TEKNOFEST

AEROSPACE and TECHNOLOGY FESTIVAL

ROBOTAXI-FULL SCALE AUTONOMOUS VEHICLE COMPETITION

(AUTONOMOUS-READY VEHICLE CATEGORY)

CRITICAL DESIGN REPORT

APPLICATION ID: 366606

CONTENTS

1. Team Organization

The team IZTECH BOLD PILOT has started to develop the subsystems of this autonomous vehicle software system individually at least two year ago although the team foundation year was stated as 2021 in the team introduction file. The team has 8 members from 3 different department. 4 main members are graduated from Electronics and Communications, and Computer Engineering. They are responsible for designing, developing, implementing and simulating autonomous driving solutions by using the middleware ROS, the simulation environment Gazebo, and the rendering tool Blender. Then, the other 4 members are interns to adapt working environment, and learn theoretical background.

Each team member got his duty based on his background topics. All tasks are decided according to the block diagram of Bold Pilot 2.5 autonomous driving system given Figure 1.

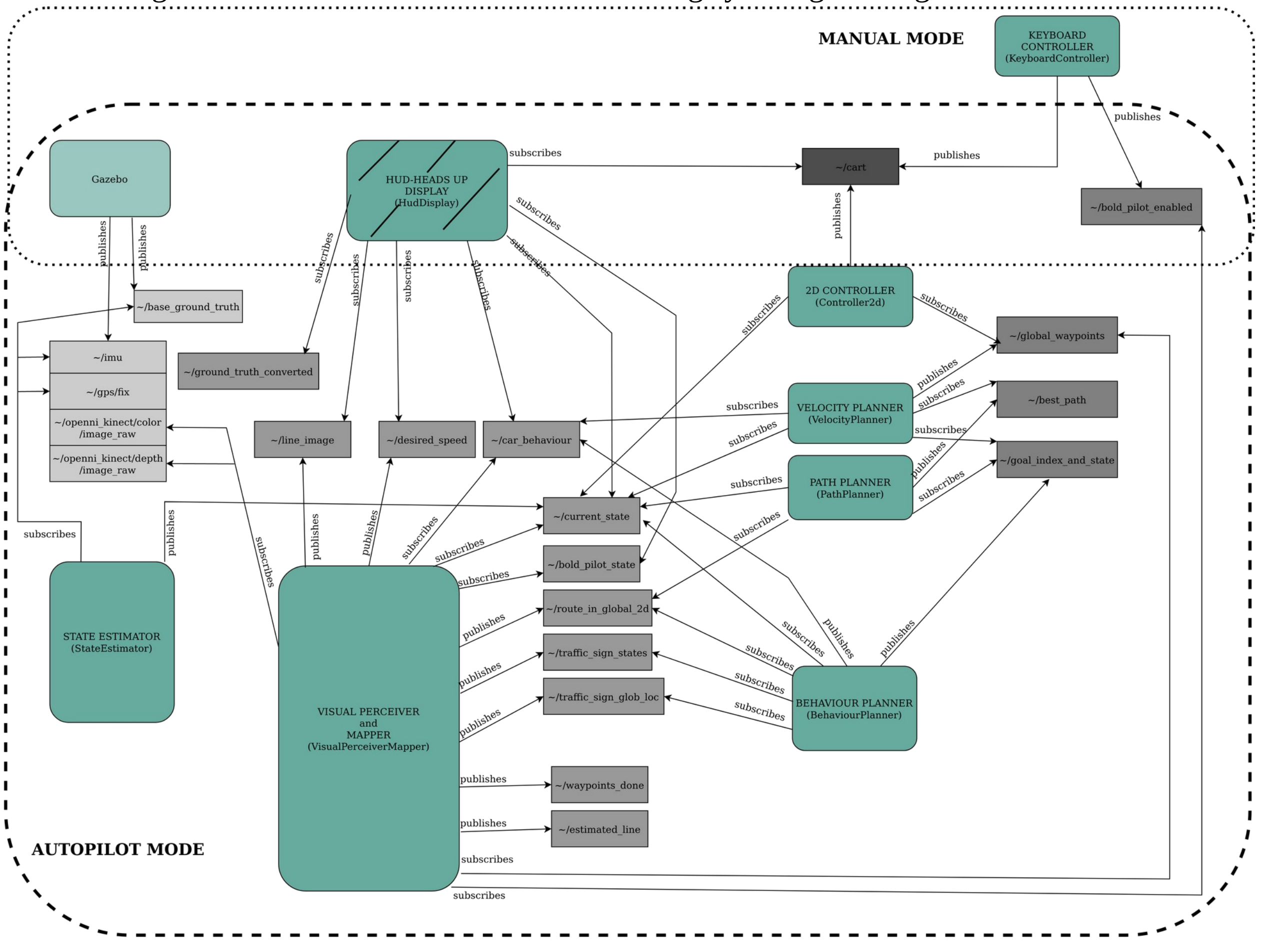


Figure 1: The Block Diagram of Bold Pilot 2.5 Autonomous Driving System

The block diagram above shows the project structure and information flow between subsystems. Whole system was designed by the team lead Ali. STATE ESTIMATION and LOCALIZATION is responsible to provide accurate speed and location estimations for vehicle under noisy sensor readings. MOTION PLANNING is responsible to plan a feasible trajectory for the vehicle under different road conditions. Finally, 2D CONTROLLER tries to maintain the vehicle's heading and speed at desired state according to given interpolated waypoints. Then, he applied image processing solutions to obtain middle of the lane information by using RGB data, developed an algorithm to output traffic sign and signal states and global positions for behavioral planner by using object detection results in VISUAL PERCEPTION subsystem. Also, he designed a route generation algorithm in MAPPING subsystem. Finally, he designed a HEAD UP DISPLAY to show whole system information in a one display in real-time.

Mustafa searched for Turkish Traffic Sign and Signals data set consisting of 15 sign and signal classes, and labeled some signs and signals. Then he searched for proper real-time object detection framework. He received data and put in order to be ready for training and testing. Besides, he developed a classifier in PyTorch for critical design report and final competitions.

Last year, whole system had been designed in manner that many subsystems or modules were dependent to other module functions. Besides, simulation environment was CARLA. However, now, every subsystem of Bold Pilot 2.5 was designed independently by using our design of ROS architecture, which was explained in detail in the chapter 'Software Architecture'. Also, new racetrack was modeled in Gazebo 11. To develop every module in the new architecture, and adapt to new simulation, new team mates helped us.

Celil designed new 3D topic architecture to make Bold Pilot have 3D waypoints msg files consisting of many 2D waypoints. Then, he rewrote every related code line in Behavior, Path, Velocity Planner, and 2D Controller modules. He developed a manual controller to roam the golf car model along the racetrack manually, and control enability of Bold Pilot. To navigate the vehicle around the racetrack, he developed behavior planner states in collaboration with team lead Ali.

İhsan Can designed a racetrack in Blender, and modeled in Gazebo for preliminary report. He also prepared a new URDF file to make a golf car model have a realistic sensor setup. Besides, he helped Bold Pilot to adapt software architecture principles. Finally, he tuned the image proccesing and computer graphics functions in the VISUAL PERCEPTION module to adapt the new ROS architecture.

Arda carried whole system to Github and wrote a guide to help the team to manage development process fully remote. Besides, he creates a Docker container for the system to work environment independent in every team members' machine and Nvidia AGX Xavier which is autonomous-ready vehicle computer.

Mert draw a new racetrack for the critical design report and final competitions according to highway regulations in Turkey.

System development pipeline until exhibition ceremony in September given in a Gantt Chart below.

