A Charging Robot with Auto-SLAM, Car Detection, and Navigation

Team10: Xupeng Zhu, Zimou Gao, Jiahao Wu

***Abstract*— The Turtlebot does SLAM to understand the environment. During the mapping process, it can detect cars in the parking lot then identify them with specific features, such as license plate or color of the car. After finishing SLAM and identification process, the Turtlebot returns to its original point.**

When we send a command with a specific car, Turtlebot will do path planning and find its way to the area of that car. At the same time, it have an ability of obstacle avoidance to prevent itself from hitting the person walking around.

I. Introduction

A. Electric car charging problem in parking-lot

We have initiated a project to solve the electric car charger problem. The electric car market is growing substantially while the number of chargers in parking-lot is limited and thus it is unable to address the charging demand. Hence, car owners always line up to use the facilities. Worse still, electric cars usually take several hours to be fully charged and some of the car owners cannot get back on time to pick up their vehicles, which creates inconveniences for other users and monetary losses for the charging service providers.

The current industry approach to solve this is either through hiring people to maintain the charging valets and remove devices when charging is done, or, through charging the car owner a much higher utility fee for additional non-charging hours.

B. Our goal

What we proposed instead, is to develop an robotic valet to fulfill automatic charging service solution – by utilizing robots, deep learning, and computer vision.

This robotic valet is a mobile wheel robot with robotic arm mounted on it. In electric car charging scenario, our robot will do automatic SLAM and navigation in the parking-lot to have its map built. During this exploration process, our robot will detect cars and locate them in its map. After finishing building map and locating all the cars parking in the lot, robotic valet can go to the specific car area and do charging manipulation with its robotic arm if specific commands are sent by operator.

In this way it helps the service providers to save huge costs of maintaining the charging valets, while at the same time providing a more flexible option for electric car owners and helping them save charging bills.

C. Overview of this report

In Background section, we state the hardware and software we are using. In Methods section, we discuss the methodology to achieve our goal. In Results section, we present our simulation and experimental results. In Discussion section, we analysis the . Finally, in Conclusion section we present a conclusion and future work that needs to be done.

II. Background

A. Mobile robot

We use Turtlebot3 as mobile robot in our project. TurtleBot3 is a small, affordable, programmable, ROS-based mobile robot for use in education, research, hobby, and product prototyping. TurtleBot3 is evolved with cost-effective and small-sized SBC that is suitable for robust embedded system, 360 degree distance sensor and 3D printing technology.

B. Robotic arm

We use UR5e as robotic arm in our project. The UR5e is a lightweight, adaptable collaborative industrial robot that tackles medium-duty applications with ultimate flexibility. The UR5e is designed for seamless integration into a wide range of applications.

C. Physical Simulation

We use Gazebo to simulate the physical environment for robotic valet and electric cars. Gazebo offers the ability to accurately and efficiently simulate populations of robots in complex indoor and outdoor environments. It provides a robust physics engine, high-quality graphics, and convenient programmatic and graphical interfaces.

D. Manipulation Simulation

We use MoveIt to simulate our robotic arm manipulation. With MoveIt's help, robotic arm can achieve the following tasks:

Generate high-degree of freedom trajectories through cluttered environments and avoid local minimums.

Analyze and interact with the environment with grasp generation.

Solve for joint positions for a given pose, even in over-actuated arms.

Connect to depth sensors and point clouds with Octomaps.

Avoid obstacles using geometric primitives, meshes, or point cloud data.

III. Method

In this section, the full pipeline of the charging bot will be discussed in detail. The flow chart is shown in figure. 1. We simplify the whole process into several parts, i.e., map building, car detection and recognition, way point recording, way point navigation, and command navigation.

