Mini Project in Robotics
on

Autonomous Car Project - Vision Implementation

Submitted by:

Harman Mehta - 2016BB50003
Tanmay Goyal - 2016ME20757

Supervisor:
Prof. Sunil Jha

Department of Mechanical Engineering

Indian Institute of Technology, Delhi

New Delhi - 110016

Acknowledgement

This is to certify that the thesis entitled ‘’ submitted by HARMAN MEHTA and TANMAY GOYAL to the Indian Institute of Technology, Delhi for the completion of Bachelor of Technology degree is a bonafide record of research work carried out by them under our supervision. This thesis has been prepared in conformity with the rules and regulations of the Indian Institute of Technology, Delhi. We further certify that the thesis has attained a standard required for a B.Tech degree. The research reported and the results presented in the thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Prof. Sunil Jha

Date: _____

Abstract

In recent years, a lot of research has been conducted on Autonomous cars and control. This report discusses a section of the parent project, based on the development of an Autonomous Controller for the self-driving Mahindra-e2o. It talks about the implementation of vision, perception, and navigation for realising autonomous driving. The primary aim is to design an autonomous navigation system which has perception (lane-detection) and motion-planning (steering) as its two main components. A video of the road is converted into the coordinates of the detected lane lines and a continuous optimal waypoint for the car is generated to achieve driverless control.

This report discusses an approach for lane following and driving, the theory and literature behind it, simulations on ROS Gazebo for testing the technology, qualitative comparison of the designed models, live ZED stereo camera feed testing of the algorithms and finally generating relevant insights and conclusions. As the results from implementing the vision-based lane detection algorithm on a live video stream were encouraging, the algorithm was integrated with car steering control, thus enabling the car to move autonomously on a straight road from vision feedback for the first time ever. Due to the Covid-19 situation, the work had to be shifted to simulation. All the functionalities and features which were being tested in the real car were successfully incorporated in the simulator. The ROS-bridge was used to control the vehicle and get the sensor data from CARLA. Then the work was directed towards further improving control by applying, tuning, and optimising the PID control in the CARLA simulator, assuming parameters for the real world can be found close to them.

Contents

List of Figures

Chapter 1: Introduction

1.1 Background

The automobile industry seems to be on the brink of a giant technological leap. It has come a long way since the development of the first commercial car in the early 20th century. Autonomous / Self-driving / Robotic cars have recently attracted a lot of attention all around the world. An autonomous car can sense its environment, interpreting different sensory information (LIDAR, GPS, Computer Vision) and guide itself safely with little or no human input.

The world's first radio-controlled car was 'Linrrican Wonder' (1926). Over the years, many companies like Mercedes, Google, Nvidia, Tesla have invested heavily in autonomous cars, and significant progress has been made. There are broadly five different levels of autonomous driving -

Level 0: The system can only issue some warnings based on certain sensor readings. The control remains mainly manual.
Level 1: The automated system can control certain systems like steering system during Parking Assistance; speed during Cruise Control
Level 2: Partial automation. The system can control the steering, speed, braking of the vehicle, but the driver has to continuously monitor the car.
Level 3: The driver can take his/her eyes off, but the driver must be ready to take control within a specified time interval.
Level 4: No human attention is needed, but the driver still remains in the car. The automated system should have the capability to follow all safety protocols.
Level 5: 100% automation. No driver in the car.

Some of the recent autonomous systems launched could operate at Level 3 and 4. The driverless system that is being implemented works at Level 2, where the car automates all the control systems but under continuous monitoring of the driver. In this report, our autonomous system will be implemented using software like OpenCV, ROS, and Gazebo, which could simulate real-life data using sensors and virtual environments.

Autonomous vehicles can help solve a major problem faced in India - traffic jams, accidents (93% of accidents are due to driver negligence), environment impacts,

parking space, etc. but the unique nature of chaotic, diverse Indian roads and transport pose a big challenge in designing successful self-driving cars.

1.2 Overview

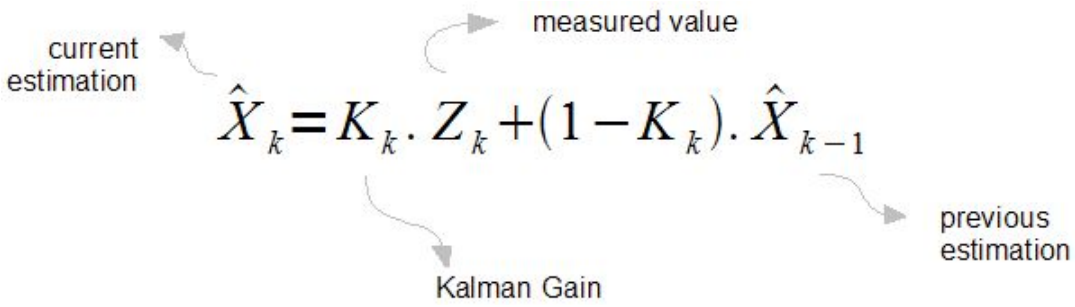
This thesis discusses a lane keep assist system, initially understanding and working on raw colour and image detection edge detection algorithms for images. Later it was tested on ROS (Robot Operating System). A simulation environment with a turtle bot along with its proper control inputs had already been created for the project. The 3D physics engine Gazebo was used to create this environment. The reason behind using Gazebo was its high flexibility and its easy integrating nature with ROS.

Later improvements were made to the lane detection algorithms by employing various strategies like feature detection and filters. These were tested on data from the Zed Stereo camera: mounted on the windshield pointing towards the front of the car. It was tested on all the roads inside IIT Delhi. Following this, the lane following was planned for the motion of the car on straight, curved, and discontinuous laned roads. With good lane detection results, the focus is now on improving the control of the car and the correction algorithm used.

Due to the Covid-19 situation, testing on the car had to be discontinued, and hence the decision was made to work on improving control in simulation using the CARLA simulator. All the functionalities and features which were being tested in the real car were successfully incorporated in the simulator. The ROS-bridge was used to control the vehicle and get the sensor data from CARLA. Then the work was directed towards further improving control by applying, tuning, and optimising the PID control in the CARLA simulator, assuming parameters for the real world can be found close to them.

Chapter 2: Literature survey

Vision-based autonomous driving has been studied, attempted, and implemented and has shown to have real applications [1] [2]. Many different resources have been referred for this attempt and are discussed in this section. For vision implementation, OpenCV, which stands for Open Source Computer Vision Library [3], was studied. The library has over 3000 optimised computer vision and machine learning algorithms that provide space for machine perception models and computer vision applications. These are used for object identification, face detection, recognition and classification of human actions, tracking of camera movements, tracking of moving objects, extracting 3D models of objects, and producing 3D point clouds from stereo cameras. Since the implementation of autonomous driving is based on vision, the selection of the camera is very crucial. While many stereo cameras were considered, it was decided to proceed with the ZED stereo camera. It is a 3D camera capable of sensing depth, tracking motion, and real-time 3D mapping. It provides a wide-angle all-glass dual-lens image with reduced distortion. For processing ZED feed (SVO format) to AVI format, SDK requirements of an Nvidia GPU with compute capability > 3.0 are required. For this, Nvidia CUDA [4] (Compute Unified Device Architecture) was used. The CUDA platform is an additional layer of software used to give direct access to the GPU's virtual set of directions and parallel computational elements, for the kernel computation. For efficient control, a Kalman filter [5] was applied to the control elements. The Kalman filter is a combination of mathematical equations that gives an efficient means of computation to predict the state of a process by minimizing the mean square error. It provides estimations of past, present, and even future states, even without knowing the exact nature of the model. This is a two-step algorithm. The first step is the prediction step in which the Kalman filter estimates the current state variables and their uncertainties. After the next measurement, which often contains some amount of error, including random noise, these estimates are updated using a weighted average, where estimates with higher certainty are given more weight. The algorithm can run recursively in real-time, using only the current measurements of input, the previously calculated state and its uncertainty matrix.



The working of the Kalman filter is explained in Fig. 2.1.

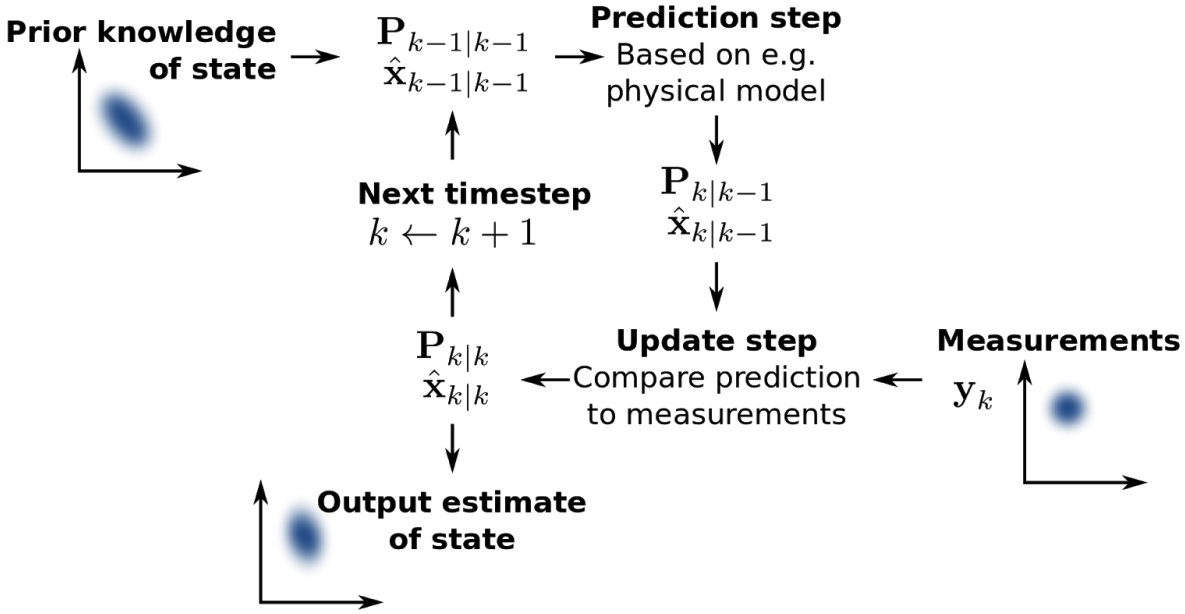


Fig 2.1 Kalman filter working

To further improve control, a PID control algorithm [6] was implemented. PID Control or Proportional Integrative Derivative control is a corrective feedback system. It calculates the error as the difference between the desired direction and heading direction of the car. The corrective control applied attempts to minimise the error over time by applying a control, which is a weighted sum of the three terms, i.e., proportional, integrative, and derivative errors. This control can be defined as follows:

$$u(t) = K_\mathrm{p} e(t) + K_\mathrm{i} \int_0^t e(t')\,dt' + K_\mathrm{d} \frac{de(t)}{dt}, \qquad \text{- (2.1)}$$

Where, u(t) = control term

e(t) = error term

Kp = proportional term’s coefficient
Ki = integrative term’s coefficient
Kd = differential term’s coefficient

Introducing the proportional gain (Kp) term has the effect of proportionally increasing the control signal for the same error level. The introduction of a derivative term to the controller (Kd) adds the controller's ability to "anticipate" error. With the addition of a derivative term, as the error begins increasing the control term increases, even while the magnitude of the error is still relatively small. This adds a damping effect to the system. The introduction of an integral term to the controller (Ki) helps reduce the steady-state error. The integrator term becomes bigger if there is a persistent, subsequently increasing the control signal and minimizing the error. To communicate with the car, ROS [7] was decided as the most suitable tool. ROS is an open-source, meta-operating system for robots. It provides capabilities like hardware abstraction, implementation of commonly-used functionality and communication between processes. It provides tools and libraries for building, writing, and running code across multiple machines. To test the performance of the code, it was decided to use two simulators, namely, Gazebo and CARLA. The Gazebo [8] is a 3D multi-robot simulator, complete with dynamic and kinematic physics, and a pluggable physics engine (Fig 2.2).

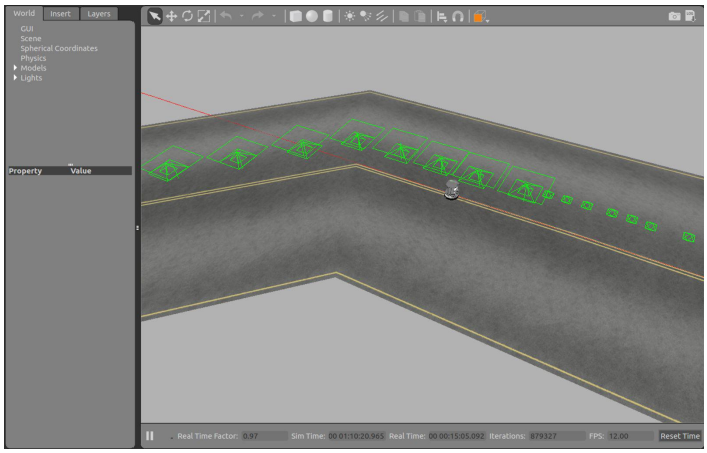


Fig 2.2 Gazebo simulation platform

The available set of Gazebo plugins provides integration between ROS and Gazebo which caters to many existing robots and sensors. The plugins provide the same message interface as the ROS ecosystem, thereby providing the ability to write ROS nodes which are compatible with simulation, stored data, and hardware. The application can be developed in simulation and then deployed to the physical robot with minimal

changes to the code. The other simulator used, CARLA [9], is an Open-source simulator created specifically to support the development, training, and validation of autonomous driving systems. The simulation platform supports the flexible specification of sensors, environmental conditions, and full control of all static and dynamic vehicle variables. CARLA has a powerful API which provides the programmer with control over all aspects of the simulation, including weather conditions, shadows, sensors, traffic generation, pedestrian behaviours and much more. CARLA is also provided with integration with ROS via a ROS-bridge, making it suitable for this study.

Chapter 3: Project Objectives and Workplan

3.1 Problem Definition/ Motivation

Recently, Mahindra took the initiative to encourage research on autonomous vehicles in India through the Mahindra Driverless Car Challenge. The challenge is divided into levels. Level 0 challenge includes simple tasks such as lane driving and reading traffic signals. The complexity of the challenge increases and level 3 will address problems such as avoiding a truck with protruding rods, identifying shallow ditches and unmarked bumps, wading through waterlogged streets and night driving capabilities. Team dLive at IIT Delhi is participating in this competition and is currently working together to develop a fully autonomous vehicle.

3.2 Objectives of the work

The objective of this project is to move a driverless car on the road at Level 2 stage of autonomous driving with the help of lane following.

Goals of the project are:

- Implementing the optimal perception (lane detection) model on the video feed
- Design the motion planning (steering) for lane following
- Improve and implement the detection algorithm in the CARLA simulator
- Improve the control and correction algorithm in the CARLA simulator
- Testing on the real-world live feed on Mahindra e2o
- Import Mahindra e2o car in CARLA

A GitHub repository is also being maintained for the project so that multiple versions of the same project can be maintained and make it easy for others to contribute as well.

Link - https://github.com/tanmay2798/Autonomous_car_JRD.git

3.3 Methodology

The goal that was opted is to make the car move autonomously using vision system feedback. First, the existing methods that were available online were studied. After surveying numerous methods and extensive discussion, it was concluded that autonomous navigation could be done using the lane mark detection and prediction technique using the OpenCV Kalman filter. So, for the first month, the work was focused on the problem of solving lane driving in a simulated environment - Gazebo. The next task at hand was to implement the PID control system on the Turtlebot in a simulated environment taking input from lane detection to verify the algorithm.

After successfully implementing the control in simulation, the next task in hand was to implement the lane detection system in the real world taking care of all the noises, disturbances, and interfering objects. This will help in providing a robust input of detected lanes and direction to steer towards the autonomous car in the real-world hence achieving autonomous movement. Kalman filter, averaging technique and PID control were implemented to achieve autonomous motion of the car. Several test runs of the car were conducted to tune the PID parameters, but it was not feasible to tune the parameters by testing on the real car from scratch. So, a simulator was to be used to tune the parameters and get the corrections.

After comparing several simulators, the CARLA simulator was selected for further improvements in the system. All of the results from the real-world testing were successfully translated into the simulation. The ROS-bridge was used to control the vehicle and get the sensor data from CARLA. Then the work was directed towards further improving control by applying, tuning, and optimising the PID control in the CARLA simulator, assuming parameters for the real world can be found close to them. Once the parameters were tuned in the simulator, they were to be tested on the real car. But, because of the COVID-19 crisis, this was not possible, and the work was continued on improving in the CARLA simulator.

Below is the Gannt chart representing the completed work.

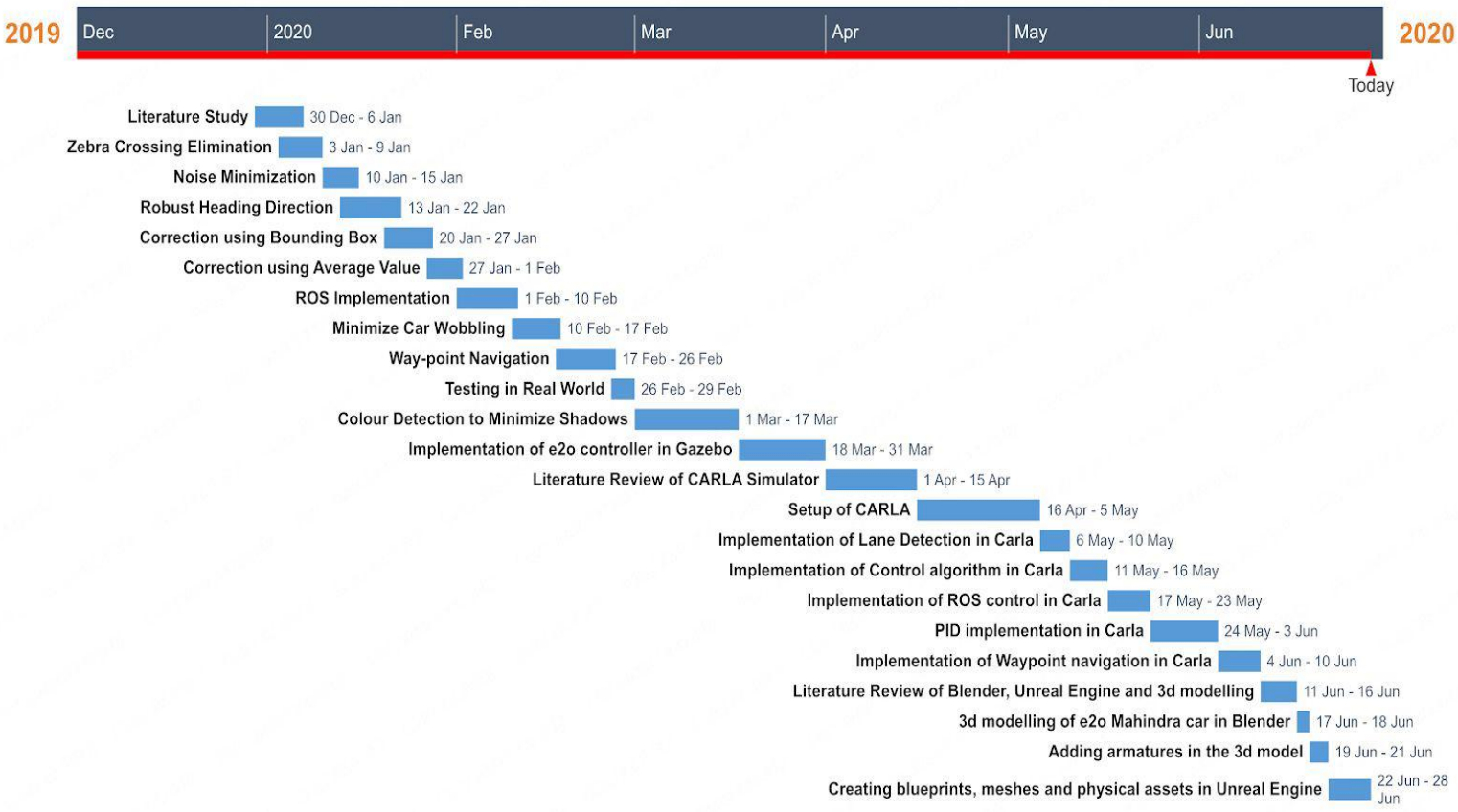


Fig 3.1 Gantt Chart

Chapter 4: Previous Work

4.1 Region of Interest

After applying Canny-Edge Detector and Hough-Line Transform, there are still many lines that are detected which are not lanes. To solve this issue, the Region of Interest was selected from the frame to exclude all the lines which are not lane marks.

