Autonomous Parking Spot Locator and Park System for Empty Parking Lot

Kazumi Malhan
Electrical and Computer Engineering Department
School of Engineering and Computer Science
Oakland University, Rochester, MI
E-mail: kmalhan@oakland.edu

***Abstract*— The project presents a system that autonomously locates a parking spot in an empty parking lot and parks the vehicle. The coding is done using C++ with ROS and OpenCV libraries, and tested with Gazebo and Rviz based simulation environment. In this paper, a navigation node is implemented to support an Ackerman drive vehicle. A camera based parking spot detection algorithm was developed and implemented. Also, camera image feedback was used to improve the accuracy of the parking control. The paper discusses methodology, implementation, development process, results, and possible improvements.**

I. INTRODUCTION

The goal of this project is to create a system that can autonomously find an empty parking slot in the parking lot, and park the car. Since most of the implementation of the concept is done for a differential drive robot in the class, an Ackermann drive robot (carlike vehicle) is chosen for the project.

The project is developed in divide-and-conquer fashion. Each component is developed and tested individually, followed by the system integration.

The author could grasp how to develop the navigation system as a similar system was developed in class for a differential drive robot. Also, professor had provided the fundamental of how to interact with OpenCV library and cv_bridge package available in ROS. However, the algorithm for parking spot detection and parking maneuver needed more extensive research to understand a useful approach.

All testing is done in simulation-based environment using Gazebo. Navigation and parking spot detection algorithms are implemented using C++ with ROS and OpenCV libraries.

The motivation behind this particular project was to demonstrate a deep understanding of ROS concepts and apply it to a real world project, to show the effectiveness of Rviz visualization tool for shorter development and debugging time, and get exposure to image processing and OpenCV library.

This report explains the vehicle configuration, design process and implementation of each component, the experimental setup, analysis of results, possible improvements, and future development plan.

II. METHODOLOGY

The project is divided into three major components: navigation, parking spot detection, and parking maneuver.

A. *Vehicle Configuration*

Audi R8 (Audibot) is used as the test vehicle throughout the project. The vehicle model itself is provided by Prof. Radovnikovich in the course repository. The vehicle is equipped with the sensors described in **Table 1** below.

Sensor	Purpose
GPS (1 Hz)	Localization
LIDAR	Costmap Generation
Front Camera	Parking Sport Detection
Back Camera	Parking Maneuver feedback

Table 1: List of sensors equipped on audibot

GPS is placed in top of the vehicle to provide the exact location of the vehicle. To make the system more realistic, the GPS operates at 1Hz. LIDAR is placed on top of the vehicle to provide object information for costmap generation. The front camera, which is actually placed at front right side of the vehicle, is used to detect the parking spot. Due to computational power limitations, the left side camera is not installed in this project. However, it can be installed on a real system with similar algorithm as used for the front right camera. The back camera is mounted at the rear of the Audibot, showing lower rear of the vehicle. This camera is used during the parking maneuver to provide feedback to parking control node. **Figure 1** shows the fully equipped Audibot.

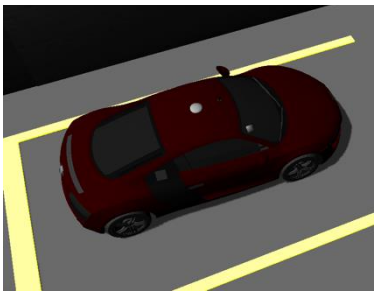

Figure1: Fully equipped Audibot

The Audibot is controlled using speed and steering angle commands, instead of the Twist message (speed and yaw rate). The controller node publishes appropriate topics, and Gazebo subscribes to them in order to move the vehicle in the

simulation. Gazebo provides the current speed, yaw rate, and steering angle as references to be used by the controller nodes.

B. Navigation

The navigation system consists of three nodes. The odom node implements state space model, the car_nav node implements costmaps and planners, and the set_nav_points node publishes appropriate goals to be used by planner. The **Figure 2** below shows the organization of the car_nav node. In order to develop the car_nav node, nav_stack_example created as an example in the class is used as reference.