When the charging bot launches, the basic packages and services (for example,turtlebot initialization, visualization,

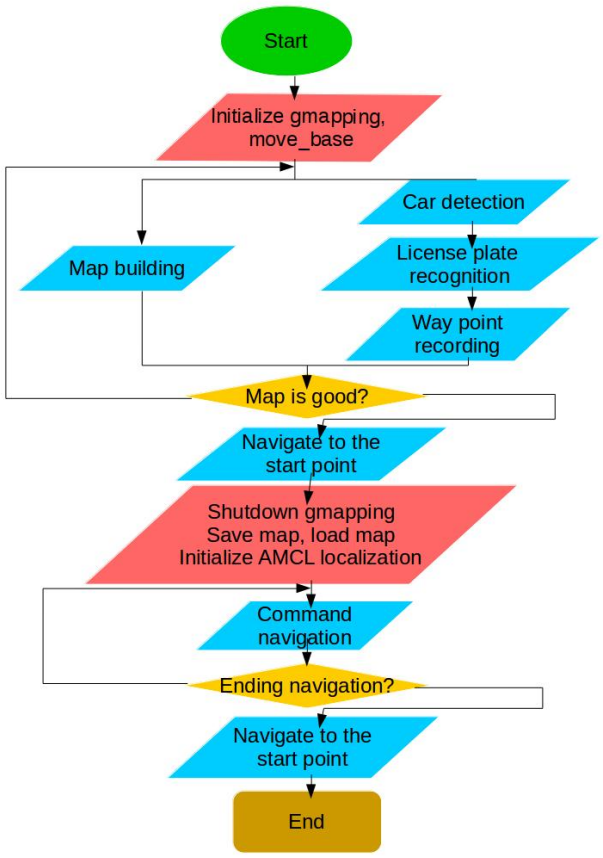

Fig. 1. The flow chart of the whole process of the service of the chargingbot. All the blue parts in the figure were implemented as functions.

gmapping, and move base) will be initialized. Later, two processes, map building and car recognition processes will ruining in parallel. In these processes was designed to explore any unknown environment and to create a map as well as record the pose of cars at same time. After created a map and obtained the pose of cars, gmapping service will be shutdown and AMCL will be initialize for localization. In the terminal, all recognized cars will be printed and user can select a car for the charging service. When the charging service finished, user can choose another car for the service, or end the service. If the service ended, the turtlebot will navigate to the start point waiting for the next service.

A. Simulation world

Compares to the real-world experiment, the simulation has the advantages on fast developing, avoiding physical equipment failure, and easy to initialize. Thus, our mainly focus is develop our robot system on simulation. For the simulation environment, we need import a car model, built the garage, and for the performance of the SLAM algorithm, we set up some objects in the empty space as landmarks.

B. Map building

The map building was achieved by using gmapping package in the turtlebot. Instead of explore the environment manually, we achieve automatic exploration by using package explore_lite.

In order to run explore_lite (IEEEexample:multirobotsystem2016-1), a map server and a move_base server needs to be configured. As shown in figure. 2 the package explore_lite subscribe to the slam node, which will be initialized be gammaping and send navigation command to move_base service. The move_base service was configured under the navigation stack.

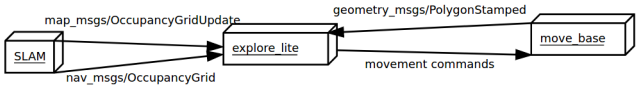

Fig. 2. The package explore_lite reliance.

Explore lite package is based on frontier exploration. During exploration, frontiers will be generated according to the Lidar reading. The package calculates the cost for each way-point based on the frontiers. The package will sent the cosiest way point to the move_base service. Then the turtlebot was navigate to the cosiest way point.

The gmapping package provides laser-based SLAM (Simultaneous Localization and Mapping), as a ROS node called slam-gmapping. Using slam-gmapping, you can create a 2D occupancy grid map (like a building floorplan) from laser and pose data collected by a mobile robot.

GMapping solves the Simultaneous Localization and Mapping (SLAM) problem. Unlike, say Karto, it employs a Particle Filter (PF), which is a technique for model-based estimation. In SLAM, we are estimating two things: the map and the robot's pose within this map. You can kind of think of each particle in the PF as a candidate solution to the problem. Together, the set of particles approximates the true probability distribution (see Importance Sampling). To clarify, I'm talking about the probability of the map and the robot's pose given the control inputs (e.g. motor encoder counts) and sensor readings (e.g. LiDAR). In addition, there's a motion model and a sensor model involved in the calculation of the probability distribution.