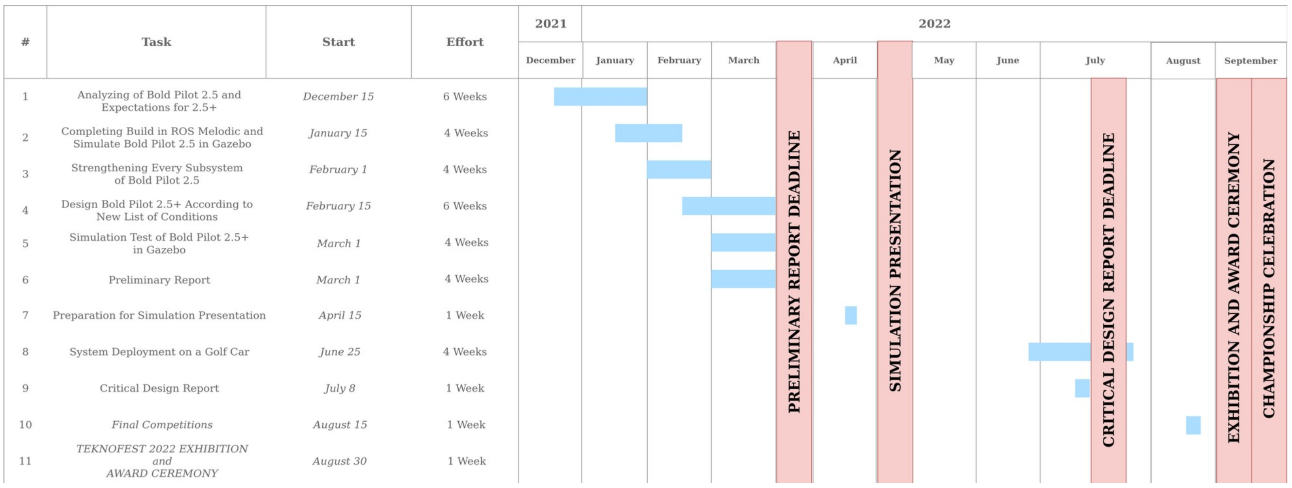

#	Task	Start	Effort
1	Analyzing of Bold Pilot 2.5 and Expectations for 2.5+	*December 15*	6 Weeks
2	Completing Build in ROS Melodic and Simulate Bold Pilot 2.5 in Gazebo	*January 15*	4 Weeks
3	Strengthening Every Subsystem of Bold Pilot 2.5	*February 1*	4 Weeks
4	Design Bold Pilot 2.5+ According to New List of Conditions	*February 15*	6 Weeks
5	Simulation Test of Bold Pilot 2.5+ in Gazebo	*March 1*	4 Weeks
6	Preliminary Report	*March 1*	4 Weeks
7	Preparation for Simulation Presentation	*April 15*	1 Week
8	System Deployment on a Golf Car	*June 25*	4 Weeks
9	Critical Design Report	*July 8*	1 Week
10	*Final Competitions*	*August 15*	1 Week
11	*TEKNOFEST 2022 EXHIBITION and AWARD CEREMONY*	*August 30*	1 Week

Figure 2 : Development Pipeline Plan of Bold Pilot 2.5+ Along Time

2. Evaluation of Preliminary Report

Bold Pilot has 5 subsystems working in a well synchronized way. Therefore, every subsystem has nodes subscribing and publishing to other topics of nodes. We had not enough time to desgin and run whole system to finish whole racetrack. Besides, our system code was written in Python3 and run in ROS Noetic so we have to use Ubuntu 20.04. However, Nvidia Xavier AGX is only compatible with Ubuntu 18.04. To solve this problem, we prepared a docker container to isolate system packages from the computer environment. Also, this solution provided interoperability between every team mate because we worked in one system that can run everyone's computer without any extra effort. Besides, to manage this new system compilation and run process, we opened a Github account and prepared a guideline to avoid from waste of time. Finally, we were able to develop a complex system fully remote.

3. Analyze and Specifications of Autonomous-Ready Vehicle

This category does not require making an autonomous driving platform from scratch. Instead, we are responsible for deriving, developing and applying autonomous driving algorithms on a autonomous-ready electric vehicle. Figure 3 below shows the related vehicle's fundamental hardware parts. However, it just state what the vehicle consists of so some hardware parts except to sensors **may not be located** in vehicle's proper partitions.

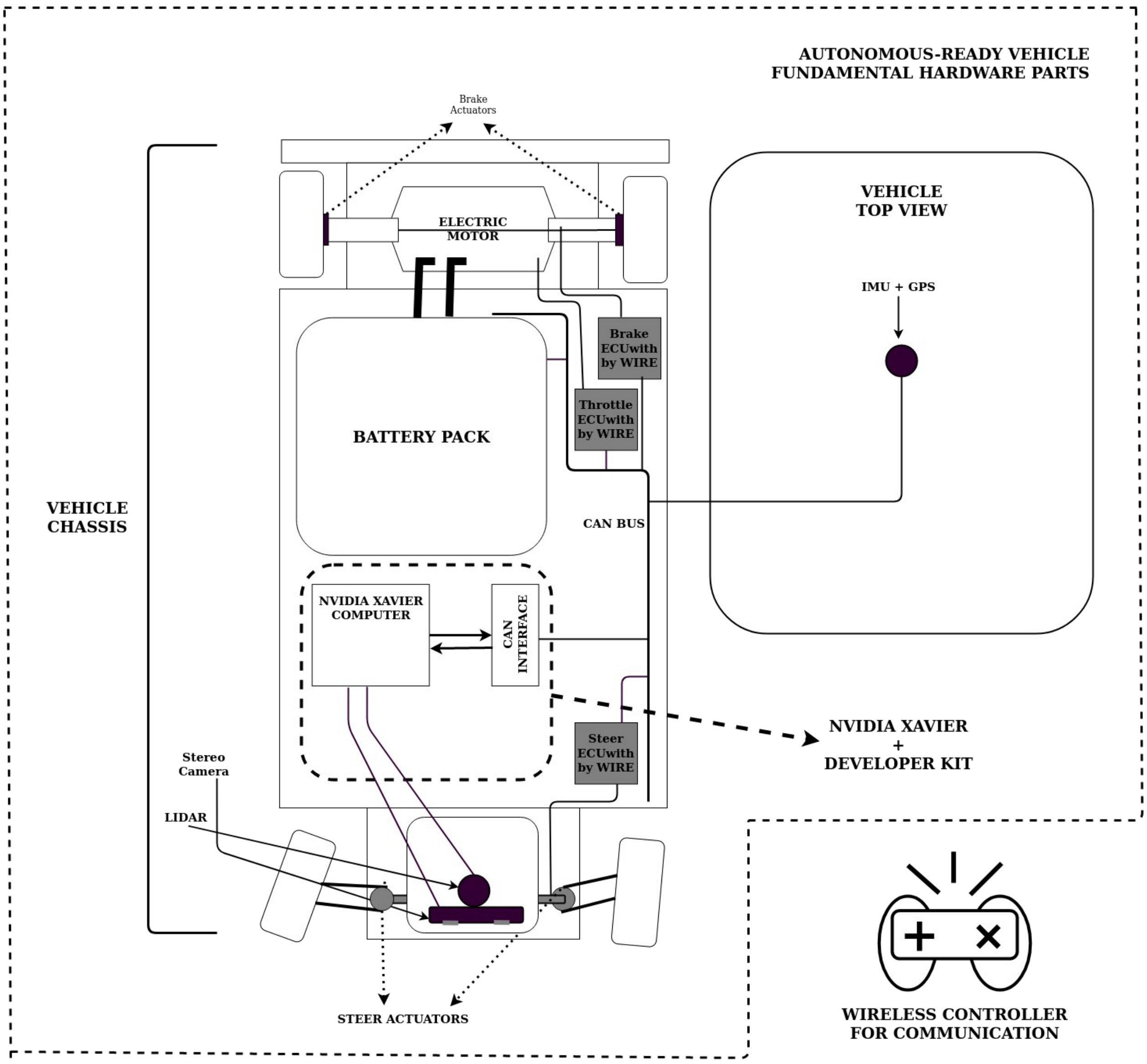


Figure 3: Fundamental Parts of the Autonomous-Ready Vehicle

This vehicle consists of the parts such that:

- An electric motor with throttle actuator providing desired torque through axes by converting electric energy to kinetic energy.
- Two park brake actuators to manually squeeze front wheels to fully stop.
- Steering actuators in front wheels to provide turn.
- A battery pack giving required electric energy from battery cells.
- Throttle, steer and brake actuators and Electrical Control Units with drive by wire interfaces for each.
- A CAN bus communication system to ensure reliable communication between vehicle peripherals and master.
- Nvidia AGX Xavier embedded computer with developer kit consisting a CAN interface to communicate to vehicle CAN bus to give productive computation ability for autonomous driving software.