Region of Interest is a polygon that defines the area in the image which contains the edges of interest. The image has the origin (0,0,0) coordinate in the top-left corner of the image. Row coordinates increase from top to down, and column coordinates increase from left to the right.

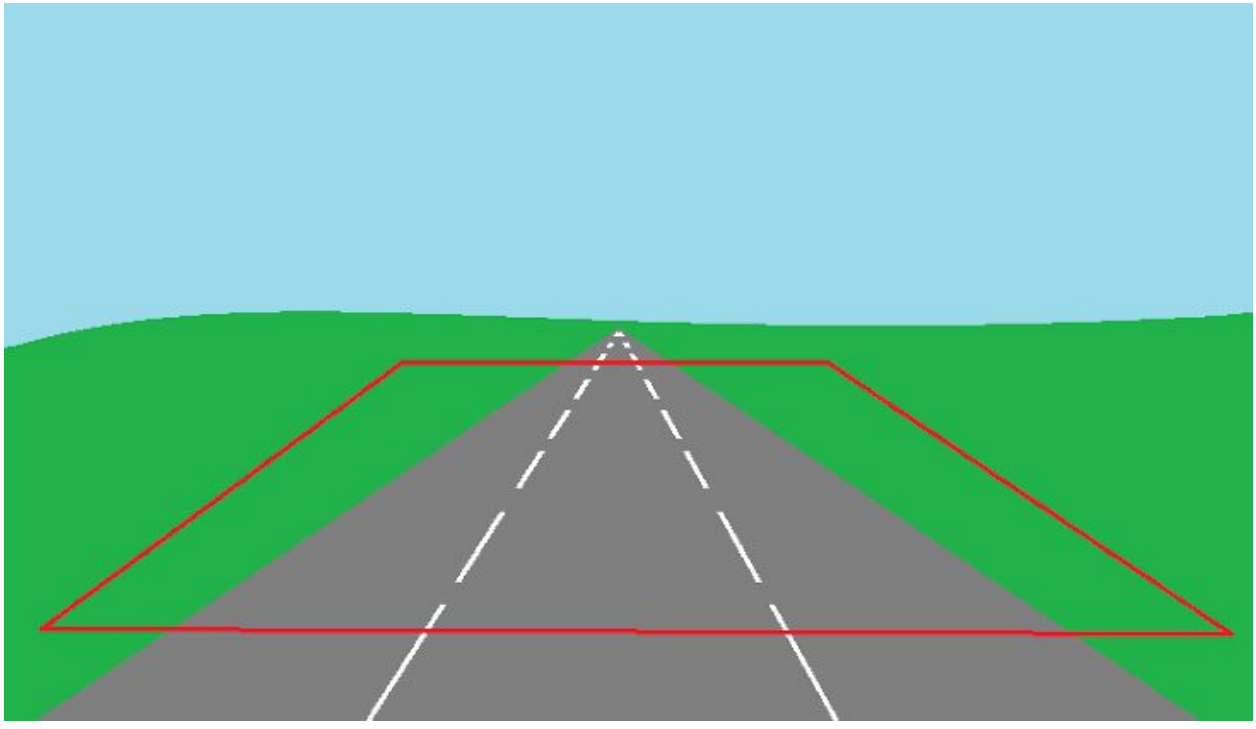


Fig 4.1 Region of interest inside the polygon

Testing 1

1) The region of interest was cropped out, excluding all the surrounding items like trees, sidewalks, buildings, humans, etc. which provided interference to the detection.

2) A mobile camera was then used to record a video, and the algorithm was implemented on the video, getting promising results with slight interference of the surroundings.

3) The results were accurate, but the code loop time was too long because of each pixel of the image being scanned out to find the lane. Thus implementing this on a real car can lead to a very slow refresh rate and response.

Fig 4.2 Lane detection on video recorder from mobile

4.2 Perspective Transform

A trapezoidal region was to be defined in the 2D image. This region would go through a perspective transform to convert into a bird's eye view, like below:

Fig 4.3 Trapezoid made using the detected lanes

Then, 4 extra points were defined which form a rectangle that will map to the pixels in our source trapezoid:

np.float32([[569,IMAGE_HEIGHT],[711,IMAGE_HEIGHT],[0,0],[IMAGE_WIDTH,0]])

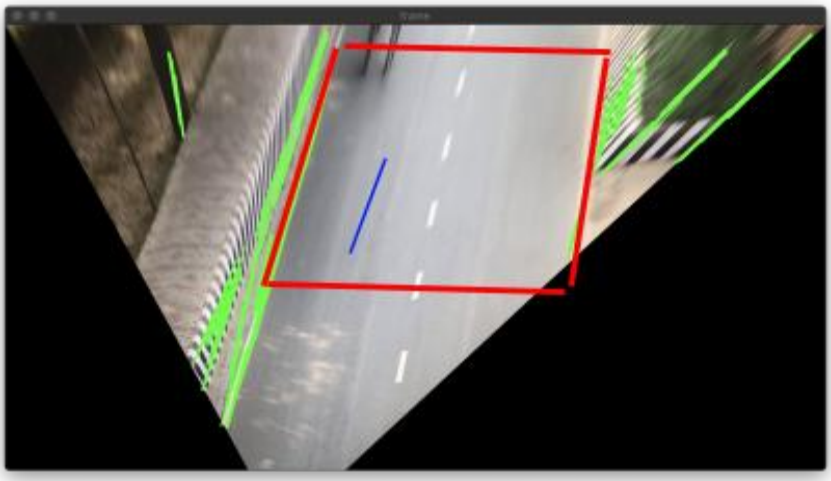


Fig 4.4 Rectangle in bird's eye view

4.3 Revised Lane Detection Algorithm

A few changes were incorporated into the lane detection algorithm in order to overcome its limitation of slow loop speed and inaccurate lane detection because of the noise and interference of objects. The changes made were -

1) **HoughLines instead of HoughLinesP** - This increased the loop speed significantly by reducing line detection time.

2) **Detection of left outlines** - Those lines, of which only one end and slope can be detected in the image frame, and the other end lies somewhere outside the image were also detected.

3) **Scaling Lines** - The farthest point of the line segment was scaled to be on the drawing horizon so that the point is not missed out. In this way, all the lines that can be found in a frame were successfully detected for getting better results.

By this change, the curved roads, and discontinuous lane marks were also detected very accurately, thus solving the issue of turning and improper lane marks as well. The issue of shadows and lighting conditions was also solved by this and made a system robust to lighting conditions.

Fig 4.5(a) Discontinuous lanes detected

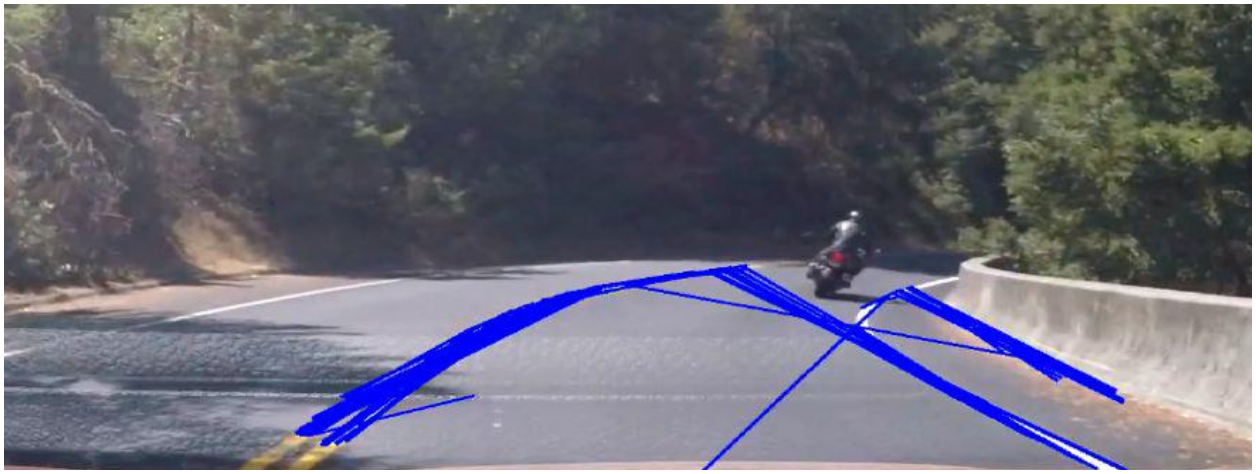


Fig4.5(b) Lane detection on curves

Testing 2

1) Due to some network/communication error, most of the ROS packaged dropped, and hence there was very less live ZED data available.

2) Only three frames were captured, and the new algorithm was tested on those three frames getting more lines detected than earlier even in dim light conditions.

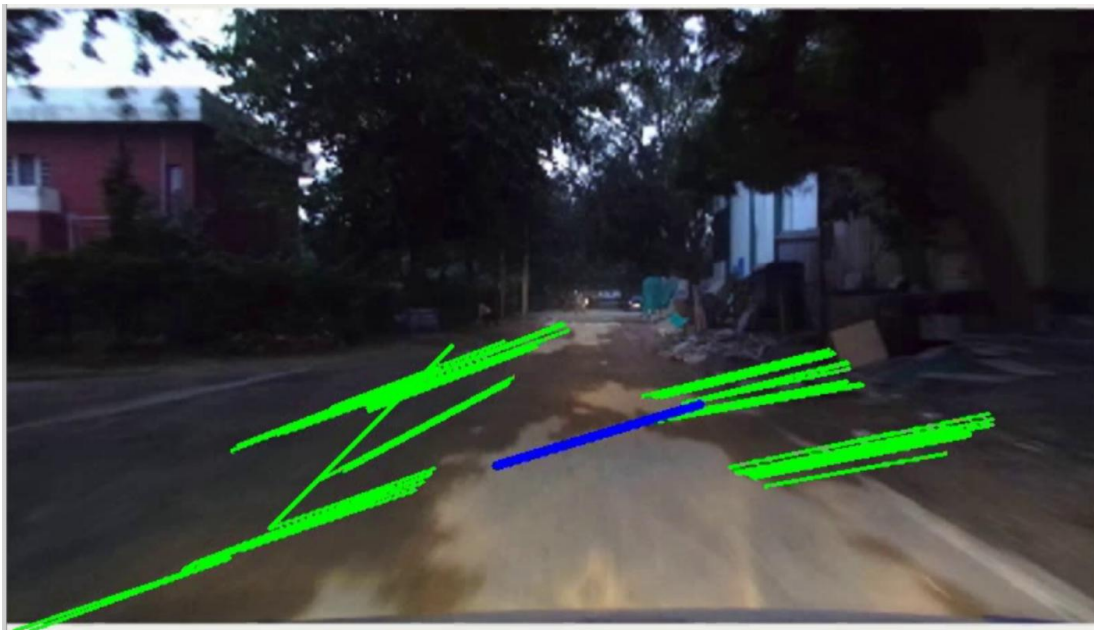


Fig 4.6 Edge detection by new algorithm

Testing 3

1) A test run was conducted using a ZED camera, and the data was recorded in SVO format using ZED Explorer software. To convert it to AVI or MP4 format on which lane detection could be carried out, the Nvidia CUDA 10.0 was used.
2) Promising results were received on running the algorithm on the recorded .avi videos.

Fig 4.7 Lane detection on ZED video feed

4.4 Kalman Filter Approach

After the detection of lanes accurately in lighting conditions, some major issues needed to be solved for making the system more robust, which were -

1) Low light intensity in some parts during the day and at night (Due to absence of street light)
2) Difficult to tune the parameters for various light intensities
3) Poor edge detection (irregular lane availability or absence of road lanes)
4) Shadows, Sudden high intensity from other vehicle headlights

Thus the approach was shifted to a more sophisticated one in which the lanes were predicted from current data and error data. This solved the issue of discontinuous lane marks completely and also to some extent, the issue of turns where lane marks are not present. Kalman Filtering was used for improved lane detection. Here, it was applied as a linear estimator for the clustered lines for the lanes to make it stable and free from any offset errors.

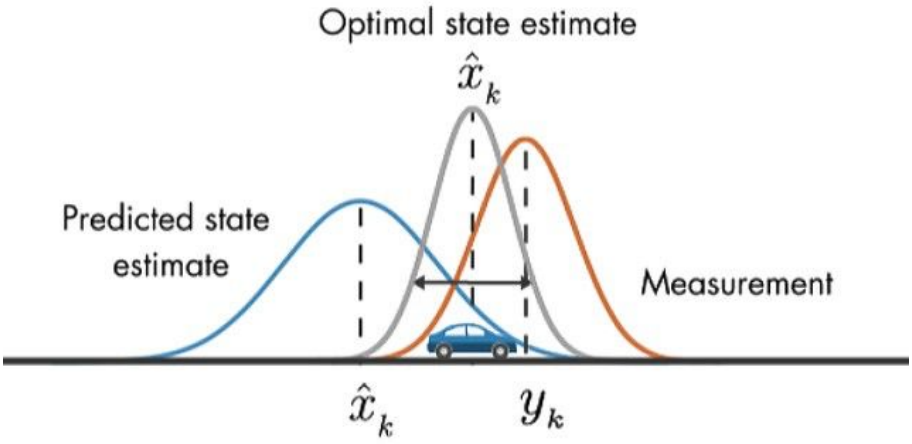


Fig 4.8 Optimal state estimation by Kalman

1) The Kalman filter uses noisy results of a system over some time for estimating the system parameters (some of which are unobservable) and predicting future results. At each time step -
    a) It makes a prediction.
    b) It takes in measurement.
    c) It updates itself based on how the prediction & measurement compare.

2) The LaneTracker class implements the Kalman filter for lane detection. Firstly, it initialises the State matrix & Measurement matrix size. Then, it calculates the transition matrix. White Gaussian noise was considered for our system. Using this noise model, the error was calculated for state and generated the predicted state using that noise measured as an estimator.

3) The variation of detected lines along the lanes is averaged out by adding up the measurement error and previous state by the Kalman filter algorithm. That's why detected lane marker lines are stable over time and by its predictor property using the previous state, at very low illumination condition, it is able to detect lanes by remembering the detected lanes from the previous video frame.

Testing 4

1) The new HD videos were recorded in better lighting conditions by ZED camera, converted them to .avi, and the Kalman filter was tested on it.

2) Especially those parts of IIT Delhi were recorded where lane marks were not present, or curves were there to test robustness.

3) Significantly greater results were observed with **absolutely 0** regions of no predicted lines. Thus the car will have a direction even if no lane marks present to some extent until the Kalman filter gets affected by extreme noise.

4) The image shown below justifies that even at turns of no lane marks, the red predicted lines gives the direction of the curve to be followed by the car.

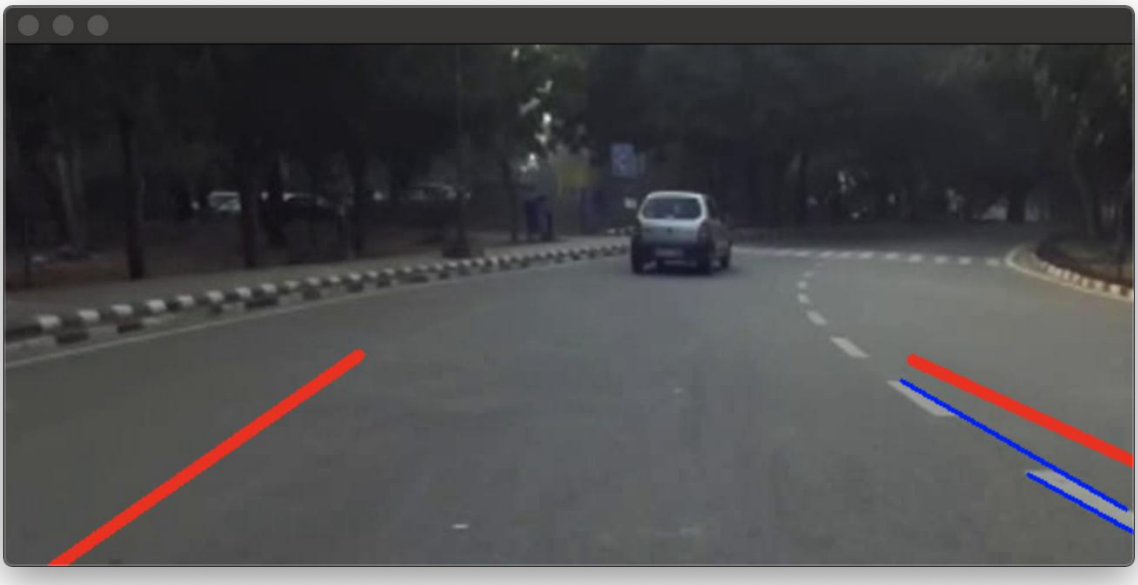


Fig 4.9 Karman filter on curves

Chapter 5: Lane Detection in Real-world

5.1 Zebra Crossing exclusion

1) The zebra crossings detected were eliminated to avoid its interference with lane detection.

2) This was done by ignoring lines that were horizontal, i.e., had an absolute angle less than 5 degrees with the horizon.