This approach uses a particle filter in which each particle carries an individual map of the environment. Accordingly, a key question is how to reduce the number of particles. We present adaptive techniques to reduce the number of particles in a Rao- Blackwellized particle filter for learning grid maps. We propose an approach to compute an accurate proposal distribution taking into account not only the movement of the robot but also the most recent observation. This drastically decrease the uncertainty about the robot's pose in the prediction step of the filter. Furthermore, we apply an approach to selectively carry out re-sampling operations which seriously reduces the problem of particle depletion.

C. Car detection and identification

1) Use Deep Learning for Object Detection: We choose SSD(single shot multibox detector) (IEEEexample:multiboxdetector-1) to detect cars and

license plates. SSD has two components: a backbone model (VGG16 (IEEEexample:simonyan15-1)) and SSD head. Backbone model usually is a pre-trained image classification network as a feature extractor. This is typically a network like ResNet trained on ImageNet from which the final fully connected classification layer has been removed. We are thus left with a deep neural network that is able to extract semantic meaning from the input image while preserving the spatial structure of the image albeit at a lower resolution.

2) Training Dataset for Deep Learning: We choose VOC as SSD training dataset because VOC has a large amount of high-quality car pictures. Before putting images into the training model, we will resize images into 600*400 size as pre-processing.

3) Car localization: After detecting the car with SSD, the detection code returns the coordination data and license plate information with respect to the car. We fuse the coordination data with depth camera to locate the detected car and tag it with its license plate number.

D. Way point recording

Once a car is detected and recognized as described previously, the pose of the turtlebot will be recorded in YAML format. The YAML file will be used for the charging service in the later section. The pose was obtained by subscribe to the topic "/amcl_pose". The pose information will be saved as a dictionary which includes the license plate, position, and quaternion.

Way point navigation is a robotic algorithm for motion planning in the case of known robotic positions. When we already know the real-time position of the machine and the attitude, we can use this fuzzy control to control the robot movement.

The main idea of fuzzy control is to translate the information and knowledge that is used by human pilots into fuzzy rules, which can be used by fuzzy controllers. It has strong robustness, good adaptability and fault tolerance. But the weakness is low steady state accuracy due to lack of integral link. It is implemented in PID position controller to tune the parameters of PID gains with the inputs are error and error of rate. There inputs are obtained from evaluating the feedback of GPS sensor and the reference position given by the user. The system has three output variables which are alpha , beta and gamma .In first stage of fuzzification, the inputs are divided into three subsets of fuzzy with trapezoidal membership function between the range from -1 until 1. For the outputs, trapezoidal membership function with two subsets of fuzzy is also used in alpha and beta between the range from 0 until 1 while triangle membership function with four subsets of fuzzy used in gamma between the range from 1 until 6. The Fuzzy Inference System (FIS) has 9 rules. The method of Mamdani with Min-Max was chosen for conducting the combination of fuzzy.

E. Command navigation

After all the position of the cars were recorded and the map was generated, the robot will move to the start point and print all the cars it observed. The print was achieved by reading all the position of the cars in the YAML file. When user chooses a car for the charging service, the ROS node /charging_bot will read the position, quaternion of the car and send it to the service move_base.

The present location of the robot will be the starting point, and the goal point i.e., the location of the car, will be given by III.5 vision. The path planning can be conducted by applying the following algorithms in the 2D occupation map obtained by part III.2.

A* is a graph traverse algorithm that is complete and optimal. A* utilizes the heuristic function h(n) plus the cost function: f(n) = g(n) + h(n) as priority to traverse the graph. However, A* has high demand on storage.

D* is Dynamic A* in short. Initially D* algorithm assumes the robot workspace is empty and plan the initial path. Then D* will include obstacles in the map and replan the path iteratively. Ultimately, D* can find a path to the goal, if it exists. D* is more efficient than apply A* iteratively to deal with dynamic map.