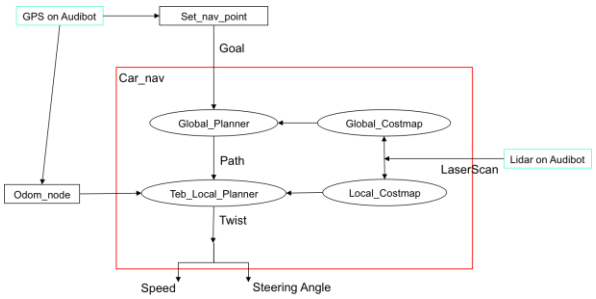

Figure2: Organization of car_nav node

1) Bicycle State Space Model

As Audibot is an Ackermann drive vehicle, the state space model for a bicycle type vehicle is implemented to estimate the vehicle position and heading. Only initial heading is provided to the system. The state vector, control vector, and state space equation are shown in **Equations 1, 2, and 3** below.

$$X = \begin{bmatrix} x \\ y \\ \psi \end{bmatrix} \quad (1) \qquad U = \begin{bmatrix} v \\ \alpha_s \end{bmatrix} \quad (2)$$

$$f(X,U) = \begin{cases} \dot{x} = v cos\psi \\ \dot{y} = v cos\psi \\ \dot{\psi} = \frac{v}{L} tan\left(\frac{\alpha_s}{\gamma}\right) \end{cases} \quad (3)$$

To implement these equations, discretization is necessary, as a digital program cannot implement those equations. The discretized version of state space model is show in equation 4. Here, the state space node is running at 50 Hz, so sample time Ts is 0.02 seconds.

$$\begin{aligned} x_{k+1} &= x_k + T_s * v_k * cosv_k \\ y_{k+1} &= y_k + T_s * v_k * sinv_k \\ \psi_{k+1} &= \psi_k + T_s * \frac{v_k}{L} tan\left(\frac{\alpha_{s|k}}{\gamma}\right) \end{aligned} \quad (4)$$

To improve the accuracy of location estimate, GPS reading is incorporated to the node. As state space node is running at 50 Hz while GPS is at 1Hz, x and y values are replaced with GPS reading whenever GPS updates. Heading estimation is done purely by state space equation.

The node publishes the information using an odometer message. Also, the node publishes the transform from base_footprint frame to map frame.

2) Goal Publisher

Since the vehicle has the front camera on the right side of the vehicle, Audibot must drive itself in a way that the parking spot comes to the right side of the vehicle. Also, a goal point is needed for the global planner to publish the path.

A node that performs a similar task was developed during the GPS simulation project. Therefore, the author had a solid reference to implement the idea. Currently, goal location is hard coded in the node, but the code will be updated in the future to take them as yaml file parameters.

The goal publisher node subscribes to odometer message, and calculates the distance till goal point. If the distance is less than the threshold value, then the node updates the goal to next point. Once the node reaches the last goal, it resets the goal to first point for further searching.

3) Global and Local Costmap

Costmap_2d package available in ROS is used for global and local costmap generation. The class is initialized inside the car_nav node. Costmap is generated with laser scan data obtained from the LIDAR on top of the vehicle. The main difference from costmap for differential drive is that Ackerman drive vehicle has a rectangular footprint instead of circular, and base_footprint is located at the rear of the vehicle instead of near the center. A few parameters are modified to accommodate these changes. The **Table 2** shows the summary of important parameter changes. These parameters are set in both the global and local costmap.

Parameter	Value
Footprint:	[[1.0, 2.0], [1.0, -2.0], [-1.0, -2.0], [-1.0, 2.0]]
inflation:	
inflation_radius:	3.0
cost_scaling_factor:	1.0

Table 2: Summary of important parameters

The footprint is specifically defined in costmap to generate a wider lethal cost area. The inflation radius is defined bigger as vehicle base_footprint is located at the rear. During the test, there were many incidents where Audibot hit the wall while base_footprint was outside the inscribed cost area.