- An Inertial Measurement Unit (IMU) and Global Positioning System (GPS) sensor at the top of vehicle to support autonomous driving system.
- A stereo camera with an integrated IMU and Light Detection and Ranging (LIDAR) sensor strapping-down to the in front of vehicle to support autonomous driving system.
- A wireless controller connected CAN bus to drive it manually.

3.1 Driving Information Flow Through Vehicle Parts: From Raw Data to Steering and Speed

Autonomous driving software development requires processing and manipulating raw sensor data to take control of vehicle. The autonomous-ready vehicle in this competition have a stereo camera providing both RGB and depth information in various resolutions, a LIDAR sensor gives point cloud data, a GPS sensor to provide location information, and an IMU to give acceleration and rotation rates in Easting, Northing and Up. Of course, all software system will be run via Nvidia AGX Xaiver that gathers LIDAR data through its ethernet port, RGB and depth information through its USB 3.0 port directly. However, it receives IMU and GPS data via CAN bus. Xavier has a CAN bus transceiver to read or send data through CAN bus. Besides, steering angle and speed coming from the autonomous driving algorithms are sent via this CAN bus. Then, electric motor and steering ECUs receive the commands and converts proper to desired torque and turning via by wire systems. Finally, the vehicle is expected to be driven autonomously.

4. Vehicle Control Unit

Vehicle can be driven by wireless controller. It is possible with translating the input signals of the controller into the information acceptable for CAN bus. However, this communication is possible without using Xavier because of security precaution. Wireless controller is more superior than Xavier in CAN bus so any undesired command coming from Xavier can be blocked by wireless controller. The CAN message ID to provide vehicle control is 0x560 whose abstract documentation is given in Table 1 below.

Name	Start bit	Length	Byte Order	Value Type	Initial Value	Factor	Offset	Minimum	Maximum
Operational	0	1	Intel	Unsigned	0	1	0	0	1
Steer Angle	8	8	Intel	Unsigned	128	1	128	0	255
Speed	16	8	Intel	Unsigned	128	1	128	0	255
Emergency Brake	24	8	Intel	Unsigned	0	1	0	0	255

Table 1: Motor Control Message 0x560 Documentation Table

Operational is used to activate electric motor, and can be 1 to drive vehicle. Steering angle changes between 0 and 255. Maximum left turn requires sending 0 whereas maximum right turn to send 255. Therefore, straight move requires to send 128. CAN bus does not ask for throttle or brake to control speed in this vehicle. Vehicle control unit is able to provide speed control by applying embedded Proportional-Integral-Derivative (PID) algorithm. Therefore, it asks for just desired speed to apply motor control. Speed is set to have maximum 12 km/h in forward direction with decimal 255, 0 km/h with decimal 128 to stop, and 12 km/h in backward direction with decimal 0. Namely, there is just one gear state, and the vehicle has no gears forward, reverse, and neutral as given in Gazebo simulation model and ROS Noetic package CartSim. Emergency brake provides motor brake to block undesired maneuvers by taking 255. Therefore, it has to be 0 for proper movement.

When we want to control the vehicle by AGX Xavier, we have to listen ros topics of ZED2 camera via USB port, and IMU and GPS data via the CAN tranciever of Xavier. Besides, the desired speed and steer commands to navigate the vehicle should be sent via the same way.

5. Modeling and Control of Vehicle Dynamics

An autonomous driving system needs a vehicle model to control the model’s parameters. There are two modeling techniques to represent a vehicle behavior such as bicycle kinematic model and dynamic model. The racetrack which the vehicle has to move on, most probably have good road conditions i.e. no rainy and snowy weather. Therefore there will be no need for tire slipping model. Furthermore, model simplicity is another important criteria to begin an autonomous driving system development. All in all, kinematic bicycle model was preferred because it just handles the vehicle and road geometric characteristics. Firstly, kinematic bicycle model will be introduced, then 2D Controller will be developed on this model.

5.1. Modeling the Vehicle: Bicycle Kinematic Model

The bicycle kinematic model that will be develop is called the front wheel steering model. Assuming that the vehicle operates on a 2D plane denoted by the inertial frame F_i. In this bicycle model, the front wheel represents the front right and left wheels of the car, and the rear wheel represents the rear right and left wheels of the car. Figure 4 below shows the setup of the bicycle kinematic modeling.

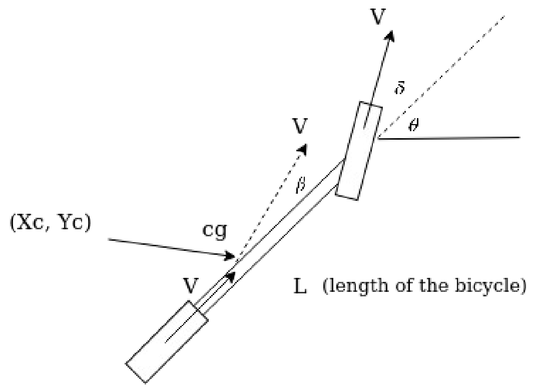


Figure 5: Bicycle Kinematic Modeling, Step 2 [4]

5.1.a State Space Representation

It is not usually possible to instantaneously change the steering angle of a vehicle from one extreme of its range to another, as is currently possible with our kinematic model. Since δ is an input that would be selected by a controller, there is no restriction on how quickly it can change which is somewhat unrealistic. Instead, the kinematic model can be formulated with four states: X, Y, θ, and the steering angleδ. If assuming that the rate of change of the steering angle φ can be the only controllable parameter, the model can be simply extended to include δ as a state and use the steering rate φ as modified input[5].

State: $[X, Y, \theta, \delta]^T$ Inputs: $[v, \varphi]^T$

$$\dot{X}c = v\cos(\theta+\beta)$$
$$\dot{Y}c = v\sin(\theta+\beta)$$
$$\dot{\theta} = \frac{v\cos\beta\tan\delta}{L} \tag{2}$$
$$\dot{\delta} = \varphi$$

Now, this model can be used to design 2D controller.

5.2. 2D Controller

The controller of Bold Pilot 2.5 consists of two parts: Longitudinal Speed Control with Proportional-Integral-Derivative (PID) and Lateral Control. However, the CAN bus of autonomous-ready vehicle accept just speed and steer. Vehicle controls speed by an embedded PID algorithm. Therefore, we have to downgrade controller to just control laterally in real system even if the system uses longitudinal control in simulation .

5.2.a Longitudinal Speed Control with PID

Vehicle speed will be controlled by keeping at a reference speed by throttling and braking. The diagram in Figure 6 below shows how to apply control.

