Audibot Adaptive Cruise Control Final Report

April 26th, 2021

Team Members:
Lisa Branchick - BSME
Aaron Garofalo - BSME
Brian Neumeyer - BSME

Prepared for:
ECE 5532
Winter 2021
Dr. W. Geoffrey Louie

Abstract:

To demonstrate a sufficient understanding of autonomous vehicle navigation systems, a simulation of audibot adaptive cruise control was developed. Level 1 required an understanding of state space modeling, xacros and URDFs, as well as subscribing/publishing the required topics for controlling Audibot. To achieve this, the audibot_path_following package was utilized and modified accordingly. Gazebo's ModelStates data could then be used to determine vehicle location, calculate a distance, then pass the distance to PID to control the trailing vehicle's throttle position to maintain a desired distance. Level 2 required the implementation of a LIDAR sensor borrowed from RoundBot. To determine the distance between vehicles, the average of the valid returned data points from the LIDAR was taken. Since the simulation is based on a world with no trees or other obstructions, it was assumed that the returned data would only represent the leading car. To further ensure the validity of the data, the LIDAR's scan angle was restricted to only scan the region where the lead car is expected to be. Lastly, level 3 utilized a camera to locate the leading car. By using filters and cropping the image, it was possible to obtain a dataset describing only the leading car. Based on the vertical position of the average set of points returned by the camera, the distance between the two vehicles was able to be determined. All three levels utilized the same PID library and settings; however, the methodology used to determine the distance varied between levels.

Introduction:

Adaptive cruise control is used in many production vehicles as part of their ADAS package. However, different OEMs utilize different sensors and/or different methodologies to perform the essential functions. This project shows three different methods which could be used either individually or in combination to achieve the desired result. Level 1 extracts vehicle locations from Gazebo. Level 2 implements a LIDAR sensor which was borrowed from the RoundBot robot used for previous course examples and projects. Finally, level 3 uses the camera mounted on the Audibot.

For level 1, the vehicle's GPS information could have been used to determine the vehicle location; however, for additional accuracy within the simulation, as well as to expand on the knowledge acquired throughout the course, it was decided to instead pull location information directly from Gazebo. Level 1 also required an understanding of state space modeling, Ackermann drive, as well as subscribing/publishing the required topics for controlling Audibot. To achieve this, the audibot_path_following package was utilized and modified accordingly. Gazebo's ModelStates data could then be used to determine vehicle location, calculate a distance, then pass the distance to PID to control the trailing vehicle's throttle position to maintain a desired distance.

Level 2 required the implementation of a LIDAR sensor extracted from ROS RoundBot. To determine the distance between vehicles, the average of the valid returned data points from the LIDAR was taken. Since the simulation is based on a Flat World™, with no trees, signs, pedestrians, "birds", or other obstructions, it was assumed that the returned data could only represent the leading car. To further ensure the validity of the data, the LIDAR's scan angle was restricted to only scan the region where the lead car is expected to be.

Lastly, level 3 utilized a camera to locate the leading car. By using filters and cropping the image appropriately, it was possible to obtain a dataset describing only the leading car. Based on the vertical position of the average set of points returned by the camera, the distance between the two vehicles was able to be determined.

All three levels utilized the same PID library and settings; however, the methodology used to determine the distance varied between levels.

Level 1: Vehicle Position

The first portion of the project is to use the positions of the two Audibot vehicles to calculate and maintain the distance between them. Considering the variability in the vehicle behavior observed in the GPS project, it is assumed that some unknown “noise” is added for more-realistic operation in the Audibot simulation. The additional noise is not ideal for calculation accuracy, so it was decided to bypass the sensors and pull position directly from Gazebo’s model states, as the project definition did not specifically require use of any vehicle sensors. However, a GPS implementation would have been trivial to add as all team members implemented it in the Midterm Project. It was more entertaining to implement a new method to obtain positions. To find the vehicle positions, the gazebo_msgs::ModelStates command was used. In order to be able to use this command and extract the necessary data, two additional header files need to be included: <geometry_msgs/Pose.h> and <gazebo_msgs/ModelStates.h>. Through ROS_INFO trials, it was determined that the final two parts of the returned data contain the desired positions for the respective vehicles. The returned data was then placed into variables, so it could easily be used and updated for future calculations.

```
78  void recvModelStates(const gazebo_msgs::ModelStates& msg){
79    int arraySize = msg.name.size();    //vehicle models are at end
80    //last in array is lead car
81    //second last is follow car
82    double a1x = msg.pose[arraySize-2].position.x;
83    double a1y = msg.pose[arraySize-2].position.y;
84    double a1z = msg.pose[arraySize-2].position.z;
85
86    double a2x = msg.pose[arraySize-1].position.x;
87    double a2y = msg.pose[arraySize-1].position.y;
88    double a2z = msg.pose[arraySize-1].position.z;
89
90    a1_a2_separation = cartDistance(a1x, a2x, a1y, a2y);
91
92    pid_source = sep_target - a1_a2_separation;
93
94    //ROS_INFO("recvModelStates: %f", a1_a2_separation);
95  }
```