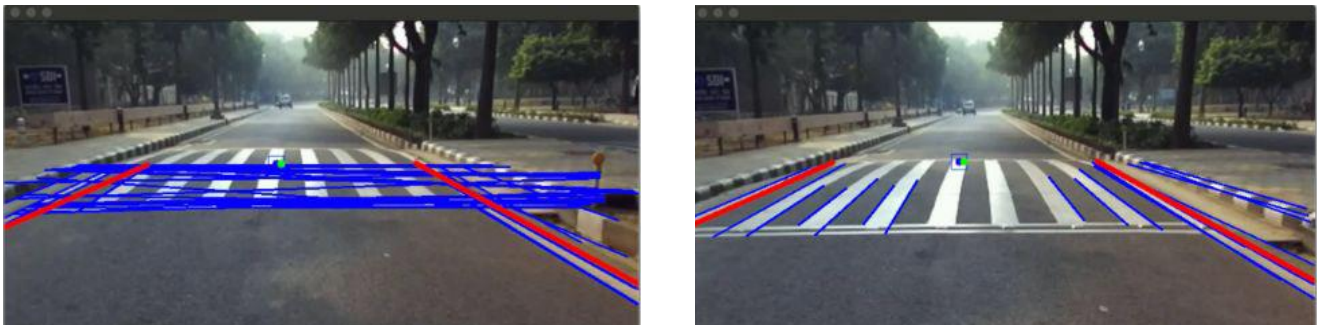
3) The noise observed in lane detection was reduced by ignoring lines that had an absolute angle difference greater than 20 degrees with the line predicted by the Kalman filter for both left and right boundaries of the lane detected.

Testing 6

1) The new HD videos were recorded in better lighting conditions by ZED camera, converted them to .avi, and the Kalman filter was tested on it.

2) Parts of IIT where zebra crossings were there were recorded to test robustness.

3) It was observed that zebra crossings had been successfully excluded and thus, the horizontal lines didn't deviate the predicted Kalman filter lines.



A **B**

Fig 5.1: Zebra crossing detection
A) before elimination B) after elimination

5.2 White colour detection

1) There were issues observed of detecting lines in shadows, footpaths and other obstacles that were interfering with our algorithm, giving incorrect values. To solve this, a colour detection algorithm was used in which a range of white colours were detected in the video captured and then only those areas were processed for lanes.

2) This gave better results as footpaths, and other areas were ignored. But this posed a few major issues -

    a) As the entire image is traversed pixel by pixel to find the white region, this reduces the speed of the algorithm significantly increasing the time of giving output. As a result, the car movement couldn't be handled.

    b) Also, the colour density of side lanes was dependent significantly on sunlight and also varied highly for different side lanes. Therefore the colour code tuned for a particular side lane will be of no use for other side lanes.

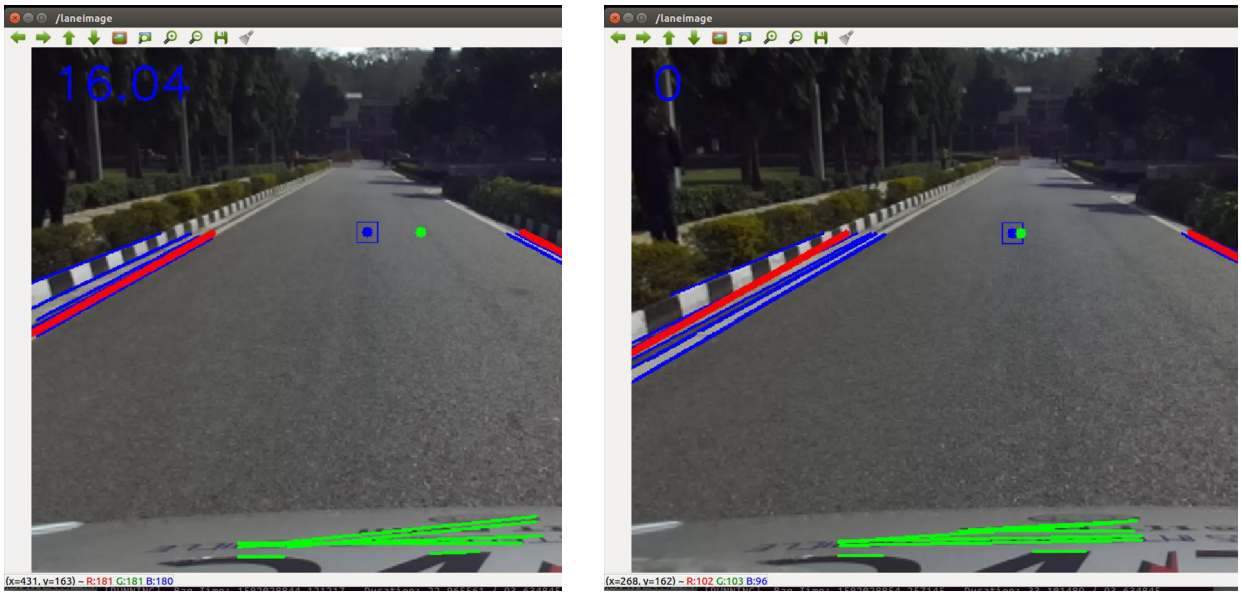
3) As a result, this approach was discarded because it was slow and dependent on sunlight and side lanes highly.

Testing 7

1) The recorded HD videos of the road in front of the security room were used, and the colour code was tuned for it. But this colour code failed for the previous video recorded behind the blocks and slowed the code.

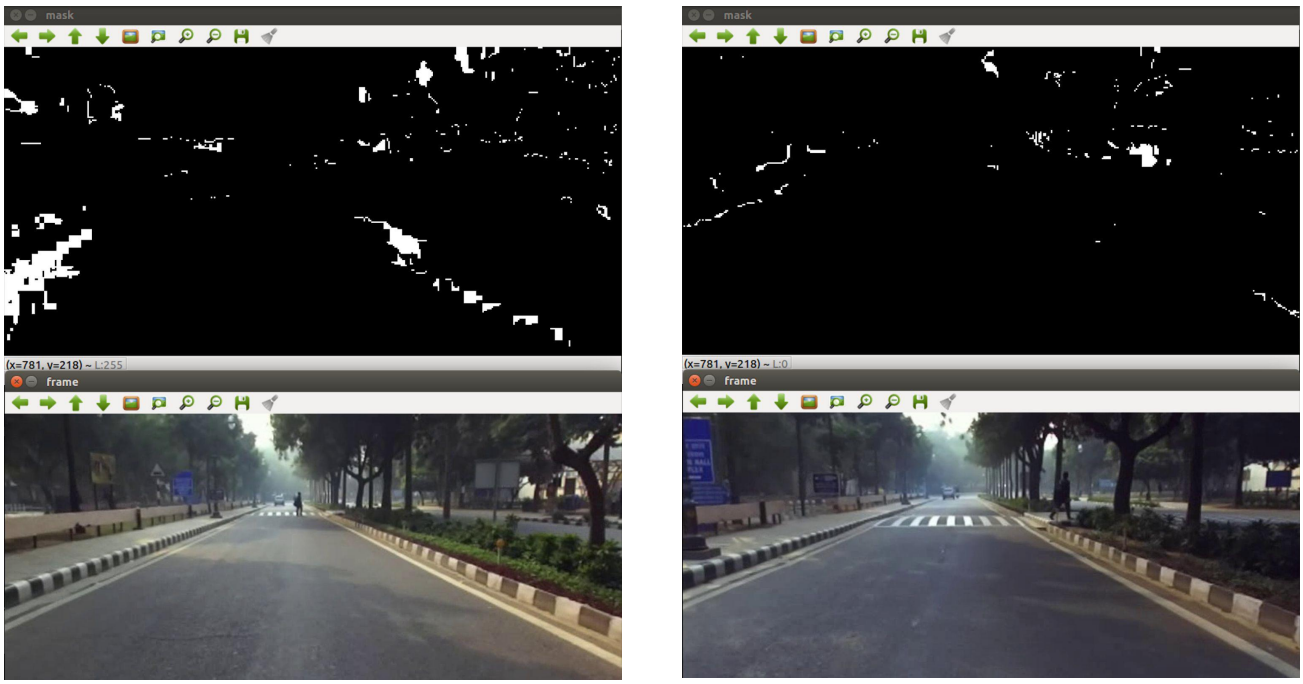
2) In the image shown below also, in Fig 5.2 A, white colour was detected on the left side of the lane, and the RGB values were (181,181,180). On the other white colour of the right side of the lane had RGB values (102,103,96), as shown in Fig 5.2 B.

3) In the image shown below also, in Fig 5.3 A, sidelines were detected well because of white colour detection, but the same was not detected 100 meters away on the same road as shown in Fig 5.3 B.



A **B**

Fig 5.2: Variation in white colour
A) Lane's left side B) Lane's right side



A **B**

Fig 5.3: White colour detection

Chapter 6: Control in Real-world

6.1 Motion Planning: Steering with instantaneous values

Knowing the exact location of lanes, the car needed to be steered so that it stays in the middle of the road. An algorithm was designed to give a steering angle to the car from the detected lanes.

In the figures below -

Green pointer - X coordinate = (ZED frame width) / 2 - (6.1)
- Y coordinate = (avg Y coordinate of lane lines) - (6.2)

Blue pointer - X coordinate = Avg[min(X coord of left lane) , max(X coord of right lane)] - (6.3)
- Y coordinate = (avg Y coordinate of lane lines) - (6.4)

The green pointer represents the heading direction (the dashcam or ZED camera is in the middle of the car), and blue is the desired direction of motion. Therefore the car needs to move in the direction that green follows the blue pointer. Ideally, they both should coincide.

$$\tan(\theta) = (x(\text{blue}) - x(\text{green})) / (\text{avg Y coordinate of lane lines}) \quad - (6.5)$$

A negative θ means a left turn, and a positive value means a right turn.
Once this θ is achieved, the rotation angle of the steering wheel is mapped with its value to get the angle that the steering wheel should rotate with.

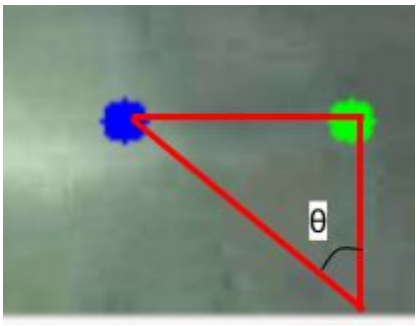


Fig 6.1 Calculation of steering angle

Testing 8

1) Tested the motion planning algorithm by subscribing to rostopic "/zed/stereo/image_rect_color" realtime, generated using the ZED Stereo camera mounted on the car.

2) This provided the image frame in the form of **img_msg,** which was converted to **cv2** form for further processing in OpenCV.

3) Fig 6.2, shows that the desired direction of motion is towards the left, as verified visually, and using the equation below, generated the angle of steering θ. A steering angle of -12.4 degrees (to the left) is given as output by the algorithm which is published to the car controller.

Fig 6.2 Steering angle generated on a curved road

4) On a straight road, both the points nearly overlap and the calculated angle is 0.51 degrees (Fig 6.3). To solve this, a minimum threshold was set for the car to make a decision; thus, it keeps moving straight in case of small errors.

Fig 6.3 Steering angle on a straight road

5) Once this angle was found, it was published on a rostopic “/steer” which then can be used later on by the autonomous car control to get the steering rotation angle.

Testing 9

1) A test of lane following was conducted on the road behind IIT Delhi blocks by integrating the steering angle with the control system of the car.

Fig 6.4 Live demonstration

6.2 Robust Heading Direction

The previous method of finding the heading direction was not robust as it was highly dependent on the width of the road. To solve this issue, the front part of the car was used as a reference for the heading direction. The frame was divided into 2 frames - lower and upper half by frame coordinates -

1) **Upper half** – Only the upper half of the frame was used for lane detection. This allowed the neglect of all the horizontal lines that are encountered from the front of the car. This is then used to predict the direction of motion. (blue lines)

2) **Lower half** – the lower half of the frame was used for detecting the horizontal line on the front of the car to detect the car centre. This provided the heading direction of the car. This heading direction of the car was independent of the width of the road and frame that was selected. (green lines)

Testing 10

1) This approach was tested to find the heading direction of the car (green dot) on roads of different width and displayed that it changed dynamically according to road width, making the system highly robust.

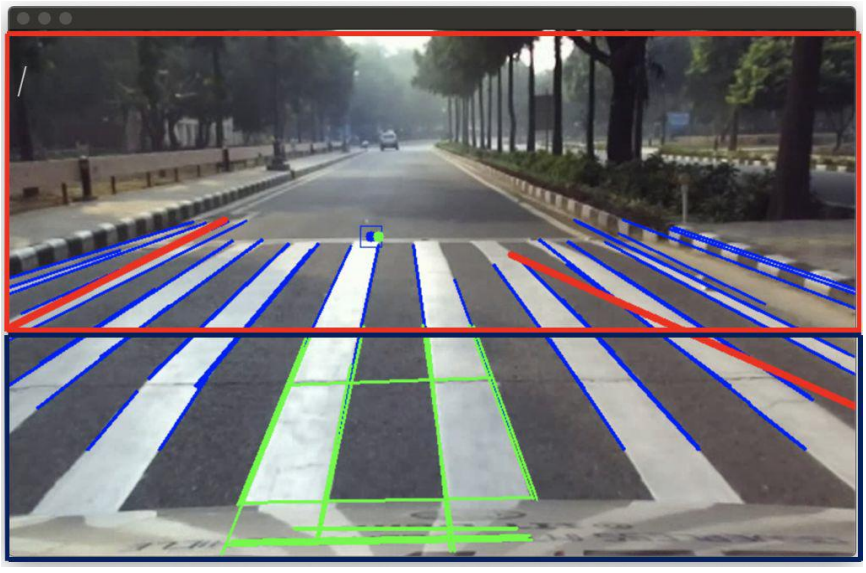


Fig 6.5 Divided Frames approach

2) But the centre of the car and the road that was calculated from this varied highly because of certain lines which were not identified correctly giving incorrect values. This made the car wobble a lot.

6.3 Control using One Side of Lane

After careful deliberation and discussions, it was decided to try a different approach for controlling the car using lane detection. The approach of one-sided lane detection was followed. Only the left side of the lane was selected, and the car was coded to always move at a fixed distance from the left side. In this way, the car was given corrections so that ideally it can move on an imaginary straight line parallel to lane marks on the left side.

But it posed some issues as it had a lot of variations and incorrect readings many times due to variations in the predicted line. It was also dependent on road width and was a problem when one side had no lane marks. It was also a problem at turns and roundabouts where only one side had lane marks.

Testing 11

1) This concept was tested on the pre-recorded videos. But a lot of variation in the values was observed.

2) Also, if the initial value detected was wrong, then the correction was an issue as there was no right side value to average out and eliminate the errors.

Fig 6.6 One-sided lane detection results

6.4 Minimizing Car wobbling using the Bounding box

In order to reduce the variations of the car heading direction and centre of the road, the values of centres that were more than a specified distance from the initial value were ignored as such a variation meant incorrect centre. But this only reduced the incorrect values and not the close variation in centres.

To solve this issue, a contour (20 * 20 pixels) was created around the centre of the road, and only a steering angle was published for the car if the centre of the heading direction was outside the contour. This reduced the centres robust to small deviations in values, but after a certain value, it varied highly.

Also, in this approach, if the centre is in some case stored at a side value, then its correction was an issue as the initial value was wrong. So there was no correct basis to measure the distances.

Testing 12

1) This was tested on the recorded videos, and it was observed that this bounding box could reduce the steering wobbling for very small deviations giving better control of the car.

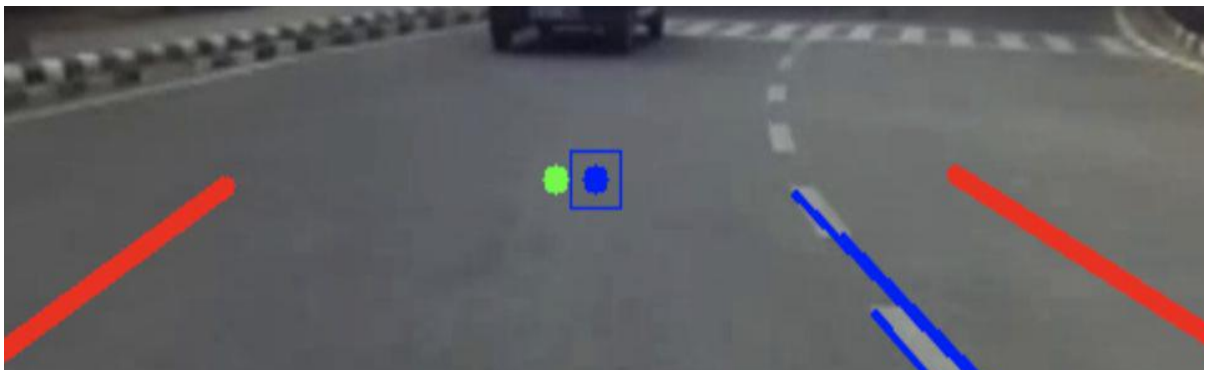


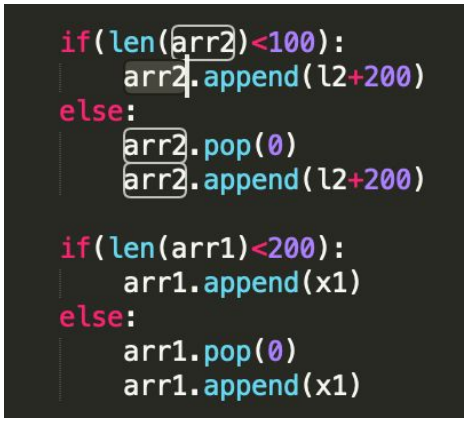
Fig 6.7 Bounding Box

6.5 Minimizing Car wobbling using Average values

In order to reduce sudden variations in centre detection, the x coordinate of the centre of the road (blue dot) and the heading direction of the car (green dot) were stored in 2 separate arrays of length 200 and 100 respectively.

When there was no sudden deflection in the detected value, the array was updated by adding the new detected value and removing the oldest value in the array. Then the average of all the values was used in both the arrays for control instead of the instantaneous values.

This posed a problem when the points deviated, and wrong values were inserted in arrays. As the average value was considered, it took a while for the points to be corrected. To solve this, if wrong values are inserted in arrays giving a turning angle of more than 10 degrees instantaneously, all the values of an array are flushed to start a new set of values, so that incorrect values are not used for averaging.



```
if(len(arr2)<100):
    arr2.append(l2+200)
else:
    arr2.pop(0)
    arr2.append(l2+200)

if(len(arr1)<200):
    arr1.append(x1)
else:
    arr1.pop(0)
    arr1.append(x1)
```

Fig 6.8 Code for using Arrays as Queues

$$x\_avg = sum(arr1)/len(arr1) \quad -(6.6)$$

$$l2\_avg = sum(arr2)/len(arr2) \quad -(6.7)$$

$$dif = l2\_avg - x\_avg \quad -(6.8)$$

Testing 13

1) This concept was tested on the car in front of the security room, IIT Delhi. The car which was correcting itself was able to move on a straight road for 100 meters.