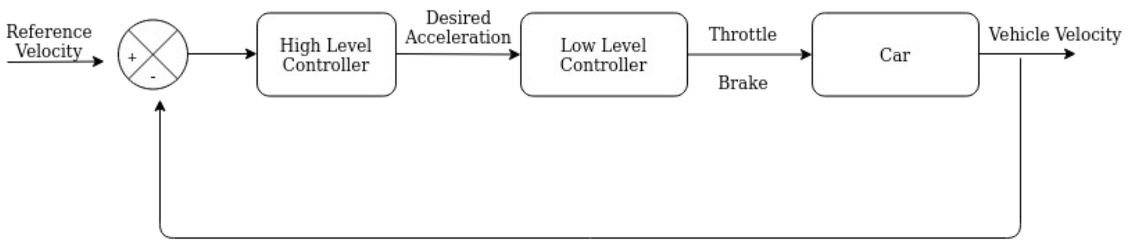


Figure 6: The Longitudinal Controller Design

5.2.a.I High Level Controller

The high-level controller determines the desired acceleration for the vehicle based on the reference and the actual velocity where the PID gains are K_P, K_I and K_D [6].

$$X\ddot{d}es = K_P\left(X\dot{r}ef-\dot{X}\right) + K_I\left(\int_0^t\left(X\dot{r}ef-\dot{X}\right)\mathrm{d}t\right) + K_D\left(\frac{d\left(X\dot{r}ef-\dot{X}\right)}{dt}\right) \tag{3}$$

5.2.a.II Low Level Controller

The low-level controller will just convert the desired acceleration to throttle and brake by limiting the acceleration in a boundary and outputting as the throttle or brake in software.

5.2.a.III Parameter Preferences of Bold Pilot 2.5

Bold Pilot 2.5 uses K_P, K_I and K_D equal to be 0.2, 0.035, 0.25, respectively. These values were decided experimentally.

5.2.b Lateral Control Problem

To design a lateral controller for an automobile, a reference path is needed to track. Also, ego vehicle has already in a direction in the path so to achieve the reference path there has to be an error term relative to the reference path. Then, a control law is required to drive the errors to zero and satisfy input constraints. Finally, adding dynamic considerations help the vehicle manage forces and moments acting on itself.

5.2.c A Geometric Path Tracking Solution to Lateral Control: Stanley Approach

Any controller that uses only the geometry of the path and the vehicle kinematics to track a reference path is a geometric path tracking solution. These approaches ignore dynamic forces on the vehicles and assumes there are no slip holds at the wheels. Therefore, their performance suffer from slip conditions when the vehicle is aggressively maneuvered.

Stanley approach was firstly used by Stanford University's Darpa Grand Challenge Team. It uses the center of the front axle as a reference point and looks at both the heading and cross track errors. It defines a steering law by following up the process such that:

- Correct heading error
- Correct cross track error
- Obey maximum steering angle bounds

Combining the three steps into one line gives us Stanley control law.

$$\delta(t) = \psi(t) + \tan^{-1}\left(\frac{ke(t)}{v_f(t)}\right),\ \delta(t) \in \left[\delta_{min}, \delta_{max}\right] \tag{10}$$

When we look into the error dynamics of the law when the steering angle is not at the maximum:

$$\begin{aligned}\dot{e}(t) &= -v_f(t)\sin(\psi(t) - \delta(t)) = -v_f(t)\sin\left(\tan^{-1}\left(\frac{ke(t)}{v_f(t)}\right)\right) \\ &= \frac{-ke(t)}{\sqrt{1+\left(\frac{ke(t)}{v_f}\right)^2}}\end{aligned} \tag{11}$$

For small cross track errors the denominator can be simplified by assuming the quadratic term is negligible. Then, the equation turns out the ordinary differential equation such that [13],

$$\dot{e}(t) \approx -ke(t) \tag{12}$$

5.2.c.I Parameter Preferences of Bold Pilot 2.5

Bold Pilot 2.5 uses a cross-track dead-band or threshold as 0.01 to prevent from oscillations in lateral direction. Therefore, any cross-track error smaller than 0.1 is set to 0.0. Also, system prefers the gain k equal to be 10 at numerator is set experimentally. Controller was oscillating much in lower speeds like 0.1 m/s or smaller because arc tangent term enlarges suddenly when the vehicle starts to movement. Therefore, 1 added to speed to smooth control in smaller speeds.

6. Autonomous Driving Algorithms

Bold Pilot 2.5 needs data of 4 sensors to output driving commands. This system is set on the collaboration of the subsystems such that visual perception, state estimation and localization, motion planning and 2D controller as stated in Figure 1 in *Team Organization* section. 2D controller was introduced in detail under Modeling and Control of Vehicle Dynamics topic. Controller needs two information to output: Current state and interpolated waypoints, as stated in Figure 1. Current state consists of current east, north, and yaw that are coming from the state estimation subsystem. State estimation obtains current state by fusing noisy IMU and GPS data. Besides, motion planner uses 3 information to output planned trajectory. These are trajectory in global frame, a certain point of object bounding boxes in global frame, and objects' class names. Visual perceiver obtains these three outputs by processing and manipulating RGB and depth raw data. After this abstract introduction, every subsystem with their novel algorithms will be explained in detail by the order such that visual perceiver, motion planning, and state estimation and localization.

6.1 State Estimation and Localization

6.1.c Sensor Fusion using EKF

After getting ready to use odometer topic via ZED2 camera node, we by-passed our algoritm uses GPS and IMU fusion in preliminary report . This odometer information is obtained by visual odometer and IMU sensor information [x]. However, the odometer information published by ZED2 does not contain velocity information so we derivated the position by time to obtain this velocity then published in our state estimator node.

6.2 Visual Perception and Mapping

While driving, human drivers act with what they hear and see around them. They keep the vehicle in line by processing what they see, and create a following distance by calculating the distance of the vehicle in front. They plan their movements by looking at the signs. Autonomous cars are required to behave like human drivers, or even better than them. For this reason, visual perception is the most important component of the system. This module allows us to calculate where pixels are in the real world using camera images and parameters. It calculates way-points according to the data coming from the camera and decides its movement according to the signs.

When the autopilot activated , in the Visual Perception package, the first thing we see is the img_pipeline function, where we perform all the calculations for lane detection. In this function, we first convert the RGB camera image to binary format, which is a format that we can process more easily. In other words, we obtain a binary image from the RGB image according to certain thresholds. When obtaining a binary image, we first apply Gaussian Blur to the camera image. Gaussian blur (Gaussian smoothing) is pre-processing step used to reduce the noise from image (or to smooth the image)[1]. A Gaussian blur with kernel size 3 is applied in the corresponding function. Then, a Sobel filter was applied on the blurred image with the threshold between 0 and 100. The Sobel filter is used for edge detection. It works by calculating the gradient of image intensity at each pixel within the image. It finds the direction of the largest increase from light to dark and the rate of change in that direction[2]. Figure 1 shows the flowchart of this process.

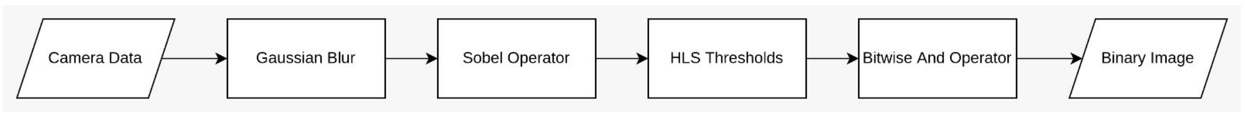


Figure x

On the blurred image, for HLS (hue, saturation, lightness), h value is kept constant, s value is set between 0 and 80 and l value is between 115 and 225.Thus, we obtained two separate binary images from the blurred image, one according to the HLS thresholds and one according to the Sobel thresholds. Then, by processing these two images with the “bitwise_and()”[3] operator provided by cv2, we obtain the final binary image. Figure 2 has , original image with region of interest, binary image and birds-eye view of the image respectively.