Figure 1: Function to receive model state positions from Gazebo

Once the vehicle positions are known, the distance between them is able to be calculated using the function cartDistance, which uses a basic pythagorean calculation, as is shown in Figure 2 on the following page. The returned distance is then used to determine the difference between current distance and desired distance, which is then used as the input to the PID controller to control the throttle position of the vehicle.

```
62  //finds pythagorean distance between two xy points
63  double cartDistance(double x1, double x2, double y1, double y2)
64  {
65    double xDiff = pow(x1-x2,2);
66    double yDiff = pow(y1-y2, 2);
67
68    return sqrt(xDiff+yDiff);
69  } //nice
```

Figure 2: Distance calculation between vehicles

Level 2: LIDAR

A LIDAR (LIght Detection And Ranging) sensor generates a mapped image of the surrounding environment using a pulsed laser. In the ROS simulated environment, LIDAR information can be utilized via the message sensor_msgs/LaserScan. However, the LIDAR must first be present within the Audibot model before it can be used within a function. To do this, the URDF files and associated launch files must be modified.

```
1  <?xml version="1.0"?>
2  <robot name="audibot_mod" xmlns:xacro="http://www.ros.org/wiki/xacro">
3
4    <xacro:include filename="$(find ugv_course_sensor_description)/urdf/hokuyo_utm_30.urdf.xacro"/>
5
6    <xacro:hokuyo_utm_30 name="laser_front" parent_frame="base_footprint" x="3.5" y="0.0" z="0.75" roll="0.0" pitch="0.0" yaw="0.0" />
7
8  </robot>
```

Figure 3: audibot_sensors_mod.urdf.xacro containing the xacro for the Hokuyo LIDAR sensor

Within this portion of the simulation project, there are several launch files to consider. The main launch file is “level_2.launch.” This file then calls on additional launch files that contain the Audibots and the world simulation in which they operate.There are then two xacro macro of interest called “audibot_gps_and_camera_and_lidar.urdf” and “audibot_sensors_mod.urdf.” The file “audibot_sensors_mod.urdf” contains the model for the Hokuyo UTM-30LX, the same LIDAR sensor which is used on the ROS Roundbot robot. This sensor is then called within

“audibot_gps_and_camera_and_lidar.urdf” and in turn, the launch file “spawn_audibot.launch.” This is the file in which the LIDAR and Audibot are joined. The location of the LIDAR sensor with respect to the Audibot can be altered within the xacro file “audibot_sensors_mod.urdf” by adjusting the parameters x, y, and z. Only the following vehicle has a LIDAR sensor, the lead vehicle does not.

```
123  int main(int argc, char** argv){
124    ros::init(argc, argv, "lidar_nav");
125    ros::NodeHandle nh;
126
127    //initializing PID object, passes pointer
128    PIDController<double> *ip = &vel_PID_controller;
129    initPID(*ip);
130
131    //timer to refresh PID
132    ros::Timer PID_timer = nh.createTimer(ros::Duration(0.01), PIDTimerCallback);
133
134    //for publishing steering and throttle messages
135    pub_vel = nh.advertise<geometry_msgs::Twist>("/a1/cmd_vel", 1);
136
137    ros::Subscriber sub_cmd_vel = nh.subscribe("/a1/cmd_vel", 1, recvThr);
138
139    ros::Subscriber sub_lidar = nh.subscribe("a1/laser_front/scan", 1, recieveLaserScan);
140
141    ros::spin();
142  }
```

Figure 4: LIDAR main() incl. file setup

Once the LIDAR is in place, a file can then be written to determine the speed at which the Audibot a1 follows the Audibot a2 using data obtained from the LIDAR message, sensor_msgs/LaserScan. The idea is to create a closed-loop system that adjusts the output speed based on the current speed and the current distance the following vehicle a1 is from the leading vehicle a2. To do this, the node must be subscribed to /a1/cmd_vel and a1/laser_front/scan, which gives the speed and the LIDAR information of the Audibot a1. There also needs to be an advertiser that controls the speed of the vehicle via /a1/cmd_vel. The centerpiece of the three is the subscriber for a1/laser_front/scan, which is handled in the function recieveLaserScan().

