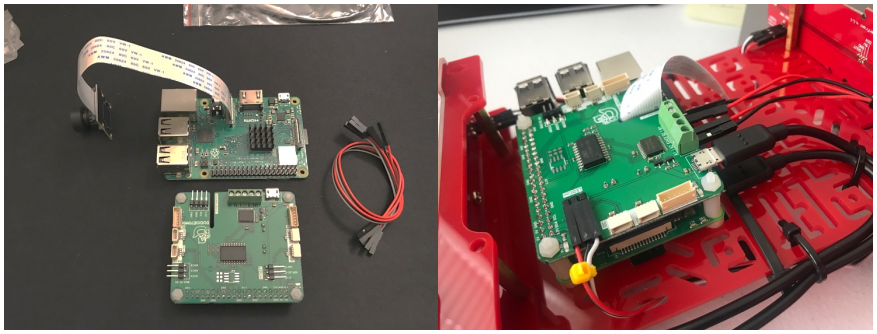

Figure 5. Raspberry Pi

A-3: Chassis assembly (shown in figure 6)

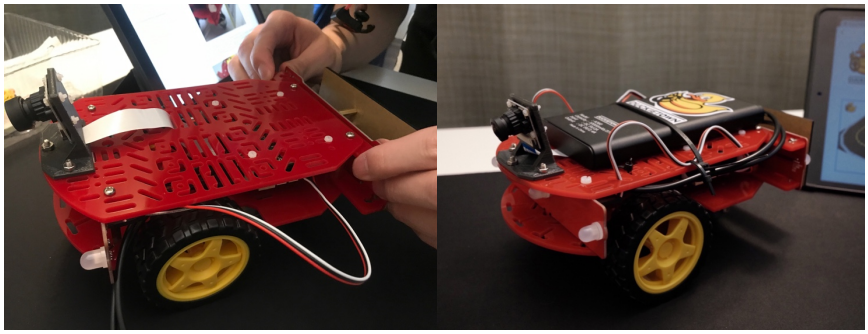

Figure 6. Assemble chassis

A-4: Mount the additional chassis and LiDAR (shown in figure 7)

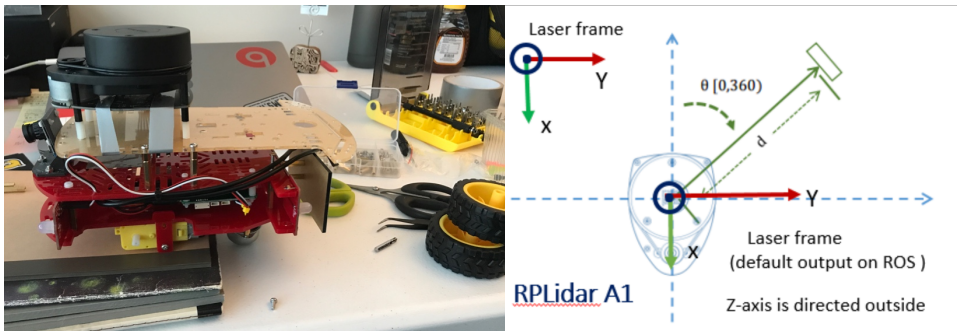

Figure 7. Assembly the Lidar (left); RPLidar A1installation manual (right)

A-5: Test the LiDAR, assembly battery parts and modified easy-switch (shown in figure 8)

Figure 8. Test the Lidar

A-6: Assembly the Duckietown road tiles and traffic signs following the Duckietown Appearance Specifications. The finished Duckietown are shown in figure 9.

Figure 9. Build Duckietown environment

Some problems we encountered: The first one is when mounting the motors the bottom chassis is not fit the motor mounts well. We have to only install one screw instead of two. The second is when connecting the motor cable with Duckiebot Hut, the instruction shows the wrong order of the Hut connector. And when we running the Duckiebot in the next step, we found it runs with inverse direction.

B. Duckiebot Preliminary Setting (Shaoshu Xu & Xingyu Lu, 15 hours)

The second step is setting up the Duckiebot. This part is much more time-consuming than we expected. It is not only because there are so many steps, but also there are many obscure places in the online instructions. So, we have to spend much time on figuring out those issues by ourselves and with only limited online resources. We also spend a lot of time getting familiar with Docker. Several main steps will be discussed following:

B-1: Laptop setup

Set up the Ubuntu 16.04 laptop: Get basic dependencies (installs pip, git, git-lfs, curl, wget) and Install Docker and Duckietown Shell. We took some time in installing the Duckietown Shell. Because the instruction warned us not using "sudo pip" to install the Duckietown Shell. Finally, we still used the sudo pip command to install the Shell without any issue. Details about some problems we met will be described below:

First, according to the tutorial it said that "don't use the sudo pip ** command to install Duckietown Shell". However, the installation required the permission to install files in the system. Without using "sudo" command, there was always permission denied error. So, we chose to use "sudo pip **" command to install Duckietown Shell successfully. Until now we have not faced any trouble caused by this issue, so we didn't spend more time in figuring out how to solve the permission problem.

Second, after the installation we could not run the 'dts version' command to view the Duckietown Shell version. It showed that there were no dt_shell and other corresponding packages. At the beginning, we thought it could be due to the path problem because we skipped the path adding step by accident. So, we edited the profile to add '~/.local/bin: $PATH' at last. However, the 'dts version' still couldn't work and showed the same error, which means there were errors in installation. Then we restarted the installation: returned profile to initial one, reinstalled the Duckietown Shell and then added the proper path again. However, it still showed 'No packages ***' errors as before. Since we followed the tutorial seriously, the installation process could not be wrong and the packages were actually installed. The only explanation could be: the packages were at the different path from the one that the system searched for the target packages. We googled some explanations of the same case and looked at some system directories. Finally, we figured out the reason which is: by following the

commands in tutorial, we installed the Duckietown Shell packages within home directories under "~/.local/lib/python3.7/site-packages" because we used "--user" flag. But the system searched for them under "/usr.lib/python3.7/dist-packages" when we ran the dts code. There were two solutions: 1. We could simply copy the packages installed within the home directory to the /pyhton3.7/dist-packages in system directories. 2. We could reinstall the dts packages without the "--user" flag to let them be in our target path, and added the new path into the profile file.

After the following steps, we successfully installed the Duckietown Shell and worked properly.

B-2: Duckietown account

Sign up on the Duckietown and setting the Duckietown Shell using the token.

B-3: Duckiebot Initialization

Burn the SD card and Booting the Duckiebot. We spent a lot of time in booting the Duckiebot, because the instruction is unclear and we have to try multiple times to successfully boot the Duckiebot.