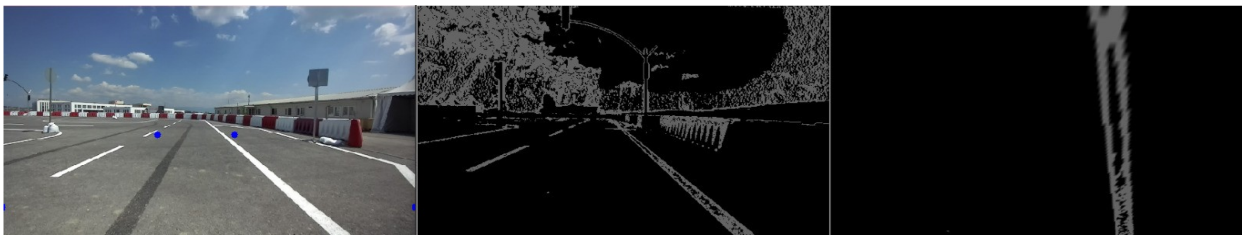


Figure x

Then we pass the binary image we have obtained to the warp_image function to obtain a birds-eye view. In this function, we set the destination points and source points that we want to warp, which we also set parametrically .In this figure, dots represents the region of interest. Then we warp the binary image with the perspective transform function that OpenCV offers us. So we get a birds-eye view of the road. Figure 3 shows the algorithm behind this process.

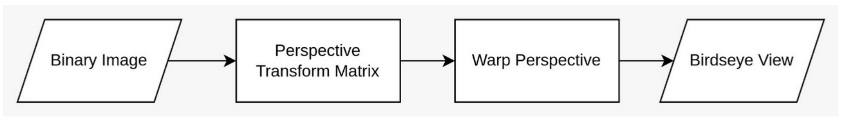


Figure x

On the left side of Figure 4 , binary white image, white areas are expected to be stripe lines. It then detects lane lines using an efficient search method, window and margin search. Of course, before that, when the module first starts, the strip widths are calculated and saved in RAM, thanks to the camera parameters.

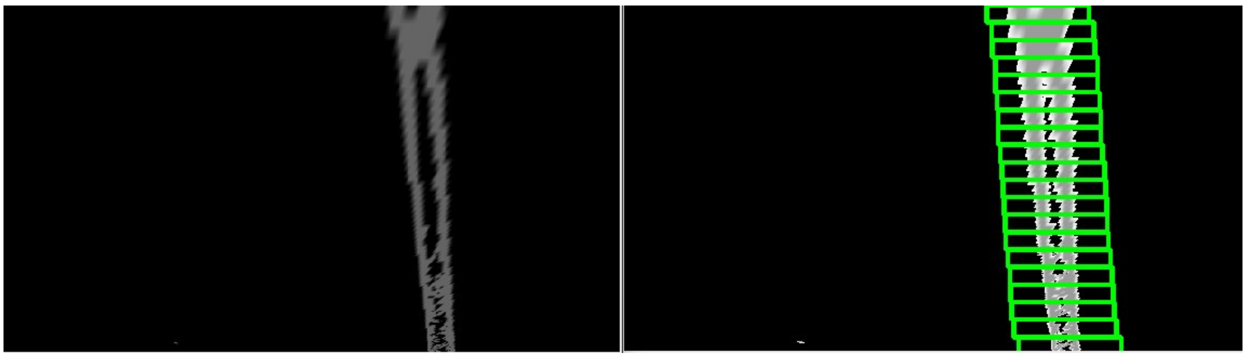


Figure 4

Thus, a polynomial is drawn parallel to the lane lines (3rd image in Figure 6) , depending on the speed. This polynomial contains way-points. This polynomial curve updated by images.

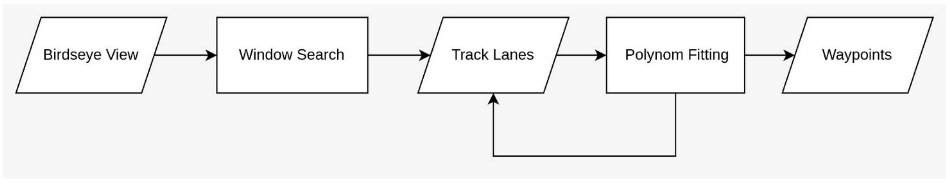


Figure 5

In summary, the img_pipeline() function detects the bird's-eye view of the road and the position of the lane lines by applying certain filters to the camera images. It keeps the vehicle in line with the lane by constantly updating it. (Figure 6)

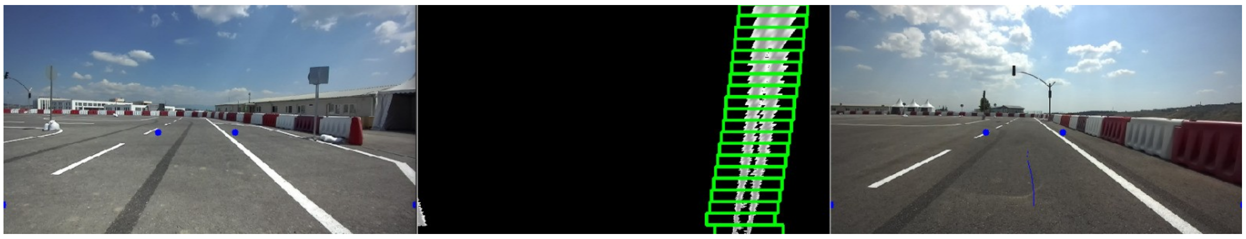


Figure x

Now, we have left and right polynomials of the lane. However, the car has to follow the middle of lane whose algorithm was given in the following section.

6.2.a.I Middle of Lane Estimation Method

Our novel method illustrated in Figure 11, uses two lane lines only when the vehicle starts to move. The algorithm flow could be given such that:

- Two lane lines estimated at the first system loop, and are used to evaluate lane width. This lane width information is kept until program dies.
- The half of lane width is substracted from the right lane line pixels so that middle of the lane is reached. Algorithm uses right lane line in default so that it does not provide extra effort when it receives a right turn.

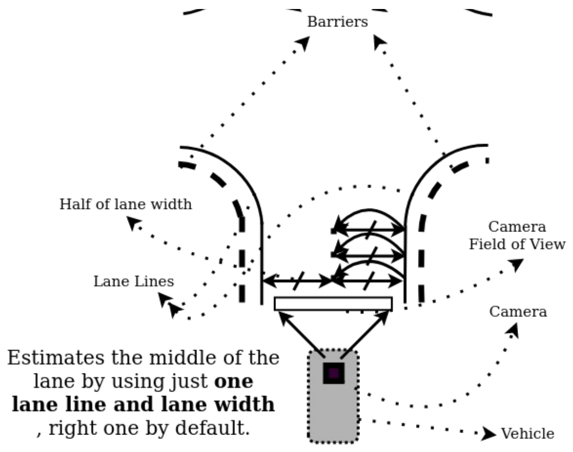

Barriers
Half of lane width
Lane Lines
Camera Field of View
Camera
Estimates the middle of the lane by using just **one lane line and lane width** , right one by default.
Vehicle

Figure 11 : Bird-Eye View Illustration of Our Novel Middle of Lane Estimation

Bold Pilot 2.5 uses the algorithm above for lane keeping behaviour. Actually, more important advantages are used to manage 4-way intersection. The perceiver orientates this algorithm according to behavior planner commands. Perceiver manages the three algorithms by using three flags such as *estimate_right_line*, *estimate_left_line*, and *keep_lane_middle*. It makes each flag 'True' or 'False' according to the command received from the behaviour planner. Because of boring details in software part of usage of these flags , there will be no further detailed information for this section. Every process stated in above is done in pixelwise coordinates. Figure 12 below shows the illustration of the up-to-date middle of the lane estimation. The green line in the middle of the lane is the algorithm's output.