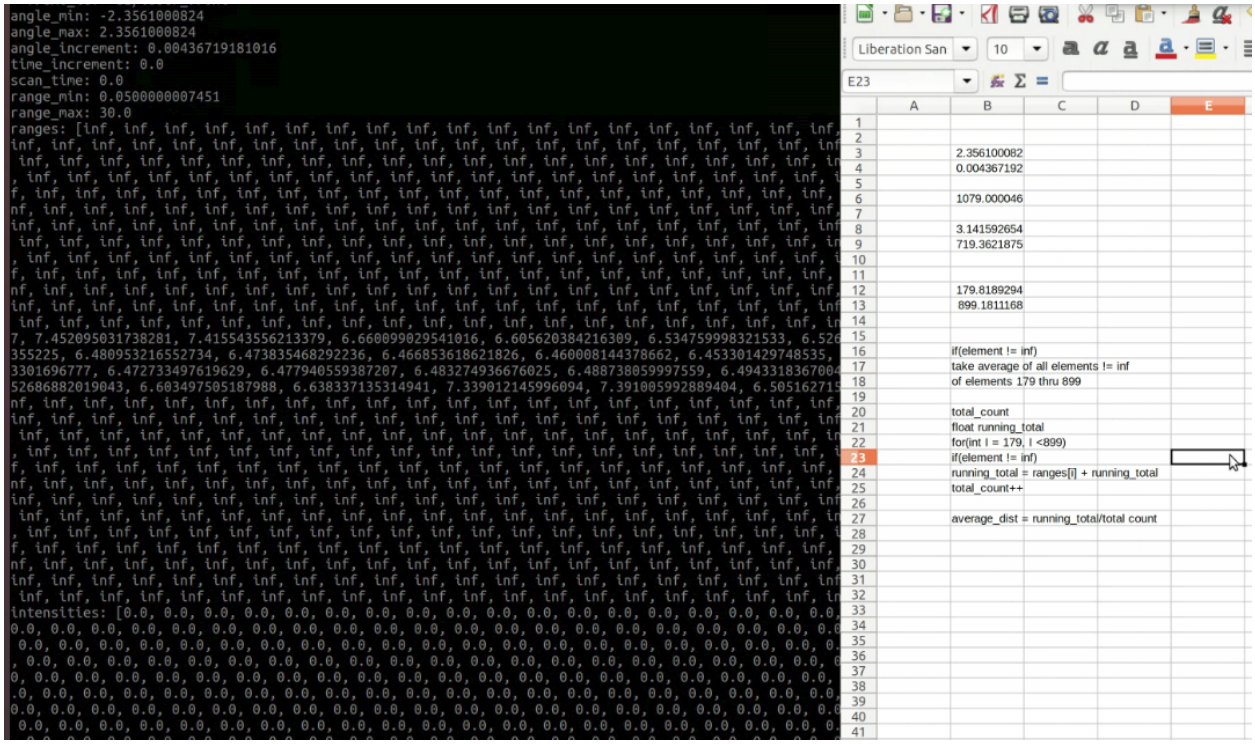

Figure 5: LIDAR data and calculations to determine the algorithm for distance/speed controller

In the function recieveLaserScan(), the LIDAR array called “ranges” is used to determine the position of the vehicle “a1” to “a2.” The array for “ranges” is based on the angles which the lidar scans multiplied by the angle incrementation. A for loop is then established to only focus on the array elements of interest (excluding “inf” values, > 30m) ranging from the elements 179 to 900. This range correlates to an angle range of ±90 degrees. There is then an if statement that then averages all the elements in the given range of the array and outputs this value as “a1_a2_separation.” There is a fail-safe in place if the return is not a number to keep the vehicle at speed. The “a1_a2_separation” is then fed into the PID controller for further refinement of the vehicle’s speed.