Rapidly-exploring Random Tree(RRT) is a path search algorithm used in search space (in contrast to A* and D* in a graph). It is suitable for high dimension space path search. Literally, RRT grow a tree rooted at the start point by randomly sampling point and connect the point to a branch if it is obstacle free.

F. Obstacle avoidance

Obstacle avoidance was realized by using the package navigation_stack/move_base. The sercive move_base will generate a global path according the global map and plan a local path according to the sensor (Lidar) readings.

When the robot is driving, it will encounter obstacles of different characteristics. In order to accomplish our goal, the robot needs to avoid obstacles autonomously and continue to remember before the path planning. After building a model of the environment and generating a pre-computed path according to all the methods described in this and the preceding chapter, the robotstarts following the path while it is capturing data from the Kinect sensor. Thisdata is continuously converted to a point cloud. If the path is being intersected by any circle with a center of point cloud and the radius of the robot, and the distance between the current pose and the intersection point is less than some threshold,then robot stops, adds the current point cloud to the world and performs all the steps to build a new model of the environment. The previous path is retained and if during re-planning the obstacle which blocked the path is removed (for example a dynamic obstacle) then the robot will continue the original path. In other words, the dynamic changes in the environment which do not have any influence on the pre-computed path either they cannot be perceived by the Kinect field of view or they don't interfere with the path.

IV. Results

A. SIMULATION RESULT

1) Auto-SLAM: The auto-SLAM is succeed in the GAZEBO. Explore lite package is based on frontier exploration as shown in figure. 3. During exploration, frontiers (blue points in figure. 3) would be generated according to the Lidar reading. The package calculates the cost for each way-point based on the frontiers, as shown as the green balls in figure. 3. The package would sent the cosiest way point to the move_base service. Then the turtlebot was navigate to the cosiest way point.

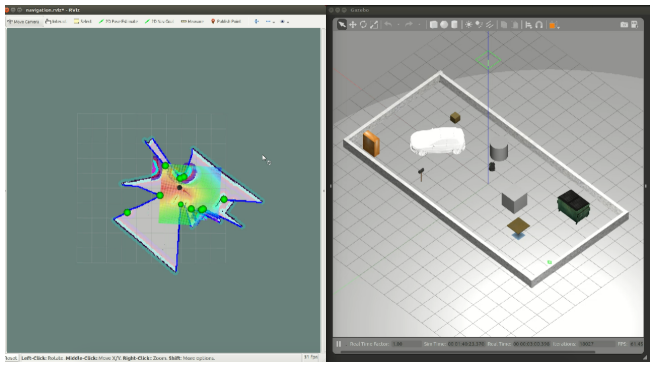

Fig. 3. The visualization of the auot-SLAM process. In the left shows the frintiers and way-points generated by the explor_lite package, and the right shows the simulation environment for the charging robot.

The auto-SLAM process is vary slow. From observation, we find that in most of the time the turtlebot keeping rotating during moving to a new way-point. At first we assume this may due to the poorly-tuned controller or too large inflation of the obstacles. After tuning these parameters in the navigation stack, the turtlebot is still rotating. Later, we found two reason lead to this problem. The first is due to turtlebot2 in simulation used x-box kinnect as Lidar that has limited view-angle, the turtlebot needs using self-rotation to realize a 360^o Lidar. Another reason is when the turtlebot think it is trapped on obstacles in the belief map, it will execute self-rotation to obtain true obstacle information to clear out the belief map. So this self-rotation phenomenon cannot be solved unless use a 360^o Lidar.

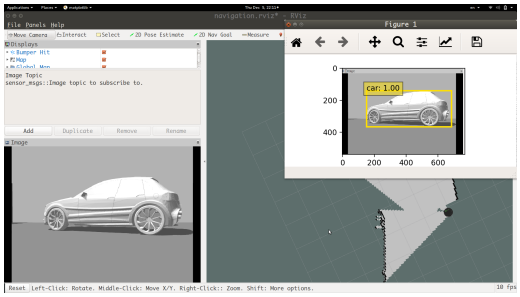

Fig. 4. Detect car in Gazebo world

2) Car detection and recognition: In this section, we present the result of object detection with deep learning.