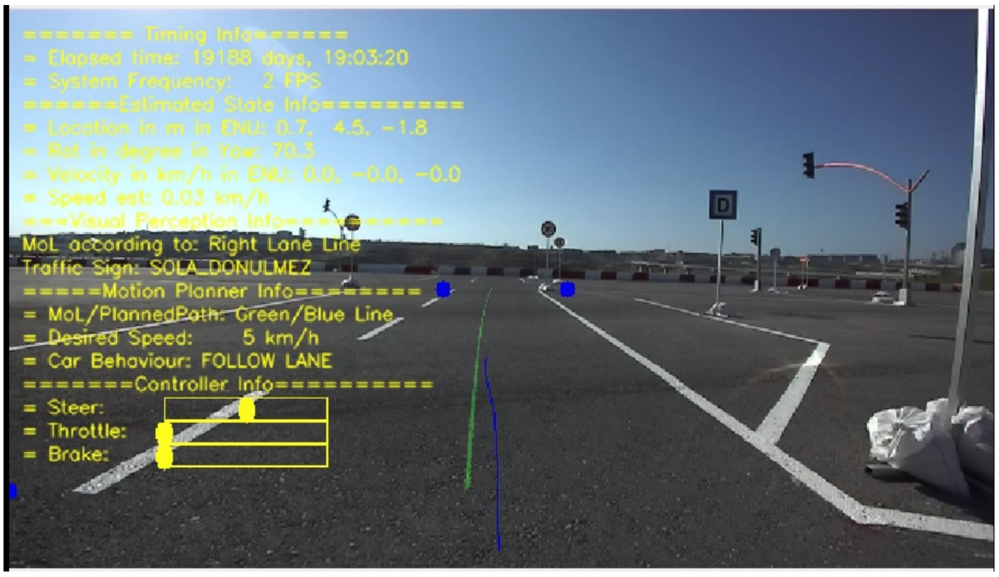
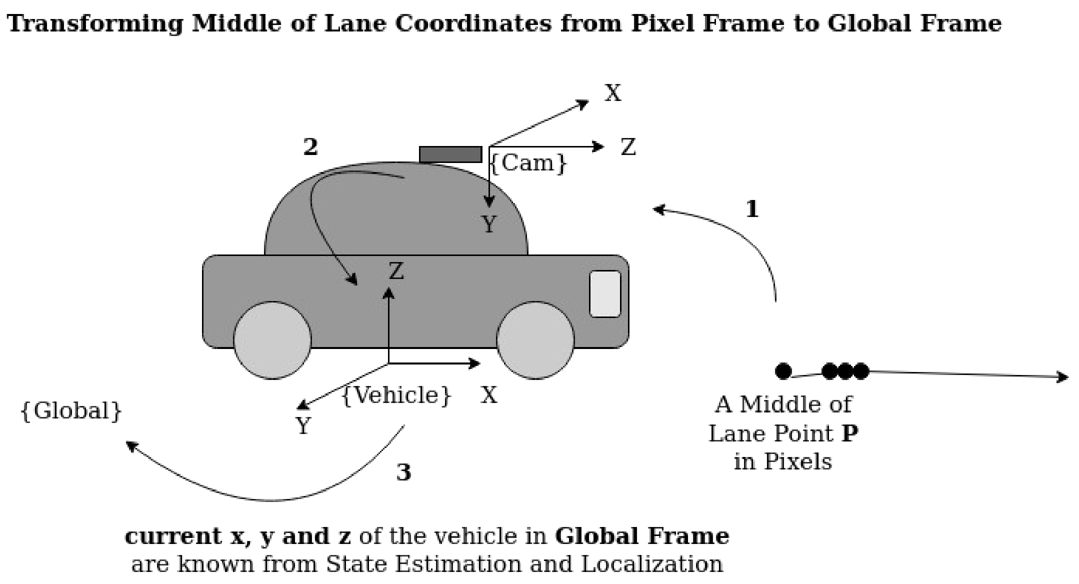

```
======= Timing Info=======
= Elapsed time: 19188 days, 19:03:20
= System Frequency:   2 FPS
======Estimated State Info==========
= Location in m in ENU: 0.7,  4.5, -1.8
= Rot in degree in Yaw: 70.3
= Velocity in km/h in ENU: 0.0, -0.0, -0.0
= Speed est: 0.03 km/h
===Visual Perception Info==========
MoL according to: Right Lane Line
Traffic Sign: SOLA_DONULMEZ
=====Motion Planner Info==========
= MoL/PlannedPath: Green/Blue Line
= Desired Speed:     5 km/h
= Car Behaviour: FOLLOW LANE
=======Controller Info==========
= Steer:
= Throttle:
= Brake:
```

Figure 12: Illustration of the Middle of the Lane (MoL) Estimation in Real Racetrack

Perceiver and behaviour planner collaboration will be explained in *Behaviour Planning Solution: Finite State Machines* section of *Motion Planning* topic by using related figures of scenarios to make communication flow between the two systems more concrete.

6.2.b Transform Pixelwise coordinates to Real World Trajectory Coordinates

In this section, this step of perception will transform the pixel obtained from above algorithm into global frame. Because in the algorithm above we just get coordinates in images not real world so we need to transfer the coordinates into the real world coordinate system. Figure 13 shows the related algorithm of this step.


Transforming Middle of Lane Coordinates from Pixel Frame to Global Frame
X
Z
{Cam}
Y
2
1
Z
{Vehicle}
X
Y
{Global}
3
A Middle of Lane Point P in Pixels
current x, y and z of the vehicle in Global Frame are known from State Estimation and Localization
Figure 13: Coordinate Transformation Between Pixel and Global Frame

1. **Transformation from Pixel Frame to {Cam} Frame Procedure**

In this stage, transform the middle line (pixelwise) that obtained from the algorithms described above into the cam frame.

2. **Transformation from Depth Camera Frame to {Vehicle} Frame Procedure**

Rotate each point by current heading (yaw) in radian to obtain each trajectory point in {Vehicle} frame in such a way that:
- Rotate the point by 90 in degree about X axis regarding right-handed frame rule.
- Rotate the point by 90 in degree about Z axis regarding right-handed frame rule.
- Translate the point from the camera center location to the vehicle center of gravity location.
- Rotate each point by current heading (yaw) in radian to obtain each trajectory point in {Vehicle} frame

3. **Transformation from {Vehicle} Frame to {Global} Frame**

Transform the point from {Vehicle} frame to {Global} frame using current location information coming from State Estimation and Localization subsystem.

6.2.c Draw Trajectory

This section will cover how to draw planned trajectory. Actually it will do the opposite of the previous operation. The obtained trajectory in global frame will be transformed to pixel frame. Figure 14 illustrates how to apply the algorithm.