2) The correction was a problem when any person or shadows came on the way. This was because of the extremely slow speed of the car, which gave a lot of time for the errors to get accumulated and give wrong corrections.

3) When the car speed was increased, the car was able to cover more distance as it moved past some disturbances like shadows very quickly and the problem did not aggregate. But because of the limit of the car speed in the real world, the plan is to test and debug further in the Gazebo simulator.

Fig 6.9 Mahindra e2o car live testing

6.6 Rejecting the Bounding Box Method

In order to reduce the wobbling nature of the correction of the heading direction of the car, earlier the bounding box approach was adopted. But, it was observed that this was leading to the car direction not being corrected completely and the car instead kept

moving at a non-zero very small angle with the heading direction. This eventually led to a significant deviation in the long run of the car.

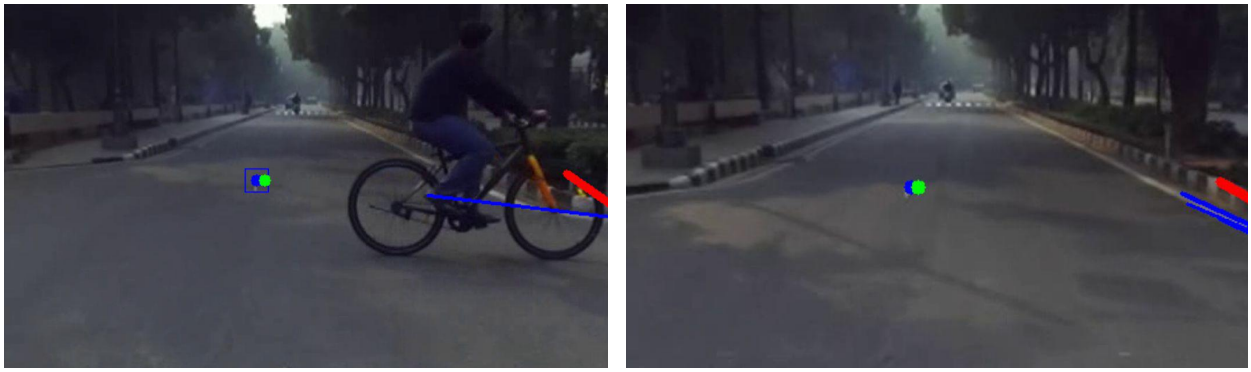
Hence, to overcome this error in control in order to prevent the car from moving at a non zero small angle, the bounding box was removed, and only the average value method was continued for the minimization of wobbling.

Testing 14

1) The recorded HD videos of the test run were used to test correction without the bounding box.

2) It was observed that although the car kept on getting very small correction angles continuously, the issue of very small angles the car made with the heading direction were solved and it coincided perfectly with the path direction.

3) In Fig 6.10 A, the heading direction does not exactly coincide with the centre of the lane, thus having a small angle. But in Fig 6.10 B, both coincide exactly having a 0-degree angle with each other.



A B

Fig 6.10: Rejection of bounding box
A) With bounding box B) Without bounding box

6.7 Way Point Navigation

In order to further improve the control, it was decided to incorporate a waypoint navigation technique in which for giving any steering angle to the car, a set of points were considered rather than a single point.

1. At the initial point, the path that was visible in the frame was considered, and 10 centre points (blue) of the path which are to be followed were stored in an array.
2. Also at the same “y” coordinate of the 10 path points, the 10 points (green) showing car direction were stored in another array.
3. Then the 10 angles were calculated from those 10 path and direction points and stored in a separate array.
4. As soon as the car reaches the stored “y” coordinates of a point, the car is given the angle it should steer which was calculated from the point just ahead of the point. This correction was based on the initial position of the car only. New steering values are calculated and used only after all the previous 10 values were used and the car reaches a point which was not in the view of the car earlier. This prevented the deviation of the car from the intermediate values and helped the car follow a path not only a point.
5. A time delay was taken assuming this is the time taken by the car to reach the next point from the starting point.

Testing 15

1) The recorded HD videos of the test run were used to test the waypoint navigation method.

2) It was observed that 10 angles were calculated and they were getting published after a fixed time delay only after the car reached the next waypoint.

3) Fig 6.11 shows the waypoints, and in the terminal, the same angle is visible being printed 10 times with a delay of 0.1 seconds and after 1 second when it is assumed that the car reaches the next point, the next angle is published.

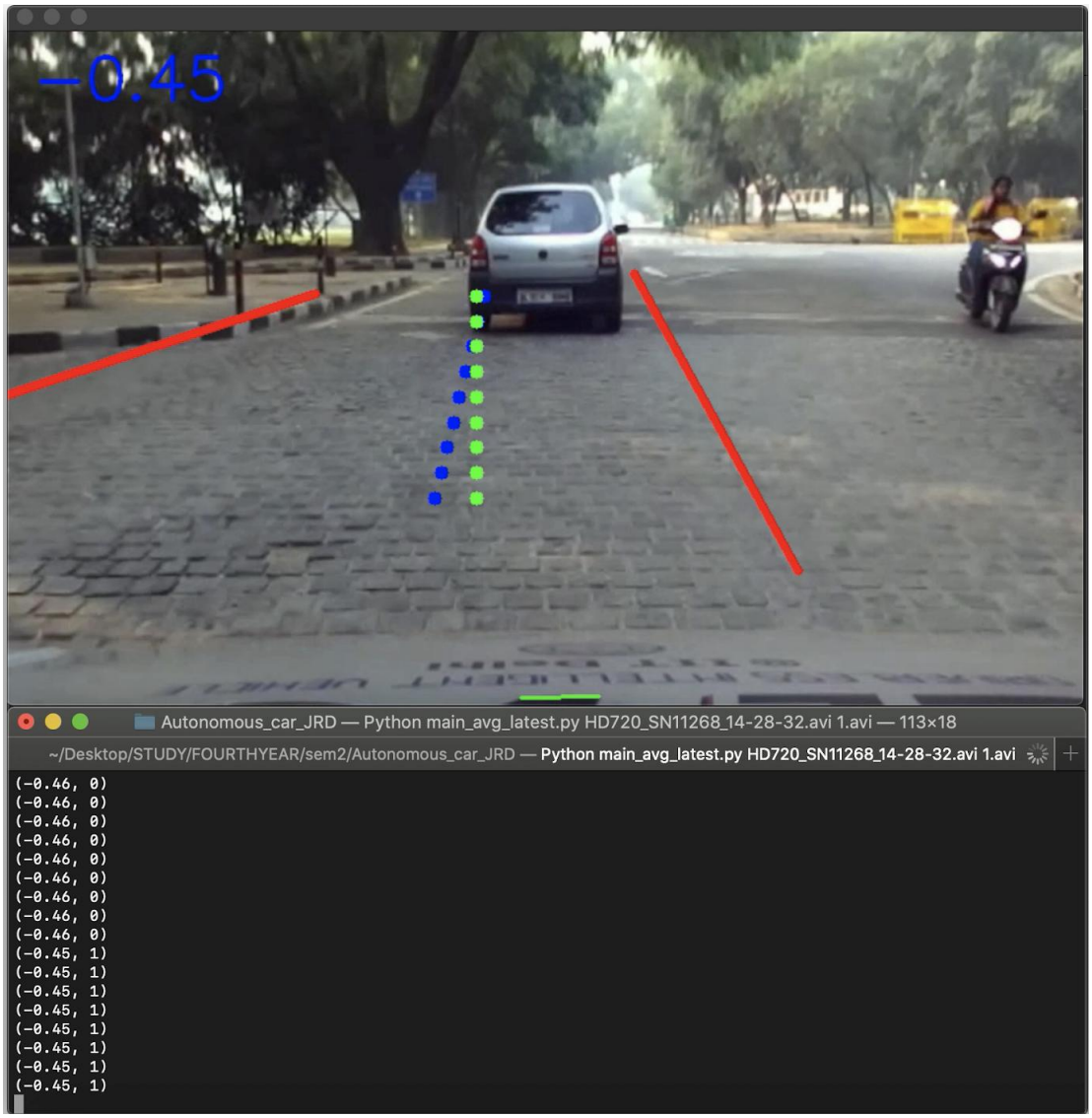


Fig 6.11: Waypoint Navigation method

6.8 PID control implementation

In order to improve the control over the car, PID control was implemented on the steering angle (θ) calculated by the system. This is a control loop mechanism employing feedback. It calculates the error as the difference between the desired direction and heading direction of the car. The time of 1 loop is taken as dt. kp is used for proportional control,, and it is deduced using kp and θ. ki is used for integral control, and it is deduced using ki, θ, and added error. kd is used for derivative control, and it is deduced

using kd, θ, and the added error. The sum of the three terms is returned as the output of the PID function. This returned value is published as the steering angle of the car.

```
def pid(dif):
    global sum1
    global elapsed_time
    global diff
    global prev_err
    global kp
    global ki
    global kd
    global check
    #print(dif)
    #print("f",sum1)
    if(abs(dif)>0.1):
        if(check==True):
            deltaT=0.1
            check=False
        else:
            deltaT = (time.time()-elapsed_time)
        #print("dt",deltaT)
        #print("dif",dif)
        elapsed_time=time.time()
        sum1 = sum1 + dif * deltaT;
        diff = (dif - prev_err) / deltaT;
        #print("sum",sum1)
        #print("dif",dif)
        prev_err = dif
        output = (kp * dif + ki * sum1 + kd * diff);
        return output
    else:
        sum1=0
        check=True
        return 0
```

Fig 6.12 Code for PID control

The PID parameters used in the code for moving car in the simulator are -

$$kp = 0.00000001 \quad \text{- (6.9)}$$

$$ki = 0.05 \quad \text{- (6.10)}$$

$$kd = 1 \quad \text{- (6.11)}$$

These parameters were tuned using the repeated runs done in the Carla simulator and are expected to run in the real world with minimal changes.

Chapter 7: Implementation using ROS

7.1 Input from ZED camera

The ZED camera was connected to the system, and the ZED wrapper libraries were installed in the system. The NVIDIA drivers and CUDA 10 were also installed which were needed for the ZED camera to run in the system

The **zed.launch** file was launched in the system that starts all the rostopics of the ZED camera and sets up communication between the ZED camera and the system.

roslaunch zed_wrapper zed.launch -(7.1)

The rostopic - "**/zed/zed_node/stereo/image_rect_color"** was used. This rostopic publishes the stereo image captured by the ZED camera which can be subscribed to get the input from the ZED camera.

7.2 Input to python code

This image rostopic was taken as the input to the python code. A node - **"autonomous_car"** was initiated. Using the node, the rostopic - "**/zed/zed_node/stereo/image_rect_color"** was subscribed.

rospy.init_node('autonomous_car', anonymous=True) -(7.2)

The input image was given as a ROS message. To process it using OpenCV, it was converted into cv2 type which can then be processed for lane detection.

frame = self.bridge.imgmsg_to_cv2(data,'bgr8') -(7.3)

The frame was processed according to the need to remove all the disturbances due to objects on the sides of roads.

7.3 Output from python code

Once the image was processed, and the steering angle was successfully found using the control methods as discussed in chapter 5, it is to be published to the car so that the steering wheel can rotate through the specified angle.

A rostopic - **"/steer"** of Float32 data type was started. The steering angle was published on the rostopic.

Another python file designed to take input from a PS4 controller was created. A node was initiated in it to subscribe to rostopic - **"/steer"**, converting the value into Integer data type, and publishing it to rostopic - **"e2octrl"**.

7.4 Input to e2o Mahindra Car

This rosoptic - **"e2octrl"** has the message type - **"e2o_ctrl"**. This message is a customized ROS message having the following variables -

string RNDB
uint8 Indicator
uint8 Lamp
int8 Steer
uint8 Brake
uint8 Accel
uint8 Wiper
uint8 Horn

Fig 7.1 e2o_ctrl ROS message

All the other variables are input from PS4 joystick except Steer. Steer is assigned the value of steering angle which is published by the main python code by rostopic - **"/steer"**. This is done in the file - **"ps4_lane_e2o_ctrl"**.

The car then, through Controller Area Network (or CAN) communication linked all of the electronic systems within a car together to allow them to communicate with each other. Through this car moved according to commands given by the user and vision code.

Chapter 8: Lane Detection in CARLA Simulator

8.1 Frame and Car Heading Direction Detection

1) The coordinate system of the frame in CARLA was different from the ZED camera frame. Therefore the coordinates of the cropped frame were changed.

2) Since CARLA is an ideal system, there was no image of the car front to use to find the heading direction of the car.

3) Hence the centre of the frame was used as the heading direction due to the placement of the camera at the centre of the car.

Testing 16

1) The control and lane detection were tested in the CARLA simulator in real-time. The centre of the frame was successfully detected, and the frame width was changed according to the road width.

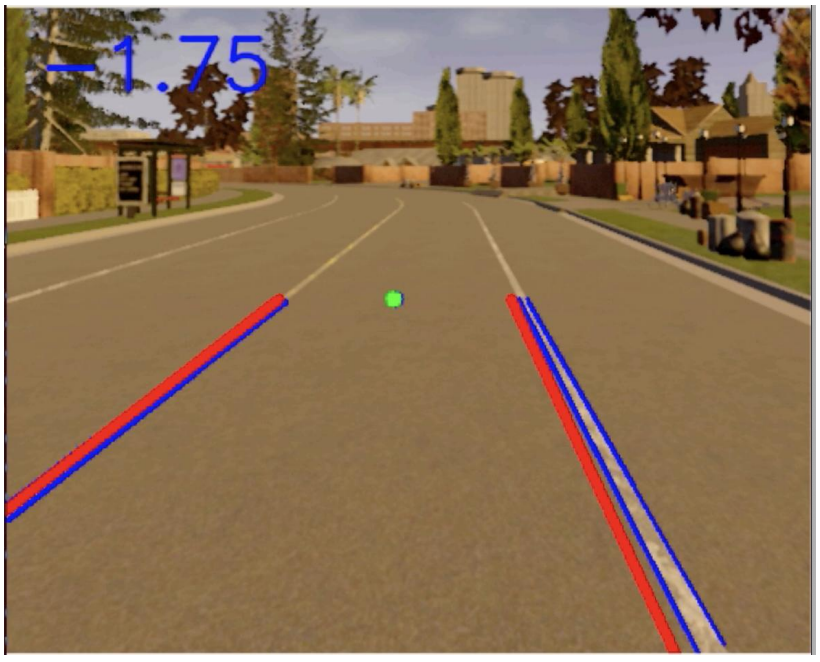


Fig 8.1: Heading direction by green colour

Chapter 9: Control in CARLA

9.1 Motion Planning: Steering in the simulator

Knowing the exact location of lanes, the car needed to be steered so that it stayed in the middle of the road. An algorithm was designed to give a steering angle to the car from the detected lanes.

In the figures below -

Green pointer - X coordinate = (frame width) / 2 - (9.1)

- Y coordinate = (max Y coordinate of lane lines) - (9.2)

Blue pointer - X coordinate = Avg[(X coord of left lane at max Y coord) , max(X coord of right lane max Y coord)] - (9.3)

- Y coordinate = (max Y coordinate of lane lines) - (9.4)

The green pointer represents the heading direction (centre of frame width) and blue is the desired direction of motion from lane detection. Therefore the car needs to move in the direction that green follows the blue pointer. Ideally, they both should coincide.

$$\tan(\theta) = (x(\text{blue}) - x(\text{green})) / (\text{max Y coordinate of lane lines}) \quad \text{- (9.5)}$$

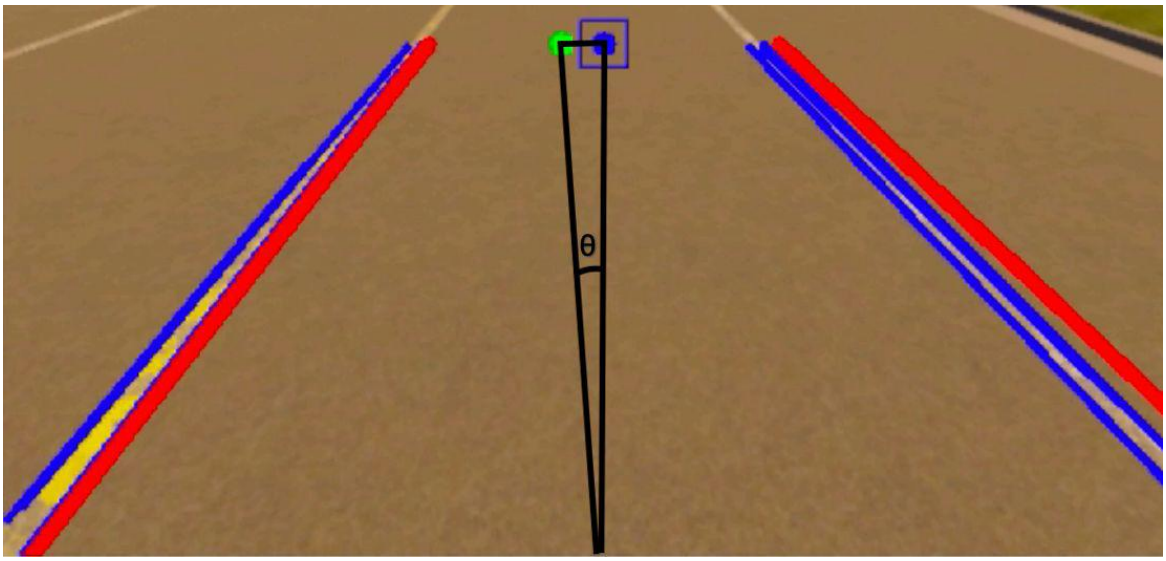


Fig 9.1 Calculation of steering angle

Testing 17

1. The control and lane detection were tested in the CARLA simulator in real-time. The lanes were detected very well in CARLA, but numerous issues were found with the control.
2. The car always spawned at random points and many times not oriented in the correct angle facing towards the roads. Many times it spawned at points where there were no lane marks.
3. As a result, the control was not very accurate as if the car started from an off-centre point, then the car was not able to move accurately.