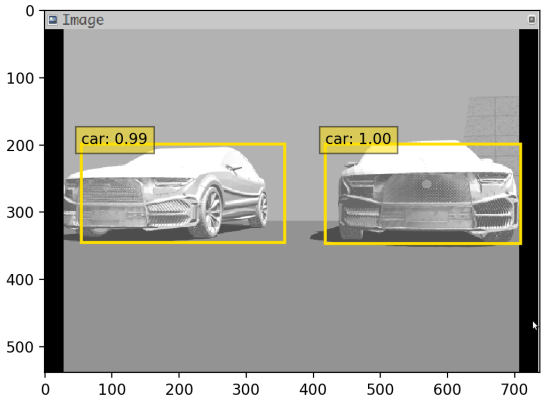

Fig. 5. Detect cars from the view of robot

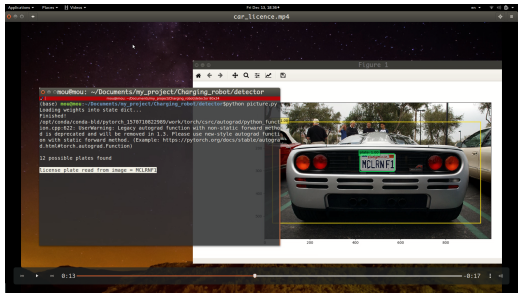

Fig. 6. Identify car with its license plate number

The car detection script will subscribe "raspicam-node/image" topic send from turtlebot3, transferring the data to OpenCV format through "cv-bridge". Then it will detect cars and recognize the license plate number during mapping process as shown in Fig 4, Fig 5 and Fig 6.

After detection and recognition, our script will publish an new "/raspicam-node/image/image-detect" topic with coordinates and license data, which will be recorded as new Way-points.

3) Way point navigation: The way point navigation was realized by the ROS node /charging_bot reading the position, quaternion of the car and send it to the service move_base. During this process, the sercive move_base was generated a global path according the global map and plan a local path according to the sensor (Lidar) readings as shown in figure. 7. In the figure. 7, the global map was shown as black points, the purple shadows are the inflation of the global map. The global path planning was shown as green curve. In the other hand, the local map was show in the colored square. The cost to each point in the local cost map was color-coded according to the cost by the Dijkstra algorithm (IEEEexample:dijkstra1959note-1). Moreover, the local path planning was shown as red curve in figure. 7.

4) Robotic arm manipulation: In this section, we present the result of grasping a can with UR5e robotic arm.

To achieve our goal of doing arm manipulation near the car, we want the robotic arm can detect and locate specific objects and do manipulation on them. Results are shown in the follow figures:

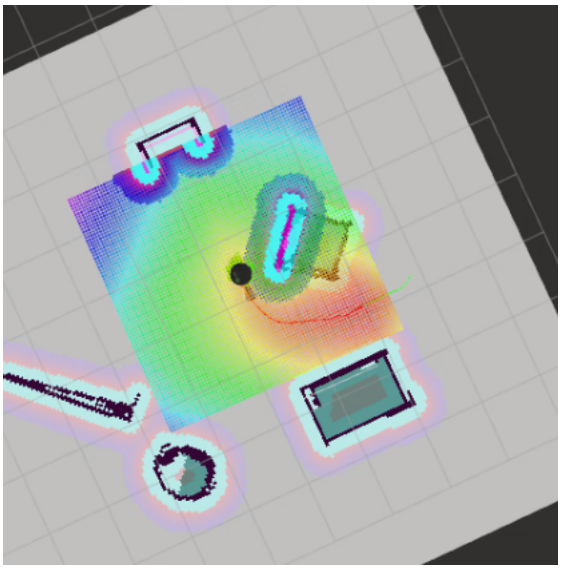

Fig. 7. Way point navigation experiment in simulation. The global map was shown as black points, the purple shadows are the inflation of the global map. The global path planning was shown as green curve. The local map was show in the colored square. The cost to each point in the local cost map was color-coded. The local path planning was shown as red curve in figure.