4) Global Planner

The global planner used for this project is the same global planner used by a differential drive robot. It means that the global plan developed assumes that the robot can change directions while stationary without moving. This decision is acceptable for the project because the global planner is not generating the vehicle control command.

In order to develop the plan, global planner takes current location from odometer node, goal location from nav_set_point node, and generates the pose stamped message. Originally, global planner outputs error message stating that there is no goal provided to the terminal. In order to prevent this issue, code segment where global plan is generated is placed inside the conditional statement so that it only runs when there is a valid goal available.

5) Teb Local Planner

One of the most difficult parts of navigation of Audibot was to implement a teb local planner. Teb stands for Timed Elastic Band. The implementation of local planner that supports carlike robot was critical to this project as changing direction while stopped is impossible for Audibot. Originally, the author planned to develop own local planner by inspecting the architecture of base local planner package for differential drive robot. However, after extensive research, a teb local planner that experimentally supports carlike robot was found.

Currently, teb local planner is developed as plugin to the move_base node. However, the move_base includes a recovery feature that does not support carlike movement. Also, implementing the move_base node makes it harder to customize the behavior of the node. By mimicking the implementation of base local planner into the nav_stack node and finding similar functionality on teb local planner, the author was able to implement into car_nav node. One addition made to car_nav node is the conversion from yaw rate to steering angle. This is because teb local planner outputs the vehicle control command in speed and yawrate.

Resolving the dependency for the teb local planner was another big challenge that author has faced. The planner utilizes C++11 and GNU++11, and had dependency to SuiteSparse, which is outside of ROS ecosystem. To use new version of C++, following code described in **Figure 3** is added to CMakeLists.txt, and path to SuiteSparse is added in include_directories section.

```
include(CheckCXXCompilerFlag)
CHECK_CXX_COMPILER_FLAG("-std=c++11" COMPILER_SUPPORTS_CXX11)
CHECK_CXX_COMPILER_FLAG("-std=gnu++11" COMPILER_SUPPORTS_CXX0X)
if(COMPILER_SUPPORTS_CXX11)
    set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=c++11")
elseif(COMPILER_SUPPORTS_CXX0X)
    set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=gnu++11")
```

Figure 3: Enable C++11 and GNU++11 in CMakeLists.txt

Similar to base local planner, teb local planner takes various settings via yaml file. The important change made here is to allow large amount of deviation from the global plan. It is because global plan generates curves that are not feasible for Audibot to take.

Additionally, teb local planner takes wheel base or minimum turning radius to accurately plan local path that Audibot can follow. Enable_homotopy_class_planning parameter enables multiple local path planning, however, it uses so much computational power that the Gazebo simulation cannot run. Thus this parameter is disabled for the project.

As teb local planner is still experimental, the performance of the package can be a little slow, and decent time to create local path if there is objects around, or global plan is very tight for Audibot. However, this package is trying to solve a difficult problem that many people want the solution for, and the author is looking forward to contributions for making the package better.

C. Parking Spot Detection

Image processing is the area where the author has less experience, so extensive research on IEEE, SAE, and other technical papers are done to prepare a few methods to try. Also, professor provided a few guides on how to place camera in Gazebo and work with them. In this project, a front camera located at front right is utilized to detect the empty parking sport. Ultrasonic sensors will not work with this project because there is no other car to measure the distance.

1) Flow of Image Processing

The raw image captured from front camera is a rich image that contains details of various objects, and it is a great resource for humans to detect a parking spot. However, as more processing is necessary for the system to detect the parking spot, the camera image is converted to Mat data type to work with the OpenCV library. **Figure 4** below shows the raw image captured from the front camera. **Table 3** shows the flow of image processing.