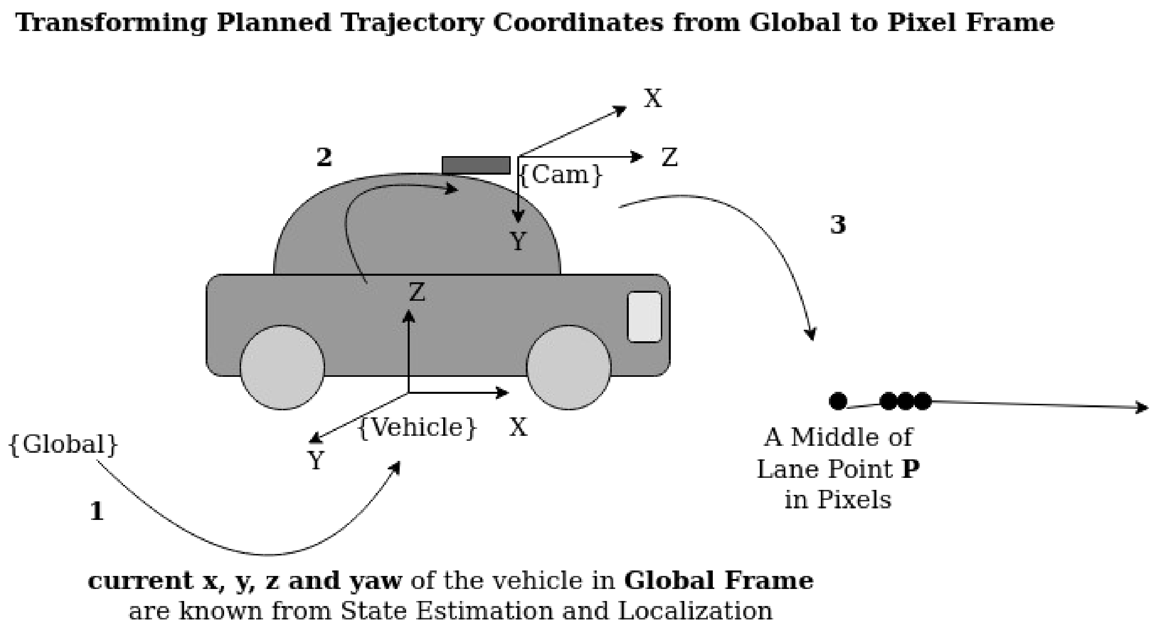

Transforming Planned Trajectory Coordinates from Global to Pixel Frame
X
Z
2
{Cam}
Y
3
Z
{Global}
Y
{Vehicle}
X
A Middle of Lane Point P in Pixels
1
current x, y, z and yaw of the vehicle in Global Frame are known from State Estimation and Localization

Figure 14: Coordinate Transformation Between Global and Pixel Frame

1. **Transformation from Global Frame to Vehicle Frame Procedure**

- Transform the point from {Global} frame to {Vehicle} frame using current state information coming from State Estimation and Localization subsystem.
- Rotate the points -current yaw about Z axis so that the vehicle has zero heading in local frame.

2. **Transformation from Vehicle Frame to Depth Camera Frame Procedure**

- Translate the point from the vehicle's center of gravity location to the camera location
- Rotate the point by -90 in degree about Z axis regarding right-handed frame rule.
- Rotate the point by -90 in degree about X axis regarding right-handed frame rule.

3. **Transformation from Depth Camera frame to Pixel Frame Procedure**

Use stereo camera model to obtain back the points in X and Y pixels [9].

$$X = (x\_3d * fx) / z\_3d + image\_width / 2 \tag{13}$$
$$Y = (y\_3d * fy) / z\_3d + image\_height / 2 \tag{14}$$

where x_3d, y_3d and z_3d are the trajectory points' location in the depth camera frame; fx and fy are the focal lengths of the left lens of ZED2 camera that is given by the ROS topic */zed2/zed_node/depth/camera_info*. Finally, image_width and image_height are the length of the image dimensions.

6.2.d Object Detection

This part will cover How model differs from preliminary design and simulation report and how new model works.

Firstly, we changed our model from Tiny-YOLOV4 to YOLOV4. Because in some cases like the traffic signs that far away from the camera aren't recognized by tiny-yolov4 due to lack of feature extraction layers. Also some specific traffic signs like ‘sağa dönülmez’ and ‘sola dönülmez’ needs more features to determine the class correct because these two traffic signs have lots of common features and so we need more feature extraction layers to make model recognizing the traffic signs like these correctly. That’s why we prefer yolov4 over tiny-yolov4. We used darknet_ros repository[40] to run our model efficiently and it allows me to integrate the model easier to our system because it works in ros environment already.

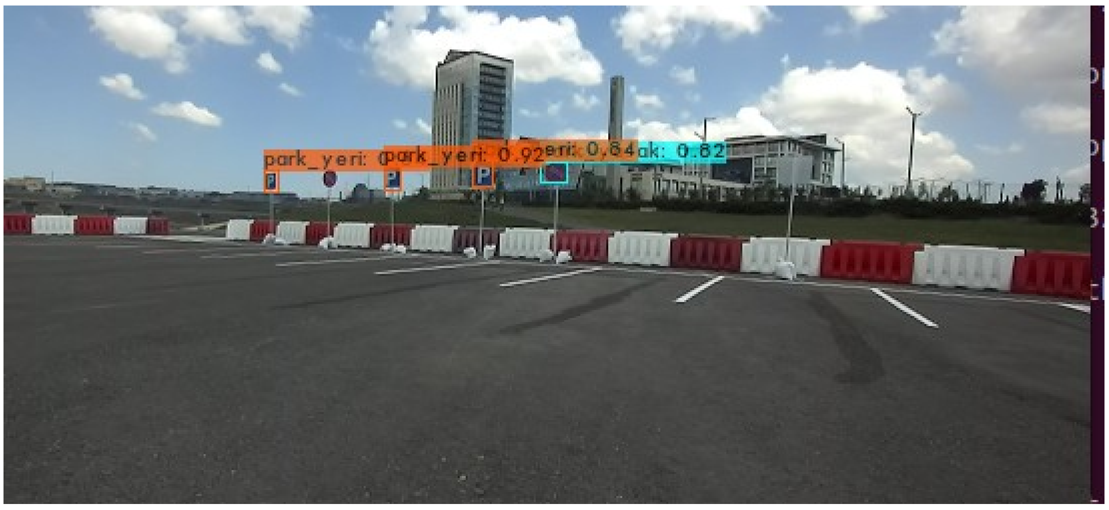
6.2.d.I How to Predict Bounding Box

YOLOV4 is working as same as tiny-yolov4 but extra layers.

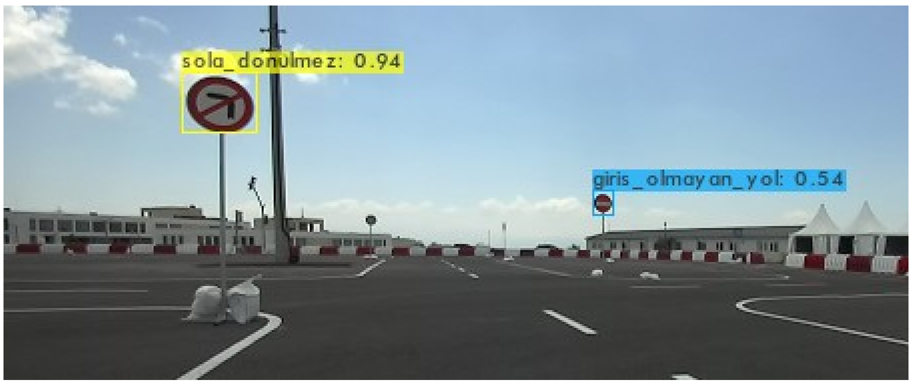
- Darknet_ros is subscriped to /zed2/zed_node/right_raw/image_raw_color topic which comes from the zed camera and its type is Image.
- Then divides image into n pieces.
- Then, for each pieces, it divides to n anchor boxes
- For each anchor box, it tries to find the object
- Then, if confidence score is below the threshold, it removes the prediction
- Finally, we will have objects that have high confidence score with bboxes.
- Then it publishes bbox results to /darknet_ros/bounding_boxes
- Also it publishes image with bbox to /darknet_ros/detection_image.

These are the some results from our model.