During the last process of SD card burning, an error existed which shows 'Could not find docker-compose, cannot validate file'. However, the burning process did not interrupt at that time. Because the tutorial did not mention any requirement or installation need of this package, we just skipped it. After we inserted the card into the robot and started initialization, some little problems occurred which were caused by the lack of docker-compose. Then we had to install this package and burned the SD card again to solve this problem.

Things need to be cautious in SD card burning step: when we burn the SD card, it would be better to add our own WiFi network for connecting the robot with the remote laptop. Because SD card burning and initialization are both time consuming, we should add all the configurations we might use in the future to eliminate time wasting and extra underlying trouble.

B-4: Networking setting

Connect Duckiebot to the internet through a WiFi router that we controlled, and Push Docker Images from Laptop. We took some detours in setting the network. Initially, we bridged the internet connection through laptop with Ethernet, and after that we found that the ROS cannot run anymore due to some network IP address issue.

B-5: Setting up the Docker workflow

Use the Docker functionality and introducing some monitoring tools and workflow tips. This part was a bit obscure, because we were not familiar with Docker.

B-6: Making the Duckiebot move

This part was straightforward by following the operation manual, and it didn't take too much effort to make the robot move.

B-7: See what the Duckiebot sees

View the image stream on laptop and verify the data streams output in ROS. At first, we didn't see anything because the duckiebot-interface container was not running. The robot needs some time to finish initialization after turning on the power. After the initialization, the basic containers including duckiebot-interface would be running. It could be checked by viewing the container status in local at Portainer: https://speedhunter.local:9000/#/container. Then, the remote laptop was able to connect to the robot and receive the image data which could also be viewed from the image view window. The image view window could be recalled by "rqt_image_view" in duckiebot terminal.

B-8: Camera calibration and validation

Focus the image by rotating the mechanical focus ring on the lens of the camera. Calibrate the camera intrinsic and extrinsic parameters and stored on the Duckiebot.

Figure 10. Camera calibration step

B-9: Wheel calibration

Place the Duckiebot on one end of the tape. Make sure that the Duckiebot is perfectly centered with respect to the line. Command the Duckiebot to go straight for about 2 meters. The Duckiebot drifted by less than 10 centimeters, we could stop calibrating the trim parameter.

Figure 11. Wheel calibration step

C. Hector-SLAM (Shaoshu Xu, 6 hours)

C-1: RPLiDAR A1 Device Settings and Testing

RPLIDAR is a low cost 2D LIDAR solution developed by RoboPeak Team, SlamTec company. It can scan 360° environment within 6meter radius. It has 12-meter detection range, 8000 times sample

rate and 2-10Hz scan rate. The output of RPLIDAR is very suitable to build map, do slam, or build 3D model.

First we connected the RPLiDAR with laptop via a micro USB cable and remaped the USB serial port name to /dev/ttyUSB0. Next, we ran rplidar ROS package and view in the RVIZ. An example of the laser scan result shows in figure 10.

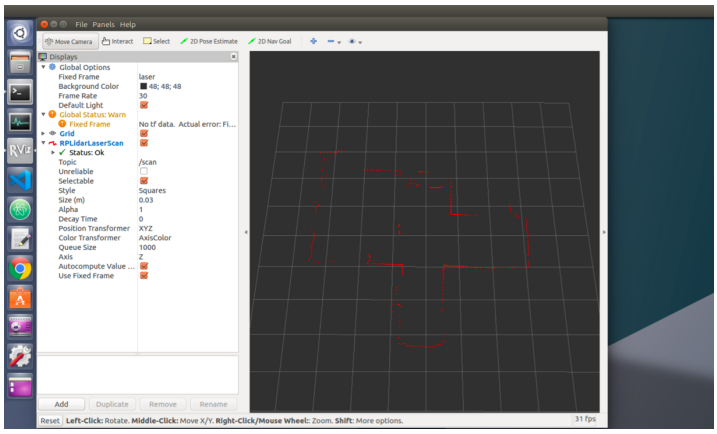

Figure 12. 2D laser scan

C-2: Hector-SLAM

Hector_mapping is a SLAM approach that can be used without odometry as well as on platforms that exhibit roll/pitch motion (of the sensor, the platform or both). The system produces 2D map of environment precisely that can be used for navigation of mobile robot. While the system does not provide explicit loop closing ability, it is sufficiently accurate for many real world scenarios. The system has successfully been used on Unmanned Ground Robots, Unmanned Surface Vehicles, Handheld Mapping Devices and logged data from quadrotor UAVs.

The system subscribes to the sensor "msgs/LaserScan" message and publishes "nav msgs/OccupancyGrid", tf transformation, and the pose with covariance message geometry "msgs/PoseWithCovarianceStamped". This information can be used for sensor fusion for systems based on Hector SLAM data. To properly set up and start the hector-slam, we also created a slam.launch file under the "rplidar_ros/launch/" directory.

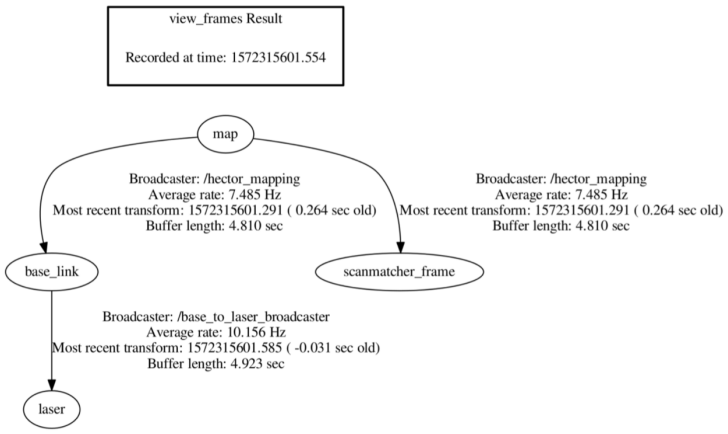

Figure 13. Frame relationship

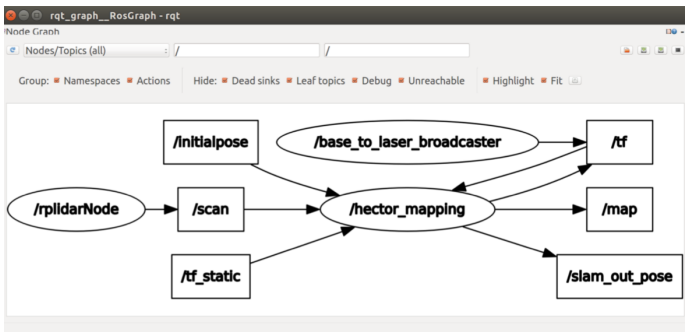

Figure 14. The rqt node graph

A visualization of the frame relationship of the system is shown in figure 13. The rqt node graph is present in figure 14. Figure 15 shows a view of the mapping process. Because the LiDAR data and mapping process are implemented on the laptop not Duckiebot Hut, it is necessary to control and follow the Duckiebot while mapping. Three final mapping results are analyzed and presented in Section III-C. If we want to get the translation and rotation in Quaternion of the base_link, we could get by running this command “rosrun tf tf_echo /map /base_link”.