Figure 4: Raw Image from Front Camera

1	Convert image from BGR8 to HSV
2	Split channel, take Hue
3	Threshold Hue (Center: 30, Width: 10) => Yellow
4	Erode and dilate to remove noise
5	Apply Canny Edge Detection algorithm
6	Erode and dilate to remove noise
7	Apply Template Matching, get highest possible location
8	Inside the location, apply Probabilistic Hough Transform

9	Calculate angles of each line, and delete similar ones
10	Check angle and number of lines detected

Table 3: Flow of Image Processing

First, the raw image is converted from BGR format to HSV format, and split to each channel. Next, the Hue channel is thresholded with center at 30 with width of +/= 10 to extract the yellow line color. **Figure 5** shows the image after Hue threshold is applied. Dynamic reconfigure server is established for this process to dynamically changes the threshold values to select best values to use. The resulting image is eroded to remove the small noise generated during the previous process, and dilated to bring back the width of the line.

Originally, Probabilistic Hough Transform was applied at this stage to detect the lines, but the algorithm detected hundreds of lines, as lane marking are still wide. In order to the reduce the number of lines, Canny Edge Detection is applied to the image to extract the edges of the lane mark. **Figure 6** shows the image after Canny Edge Transform is applied.

Figure 5: Image after Hue Threshold

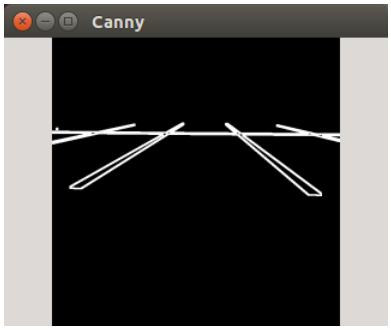

Figure 6: Image after Canny Edge Transform

After the canny transform, Probabilistic Hough Transform is applied to extract the lines. Although the number of lines detected is reduced, it was still difficult to select which pair of lines are two sides of parking spot that the car is searching for.

2) Template Matching

The method to apply Template Matching algorithm was developed and implemented. This method worked well as parking spots is repeat of similar shapes, and one template works for all of the parking spots.

Template Matching is an algorithm that consists of two steps. In OpenCV, there are six methods available to perform the task, and the CV_TM_SQDIFF method is used as it most accurately detected the parking spot. **Figure 7** shows the template image used for the project, and **Figure 8** shows the detected parking spot image.

- Step1: Compares a template against overlapped image regions.
- Step2: Finds the global minimum in the array

Figure 7: Template Image

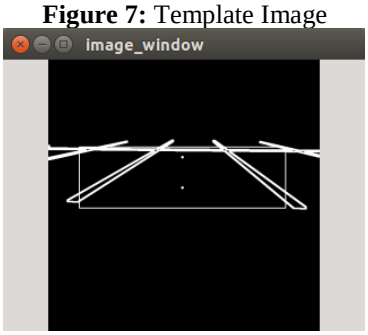

Figure 8: Sample Detected Parking Spot

The single problem left after the Template Matching was that the algorithm always returns the highest possible position where parking spot is detected even if there is no parking spot. In order to identify when actually parking spot is detected, Probabilistic Hough Transform is applied inside the detected rectangle. Also, angle of each line is calculated and all similarly angled lines are deleted. The parking spot is considered to be detected when there are more than three differently angled lines inside. **Figure 9** below shows the result of Probabilistic Hough Transform.

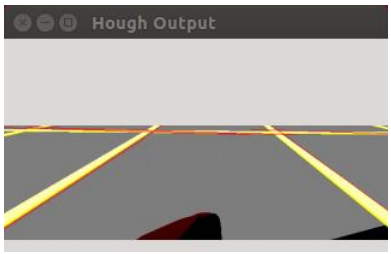

Figure 9: Result of Probabilistic Hough Transform

3) Feature 2d Method

Feature 2d is another method in OpenCV to recognize the object in the image. The main difference between Template Matching and Feature 2d is that Feature 2d finds the key points in both template and source, and tries to match similar key points. **Figure 10** shows the result of implementation.