Level 3: Camera

For Level 3, the OpenCV library was utilized, specifically the Canny Algorithm and findContours. Luckily there was already a forward facing camera on the Audibot used for lane keeping. The view of this camera was also acceptable for vehicle detection purposes.

```
128  void recvImage(const sensor_msgs::ImageConstPtr& msg)
129  {
130    int x_0 = 0;
131    int y_0 = 0;
132    int x_f = msg->width;
133    int y_f = msg->width - 450; //cropping of vertical
134
135    cv_bridge::CvImagePtr cv_ptr = cv_bridge::toCvCopy(msg, sensor_msgs::image_encodings::BGR8);
136    cv::Mat raw_img = cv_ptr->image;
137
138    cv::imshow("Raw Image", raw_img);
139    cv::waitKey(1);
140
141    std::vector<cv::Mat> split_images;
142    cv::split(raw_img, split_images);
143
144    cv::Mat blue_image = split_images[1];
145    //cv::Mat green_image = split_images[1];
146    //cv::Mat red_image = split_images[2];
147
148    cv::Mat croppedImage = blue_image(cv::Rect(x_0, y_0, x_f, y_f));
149    cv::imshow("Cropped Image", croppedImage);
150
151    cv::imshow("Blue Image", blue_image);
152    cv::waitKey(1);
153
154    cv::Mat thres_img;
155    cv::threshold(croppedImage, thres_img, 2, 255, cv::THRESH_BINARY);
156
157    cv::imshow("Thres Image", thres_img);
158    cv::waitKey(1);
```

Figure 6: Initial image processing

To start the process, a subscriber callback function of “/a1/front_camera/image_raw” was created. The function took the raw image and separated the red, green, and blue components. As the target lead vehicle in this situation is blue, only the blue component was utilized. The blue component was cropped to remove A1’s hood and then a threshold filter placed over it to capture only the most blue pixels, i.e. the blue lead car. The remaining pixels were processed via eroding and dilating in order to reduce noise and make a more continuous block of pixels for the last steps. A useful feature from the openCV_example was a dynamic reconfigure server which allowed manual tuning of the filter values. Once proper values were found, they were hard coded in.