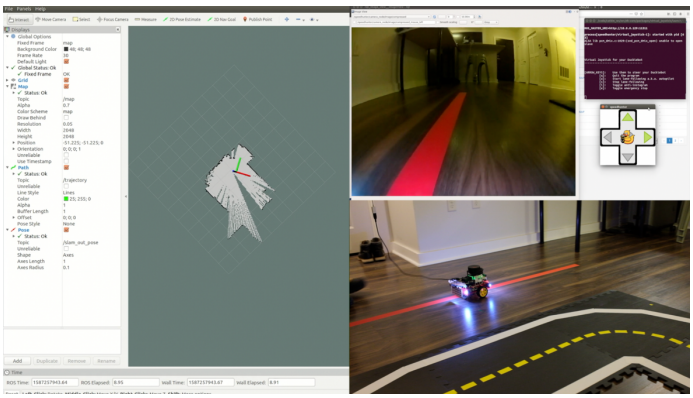

Figure 15. Screenshot of the mapping process

D. Map 2D Laser Scan Points to 2.5D Point Clouds (Shaoshu Xu, 4 hours)

Due to the limit information provided by 2D laser scanner, we proposed a method to map 2D laser scan points to 2.5D point clouds. Even though this is a pseudo-3d point cloud (this is also the reason why we call it 2.5D), it still provides us more information from 2D laser scanner. A visualization of the 2.5D point cloud in Rviz is shown in figure 16. It is more intuitionistic and clear to see the wall, person and even obstacles from the 2.5D point cloud, especially for indoor environment.

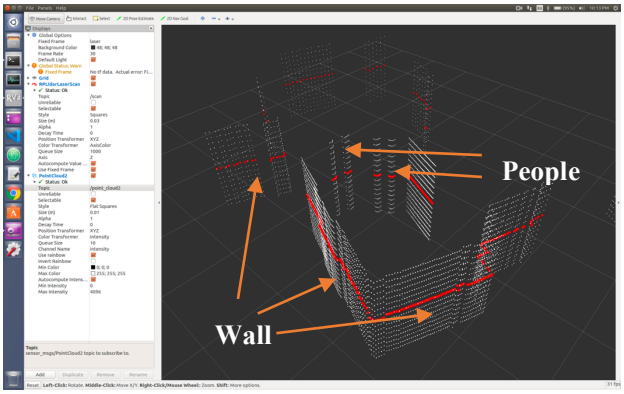

Figure 16. Visualization of the point cloud

E. Lane Following (Shaoshu Xu & Xingyu Lu, 20 hours)

The overall flow of the lane following process is shown in figure 17.

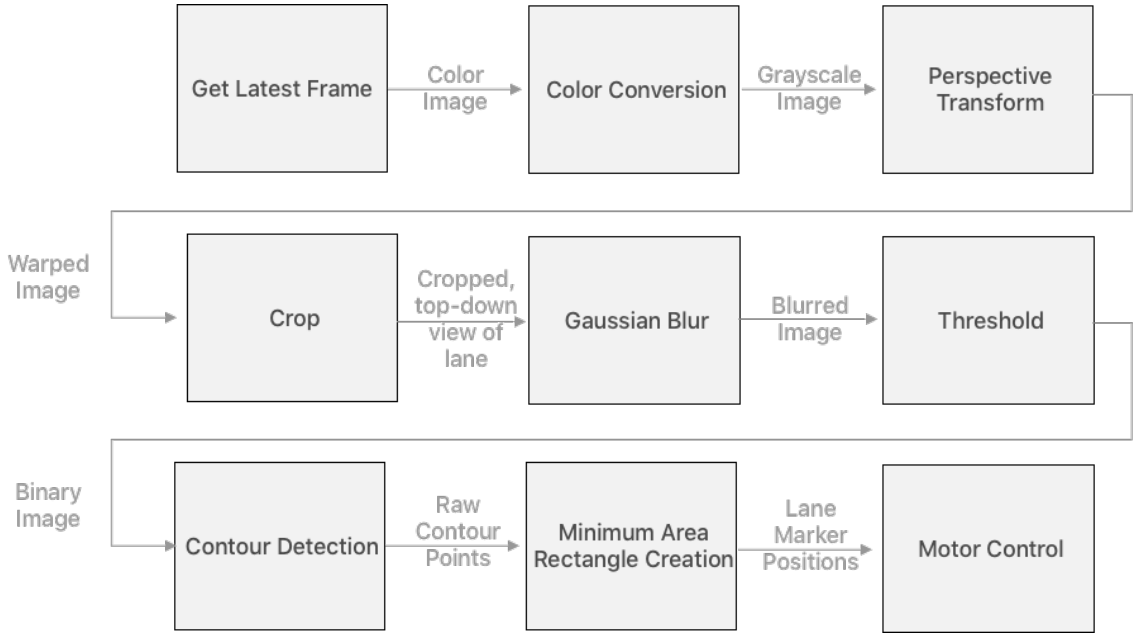

Figure 17. Flowchart of lane following

E-1: We followed the tutorial in Duckiebot operation manual and applied the lane following demo to our Duckiebot. The processes are as following:

1. At the beginning, we needed to check the connection between the robot and remote laptop by "ping duckiebot name.local".

2. Ran the lane following demo by:

```
xingye@xingye-ThinkPad-T590:~$ dts duckiebot demo --demo_name lane_following --d
uckiebot_name speedhunter --package_name duckietown_demos
```

This command started the "demo_lane_following" container. It is important to note that before running the robot we need to wait for a moment to let everything initialize.

3. Verified the "lane_filter_node" was working by running:

```
xingye@xingye-ThinkPad-T590:~$ dts start_gui_tools speedhunter
```

This node allowed us to talk with Duckiebot from our laptop through a container. Then we could run "rostopic list" to view all topics related with "lane_filter_node". By viewing all the related topics, we could make sure everything was prepared and normal to run.