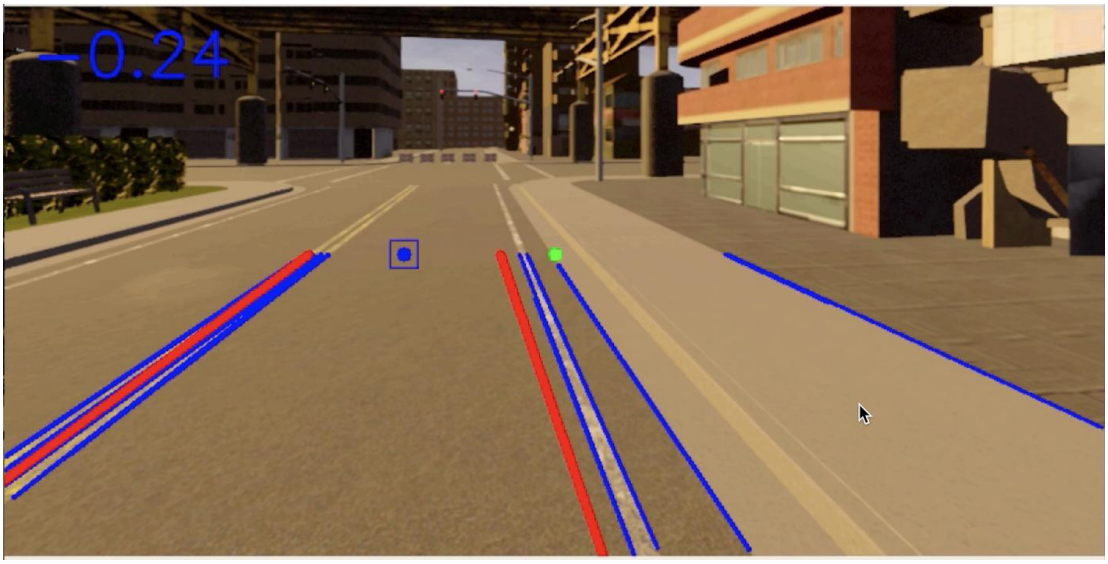


Fig 9.2 Car spawned at an off-centre location

9.2 PID control implementation

In order to improve the control over the car, PID control was implemented on the steering angle (θ) calculated by the system. This is a control loop mechanism employing feedback. It calculates the error as the difference between the desired direction and heading direction of the car. The time of 1 loop is taken as dt. kp is used for proportional control, and it is deduced using kp and θ. ki is used for integral control, and it is deduced using ki, θ, and added error. kd is used for derivative control, and it is deduced

using kd, θ, and the added error. The sum of the three terms is returned as the output of the PID function. This returned value is published as the steering angle of the car.

```
def pid(dif):
    global sum1
    global elapsed_time
    global diff
    global prev_err
    global kp
    global ki
    global kd
    global check
    #print(dif)
    #print("f",sum1)
    if(abs(dif)>0.1):
        if(check==True):
            deltaT=0.1
            check=False
        else:
            deltaT = (time.time()-elapsed_time)
        #print("dt",deltaT)
        #print("dif",dif)
        elapsed_time=time.time()
        sum1 = sum1 + dif * deltaT;
        diff = (dif - prev_err) / deltaT;
        #print("sum",sum1)
        #print("dif",dif)
        prev_err = dif
        output = (kp * dif + ki * sum1 + kd * diff);
        return output
    else:
        sum1=0
        check=True
        return 0
```

Fig 9.3 Code for PID control

The PID parameters used in the code for moving car in the simulator are -

$$kp = 0.00000001 \quad \text{- (9.6)}$$

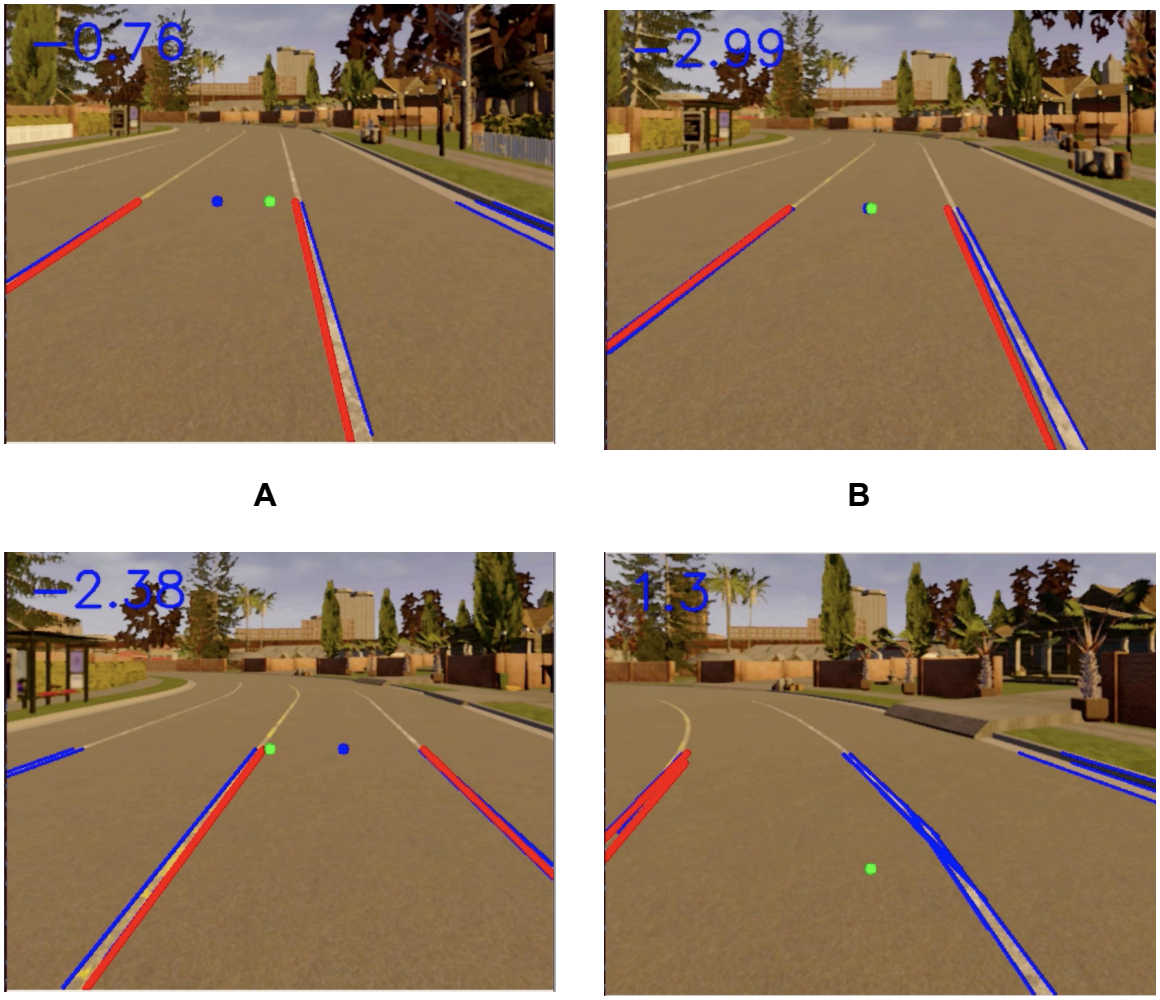
$$ki = 0.05 \quad \text{- (9.7)}$$

$$kd = 1 \quad \text{- (9.8)}$$

These parameters were tuned using the repeated runs done in the Carla simulator and minimising the wobbling.

Testing 18

1. By spawning the car at separate random locations, the coordinates of the spawning location which had the car at the centre of the lane marks and had a long straight road with a curve at its end were determined.
2. The PID control was tested in real-time in CARLA.
3. The car was able to move on a straight road, but with very high wobbling which eventually led to the car losing its path as one of the side lanes got out of the field of view of the car.



A **B**

C **D**

Fig 9.4 A) frame at start B) frame after 9 seconds C) frame after 17 seconds D) frame after 34 seconds

9.3 Improving PID control using the Bounding box

In order to improve the PID control, minimise wobbling, and minimize the effect of proportional control when the θ is in the range of -0.1 to 0.1, a bounding box method was used. If the green point is in the bounding box around the blue point, the θ is in the range of -0.1 to 0.1. Any steering angle during that duration is ignored, and 0 is published. If it is not in the range, then the PID control is applied, and the output is returned, which is then published to the car.

This ensures that if the car is given a command to turn right for correction and once its heading direction coincides with the centre of the lane, it turns further right by a very small angle thus not having a large deviation from the centre. This ensured the minimum wobbling of the car.

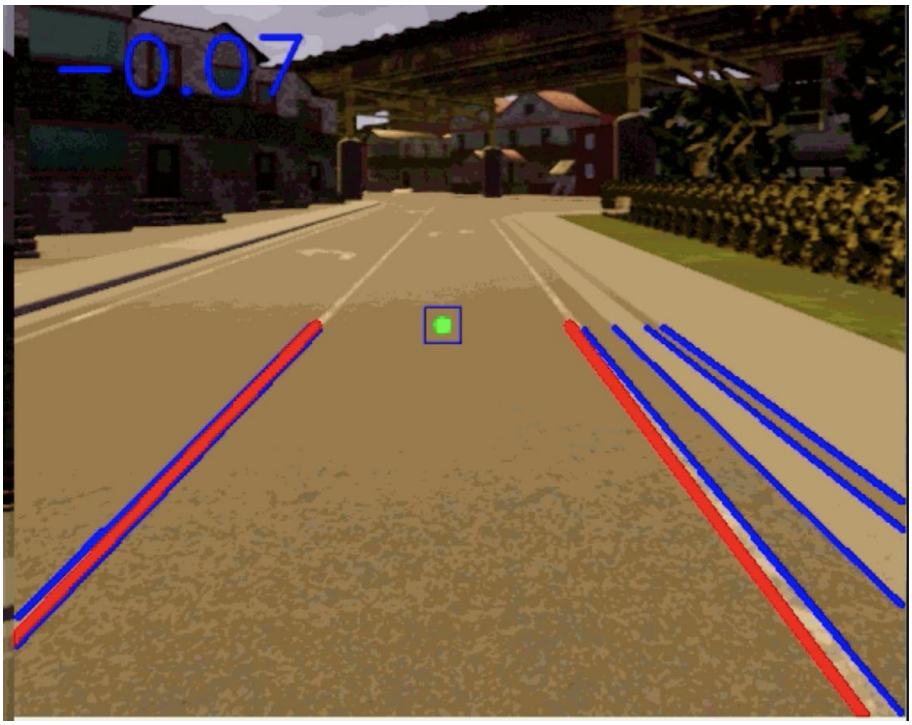


Fig 9.5 Bounding box method

Testing 19

1. The bounding box method, along with PID control, was tested in real-time in CARLA by making the car move on a straight road at a speed of 3 m/s.
2. The wobbling of the car was successfully minimised. The car moved on the straight road with minimal deviation.
3. In the curved road, when there were errors in the Kalman filter due to the curved lanes, the car got sudden random values publishing extreme turning angles. This caused problems with the PID control and the car movement on curved roads.

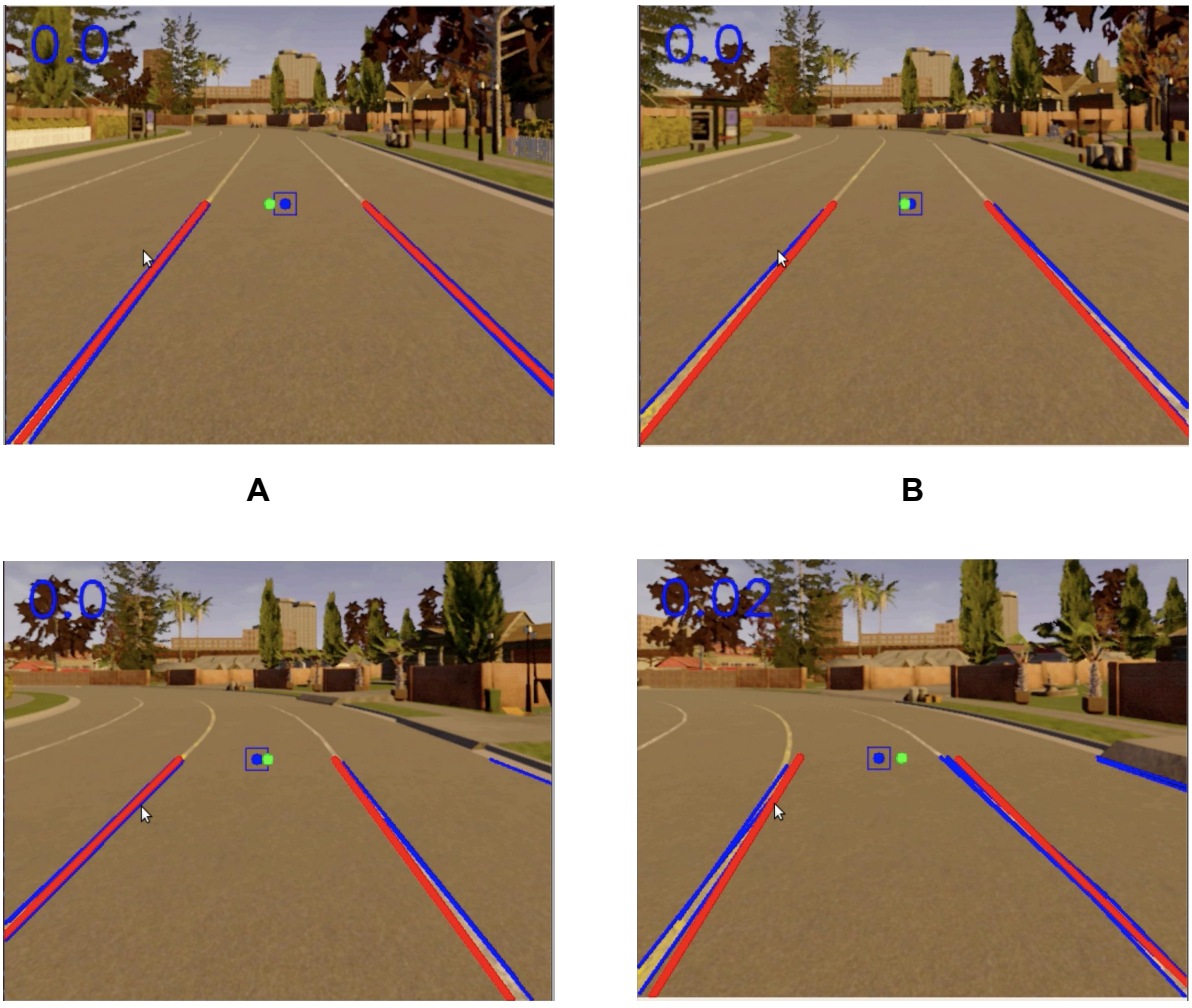


Fig 9.6 A) frame at start B) frame after 15 seconds C) frame after 25 seconds D) frame after 40 seconds

Fig 9.7 Error in Kalman Filter due to curve

9.4 Reducing error in the following direction

For curves, when other lines are detected, and there is an error in the target to be followed, the values of the direction to be followed having a sudden change more than the threshold (10 pixels) were ignored. This reduced the incorrect values coming in the direction to be followed.

```
if(abs(int(x)-prev)>10):
    x=prev
else:
    prev=x
```

Fig 9.8 Code for minimizing error

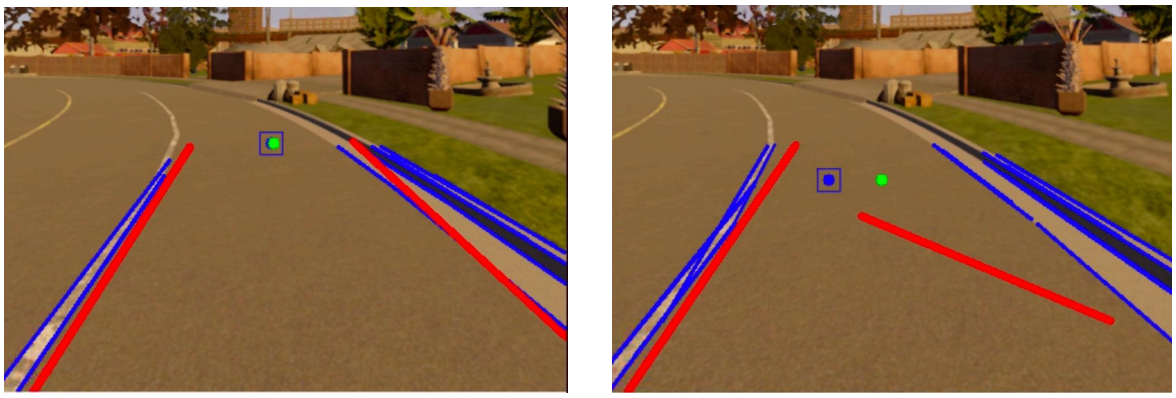


Fig 9.9 Ignoring values more than 10 pixels
A) frame before error B) frame during error

A sudden deviation is observed in the value of direction to be headed because of the curve in the road in Fig 9.8 B. This error is greater than the threshold and hence ignored.

9.5 Waypoint Navigation

A waypoint navigation technique was incorporated in which for giving any steering angle to the car, a set of points were considered rather than a single point.

1. At the initial point, the path that was visible in the frame was considered, and 10 centre points (blue) of the path which are to be followed were stored in an array.
2. Also at the same “y” coordinate of the 10 path points, the 10 points (green) showing car direction were stored in another array.
3. Then the 10 angles were calculated from those 10 path and direction points and stored in a separate array.
4. As soon as the car reaches the stored “y” coordinates of a point, the car is given the angle it should steer which was calculated from the point just ahead of the point. This correction was based on the initial position of the car only. New steering values are calculated and used, only after all the previous 10 values were used and the car reaches a point which was not in the view of the car earlier. This prevented the deviation of the car from the intermediate values and helped the car follow a path not only a point.
5. A time delay was taken assuming this is the time taken by the car to reach the next point from the starting point.

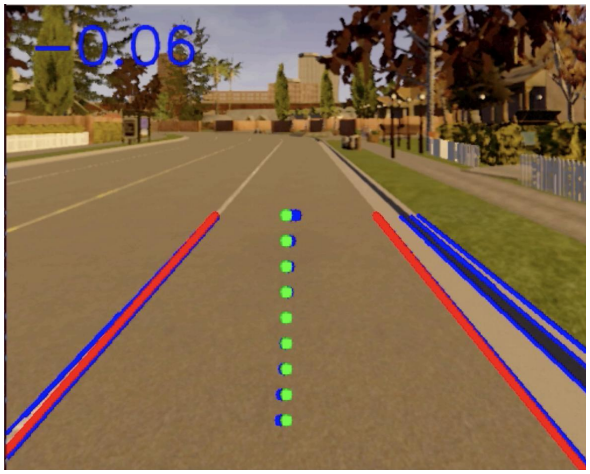


Fig 9.10 Waypoint navigation method

Chapter 10: Implementation using ROS in CARLA

10.1 Running Carla and Spawning Vehicle

The CARLA simulator needs to be launched to work in .