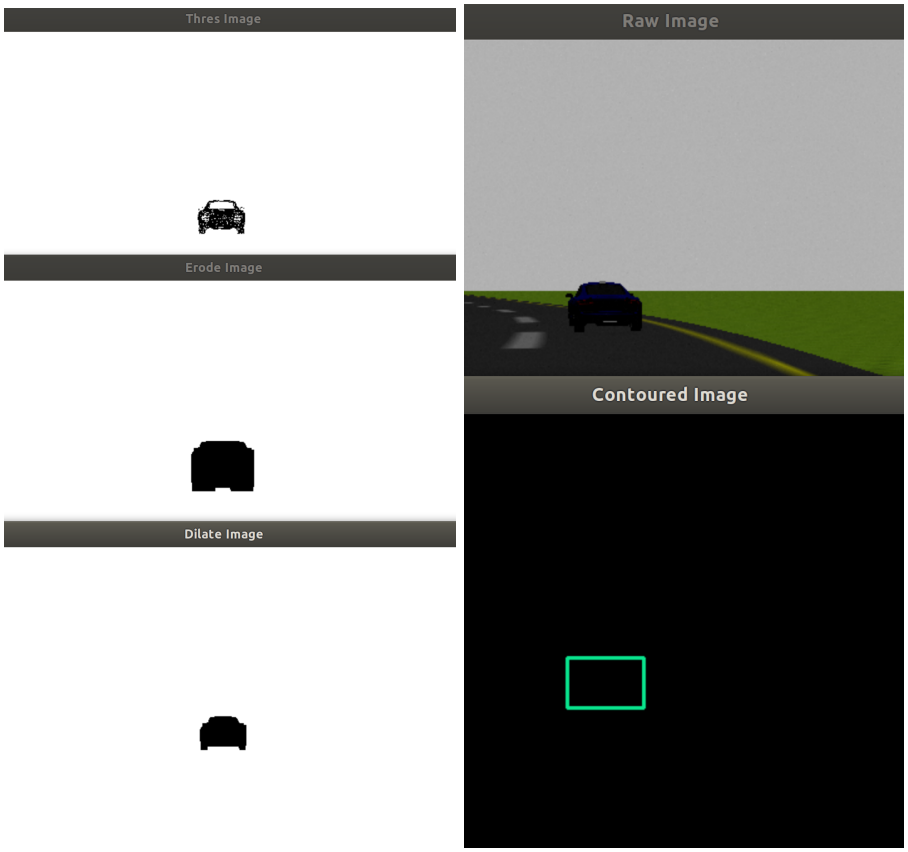

Figure 7: Left: thresholding, eroding, and dilating. Right: initial raw image, bounding box

```
183     for( size_t i = 0; i < contours.size(); i++ )
184     {
185         approxPolyDP( contours[i], contours_poly[i], 3, true );
186         boundRect[i] = boundingRect( contours_poly[i] );
187         //minEnclosingCircle( contours_poly[i], centers[i], radius[i] );
188     }
189     Mat drawing = Mat::zeros( canny_output.size(), CV_8UC3 );
190     for( size_t i = 0; i< contours.size(); i++ )
191     {
192         Scalar color = Scalar( rng.uniform(0, 255), rng.uniform(0,255), rng.uniform(0,255) );
193         //drawContours( drawing, contours_poly, (int)i, color );
194         rectangle( drawing, boundRect[i].tl(), boundRect[i].br(), color, 2 );
195         if(boundRect[i].height>max_height)
196         {
197           max_height = boundRect[i].height;
198         }
199         //circle( drawing, centers[i], (int)radius[i], color, 2 );
200     }
201
202     pix_height = max_height;
203
204     //ROS_INFO("Number of contours: %d", contours.size());
205     //ROS_INFO("max height: %d", pix_height);
206     imshow( "Contoured Image", drawing );
207
208     pid_source = pix_height - sep_target;
209
210 }
```

Figure 8: Edge detection, contour drawing, bounding box drawing, max box size filtering

With a cleaned up image, contours were drawn outlining the lead vehicle. Bounding boxes were drawn around all contours. Some noise still existed, but none of the noisy contours were larger than the vehicle's. This allowed them to be filtered out by only taking the largest bounding box. After the vehicle's bounding box was found, the height of that box was passed to the PID controller as the input. Height was used instead of width, as when the vehicle turns it presents a larger width, but height is relatively constant.

PID Library

```
25   //PID constants
26   double P = 8.5;
27   double I = 0;
28   double D = 0;
```

Figure 9: PID values

Once the vehicle positions were known and the distance between them calculated via the required method, all three levels of the project used the same PID library and settings to control the speed of the trailing vehicle. An online PID library was downloaded and included in the project files using #include "../PID/cpp/PID.h". This simplified the process, and removed the need to code a PID controller separately. In the end, only the proportional (P) control from PID was used, as it resulted in an ideal vehicle behavior while also minimizing the time and space complexity of the program. The chosen values for P, I, and D are shown above in Figure 9 and were determined based on trial and error to tune the vehicle behavior to an acceptable level. Future enhancement to implement full PID control for varying velocities and stability.

```
54   //initializes PID components
55   void initPID(PIDController<double>& myDoublePIDControllerPtr){
56     myDoublePIDControllerPtr.setTarget(pid_target);
57     myDoublePIDControllerPtr.setOutputBounded(true);
58     myDoublePIDControllerPtr.setOutputBounds(min_vel, max_vel);
59     myDoublePIDControllerPtr.setEnabled(true);
60   }
```

Figure 10: PID Controller Initialization

Using the function initPID() from the PID library, the acceptable target variance, minimum velocity, and maximum velocity are set according to the desired values. The minimum and maximum velocities ensured that the vehicle would always travel forward, while also restricting the range to a maximum variation of only up to 10 m/s. This helped to calibrate the PID to a reasonable speed and ensured that the vehicle would not drift or spin out if the controller called for a rapid acceleration. Separate global variables were created to hold these values, so they could be easily accessed. This is good practice and helps to avoid human error, especially if the same values are expected to be used in more than one location.

```
100  //refreshes PID
101  void PIDTimerCallback(const ros::TimerEvent& event){
102    vel_PID_controller.tick();
103    /*ROS_INFO("PID ticked");
104    ROS_INFO("PID Target: %f", vel_PID_controller.getTarget());
105    ROS_INFO("PID error: %f", vel_PID_controller.getError());
106    ROS_INFO("PID output: %f", vel_PID_controller.getOutput());
107    ROS_INFO("PID feedback: %f", vel_PID_controllegetFeedbackr.()); */
108  }
```