4. Recalled image window to see what camera saw by running "rqt_image_view" under the container mentioned above. To see the lane detecting image, we could make "lane_filter_node" publishing to all the image topics by running:

```
$ rosparam set /speedhunter/line_detector_node/verbose true
```

This would set the ROS parameter verbose to “true”, so that “line_detector_node” would publish the images with lines. We could also view the images with lines by selecting “/speedhunter/line_detector/image_with_lines” in image window.

5. Recalled duckiebot keyboard controller and ran the demo.

```
xingye@xingye-ThinkPad-T590:~$ dts duckiebot keyboard_control speedhunter
```

Started the lane following by choosing the ‘a’ mode by keyboard, and the robot would drive autonomously in the Duckietown lane. In this demo, Intersections and red line would be neglected.

E-2: Problems we encountered and Solutions

1. Before adding the Lidar on the duckiebot, the robot worked well on lane following with proper speed and accurate control. However, after assembled the Lidar and related parts, the performance of lane following become unstable. Not only the speed got lower but also the turning adjustment was worse and less accurate than before. Initially, we were wondering about this situation. After carefully analysis we realized that the reason could be more weight from the Lidar and change of the mass center. So, we recalibrate the wheel again and changed the gain parameters to make the control system more suitable for our Duckiebot.

2. No matter with or without Lidar, the good quality of lane following is not always guaranteed. According to our many (about 20) experiments, we found two possible impact elements: light conditions of the environment and connection status. When the environment is dark, we could observe that the number of feature points on the lane was so small, even no feature points sometimes. Under this circumstances, the Duckiebot drives on the lane very uncertainly. After improving the light conditions, the number of feature points increased greatly and the Duckiebot drives on the tiles accurately.

3. Network connection quality also matters much. Under a good WiFi connection, the communication between Duckiebot and laptop would be smooth and without delay. So, the laptop can process image stream under higher frequency and make accurate control to the Duckiebot. On the contrary, if the connection condition is poor the delay would greatly affect the performance of the lane following. Because the Duckiebot cannot adjust to the changes on the road in time. In our experiments, there will always be 2-3 seconds’ delay so that the Duckiebot would go off the road.

4. Sometimes we could not move the robot via keyboard controller, or even could not connect with the Duckiebot. One possible reason could that we did not wait until the initialization is done. Also, when the WiFi connection was not good, it was also hard to connect the robot or talked to ROS master.

F. YOLO Object Detector (Xingyu Lu, 10 hours)

YOLO (You Only Look Once) algorithm is an Object Detection algorithm based on computer vision and deep learning methods. It utilizes a single convolutional neural network to detect target objects of each class in images, with class probabilities of each predicted detection.

During the implementation, each image will be split into S*S grids, which are helpful in fast generation of bounding boxes. For each grid m, bounding boxes would be generated. Then the image with its bounding boxes will be input to the convolutional network. For each of the bounding box, the convolutional network outputs a predicted class probability and the offset values of the bounding box. In practice, a threshold will be set as confidence that if the class probability is above the threshold, the corresponding bounding box will be selected and used to locate the object.

We utilized the basic framework of YOLO from GitHub [5] and followed the tutorial of YOLO object detection in Duckietown [6]. In the YOLO GitHub repository, a well selected dataset of Duckietown environment was provided, which collected from the Duckiebot's camera in different views. It covered many cases the robot would be in, so we utilized it directly as our training data. Details about each step will be discussed as follow:

F-1: Dataset Preparation

In the dataset, there are 420 raw images, which means that we need to preprocess them first. Fortunately, some useful tools are provided in the GitHub repository and we followed the tutorial as part of our training.

First, we needed to filter out images which are too blurry to be learned anything. We ran a python script named 'detect_blurry_img.py' which is able to detect blurry images with a threshold whose value is set as 200 in our implementation. After that, blurry images and non-blurry images were split into two directories. The command is shown below:

```
xingye@xingye-ThinkPad-T590:~/Classes/CS5335_RoboticsScience/Final_Project/darknet$ python3
 detect_blurry_img.py data_4_classes blurry_img non_blurry_img --thre
shold 200
Processing image data_4_classes/203_frame0152.jpg
Processing image data_4_classes/104_000097.jpg
Processing image data_4_classes/203_frame0502.jpg
Processing image data_4_classes/103_000305.jpg
Processing image data_4_classes/202_frame0813.jpg
Processing image data_4_classes/105_000347.jpg
```

Second, we labeled the non-blurry images with the help of tool 'label_data.py' and python libraray 'easygui', as shown below:

```
xingye@xingye-ThinkPad-T590:~/Classes/CS5335_RoboticsScience/Final_Project/darknet$ python3
 label_data.py non_blurry_img labels
```

For each image, we drew a rectangle of the object we wanted to label i.e. duck, sign. Here was a stupid little problem we met: according to the tool, in the input dialog box the classes start at value 1,

while in the label file it starts at 0. At the beginning we set the class should have value 1 as 0, so after the first labeling the one of classes had negative value, which is wield. Also, the label files did not pass the check. We checked the correctness of label files with the tool 'check_annotation.py'. The command is shown as below:

```
xingye@xingye-ThinkPad-T590:~/Classes/CS5335_RoboticsScience/Final_Project/darknet$ python3
 check_annotation.py path_of_image path_of label
```

We found out that it was mentioned in the tutorial but we did not notice before. We thought it won't be any problem but actually there was and we had to re-label all images which took extra time. Then, the label files were corrected and passed the check. For each image, there was a corresponding label file in which per line has the class label and coordinates of a selected object.

```
xingye@xingye-ThinkPad-T590:~/Classes/CS5335_RoboticsScience/Final_Project/darkn
et$ python3 create_datasets.py data_4_class duckiestuff 95
```

Third, since we have non-blurry images with labels and pose information, we separated images into three subsets: train, valid and test. We also used the commands below to record the index of images for convenient recall in training, which is shown below:

```
xingye@xingye-ThinkPad-T590:~/Classes/CS5335_RoboticsScience/Final_Project/darkn
et$ ls duckiestuff/trainset/*.jpg > duckiestuff/train.txt
xingye@xingye-ThinkPad-T590:~/Classes/CS5335_RoboticsScience/Final_Project/darkn
et$ ls duckiestuff/validset/*.jpg > duckiestuff/valid.txt
xingye@xingye-ThinkPad-T590:~/Classes/CS5335_RoboticsScience/Final_Project/darkn
et$
```

Here, we have finished the preparation of the dataset with proper labels for training YOLO.

F-2: Compile and problems we faced

Now we were ready to train the data. Before that, we needed to compile the program to get the executable training file. However, when we directly ran the 'make' command as what tutorial mentioned, it showed two errors : 'No nvcc' and 'No -lcuda ***'.