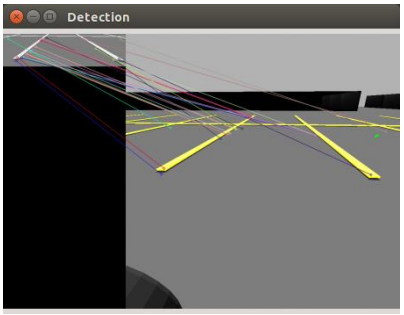

Figure 10: Result of Feature 2d

Feature 2d did not work well with this project for two reasons. First reason is that parking spot lines do not have obvious unique key points. It is obvious from the images that there are not many key points detected. Another reason is that a parking lot is consists of similar lines. This causes Feature 2d algorithms to confuse which key points belongs to which line. The image above contains the line that goes from left side of line to right side of line. As a result, this algorithm is not used in the project.

D. Parking Maneuver

Final component of the project after parking spot is detected is to park the car inside the parking spot. Since the author drives a car, one has decent idea of how the car should be controlled. Additionally, research was conducted on how current production vehicles that support automatic parking with sonar sensors work.

1) Basic Control

Since vehicle dimensions, vehicle dynamics, camera mounted location, and camera's field of view (FOV) are known to the system, time based fixed control can be implemented to park the car after a parking spot is detected. Referencing **Figure 11** below, it shows that there are only limited position where Audibot can detect the parking spot. If the car goes out of that range of position, parking spot recognition system does not locate the parking spot.

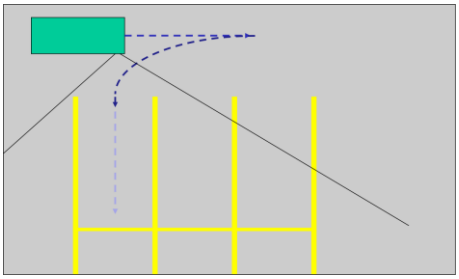

Figure 11: Trajectory for parking maneuver

Initially, manual vehicle control node with dynamic reconfigure server was developed to obtain the general idea of how long the vehicle should go forward, go backward with turn, etc. After these parameters are identified, a basic control node is implemented to perform the procedure automatically. However, the vehicle goes out of parking lane with pure time-based control as it cannot accommodate the position and heading differences when parking spot is detected.

2) Redo Process

In order to prevent vehicle from going out of the lane, feedback control needs to be developed. The back camera mounted at the rear end of the vehicle is used to provide the appropriate feedback to the parking control node to fix the position and orientation of the vehicle. Similarly to how the image processing is done for front camera, the same process up to number 6 in **Table 3** is applied to the back camera image.

Since the visible area of the back camera is significantly smaller than the front camera, Hough Line Transform is used instead of Probabilistic Hough Transform. Another reason of using line transform is because it returns rho (distance) and theta (angle). These parameters are very valuable to detect if the vehicle is oriented well, or is going out of the lane from either side. After numbers of detected lines are reduced using similar angle deletion method, logic is applied to identify the appropriate feedback signal. **Figure 12** shows the vehicle orientation and how the back camera detects the lane mark.

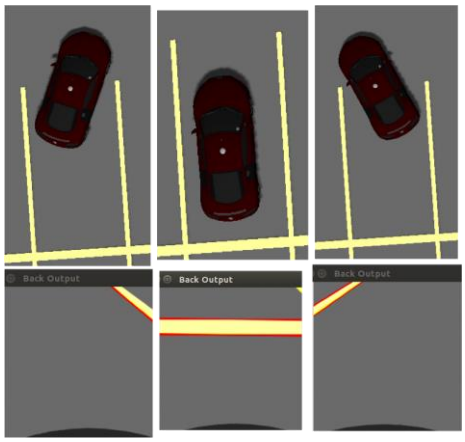

Figure 12: Relationship of Vehicle Orientation and Camera Image Lane Detection

```
std_msgs::Int16 back_status;
back_status.data = 0; // None
// Status of Back Line
for (int i=0; i<lines.size(); i++){
  if (fabs(lines[i][1]) > 1.5 && fabs(lines[i][1] < 1.6)){
    if (lines[i][0] > 200.0){
      back_status.data = 1; // Done
      break;
    }else {
      back_status.data = 0;
      break;
    }
  }else{
     if (lines[i][0] >= 0)
       back_status.data = 3; // right
     else
       back_status.data = 2; // left
  }
```