Figure 11: PID Timer Callback and Debugging

The PID function uses a timer callback to ensure the input and output data constantly refreshes according to the current calculated distance. The timer callback triggers the .tick() function, which calls a number of other PID functions to perform the necessary updates to the controller. For this project, 100 Hz was used as the update frequency. This ensured that the throttle control could be smoothly operated, thereby reducing the chance for undesirable vehicle behavior, such as drifting. In order to validate that the function was being called and updated properly, a series of ROS_INFO lines were added to print the various returned values. The values from each cycle were compared, both to each other and to the displayed simulation, and the intended behavior was confirmed.

```
97  //init PID controller
98  PIDController<double> vel_PID_controller(P, I, D, pidDoubleSource, pidDoubleOutput);
```

Figure 12: PID Controller Input/Output

A target separation was created and set to 18 meters. This distance is somewhat arbitrary, as there was not a required distance for the project. However, if the range is too small, there is a

higher chance for collision between the two vehicles while the PID controller works to modulate the throttle. Figure 12 shows the inputs for the PID controller. These include P, I, D, the input, and the output of the function. The difference between the current distance and the desired distance is used as the PID input. If the following vehicle is farther than the target distance, PID instructs the vehicle to speed up. Similarly, if the following vehicle is closer than the desired distance, it is instructed to slow down. Both act proportional to the variation (i.e. the further from target, the faster the car is instructed to go). The resulting output is calculated in terms of vehicle speed, and the result is returned to pidDoubleOutput, which then calls cmdVel to publish vehicle speed and steering angle.

```
30  //publishes velocity and steering command messages
31  void cmdVel(double v)
32  {
33    geometry_msgs::Twist vel;
34
35    vel.linear.x = v;
36    vel.angular.z = cmd_turn;
37
38    pub_vel.publish(vel);
39    //ROS_INFO("Published Velocity: %f", vel.linear.x);
40    //ROS_INFO("Distance between cars: %f", a1_a2_separation);
41  }
```

Figure 13: cmdVel Function for controlling vehicle behavior

As has been used in several class examples and projects, cmdVel() is used to control the vehicle behavior through throttle and steering angle. Since the vehicle's steering is not being controlled for this project (it is already being controlled via the path following algorithm), the steering angle is simply read and then passed through unchanged. The throttle control, after being calculated by the PID controller, is input to cmdVel() and assigned to the linear.x variable. Once both variables are calculated and passed to the appropriate location, the velocity command is published to the vehicle.

Conclusion

It can be observed from the simulations, that GPS, LIDAR, and cameras are all viable options for ADAS cruise control. Each of these solutions has their own challenges with regards to application. In the simulation, it was possible to use the base footprints of the Audibots to determine relative location. In the real world, GPS would need to convert coordinates between mapped coordinates and global latitude and longitude. In addition, a GPS controller would need to update quickly and reliably with information from satellites on both vehicles. For LIDAR, strengths and weaknesses are largely based on the model of LIDAR and how the sensor is integrated into the system. It is also important to keep in mind that, while sensors report valuable data, the instrument cannot distinguish between reliable and unreliable information. This means that outliers can appear due to unexpected reflective surfaces or fast moving objects, which were not present in the ROS simulation. Lastly, the camera would have similar issues to the LIDAR sensor. Two cameras would likely have to be implemented to allow for depth perception in a 3D plane. These cameras would also need to be calibrated to allow contrast between surfaces. If there was a dark gray object along the surface of the road or if there are unfavorable conditions (such as snow/rain) present, the camera may not be a reliable source for navigation.

For simulation purposes, all of the systems presented were implemented individually and performed without issue. However, in the real world industry, it is much more likely that a combination of some or all of the systems would be utilized to play off of strengths and weaknesses. This is generally done as a redundancy check in most systems for safety and functionality purposes. The system then can rely on the strength of each of the sub components without the drawbacks being a major issue. For further implementation of the systems presented in this paper, an additional file could be written to compare the results of each system level against each other to deem which information is reliable and determine if further action is necessary. This method could be practiced and utilized in a number of different autonomous vehicle applications.

References

1. https://github.com/robustify/audibot
2. http://wiki.ros.org/xacro
3. http://wiki.ros.org/urdf/Tutorials
4. http://wiki.ros.org/navigation/Tutorials/RobotSetup/Sensors
5. http://gazebosim.org/tutorials?tut=add_laser
6. https://www.automotiveworld.com/articles/lidars-for-self-driving-vehicles-a-technological-arms-race/
7. https://docs.ros.org/en/diamondback/api/sensor_msgs/html/LaserScan_8h_source.html
8. http://docs.ros.org/en/jade/api/gazebo_msgs/html/msg/ModelStates.html
9. https://github.com/nicholastmosher/PID
10. https://github.com/opencv/opencv