The first error is because we did not add the path of NVCC to environment PATH. To fix that, we used the command 'source nano /.etc/environment' to get into the environment file and add the nvcc local path to the PATH list. On the other hand, we could just modify the Makefile file by simply replacing 'NVCC' by the local path of NVCC.

The second error is because after installing cuda, the system did not build a link of corresponding libraries between "/usr/lib" and "/usr/local/cuda/lib64", so in compiling it could not find target library under "/usr/lib" path. To fix that, we built the link by create 'lib***.so' manually. After, the compiling worked well and executable file for training was here. But at the beginning, we had struggled for the actual reason why the second error existed for a long time, and could not directly find out the solution. Firstly, we tried to reinstall cuda because we thought it was an installation mistake for lacking a library.

However, it still didn't work. Then we searched for the meaning of some commands in Makefile file, and found out that if "-l**" was added successfully the link between cuda lib and usr lib should be built, that a file named "lib**.so" should be created under "/usr/lib". Then we examined the link file and solved this problem.

After that another problem arose. When we started training, it showed that 'The GPU driver is not suitable'. We realized that it could because our laptop doesn't equip GPU, and this could also be the reason why the cuda related libraries had no link to "/usr/lib" after installation. Therefore, we changed the parameters in Makefile to not use GUP.

F-3: Training and Testing

Now we used images and corresponding labels to train a convolutional network. The network we trained is a tiny YOLO network [7] for fast training. However, it was still extremely slow because the network is deep and need to train hundreds of images.

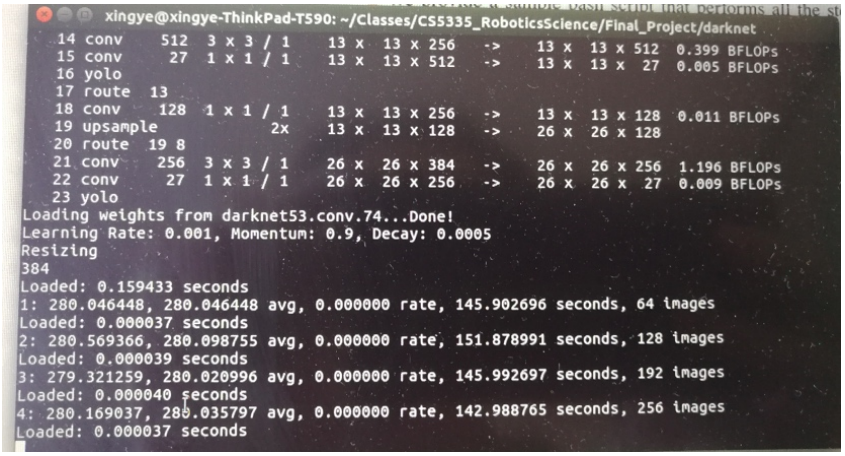

Figure 18. A screenshot of the training process

It took us several hours to finish training. After training, we tested the network with the test dataset and tried to apply it to our duckiebot. More details and results will be discussed in next section.

III. Results

All the results are presented in a more intuitionistic way in the Demo Video.

A. Duckiebot & Duckietown Hardware Assemble

The Duckiebot (with additional LiDAR) and Duckietown are properly assembled as shown in figure 2.

B. Duckiebot Preliminary Setting

The manual-verification of the calibration results is shown in figure 19.

Note the difference between the two types of rectification: In bgr_rectified the rectified frame coordinates are chosen so that the frame is filled entirely. Note the image is stretched - the April tags are not square. This is the rectification used in the lane localization pipeline. It doesn't matter that the image is stretched, because the homography learned will account for that deformation. In rectified_full_ratio_auto the image is not stretched. The camera matrix is preserved. This means that the aspect ratio is the same. In particular note the April tags are square. This rectification is needed if do something with April tags.

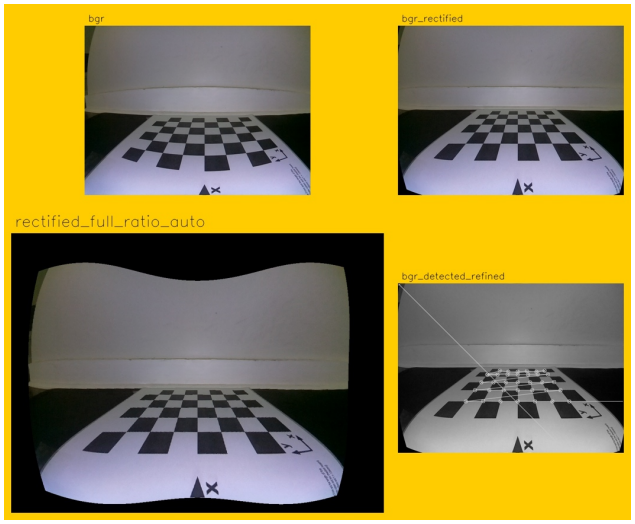

Figure 19. Manual-verification of the calibration result

Figure 20 shows the optional autonomic verification step result. We can see the calibration was correct and the robot localizes perfectly.

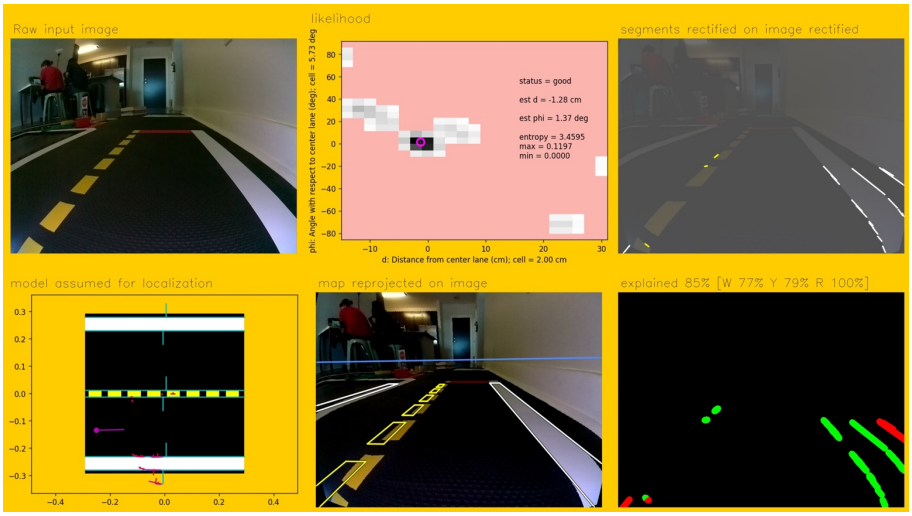

Figure 20. Optional autonomic verification step result

The goal of wheel calibration is calibrating the wheels of the Duckiebot such that it goes in a straight line when we command it to. The trim parameter and gain parameter are also store in the Duckiebot.