/CarlaUE4.sh - (10.1)

The ego_vehicle with the camera added to it also needs to be launched to test the algorithms.

source /opt/carla-ros-bridge/melodic/setup.bash - (10.2)
roslaunch carla_ego_vehicle carla_example_ego_vehicle.launch - (10.3)

10.2 Running ROS bridge

In order to control the vehicle in Carla using ROS topics, ROS bridge library was used. This started all the available ROS topics related to CARLA and the spawned vehicle. The ROS bridge enables two-way communication between ROS and CARLA. The information from the CARLA server is translated to ROS topics. In the same way, the messages sent between nodes in ROS get translated to commands to be applied in CARLA.

source /opt/carla-ros-bridge/melodic/setup.bash - (10.4)
roslaunch carla_ros_bridge carla_ros_bridge.launch - (10.5)

10.3 Input from the camera

The camera was added to the car and the ROS bridge was launched to publish the image captured by the car to a rostopic.
The rostopic - **"/carla/ego_vehicle/camera/rgb/front/image_color"** was used. This rostopic publishes the stereo image captured by the camera which can be subscribed to get the input from the camera.

10.4 Input to python code

This image rostopic was used as the input to the python code. A node - **“autonomous_car”** was initiated. Using the node subscribed to rostopic - **“/carla/ego_vehicle/camera/rgb/front/image_color”**.

rospy.init_node('autonomous_car', anonymous=True) - (10.6)

The input image was received as a ROS message. To process it using OpenCV, it was converted into cv2 type which can then be processed for lane detection.

frame = self.bridge.imgmsg_to_cv2(data,'bgr8') - (10.7)

The frame was cropped according to the need to remove all the disturbances due to objects on the sides of roads.

10.5 Output from python code

Once the image was processed and the steering angle was found using the control methods as discussed in chapter 7, it needed to be published to the car, so that the steering wheel can rotate through the specified angle.

A rostopic - **“/carla/ego_vehicle/vehicle_control_cmd”** having a message type of **“CarlaEgoVehicleControl()”** with a component named **“steer”** of Float32 data type was used. The steering angle was published on the rostopic.

msg = CarlaEgoVehicleControl() - (10.8)
msg.steer = round(deg,2) - (10.9)

A rostopic - **“/carla/ego_vehicle/twist_cmd”** having a message type of **“Twist()”** with a vector component named **“linear”** having **“x”** as one of the components as of Float64 data type was used. The linear speed was published on the rostopic.

msg2 = Twist() - (10.10)
msg2.linear.x = 1.5 - (10.11)

10.6 Input to ego vehicle

The rosoptic - **“/carla/ego_vehicle/vehicle_control_cmd”** has the message type - **“/CarlaEgoVehicleControl”** having msg type -

```
uint32 seq
time stamp
string frame_id
float32 throttle
float32 steer
float32 brake
bool hand_brake
bool reverse
int32 gear
bool manual_gear_shift
```

Fig 10.1 CarlaEgoVehicleControl ros message

This rosoptic - **“/carla/ego_vehicle/twist_cmd”** has the message type - **“Twist()”** having msg type -

```
Vector3  linear
    float64 x
    float64 y
    float64 z
Vector3  angular
    float64 x
    float64 y
    float64 z
```

Fig 10.2 Twist ros message

Chapter 11: Importing Mahindra e2o Car in CARLA

11.1 3-D modelling of Mahindra e2o car

Mahindra e2o car 3-D model in .dae extension was used. It was imported in Blender and all the bones and meshes were integrated and exported as a .fbx file. Origin of the car was fixed at the COM of the car. Also, the origin of all the 4 individual wheels were fixed at the COM of respective wheels.

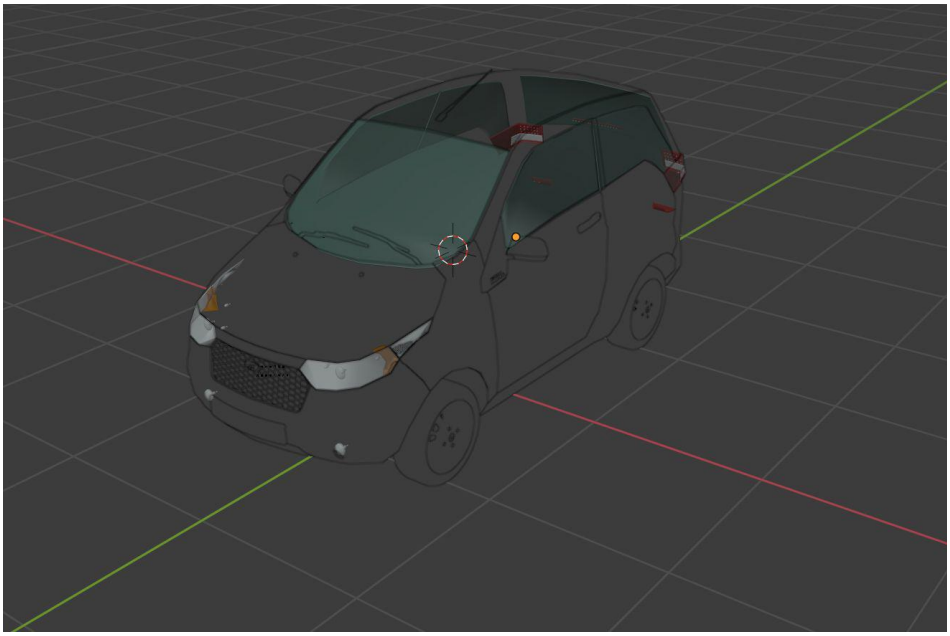


Fig 11.1 e2o model in Blender

11.2 Adding armature to Car and wheels

Skeleton of 4 wheeled vehicle was downloaded and imported as VehicleSkeleton.fbx. It has the 5 bones (armature) of wheels and the car body that were to be added to their respective COM's. The bones were bound to corresponding meshes. These meshes were then exported as a .fbx file.

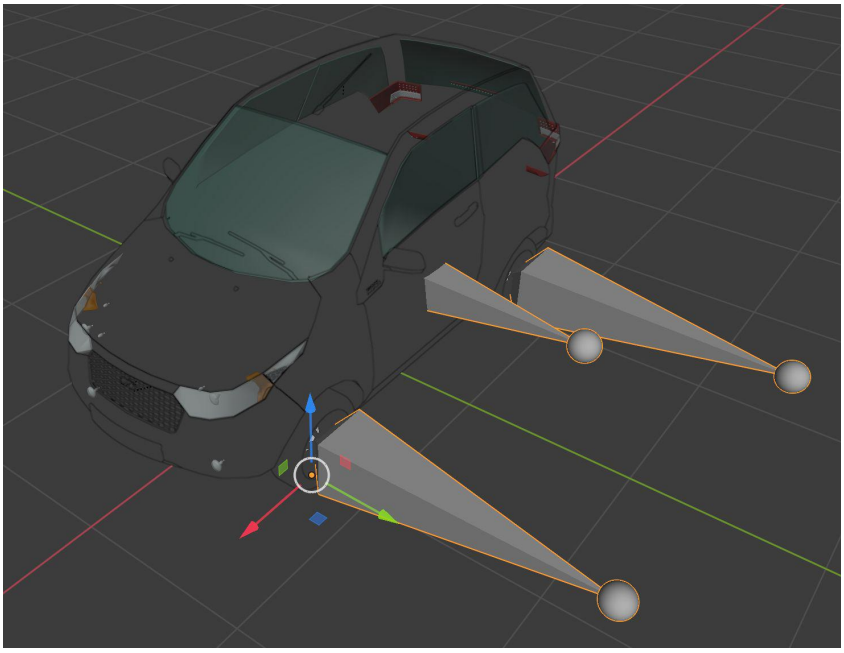


Fig 11.2 Armature added to COM's of wheel and body

11.3 Setup of Unreal Engine

The Unreal Engine was set up in the system. A new folder with the name "e2o" was created and in it the .fbx model of the car created in the above step was imported as a skeletal mesh.

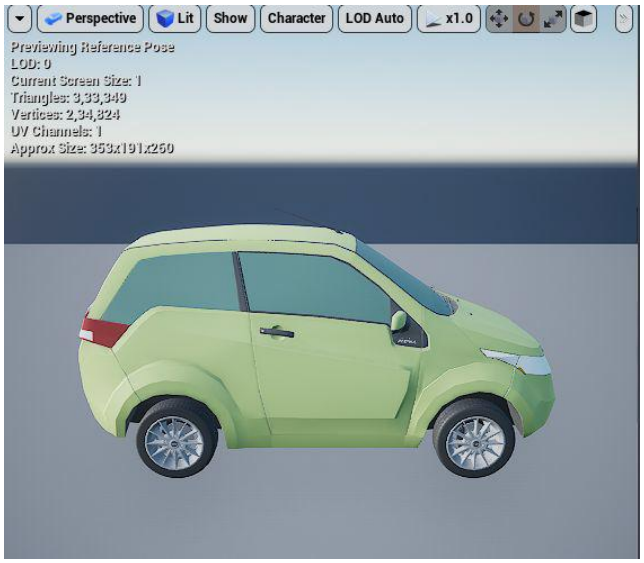


Fig 11.3 e2o model imported as skeletal mesh in Unreal Engine

11.4 Physical Assets of the e2o Car

Physical assets of the car were set to set the colliders.

1. For each wheel's collider, a spherical shape was selected and was adjusted to the shape of the wheel.
2. Physical type of each wheel was set to **"kinetic"**.
3. A general collider for the car was set with the shape of a box (cuboid).
4. ***Simulation Generates Hit Event*** was enabled for all of the physics' bodies.

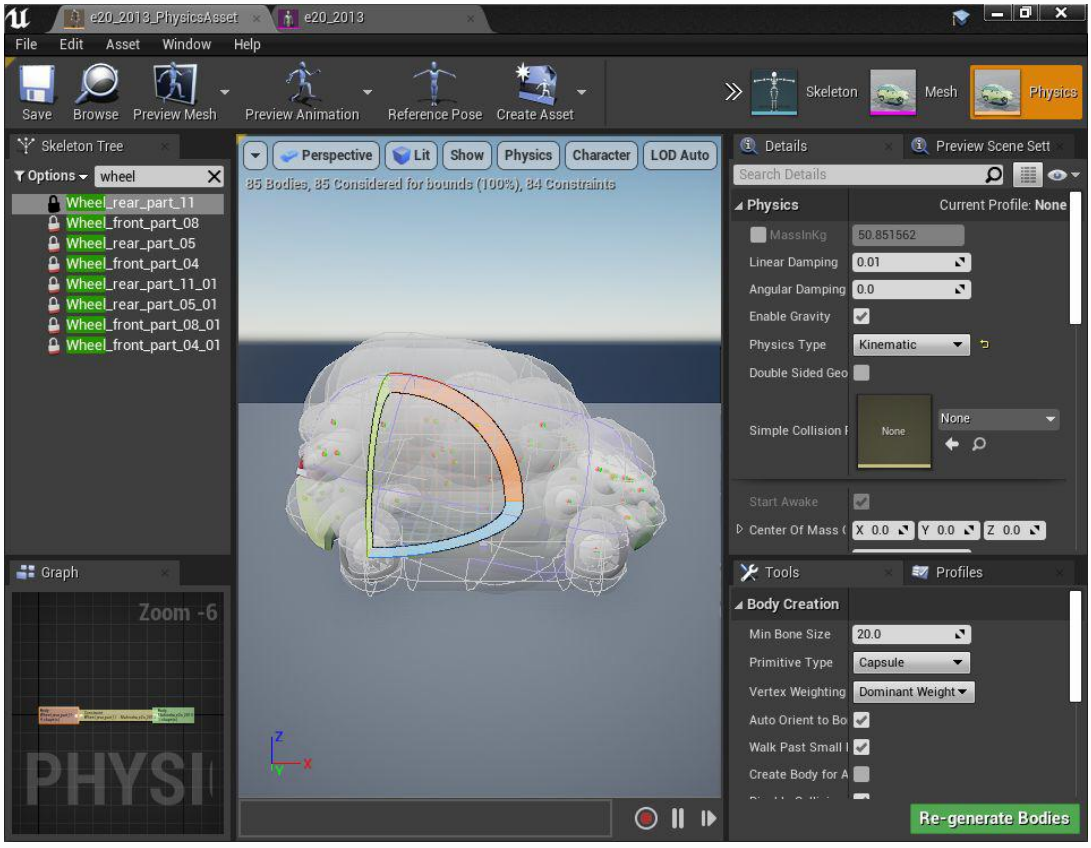


Fig 11.4 Physical assets of e2o car

11.5 Animation Blueprint creation

Develop two blueprints derived from VehicleWheel, one named wheel_f and the other wheel_b. These should have the exact shape and radius of the wheel. On wheel_f set steer angle 70 as default and set rear angle 0 as default. This would create the bp_e2o. Now compile the blueprints.

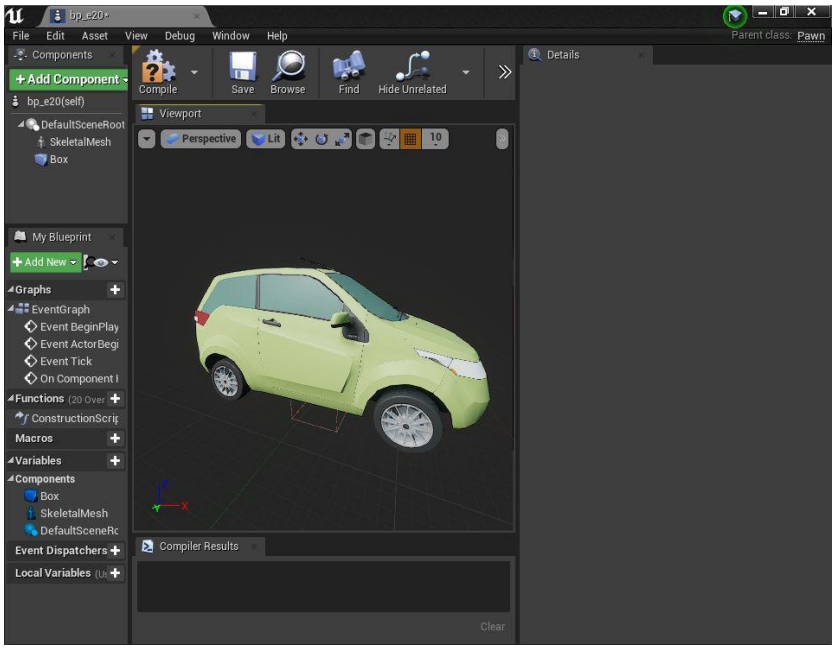


Fig 11.5 e2o model Blueprint

11.6 Set Manual Control

Open Game settings and add the following buttons for controlling the throttle and steering angle of the engine.

W = Forward

S = Reverse

A = Left

D = Right

Then open the blueprint event graph of the e2o car and add the control.

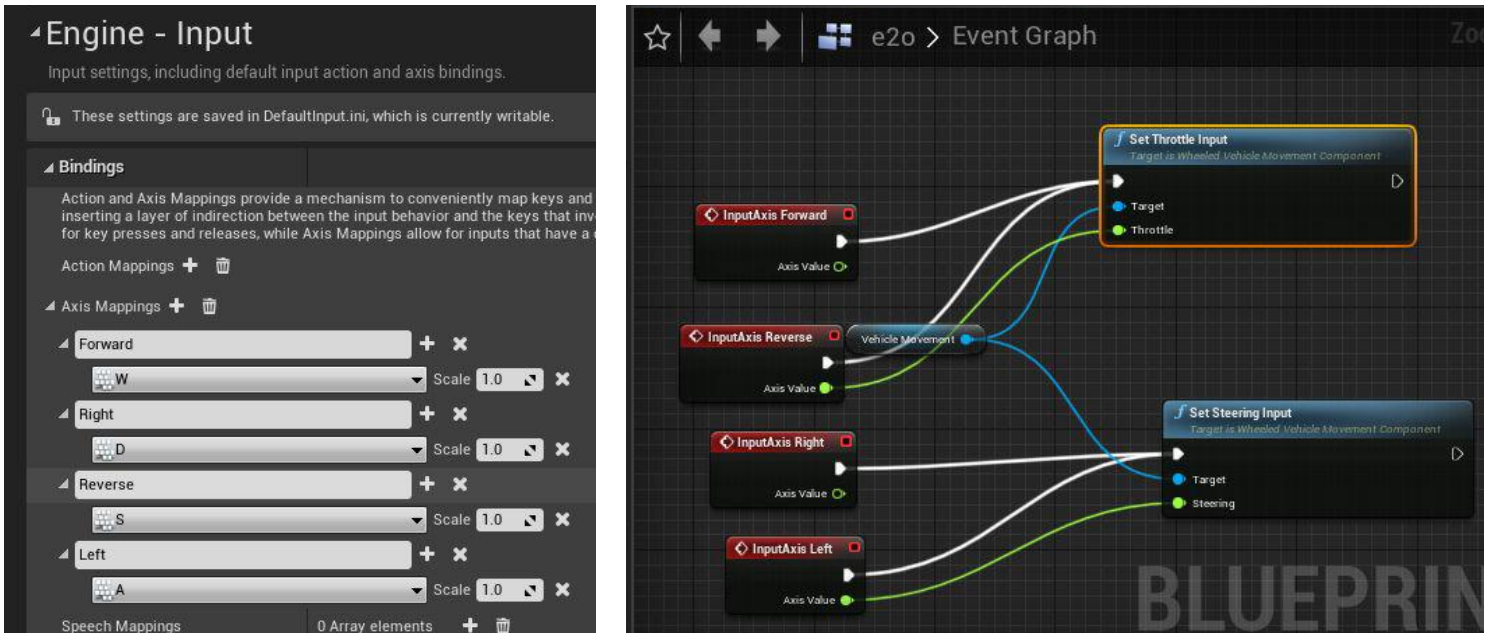


Fig 11.6 e2o car control setup

11.7 Open in Game mode in Unreal Engine

Now create a game mode blueprint in the project, add the e2o blueprint to it and import the e2o model on a new level in the game mode. Now the car can be controlled in the game mode.

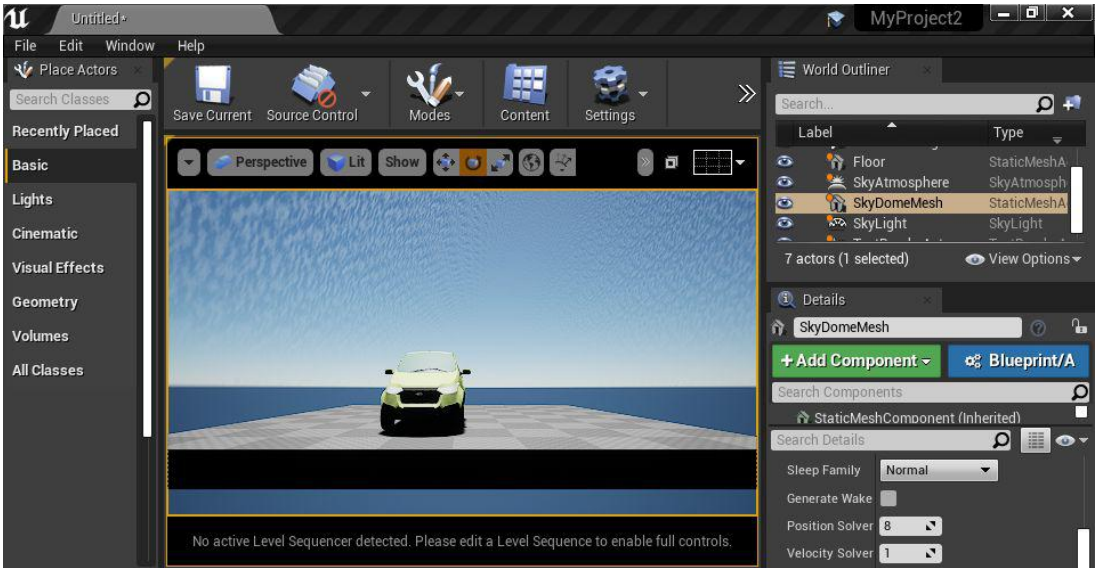


Fig 11.7 e2o model in Game mode

Chapter 12: Conclusions and Results

12.1 Conclusion

12.1.1 Lane Detection in Real-world

A robust lane detection algorithm was successfully implemented, that could work well even in dim lights and overcast conditions. It can remove all the noise lying outside the road and be unaffected by shadows and lighting conditions.
It also **successfully** handled the discontinuous lane marks, cross-sections, missing lane marks, zebra crossings, and horizontal lines by applying filters. Using the Kalman filter, it was able to generate accurate lane markers on curves.