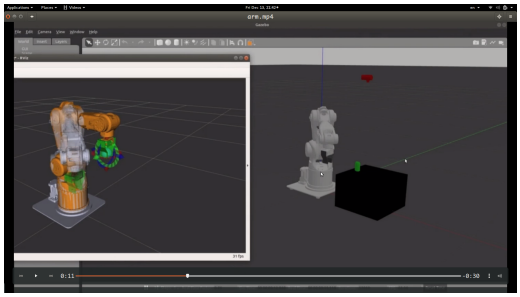

Fig. 8. Camera is set above UR5e, UR5e move aside to avoid blocking the view of camera

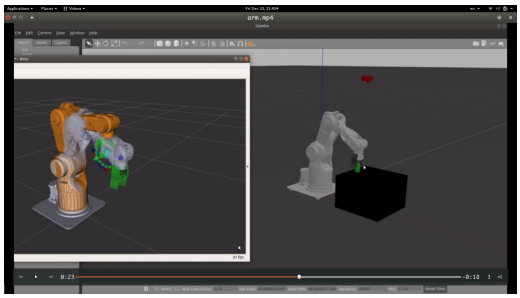

Fig. 9. UR5e receive the location of the can detected by camera

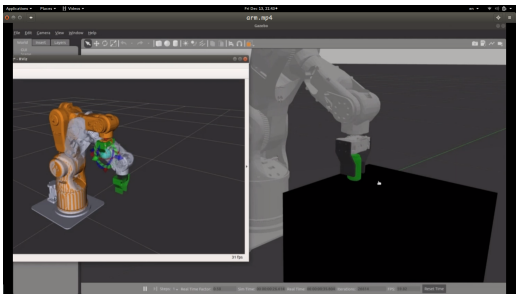

Fig. 10. UR5e grasp the can and move it to one side

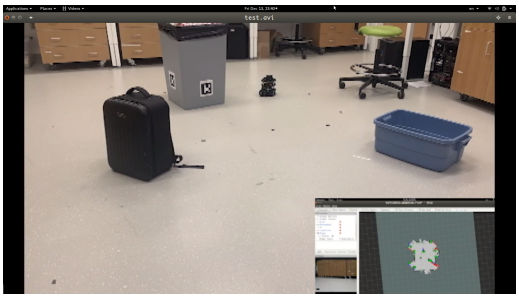

Fig. 11. Turtlebot3 is doing Auto-SLAM in the lab

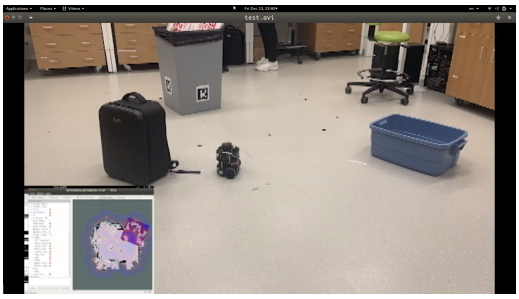

Fig. 12. Turtlebot3 is doing Way point navigation

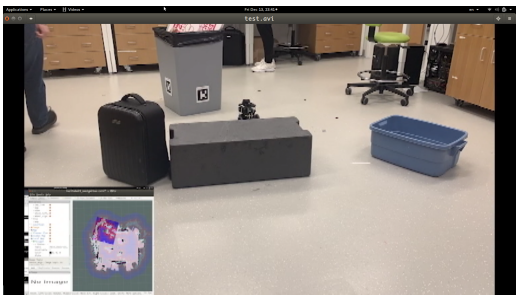

Fig. 13. Turtlebot3 fail in dynamic navigation when being blocked by huge obstacle

B. REAL-World EXPERIMENTS RESULT

1) Auto-SLAM: We tried Auto-SLAM in the real world and succeed. But the process is vary slow. And we found that the hardware performance matters a lot. Once we used a turtlebot3 of bad quality, the lidar kept drifting and Auto-SLAM failed.

2) Way point navigation: We also tried Way point navigation in the lab with the map built by Auto-SLAM. As shown in Fig12, the turtlebot can make path planning and go to the goal point successfully.