After setting the Duckiebot, we successfully connected with the Duckiebot with laptop, made the Duckiebot moving and was able to see what the Duckiebot sees.

C. Hector-SLAM

We tested the Hector-SLAM mapping approach in three different driving approach. The details will be discussed as follows:

C-1: In the first run, the Duckiebot was manually controlled on laptop and drove smoothly starting from point A (living room) to B (kitchen), as shown in figure 21. We could observe that the mapping result looks reasonable and matches the ground truth well.

Figure 21. Mapping result 1

C-2: For the second run, the Duckiebot was manually controlled on laptop and drove less smoothly starting from point B (kitchen) to A (living room). As shown in figure 22, we could notice there were some mismatches between the mapping. After careful observation of the camera view point, we concluded that it was because we made two fast turns while driving the Duckiebot, and the algorithm lost matching-features between two continuous time-steps. So, the mapping suddenly got huge errors and mismatches.

Figure 22. Mapping result 2

C-3: For the third run, the Duckiebot was manually controlled on laptop and drove unsteadily starting from point A (living room) to B (kitchen) then back to point A (living room), as shown in figure 23.

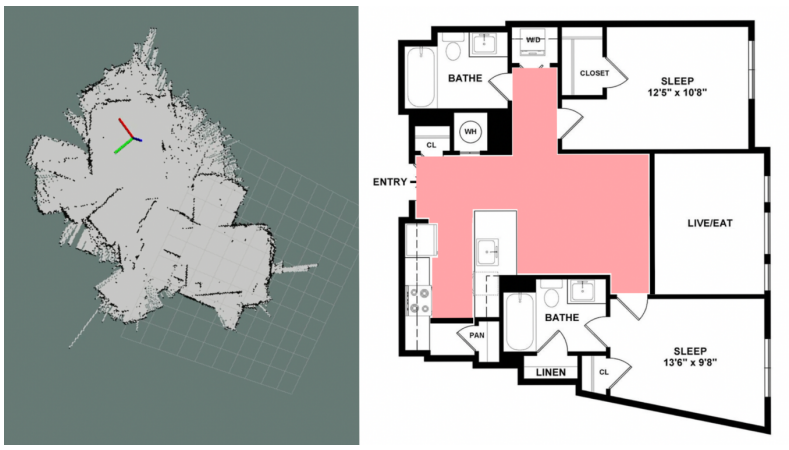

Figure 23. Mapping result 3

Given the fact that Hector-SLAM algorithm doesn't utilize odometry information. We needed to keep the Lidar moving smoothly and make sure no sudden changes of heading. Otherwise, under the condition of unsteadily moving, it would be hard to match the current leaser scan with previous map features. For more details about the mapping process and more intuitive demonstrations, please feel free to watch the demo video.

D. Map 2D Laser Scan Points to 2.5D Point Clouds

The following figure 24 shows an example 2.5D point cloud of my apartment. It is easier to gain an intuitionistic view of the environment.

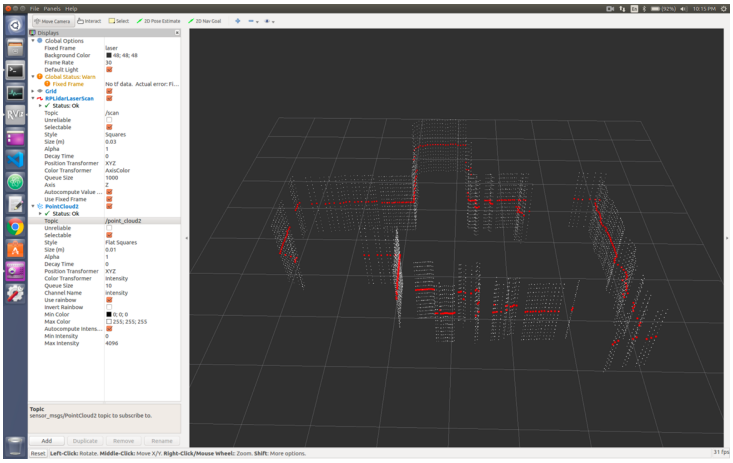

Figure 24. Example of 2.5D point clouds

E. Lane Following

For most cases, the performances are reasonable. However, due to the delay caused by unstable network connection, the Duckiebot might run out of the lanes. In navigation mode, the robot is able to detect the stop line at the crossings, but due to the delay it always brakes late and stops over the stop line. Some details about the different results will be discussed following:

E-1: Good environment light condition and Smooth WiFi connection

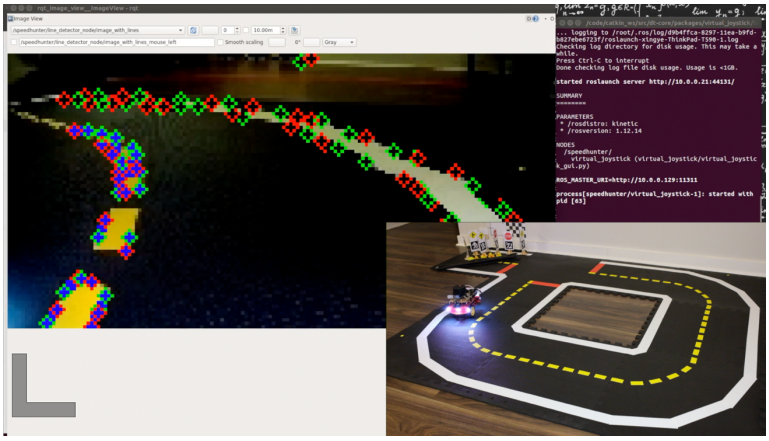

Figure 25. Lane following process 1

In our demo video, it is clear to see that the Duckiebot runs autonomously following the lanes. A screenshot of the demo video is shown in figure 25. The laptop receives image stream and send control signal back to Duckiebot without any connection delay. Also, we can notice that the good light condition helps a lot in detecting feature points. As the green and red squares shown in figure 24.

E-2: Connection delay

From the demo video and the following figure 26, we can see that although the feature points are detected successfully, the Duckiebot turns late at the left turn due to late control signal. One more thing needs to be noticed that the algorithm extract some feature points from the white wall. It is likely that the algorithm treat the white wall edge as the lane mistakenly. So, make the Duckiebot moves toward to the right side. After that, it is supervised to see that the Duckiebot turns left and go back to the right lane. The reason might be it detects the yellow line on the left side of the image, and tries to control the car back to the middle of the yellow and white lanes.