12.1.2 Control System in Real-world

A motion planning algorithm was successfully implemented to decide the optimal turning angle given to the car by comparing the calculated waypoint from the perception model and the heading direction of the car to get a steering angle on which PID control was implemented to get the turning angle. The heading direction determination was made more robust by sensing the heading direction from the front part of the car. Also, multiple techniques such as a bounding box and the average of stored values were implemented to minimise wobbling during control. These results were integrated with the car control system for achieving an autonomous lane following system. The autonomous car was **successfully** able to run for an 800 meters stretch without human interference on a path free of shadows.
Some uncertainties involved in the control system include the existence of long continuous paths with only one lane and the response time of the system in case of sharp turns. There is still some wobbling which needs to be reduced further to be able to test at higher speeds. The PID parameters could not be tuned, and the PID control could not be tested on the real car because of the COVID-19 crisis.

12.1.3 ROS Implementation in real world

ROS was successfully implemented for the integration of the control action with the car control system. The stereo image was detected by the ZED camera and was published to a rostopic, which our code subscribed to and processed the image to determine the steering angle required for correction. The steering angle was published to another rostopic which was subscribed and controlled the car. The different subsystems were successfully integrated using ROS such that the correction takes place without lag.

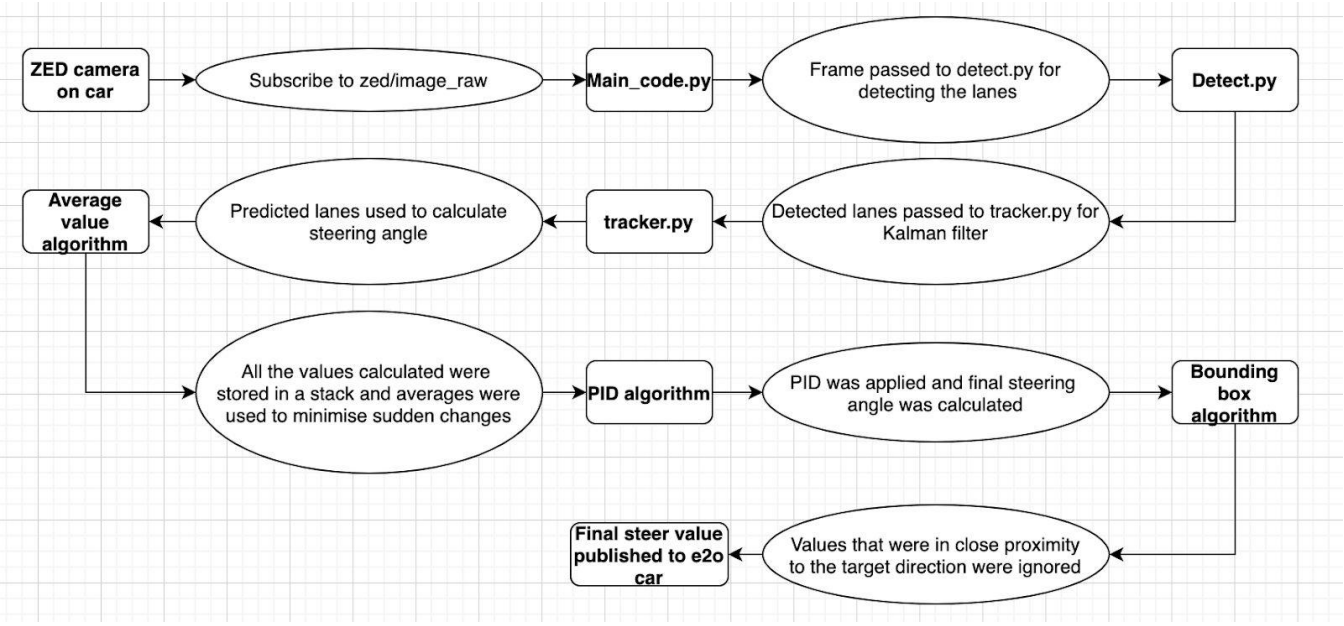


Fig 12.1 Flowchart of algorithm used in autonomous car

12.1.4 Lane Detection in CARLA simulator

The lane detection method was successfully implemented in the CARLA world with much better results than the real world. Accurate lines were detected on the straight roads. On the curved road as well, numerous short line segments were detected, but the Kalman filter prediction sometimes gave errors because of the curve and limited field of view of the camera in CARLA. There were some issues with the detection of the road marks. Because of the high clarity possessed by them in CARLA, they were also affecting the lane detection algorithm, which is not the case in the real world.

12.1.5 Control System in CARLA simulator

The motion planning algorithm designed for the real car was successfully implemented in the CARLA with minor changes. The control was further improved by comparing the calculated waypoint from the perception model and the heading direction of the car to get a steering angle on which PID control was implemented to get the turning angle. Multiple techniques such as using the average of the values were removed from the control as were not needed for the perfect lane detection in the simulated world. These results were integrated with the car control system for achieving an autonomous lane following system. The autonomous car was **successfully** able to run for a distance of 1-kilometre stretch without human interference on a path free of shadows at speeds of 1.5 m/s, 3 m/s, and 5 m/s.
The car was able to move on a curved path as well with some wobbling. Using PID, the need of the car being at the centre of the road at the start was removed. The car was able to move more accurately even if it was off-centre at the start. The PID parameters were successfully tuned by repeated runs in the simulated world and testing the PID control on the ego car.

12.1.6 ROS Implementation in CARLA simulator

ROS was successfully implemented for the integration of the control action with the car control system. The stereo image was detected by the mounted camera and was published to a rostopic, which the code subscribed to and processed the image to determine the steering angle required for correction. The steering angle and the linear speed were published to other rostopics which controlled the car.

12.1.7 Adding e2o car in CARLA

The e2o car 3D model was successfully blended with the default armature of a 4-wheeled vehicle and imported to Unreal Engine. In Unreal Engine, blueprints of the car colliders were set and manual control was set up. The car was successfully launched in the game mode in Unreal Engine.

12.2 Future Work

The primary goal of this project is to attain a fully autonomous car. Some of the problems encountered, which are planned to be solved in future are -

1) Testing the PID control on the real car and tune the PID parameters at different speeds.

2) Solve issues of lane detection on U-turns, and Sharp turns without lanes. An attempt is being made to integrate GPS with the code so that the car can take these turns using the GPS information rather than unreliable lane detection feedback.

3) Another issue faced during the trial run was that the real car speed was extremely slow (0.55m/s - 0.83 m/s). In such a situation, the shadows and all the obstacles pose a problem as a large number of values with errors are received because the car is at almost the same position for a long time. The speed of the car could be increased to a range of 1.5 m/s - 5 m/s using the PID control in the simulator and would be attempted to achieve the same in the real world.

4) The humans, road signs, and markings on the roads also pose a problem. Markings and signs were a bigger problem in the simulator. So it is being planned to use neural networks to detect humans and all other obstacles and not consider the lines generated due to them.

5) Adding camera and other sensors to the e2o model in Unreal Engine and importing it into CARLA as ego_vehicle controllable using rostopics.

Chapter 13: References

1. Gaspar, José, Niall Winters, and José Santos-Victor. "Vision-based navigation and environmental representations with an omnidirectional camera." IEEE Transactions on Robotics and automation 16.6 (2000): 890-898.
2. Huang, Edward YC. A semi-autonomous vision-based navigation system for a mobile robotic vehicle. Diss. Massachusetts Institute of Technology, 2003.
3. Bradski, Gary, and Adrian Kaehler. Learning OpenCV: Computer vision with the OpenCV library. " O'Reilly Media, Inc.", 2008.
4. Kirk, David. "NVIDIA CUDA software and GPU parallel computing architecture." ISMM. Vol. 7. 2007.
5. Welch, Greg, and Gary Bishop. "An introduction to the Kalman filter." (1995).
6. Ang, Kiam Heong, Gregory Chong, and Yun Li. "PID control system analysis, design, and technology." IEEE transactions on control systems technology 13.4 (2005): 559-576.
7. Quigley, Morgan, et al. "ROS: an open-source Robot Operating System." ICRA Workshop on open-source software. Vol. 3. No. 3.2. 2009.
8. Koenig, Nathan, and Andrew Howard. "Design and use paradigms for gazebo, an open-source multi-robot simulator." 2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)(IEEE Cat. No. 04CH37566). Vol. 3. IEEE, 2004.
9. Dosovitskiy, Alexey, et al. "CARLA: An open urban driving simulator." arXiv preprint arXiv:1711.03938 (2017).
10. Carla.readthedocs.io. 2020. *Add A New Vehicle - CARLA Simulator*. [online] Available at: <https://carla.readthedocs.io/en/latest/tuto_A_add_vehicle/>

Chapter 14: Appendix

Python Codes

14.1 Main code for real car

```
from __future__ import division

import cv2 as cv
import track
import detect
import detect2
import argparse
import sys
import time
import math
import rospy
import numpy as np
from std_msgs.msg import Float32
from sensor_msgs.msg import Image
from cv_bridge import CvBridge, CvBridgeError

parser = argparse.ArgumentParser()
parser.add_argument("--path", type=str, help="Video path")

class autonomous_car:      #class in which image is subscribed and  converted to cv
    def __init__(self):
        self.bridge = CvBridge()
         self.image_sub = rospy.Subscriber ("/zed/image_raw",Image,self.callback)
    def callback(self,data):
        try:
            frame = self.bridge.imgmsg_to_cv2(data , 'bgr8')    #rosmsg to cv
        except CvBridgeError as e:
            print(e)
        frame = mainp(frame)
        cv.imshow('', frame)
        cv.waitKey(3)
def mainp(frame):
    global ticks
    global lt
    global ld
    global l2_old
    global arr1
    global arr2


    y=0
```

```
    precTick = ticks
    ticks = cv.getTickCount()
    dt = (ticks - precTick) / cv.getTickFrequency()
    pub = rospy.Publisher('/steer', Float32)            #publisher for steer angle
    frame = frame[100:600,300:1100]           #captured frame cropping(height, width)
    predict = lt.predict(dt)  #calling predict function in track.py file this
                              #return the state and kalman filter results which are 4
                              #points stored in predict
    f=frame[0:400]  #frame cropped to remove side features from frame
    lanes = ld.detect(f)     #calling detect function to detect lanes
    l2= ld2.detect(frame[100:600,200:500])  #frame cropped to get car front only in
                                            #frame to get the heading direction and
                                            #sent to line detection function

    if predict is not None and l2 is not None:   #if we get kalman filter results

        cv.line(frame, (predict[0][0], predict[0][1]), (predict[0][2], predict[0][3]),
(0, 0, 255), 5)     #drawing left lane prediction on frame
        cv.line(frame, (predict[1][0], predict[1][1]), (predict[1][2], predict[1][3]),
(0, 0, 255), 5)     #drawing right lane prediction on frame
         x1=(max(predict[0][0],predict[0][2])+min(predict[1][0], predict[1][2]))/2
#get lane centre of x coordinate
        y=((predict[1][1])+((predict[0][3])))/2   #get avg y coordinate


        if(len(arr2)<100):          #adding the heading direction in array for having
            arr2.append(l2+200)     #recent 100 values for getting the average to
        else:                       #minimise error in heading direction
            arr2.pop(0)
            arr2.append(l2+200)

        if(len(arr1)<200):          #adding the centre of lane in array for having
            arr1.append(x1)         #recent 200 values for getting the average to
        else:                       #minimise error in centre of lane
            arr1.pop(0)
            arr1.append(x1)

        x=sum(arr1)/len(arr1)         #average lane centre
        l2_avg=sum(arr2)/len(arr2)    #average heading direction
        cv.circle(frame, (int(x),int(((predict[1][1])+((predict[0][3])))/2))
,5,(255,0,0),-1)
        cv.rectangle(frame,(x+10,y+10), (x-10, y-10), (255,0,0),1)
        cnt = np.float32([(x+10,y+10),(x+10,y-10),(x-10,y-10),(x-10,y+10)])
        dist = cv.pointPolygonTest(cnt,(int(l2_avg),int((predict[1][1])+
((predict[0][3])))/2),False)  #checking that heading direction is in contour
                              #created around lane centre to minimise wobbling or not
        print(dist)
        cv.circle(frame,(int(l2_avg),int(((predict[1][1])+((predict[0][3])))/2)),
5,(0,255,0),-1)
```

```
        dif = l2_avg-x       #getting difference b/w lane centre and heading direction
        dire = dif/(((predict[1][1])+((predict[0][3])))/2)  #getting tanθ
        if(round(-1*math.degrees(math.atan(dire)),2)<-10 or round(-1*math.degrees
(math.atan(dire)),2)>10):
            arr1=[]

        deg = (round(-1*math.degrees(math.atan(dire)),2)) #getting angle from tanθ
        deg = pid(deg)       #sending the angle to PID function to get steering angle

        if(dist==-1):        #if heading direction is not in the contour
            print(deg)
            pub.publish(deg) #publish steering angle to car
            cv.putText(frame , str(deg), (20,50), cv.FONT_HERSHEY_SIMPLEX, 1.5, (255,
0, 0), 2, cv.LINE_AA)
        else:                #if heading direction is in the contour
            print(0)
            pub.publish(0)   #publish 0 as steering angle to car
            cv.putText(frame , str(0), (20,50), cv.FONT_HERSHEY_SIMPLEX, 1.5, (255, 0,
0), 2, cv.LINE_AA)

    else:
         cv.circle(frame,(l2_old,y),5,(0,0,255),-1)

    lt.update(lanes)
    return frame

def pid(dif):        #pid function with input angle & output PID corrected steer angle
       global sum1
       global elapsed_time
       global diff
       global prev_err
       global kp
       global ki
       global kd
       global check
       if(abs(dif)>0.1):            #check if angle in bounding box
              if(check==True):
                     deltaT=0.1
                     check=False
              else:
                     deltaT = (time.time()-elapsed_time)  #getting dt
              elapsed_time=time.time()
              sum1 = sum1 + dif * deltaT;         #getting integral term
              diff = (dif - prev_err) / deltaT;   #getting differential term
              prev_err = dif
              output = (kp * dif + ki * sum1 + kd * diff);  #PID corrected steer angle
              return output
       else:
              sum1=0
```

```
            check=True
            return 0
def mainl(args):
    ic = autonomous_car()
    rospy.init_node('autonomous_car', anonymous=True)    #initiating node
    try:
        rospy.spin()
    except KeyboardInterrupt:
        print("Shutting down")
    cv.destroyAllWindows()


if __name__ == "__main__":
    ticks = 0
    rospy.init_node('autonomous_car', anonymous=True)
    lt = track.LaneTracker(2, 0.1, 500)
    ld = detect.LaneDetector(180)
    ld2 = detect2.LaneDetector(180)
    l2_old=0
    x_old=0
    arr1=[]
    arr2=[]
    sum1=0
    diff=0
    prev=240
    elapsed_time=0
    prev_err = 0
    kp=0.00000001
    ki=0.05
    kd=1
    check=True
    mainl(sys.argv)
```

14.2 Lane detection code

```
from __future__ import division
import cv2 as cv
import math
import numpy as np

class LaneDetector:
    def __init__(self, road_horizon, prob_hough=True):
        self.prob_hough = prob_hough
        self.vote = 50         #threshold to decide a detected line is actual line
        self.roi_theta = 0.3
        self.road_horizon = road_horizon  #fixing maximum y coordinate in frame

    def _standard_hough(self, img, init_vote):
        # Hough transform wrapper to return a list of points like PHough does
        lines = cv.HoughLines(img, 1, np.pi/180, init_vote)
```

```
        points = [[]]
        for l in lines:                #hough line transform application
            for rho, theta in l:
                a = np.cos(theta)
                b = np.sin(theta)
                x0 = a*rho
                y0 = b*rho
                x1 = int(x0 + 1000*(-b))
                y1 = int(y0 + 1000*a)
                x2 = int(x0 - 1000*(-b))
                y2 = int(y0 - 1000*a)
                points[0].append((x1, y1, x2, y2))
        return points

    def _base_distance(self,x1,y1,x2,y2,width):
        if x2 == x1:  #compute the point where the line crosses the base of the frame
            return (width*0.5) - x1
        m = (y2-y1)/(x2-x1)
        c = y1 - m*x1
        base_cross = -c/m     # return distance of that point from center of the frame
        return (width*0.5) - base_cross

    def _scale_line(self, x1, y1, x2, y2, frame_height):
        if x1 == x2:               # scale the farthest point of the segment to be on
                                   the drawing horizon
            if y1 < y2:
                y1 = self.road_horizon
                y2 = frame_height
                return x1,y1,x2,y2
            else:
                y2 = self.road_horizon
                y1 = frame_height
                return x1,y1,x2,y2
        if y1 < y2:
            m = (y1-y2)/(x1-x2)
            x1 = ((self.road_horizon-y1)/m) + x1
            y1 = self.road_horizon
            x2 = ((frame_height-y2)/m) + x2
            y2 = frame_height
        else:
            m = (y2-y1)/(x2-x1)
            x2 = ((self.road_horizon-y2)/m) + x2
            y2 = self.road_horizon
            x1 = ((frame_height-y1)/m) + x1
            y1 = frame_height
        return x1,y1,x2,y2

    def detect(self, frame):
        img = cv.cvtColor(frame, cv.COLOR_BGR2GRAY)
```

```
        roiy_end = frame.shape[0]
        roix_end = frame.shape[1]
        roi = img[self.road_horizon:roiy_end, 0:roix_end]
        blur = cv.medianBlur(roi, 5)          #blurring to reduce noises
        contours = cv.Canny(blur, 60, 120)    #canny edge detection

        if self.prob_hough:
            lines = cv.HoughLinesP(contours, 1, np.pi/180, self.vote, minLineLength=
30, maxLineGap=100)
        else:
            lines = self.standard_hough(contours, self.vote)

        if lines is not None:
            #find nearest lines to center
            lines = lines+np.array([0, self.road_horizon, 0, self.road_horizon]).
reshape((1, 1, 4))      #scale points from ROI coordinates to full frame coordinates
            left_bound = None
            right_bound = None
            for l in lines:
# find the rightmost line of the left half of the frame and the leftmost line of the
right half for separating left half and right half lanes
                for x1, y1, x2, y2 in l:
                    m=(y1-y2)/(x1-x2)
                    if(math.degrees(math.atan(m))>5 or math.degrees(math.atan(m))<-5):
                        cv.line(frame, (x1, y1), (x2, y2), (255, 0, 0), 2)
                        theta = np.abs(np.arctan2((y2-y1), (x2-x1)))  # line angle WRT
                                                                        horizon
                        if theta > self.roi_theta:  # ignore lines with a small angle
                                                      WRT horizon
                            dist = self._base_distance(x1, y1, x2, y2, frame.shape[1])
                            if left_bound is None and dist < 0:
                                left_bound = (x1, y1, x2, y2)
                                left_dist = dist
                            elif right_bound is None and dist > 0:
                                right_bound = (x1, y1, x2, y2)
                                right_dist = dist
                            elif left_bound is not None and 0 > dist > left_dist:
                                left_bound = (x1, y1, x2, y2)
                                left_dist = dist
                            elif right_bound is not None and 0 < dist < right_dist:
                                right_bound = (x1, y1, x2, y2)
                                right_dist = dist
            if left_bound is not None:
                left_bound = self._scale_line(left_bound[0], left_bound[1],
left_bound[2], left_bound[3], frame.shape[0])
            if right_bound is not None:
                right_bound = self._scale_line(right_bound[0], right_bound[1],
right_bound[2], right_bound[3], frame.shape[0])
            return [left_bound, right_bound]
```