3) Navigate in a dynamic world: After finishing all the experiments above, we involved dynamic obstacles during Way point navigation process to see whether our robot can handle a dynamic world properly. At the first time, Turtlebot stopped and failed to get to the goal point when we blocked it with huge obstacle, shown as Fig13. Then we tried with

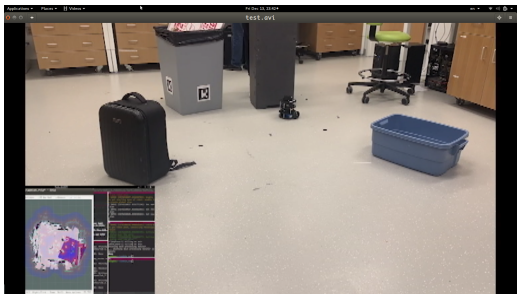

Fig. 14. Turtlebot3 succeed in dynamic navigation when being blocked by middle-size obstacle

middle-size obstacles, this time, Turtlebot stopped for few seconds in front of the obstacle but quickly planed a new path to avoid it. As shown in Fig14, Turtlebot3 reached the goal successfully.

V. Discussion

A. Car detection and recognition

In this section, we would like to discuss some problems have already been solved or to be solved

1) ROS Kinetic and Python3 Environment conflicts: ROS Kinetic is run in Python2 while our car detection script is run in python3. We figured out 2 ways to run both ROS Kinetic and Python3 script in the same time:

First solution: In Python3 script, we can call "subprocess.check-output()" to run ROS Kinetic as a sub-process with bash file. This is an easier solution but the drawback is the sub-process cannot communicate with main process until the end.

Second solution: We run ROS Kinetic and Python3 script separately while communicating each other through lcm or zmq.

2) CV2 version incompatible in ROS Kinetic and Anaconda: Due to the incompatibility of CV2 version in ROS Kinetic and Anaconda, we will encounter errors when we run CV2 with ROS path and Anaconda path. To solve this conflict, we just need to erase ROS path in our CV2 code by using: "sys.path.remove('/opt/ros/kinetic/lib/python2.7/dist-packages')"

VI. CONCLUSIONS

In this project, we built all the function of the autonomous charging robot. They are: auto-SLAM, car detection and recognition, way point recording, way point navigation, and path planning of a robot manipulator. Simulation in turtlebot2 in GAZEBO environment and physical experiment in turtlebot3 in a Lab environment validated our auto-SLAM and way point navigation. Moreover, a SSD deep neural network was trained to detect and recognize cars. The functionality of the SSD was verified by video string car detection. Further more, the experiment on the manipulator in the GAZEBO environment demonstrate the plausibility of mounting the manipulator and control it without blocking the camera.

VII. Assignment

Xupeng Zhu: Finished coding for auto-SLAM, way point recording, navigation to a point, command navigation, and take a photo. Configured move_base for auto-SLAM, adjusted parameters in navigation_stack. Wrote launch files for map building, command navigation. Wrote README and corresponding part of the report with Jiahao. Finished the physical experiment with Zimou. Prepared presentation with group members.

Zimou Gao: Finished coding for "Car detection and recognition"; Trained deep learning computer vision model; Built simulation car and robotic arm models with Jiahao; Finished "Robotic arm manipulation" part; Finished the physical experiment with Xupeng; Wrote document for Turtlebot2 and Turtlebot3; Prepared presentation with group members.

Jiahao Wu: Learn how to use the whole system about the turtlebot and the ROS system.

References

IEEEexample:multirobotsystem2016J. Hörner, "Map-merging for multi-robot system," Prague, 2016. [Online]. Available: https://is.cuni.cz/webapps/zzp/detail/174125/

IEEEexample:multiboxdetectorW. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg, "Ssd: Single shot multibox detector," 2016, to appear. [Online]. Available: http://arxiv.org/abs/1512.02325

IEEEexample:simonyan15K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *International Conference on Learning Representations*, 2015.

IEEEexample:dijkstra1959noteE. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische mathematik*, vol. 1, no. 1, pp. 269–271, 1959.