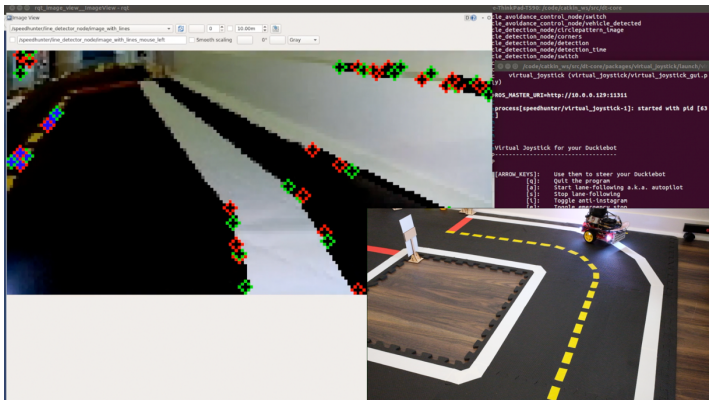

Figure 26. Lane following process 2

E-3: Dark light condition, Connection delay

This time the Duckiebot runs out of the lane and is not able to go back to the lane. Without sufficient lighting in surroundings, the algorithm has difficulty in extracted features. What's worse,

the delay connection make the Duckiebot is not able to adjust themselves back to the middle of the lanes, as shown in figure 27.

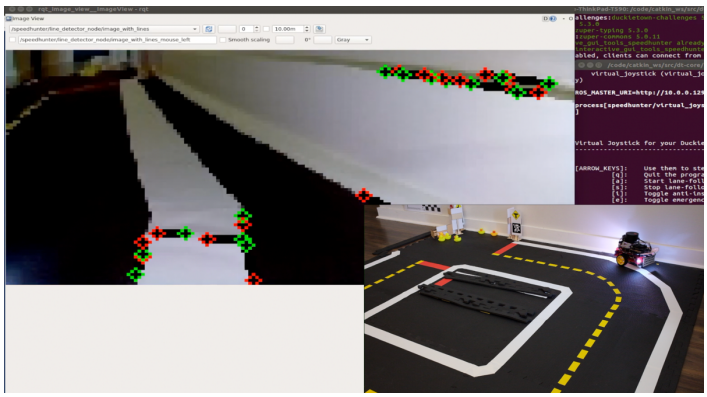

Figure 27. Lane following process 3

F. YOLO Object Detector

After training, we firstly test the network on test dataset. Some initial results are shown in figure 28:

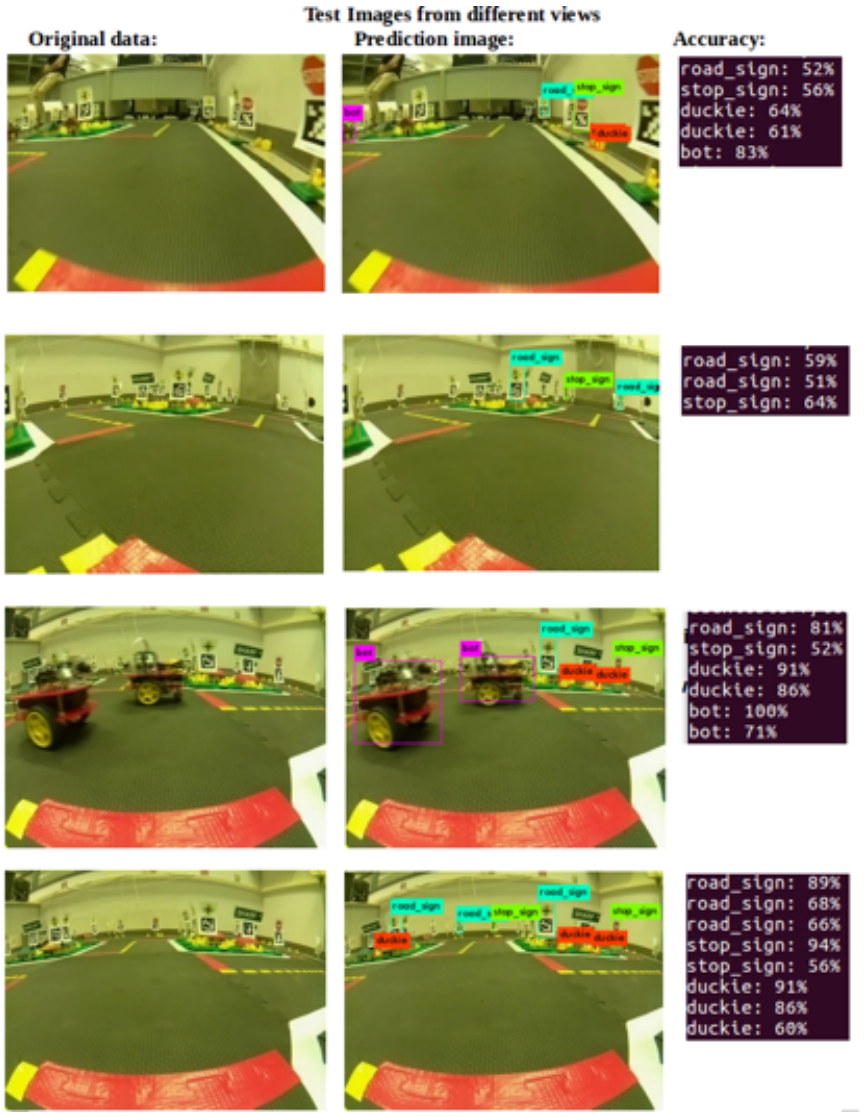

Figure 28. Initial results from YOLO Object Detector

Four test images are all at the crossroad but with different surroundings. For each detected object, it is located by a bounding box, and the classification probability (as shown on the right column) is computed based on manually created labels.

We notice that how blurry the object is in the image would affect the detection result. If the object is too far from the camera, the small objects that are of low resolution would not be detected or be detected with very low accuracy. As for the moving objects, like the left Duckiebot in third test image, we can notice that the blurry caused by moving would also have a negative influence on the accuracy. After further comparison, we can notice that camera distortion also makes influences. Objects at the edge of image are more distorted than those around the center. As shown in the first two images, it is clear to see that objects around the edge are not be detected.

Then we test the trained model on our images. Notice that some images are collected using our Duckiebot camera and others are downloaded from online. Some potential factors we found via several contrast tests are likely to influence the detector:

1. Environment light conditions: Some images in our test set were taken in dark environment, while others are from bright environment. The comparisons are shown in the figure 29 and 30.