14.3 Kalman Filter Code

```
import cv2 as cv
import numpy as np
from scipy.linalg import block_diag

class LaneTracker:
    def __init__(self, n_lanes, proc_noise_scale, meas_noise_scale,
process_cov_parallel=0, proc_noise_type='white'):
        self.n_lanes = n_lanes
        self.meas_size = 4 * self.n_lanes
        self.state_size = self.meas_size * 2
        self.contr_size = 0

        self.kf = cv.KalmanFilter(self.state_size, self.meas_size, self.contr_size)
        self.kf.transitionMatrix = np.eye(self.state_size, dtype=np.float32)
             self.kf.measurementMatrix  =  np.zeros((self.meas_size,  self.state_size),
np.float32)
        for i in range(self.meas_size):
            self.kf.measurementMatrix[i, i*2] = 1

        if proc_noise_type == 'white':
            block = np.matrix([[0.25, 0.5],
                               [0.5, 1.]], dtype=np.float32)
                 self.kf.processNoiseCov = block_diag(*([block] * self.meas_size)) *
proc_noise_scale
        if proc_noise_type == 'identity':
                self.kf.processNoiseCov = np.eye(self.state_size, dtype=np.float32) *
proc_noise_scale
        for i in range(0, self.meas_size, 2):
            for j in range(1, self.n_lanes):
                self.kf.processNoiseCov[i, i+(j*8)] = process_cov_parallel
                self.kf.processNoiseCov[i+(j*8), i] = process_cov_parallel

             self.kf.measurementNoiseCov  =  np.eye(self.meas_size,  dtype=np.float32)  *
meas_noise_scale

        self.kf.errorCovPre = np.eye(self.state_size)

        self.meas = np.zeros((self.meas_size, 1), np.float32)
        self.state = np.zeros((self.state_size, 1), np.float32)

        self.first_detected = False

    def _update_dt(self, dt):
        for i in range(0, self.state_size, 2):
            self.kf.transitionMatrix[i, i+1] = dt

    def _first_detect(self, lanes):
        for l, i in zip(lanes, range(0, self.state_size, 8)):
```

```
            self.state[i:i+8:2, 0] = l
        self.kf.statePost = self.state
        self.first_detected = True

    def update(self, lanes):
        if self.first_detected:
            if lanes is not None:
                  for l, i in zip(lanes, range(0, self.meas_size, 4)):
                      if l is not None:
                          self.meas[i:i+4, 0] = l
            self.kf.correct(self.meas)
        elif lanes is not None:
            if lanes.count(None) == 0:
                self._first_detect(lanes)

    def predict(self, dt):
        if self.first_detected:
            self._update_dt(dt)
            state = self.kf.predict()
            lanes = []
            for i in range(0, len(state), 8):
                lanes.append((state[i], state[i+2], state[i+4], state[i+6]))
            return lanes
        else:
            return None
```

14.4 Main Code in Carla

```
from __future__ import division
import cv2 as cv
import track
import detect
import detect2
import argparse
import sys
import time
import math
import rospy
import numpy as np
from std_msgs.msg import Float32
from sensor_msgs.msg import Image
from cv_bridge import CvBridge, CvBridgeError
from carla_msgs.msg import CarlaEgoVehicleControl
from geometry_msgs.msg import Twist
```

```
class autonomous_car:   #class in which image is subscribed and  converted to cv

    def __init__(self):
        self.bridge = CvBridge()
        self.image_sub = rospy.Subscriber("/carla/ego_vehicle/camera/rgb/front/
image_color", Image, self.callback)
    def callback(self, data):
        try:
            frame = self.bridge.imgmsg_to_cv2(data,'bgr8')  #rosmsg to cv
        except CvBridgeError as e:
            print(e)
        frame = mainp(frame)
        cv.imshow('', frame)
        cv.waitKey(3)
def mainp(frame):
    global ticks
    global lt
    global ld
    global l2_old
    global arr1
    global arr2
    global check1
    global prev
    y=0
    precTick = ticks
    ticks = cv.getTickCount()
    dt = (ticks - precTick) / cv.getTickFrequency()
    pub = rospy.Publisher('/carla/ego_vehicle/vehicle_control_cmd',
CarlaEgoVehicleControl)    #publisher for steer angle
    pub2 = rospy.Publisher('/carla/ego_vehicle/twist_cmd',Twist) #publisher for speed
    msg = CarlaEgoVehicleControl()
    msg2 = Twist()
    msg2.linear.x = 1.5         #linear speed in x direction
    frame = frame[200:600,150:650]  #captured frame cropping(height, width)
    predict = lt.predict(dt)   #calling predict function in track.py file this
                               #return the state and kalman filter results which are 4
                               #points stored in predict
    f=frame
    lanes = ld.detect(f)
    l2=200                     #frame centre
    if predict is not None and l2 is not None:

        cv.line(frame, (predict[0][0], predict[0][1]), (predict[0][2], predict[0][3]),
(0, 0, 255), 5)  #drawing left lane prediction on frame
        cv.line(frame, (predict[1][0], predict[1][1]), (predict[1][2], predict[1][3]),
(0, 0, 255), 5)   #drawing right lane prediction on frame
        x1=(predict[0][2]+predict[1][0])/2  #get lane centre of x coordinate
        y=((predict[1][1])+((predict[0][3])))/2   #get avg y coordinate
        x=x1
```

```
        l2_avg=l2+40
        print(int(x))
        if(check1):
             prev=x
             check1=False
        else:
              if(abs(int(x)-prev)>10):    #ignore error values
                   x=prev
              else:
                   prev=x
        cv.circle(frame, (int(x),int(((predict[1][1])+((predict[0][3])))/2)),
5,(255,0,0),-1)
        cv.rectangle(frame,(int(x)+10,int(y)+10), (int(x)-10, int(y)-10), (255,0,0),1)
        cnt = np.float32([(int(x)+10,int(y)+10),(int(x)+10,int(y)-10), (int(x)-10,
int(y)-10),(int(x)-10,int(y)+10)])
        cv.circle(frame,(int(l2_avg),int(((predict[1][1])+((predict[0][3])))
/2)),5,(0,255,0),-1)
        dif = l2_avg-x    #getting difference b/w lane centre and heading direction
        dire = dif/(((predict[1][1])+((predict[0][3])))/2)  #getting tanθ
        if(round(-1*math.degrees(math.atan(dire)),2)<-10 or round(-1*math.degrees
(math.atan(dire)),2)>10):
            arr1=[]


        deg = (round(-1*math.degrees(math.atan(dire)),2)/30)  #getting angle from tanθ
        deg = pid(deg)   #sending the angle to PID function to get steering angle
        deg=deg/2
        msg.steer = round(deg,2)
        pub.publish(msg)    #publish steer values
        pub2.publish(msg2)  #publish speed values
        cv.putText(frame , str(round(deg,2)), (20,50), cv.FONT_HERSHEY_SIMPLEX, 1.5,
(255, 0, 0), 2, cv.LINE_AA)

    else:
         cv.circle(frame,(l2_old,y),5,(0,0,255),-1)

    lt.update(lanes)
    return frame

def pid(dif):
       global sum1
       global elapsed_time
       global diff
       global prev_err
       global kp
       global ki
       global kd
       global check
       if(abs(dif)>0.1):       #check if angle in bounding box
```

```
                if(check==True):
                        deltaT=0.1
                        check=False
                else:
                        deltaT = (time.time()-elapsed_time)    #getting dt
                elapsed_time=time.time()
                sum1 = sum1 + dif * deltaT;       #getting integral term
                diff = (dif - prev_err) / deltaT; #getting differential term
                prev_err = dif
                output = (kp * dif + ki * sum1 + kd * diff);   #PID corrected steer angle
                return output
        else:
                sum1=0
                check=True
                return 0

def mainl(args):
        ic = autonomous_car()
        rospy.init_node('autonomous_car', anonymous=True)
        try:
            rospy.spin()
        except KeyboardInterrupt:
            print("Shutting down")
        cv.destroyAllWindows()


if __name__ == "__main__":
    ticks = 0
    rospy.init_node('autonomous_car', anonymous=True)     #initiating node
    lt = track.LaneTracker(2, 0.1, 500)
    ld = detect.LaneDetector(180)
    ld2 = detect2.LaneDetector(180)
    l2_old=0
    x_old=0
    arr1=[]
    arr2=[]
    sum1=0
    diff=0
    prev=240
    check1=True
    elapsed_time=0
    prev_err = 0
    kp=0.00000001
    ki=0.05
    kd=1
    check=True
    mainl(sys.argv)
```

14.5 Waypoint Navigation in Carla Code

```
from __future__ import division

import cv2 as cv
import track
import detect
import detect2
import argparse
import sys
import time
import math
import rospy
import numpy as np
from std_msgs.msg import Float32
from sensor_msgs.msg import Image
from cv_bridge import CvBridge, CvBridgeError
from carla_msgs.msg import CarlaEgoVehicleControl
from geometry_msgs.msg import Twist

arr2d=[]
for i in range(10):
        arr2d.append([])

class autonomous_car:      #class in which image is subscribed and  converted to cv

    def __init__(self):
        self.bridge = CvBridge()
        self.image_sub = rospy.Subscriber("/carla/ego_vehicle/camera/rgb/front/
image_color", Image,self.callback)

    def callback(self,data):
        try:
            frame = self.bridge.imgmsg_to_cv2(data, 'bgr8')   #rosmsg to cv
        except CvBridgeError as e:
            print(e)
        frame = mainp(frame)
        cv.imshow('', frame)
        cv.waitKey(3)

def mainp(frame):
    global ticks
    global lt
    global ld
    global l2_old
    global arr1
    global arr2
    global check1
    global prev
    flag=True
```

```
    y=0
    p=[(0,0,0,0),(0,0,0,0)]
    l2_old=0
    x_old=0
    m1=0
    m2=0
    m=0
    arr1=[]
    arr2=[]
    value=0
    value_old=0
    i_2=0
    j_2=0
    time.sleep(0.025)
    precTick = ticks
    ticks = cv.getTickCount()
    dt = (ticks - precTick) / cv.getTickFrequency()
    pub = rospy.Publisher('/carla/ego_vehicle/vehicle_control_cmd',
CarlaEgoVehicleControl)     #publisher for steer angle
    pub2 = rospy.Publisher('/carla/ego_vehicle/twist_cmd',Twist) #publisher for speed
    msg = CarlaEgoVehicleControl()
    msg2 = Twist()
    msg2.linear.x = 1.5      #linear speed in x direction
    frame = frame[200:600,150:650]     #captured frame cropping(height, width)
    predict = lt.predict(dt)     #calling predict function in track.py file this
                               #return the state and kalman filter results which are 4
                               #points stored in predict

    f=frame
    lanes = ld.detect(f)
    l2=200       #frame centre
    if predict is not None and l2 is not None:

        cv.line(frame, (predict[0][0], predict[0][1]), (predict[0][2], predict[0][3]),
(0, 0, 255), 5)     #drawing left lane prediction on frame
        cv.line(frame, (predict[1][0], predict[1][1]), (predict[1][2], predict[1][3]),
(0, 0, 255), 5)     #drawing right lane prediction on frame
        pub
        if flag:
            flag=False

            x1=(max(predict[0][0],predict[0][2])+min(predict[1][0],predict [1][2]))/2
#get lane centre of x coordinate
            y=((predict[1][1])+((predict[0][3])))/2    #get avg y coordinate

             #left lane y coordinate difference
            dif1 = (predict[0][3][0]-predict[0][1][0])/10
            #right lane y coordinate difference
            dif2 = (predict[1][1][0]-predict[1][3][0])/10
```

```
            m=((predict[0][1]+predict[1][1])+((predict[0][3]+ predict[1][3])))/4
            m1=((predict[0][1][0]- predict[0][3][0])/(predict[0][0][0]-
predict[0][2][0]))   #slope of left predict lane
            m2=((predict[1][1][0]- predict[1][3][0])/(predict[1][0][0]-
predict[1][2][0]))   #slope of right predict lane

            arr=[]
            arr_angles = []

             #appending 10 y coordinates between max and min y coordinates of left and
              right lane in 2d-array
            for i in range(2,12):
                    arr.append([int(((i*dif1/m1+predict[0][0][0])+(i*dif2/m2+
predict[1][2][0]))/2),int((predict[0][1][0]+i*dif1+predict[1][3][0]+i*dif2)/2)])

            if(len(arr2)<100):       #adding the heading direction in array for having
            arr2.append(l2+200)     #recent 100 values for getting the average to
        else:                       #minimise error in heading direction
            arr2.pop(0)
            arr2.append(l2+200)

        if(len(arr1)<200):          #adding the centre of lane in array for having
            arr1.append(x1)         #recent 200 values for getting the average to
        else:                       #minimise error in centre of lane
            arr1.pop(0)
            arr1.append(x1)


            for i in range(10):      #for taking average values
                print(len(arr2d[i]))
                if(len(arr2d[i])<40):
                    arr2d[i].append(arr[i][0])
                else:
                    arr2d[i].pop(0)
                    arr2d[i].append(arr[i][0])

            x2d=[]
            y2d=[]


            for i in range(10):
                x2d.append(sum(arr2d[i])/len(arr2d[i]))  #getting 10 average x
                                                         coordinates of waypoints
                y2d.append(int((predict[0][1][0]+i*dif1+predict[1][3][0] +i*dif2)/2))
#getting 10 average y coordinates of waypoints
            x=sum(arr1)/len(arr1)      #average x coordinate of lane centre
            l2_avg=sum(arr2)/len(arr2) #average x coordinate of heading direction
            cv.circle(frame, (x,((predict[1][1])+((predict[0][3])))/2)
,5,(255,0,0),-1)
```

```
            cv.circle(frame,(int(l2_avg),((predict[1][1])+
((predict[0][3])))/2),5,(0,255,0),-1)
            m=((predict[0][1]+predict[1][1])+((predict[0][3]+predict[1][3] )))/4
            m1=((predict[0][1][0]- predict[0][3][0])/(predict[0][0][0]-
predict[0][2][0]))   #slope of left predict lane
            m2=((predict[1][1][0]- predict[1][3][0])/(predict[1][0][0]-
predict[1][2][0]))   #slope of right predict lane
            for i in range(10):
                dif = l2_avg-x2d[i] #getting difference b/w lane centre and heading
                                    direction
                dire = dif/(y2d[i]*(10-i))  #getting tanθ
                arr_angles.append(pid(round(-1*math.degrees(math. atan(dire)))))
                 #appending pid angles from θ
                if(abs(value-value_old)>10):   #rejecting error extreme values
                    value=value_old
                if(round(-1*math.degrees(math.atan(dire)),2)<-10 or round(-1*math.
degrees(math.atan(dire)),2)>10):
                    arr2d[i]=[]
                    arr1=[]
        for i in range (len(arr2d)-1):
            cv.circle(frame, (int(x2d[i]), arr[i][1]) ,5,(255,0,0),-1)
            cv.circle(frame,(int(l2_avg),arr[i][1]),5,(0,255,0),-1)
        print(arr_angles[i_2],i_2)
        msg.steer = round(arr_angles[i_2]/10,2)
        pub.publish(msg)    #publishing steering angles
        pub2.publish(msg2)  #publishing linear speed
        time.sleep(0.001)   #time delay which is added to get total time needed for
                             car to reach next waypoint
        j_2=j_2+1
        if j_2==10:
            i_2=i_2+1
            j_2=0
        if i_2==10:
            i_2=0
            flag=True
    lt.update(lanes)
    return frame


def pid(dif):   #pid function with input angle & output PID corrected steer angle
    global sum1
    global elapsed_time
    global diff
    global prev_err
    global kp
    global ki
    global kd
    global check
    if(abs(dif)>0.1):  #check if angle in bounding box
```

```
        if(check==True):
            deltaT=0.1
            check=False
        else:
            deltaT = (time.time()-elapsed_time)  #getting dt
        elapsed_time=time.time()
        sum1 = sum1 + dif * deltaT;          #getting integral term
        diff = (dif - prev_err) / deltaT;    #getting differential term
        prev_err = dif
        output = (kp * dif + ki * sum1 + kd * diff);   #PID corrected steer angle
        return output
    else:
        sum1=0
        check=True
        return 0

def mainl(args):
    ic = autonomous_car()
    rospy.init_node('autonomous_car', anonymous=True)   #initiating node
    try:
        rospy.spin()
    except KeyboardInterrupt:
        print("Shutting down")
    cv.destroyAllWindows()


if __name__ == "__main__":
    ticks = 0
    rospy.init_node('autonomous_car', anonymous=True)
    lt = track.LaneTracker(2, 0.1, 500)
    ld = detect.LaneDetector(180)
    ld2 = detect2.LaneDetector(180)
    l2_old=0
    x_old=0
    arr1=[]
    arr2=[]
    sum1=0
    diff=0
    prev=240
    check1=True
    elapsed_time=0
    prev_err = 0
    kp=0.00000001
    ki=0.05
    kd=1
    check=True
    mainl(sys.argv)
```