Figure 13: Lane Classification Algorithm

Figure 13 above shows the code snippet for lane classification and feedback signal generation code. In order to classify and parking complete, the lane park needs to come near the vehicle (rho > 200) with horizontal orientation (1.5 < theta < 1.6). If line is not horizontal, then sign of the angle is checked to classify if car is going out from left side or right side of the lane. Once it is detected, feedback is provided to control node, and the node performs a redo procedure.

3) Image Transport

The image transform is a method to project camera pixels into map frame with various mathematical operations and frame transformations. With this functionality, more accurate parking trajectory can be performed. Unfortunately, the node was not fully completed within the project timeline, but this section describes the development effort and current results.

In order to perform image transport, the following steps need to be done:

1. Each pixel to field of view (FOV)
2. FOV to Camera frame
3. Camera frame to Base_footprint frame
4. Base_footprint frame to map frame

Steps 3 and 4 are done using TF transform as Gazebo publishes transform from camera frame to base_footprint frame, and odometer node publishes transform from base_footprint frame to map frame. Image geometry package available in ROS ecosystem is used to transform each pixel to FOV then to camera frame. **Figure 14** shows the result of image transform visualized with Rviz. It is visible that parking spot is compressed. The cause of this problem seems to be the miscalculation of distance to the ground for each pixel as camera is mounted at angle.

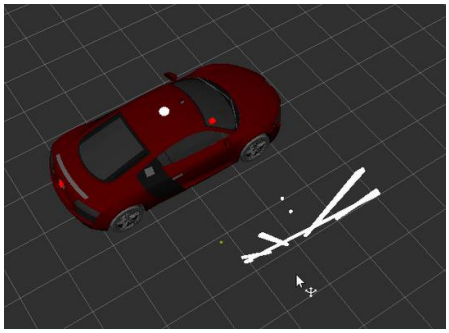

Figure 14: Result of Image Transform with Rviz

E. Integration

Once all components are ready to use, final part of the project is the integration of all components. **Figure 15** below shows the relationship of how each node interacts with other nodes. Since the system contains two control nodes that send the speed and steering, a master node is added to the system to switch between navigation control and parking control. The criteria to switch from navigation mode to parking mode is the five consecutive receiving of parking spot detection signal from front camera node. The figure also shows the nodes that are currently under development.

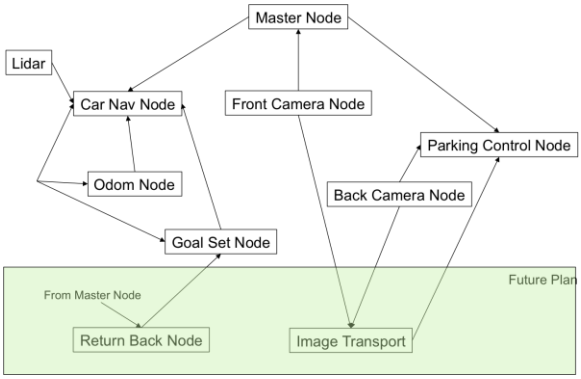

Figure 15: System Node Relationship

III. Experimental Setup

The project is developed and tested in Gazebo and Rviz based simulation environment. The simulation-based development is chosen because of the short development timeline, high cost of equipment, and safety of testing. As each component is developed, sectional testing was done to ensure the functionality. This reduces the debug time during the software integration process. **Figure 16** shows the Gazebo simulation environment developed for the project. The walls are randomly placed to test the obstacle avoidance feature of navigation. Parking lot model is developed from meshes using FreeCAD program. In order to make the simulation more realistic, world frame published by Gazebo is disabled, so that vehicle does not know the exact location without localization. The challenges the author faced with Gazebo simulation is the frequent crash of Gazebo, and model color not reflected by camera image. These issues should be resolved once the system upgrades to ROS Kinetic, which supports Gazebo 7.