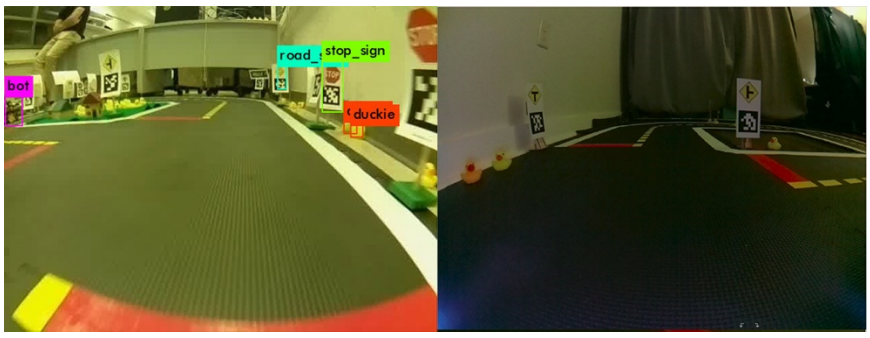

Figure 29. A comparison of our data with the training data

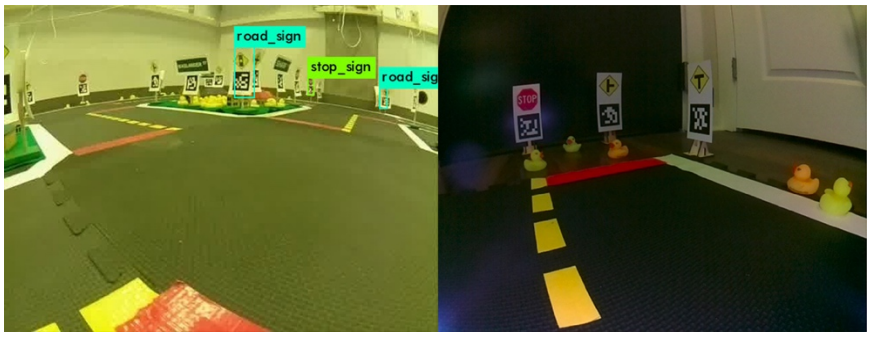

Figure 30. A comparison of our data with the training data

2. Image size: Initially we test the images of which size are both 640*480. However, the test accuracy is pretty low on our test set and most of the objects are not detected. Then we realize that out images are of different size, which could be a trouble for this model. However, after changing the size of test images the good result is still not guaranteed neither.

3. Background of images: We also test some images with black or white background and there is still no any prediction result. So, it is hard to tell whether the different background make any influences on the result.

4. Diversity of training dataset: The variety of training data is another critical factor. Perhaps our training dataset is not variable enough so that the model only works on identically distributed data set. We could add more images with different environment conditions to the training dataset to improve the robustness of this model.

5. Complexity of network structure: We could also modify the network structure to train a better model which could extract more complex and deep features.

IV. Resources

A. Duckiebot Assemble and Setting

This part is mainly build on Duckietown operation manual and Duckietown official GitHub homepage:
[https://docs.duckietown.org/DT19/opmanual_duckiebot/out/demo_sysidII.html]
[https://github.com/duckietown]
The RPLidar setting is based on the:
rplidar roswiki [http://wiki.ros.org/rplidar]
rplidar HomePage [http://www.slamtec.com/en/Lidar]
rplidar Tutorial [https://github.com/robopeak/rplidar_ros/wiki]

B. Hector-SLAM Mapping

Hector-SLAM is proposed in this paper: A Flexible and Scalable SLAM System with Full 3D Motion Estimation
[http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.302.2579&rep=rep1&type=pdf]
Hector-SLAM Mapping is implemented based on the ROS Wiki documentation:
[http://wiki.ros.org/hector_slam] and [http://wiki.ros.org/hector_mapping]

C. Map 2D Laser Scan Points to 2.5D Point Clouds

The algorithm is changed based on RPLidar original node.cpp code.
rplidar roswiki [http://wiki.ros.org/rplidar]

D. Lane Following

Duckietown operation manual:

[https://docs.duckietown.org/DT19/opmanual_duckiebot/out/demo_lane_following.html]

E. YOLO Obstacle Detector

Duckietown operation manual:

[https://docs.duckietown.org/DT19/exercises/out/exercise_object_detector.html]

[https://github.com/marquezo/darknet]

YOLO official website:

[https://pjreddie.com/darknet/yolo/]

F. Other references

[1] https://en.wikipedia.org/wiki/Self-driving_car

[2] https://github.com/marquezo/darknet

[3] http://www.slamtec.com/en/Lidar/A1

[4] http://wiki.ros.org/hector_mapping

[5] https://github.com/marquezo/darknet.git

[6] https://docs.duckietown.org/DT19/exercises/out/exercise_object_detector.html

[7] https://pjreddie.com/media/files/darknet53.conv.74

V. Reflection

For the future directions: Firstly, we hope to dig more deeply and understand internal code of Duckiebot, and run our own algorithms on it. For instance, integrating the SLAM algorithm into the Duckiebot and making the Duckiebot move autonomously, writing an YOLO object detector container and deploying on the Duckiebot. Secondly, last semester we implemented Proximal Policy Optimization deep reinforcement learning algorithms in CarRacing Game environment to train an agent driving on the track. So, we want to implement a similar reinforcement learning based decision-making algorithm on Duckiebot to make it autonomously drive in Duckietown. Third, in the future we are excited to see multiple Duckiebots safely navigating in a big Duckietown system. That would be closer to the real environment but also more complex. Our project is almost involved in all the subtasks of perception and localization part. For the sensor fusion part, it would be better to combine IMU data with Lidar point cloud while building map.

We think Duckietown is a very good platform for us to step into the autonomous driving field. We can integrate whatever parts we wanted with the Duckiebot and have a high-level understanding of the whole system. However, not like Turtlebot, the resource of Duckietown is very limited and the official

operation manual still needs to be perfected. So, we have to speed a lot of time in setting up the Duckiebot and going through a lot of detours inevitably. We really learned a lot during this project and enjoyed the process of solving all kinds of problems. In the future, we still want to focus on autonomous mobile robot field and expect to tackle more complex problems.

Last, thanks professor again for providing us the Duckiebot and all the related parts. This is a rare experience!

VI. Others (24 hour):

Video Shooting and Making (Shaoshu Xu & Xingyu Lu, 5 hour)

Write Report (Shaoshu Xu & Xingyu Lu, 20 hour)