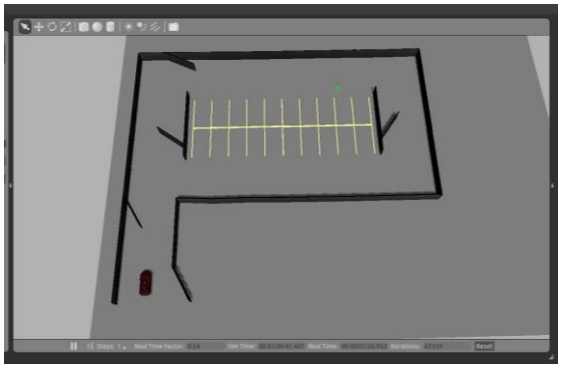

Figure 16: Gazebo Simulation Environment

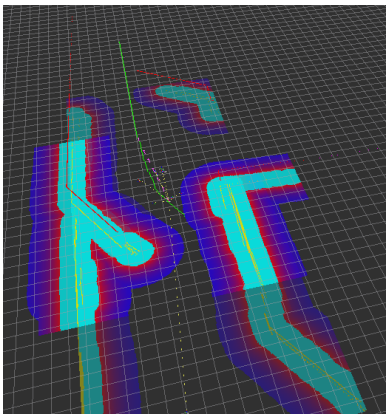

Figure 17: Rviz Visualization of Navigation Phase

Figure 17 above shows the visualization of navigation phase using Rviz. Green path is global path while pink arrows are local planner's path. With Rviz, it is possible to visualize how local planner is deviating from global plan to support Ackerman drive robot. Also, it is observed that costmap is deformed a little as vehicle goes because heading is purely based on bicycle state space model, and it accumulates error over time. Although the simulation runs at 1/3 of real speed when complete system is loaded, it is sufficient to test the system safely.

IV. RESULTS AND CONCLUSION

The project was able to reach the point where Audibot navigates the parking lot by itself by GPS points, identify the parking spot, and park the vehicle without touching any side of the line within a short development timeline. With current simulation environment, it takes 3 minutes in simulation time (1 minute in real time) to complete whole process of parking.

Future improvements are as follows:

- Full implementation of image transport node
- Functionality to call the vehicle back to the original location
- Re-factor the code using classes

Through this project, the author was able to demonstrate each component (navigation, parking spot detection, and parking maneuver) using the concepts learned in class, integrate all components, and make them work together. Experience was gained working with OpenCV library and was exposed to the field of image processing. The author has learned the difficulty of incorporating all sensors information and producing efficient control signals.

REFERENCES

[1] Teb_local_planner ROS documentation. [http://wiki.ros.org/teb_local_planner]

[2] Navigation package ROS documentation [http://wiki.ros.org/navigation]

[3] Image geometry ROS documentation [http://wiki.ros.org/mage_geometry]

[4] Hough Transform OpenCV documentation [http://docs.opencv.org/2.4/doc/tutorials/imgproc/imgtrans/hough_lines/hough_lines.html]

[5] Template Matching OpenCV documentation [http://docs.opencv.org/2.4/doc/tutorials/imgproc/histograms/template_matching/template_matching.html]

[6] Feature2d OpenCV documentation [http://docs.opencv.org/3.1.0/d0/d13/classcv_1_1Feature2D.html#gsc.tab=0]

[7] M.Radovnikovich, P. K. Vempaty, K.C.Cheok, "Auto-Preview Camera Orientation for Environment Perception on a Mobile Rebot"