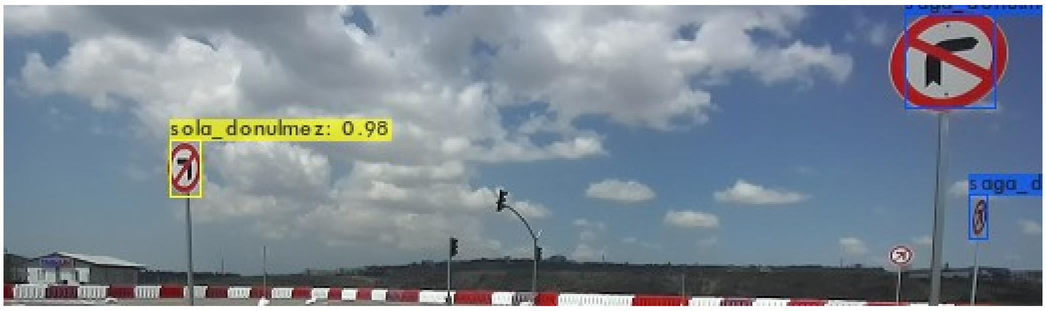
Scenario 1: Park scenario



Scenario 2: Turn Right



Scenario 3: Move Forward



As seen in the picture above, false positive results still exists and we are working on these problems to solve.

6.2.d.IV 3D Coordinates of Bounding Boxes

Visual perception uses the bounding box information of objects to keep their locations to assist behaviour planner. Firstly, it outputs every detection result as a list such that:

[‘class_name’, ‘X_min’, ‘Y_min’, ‘X_min + width’, ‘Y_min + height’, confidence_score]

‘X_min and Y_min’ are the left-most, and ‘X_min + width’ and ‘Y Min + height’ are the rightmost pixels, and ‘confidence score’ is the reliability in percent. After getting those information of objects, the perceiver let some go to behaviour planner to assist maneuvers in driving scenarios after getting 3D coordinates of them. These class names are ‘durak’, ‘kırmızı_ısık’, and ‘park’. Passing from 2D to 3D is done with the same way stated in 6.2.b *Transform Pixelwise coordinates to Real World Trajectory Coordinates.*

6.3. Motion Planning

6.3.a Behaviour Planning Problem

A behaviour planning system plans the set of high level maneuvers to safely achieve the driving mission under various driving situations. The planner for this competition will be enough by considering:

- Rules of the road. i.e. Turkish traffic signs and signals
- Static objects around the vehicle. i.e. Bus stop and red signals
- Static objects on the racetrack that the vehicle must avoid

Figure 16 below, shows the vehicle’s probable actions through the racetrack in bird’s eye view consisting Follow Lane state. The figure illustrates two distinct lane line types: dashed and straight lines. The dashed ones can be driven in current Follow Lane state. However, the straight ones need new solutions that will be determined as new states because there is no enough lane line conditions to apply the current Follow Lane state. To explain all the states except with ‘Park’ state, one 4-way intersection scenario is enough. There are three trajectory options at an intersection. In Figure 17, left, forward and right trajectories are given. There will be many Turkish traffic signs at this intersection or through a straight road to choose proper trajectory such as:

- ‘sola_mecburi_yon’, → ‘saga_mecburi_yon’,
- ‘sola_donulmez’, → ‘saga_donulmez’,
- ‘ileri_veya_sola_mecburi_yon’, → ‘ileri_veya_saga_mecburi_yon’,
- ‘dur ’, → ‘kirmizi_isik’ → ‘yesil_isik’
- ‘giris_olmayan_yol’, → ‘tasit_giremez’
- ‘park_yeri’ → ‘park_yasak’
- ‘durak’

4.3.b Behavior Planning Solution: Finite State Machines

Bold Pilot orients itself using a behavioral planner in the competition racetrack. This planner manages which behavior will take on action in which scenarios. A behavior means a distinct action taken via a distinct router. In the racetrack of this competition, there are many Turkish Traffic signs and signals, that are routers in our planner. Different traffic signs and signals requires different actions. Basically, a ‘saga_mecburi_yon’ sign requires a vehicle to turn right from the closest intersection while an ‘sola_mecburi_yon’ let the vehicle turns left. In upcoming sections, every behavior will be explained with related signs or signals.

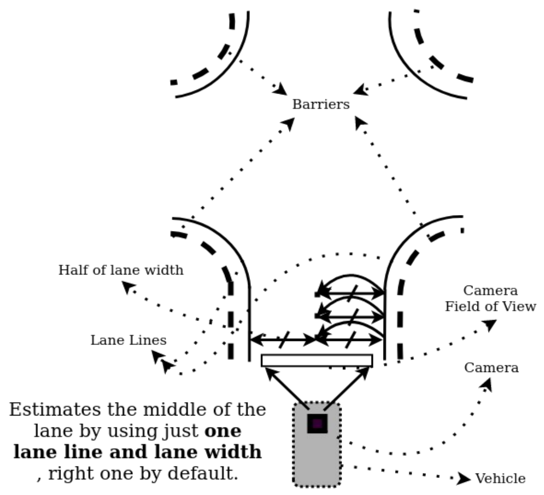
State Machine States
There are 12 states sorted as in software layout. Let’s examine each, respectively.

- Follow Lane: Maintain current trajectory.
- Stay Stopped: Keep stopping for 30 seconds.
- Decelerate to Bus Stop: Decelerate speed to zero and stop near bus stop sign.
- Turn Right: Turn right from the closest intersection.
- Turn Left: Turn left from the closest intersection.
- Bus Stop: Keep bus stop location after bus stop sign detection.
- Stop At Red: Keep red signal location after red signal detection.
- Decelerate to Red Stop: Decelerate speed to zero and stop in front of red signal.
- Stay At Red: Stay in front of red signal until green signal is on.
- Go Ahead: Go ahead through intersection without any turn.
- Park: Find a parking location and plan the route.
- Decelerate to Park: Decelerate speed to zero at the parking location.
- *Collision Detected (New with Critical Design)*
- *Avoid From Collision (New with Critical Design)*
- *Leave Bus Stop (New with Critical Design)*

Every state will be explained under the titles that are Follow Lane, Turn Right, Turn Left , Move Forward, Bus Stop, Red/Green Signal in upcoming sections, respectively.

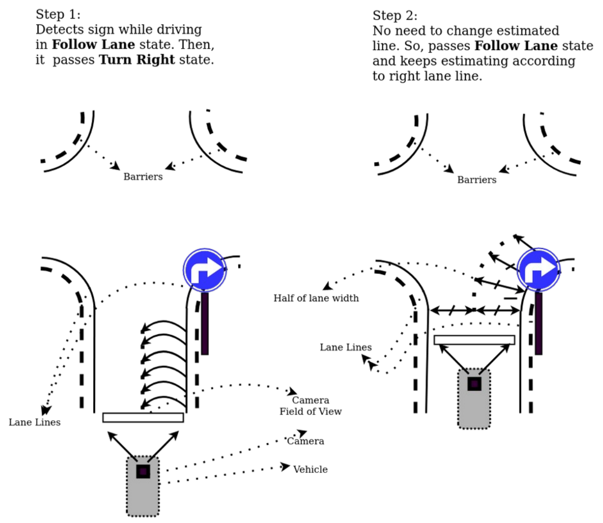
4.3.b.I Follow Lane Scenario

Every scenario starts from and finishes with Follow Lane state. Therefore, vehicle drives itself in Follow Lane state, mostly. Vehicle keeps itself between the lane lines and it’s speed under desired speed in this state.
In order to estimate the middle of lane even when there is one lane line either left or right, there must be new approach. That is because, the camera on the vehicle will see just one lane line while turning left or right from the 4-way intersection. Yet, this estimation method is not scope of the behavior planner so it was explained in detail in visual perceiver


Barriers
Half of lane width
Lane Lines
Camera
Field of View
Camera
Estimates the middle of the lane by using just **one lane line and lane width**, right one by default.
Vehicle
Figure 18 : Follow Lane Approach Illustration
subsystem. This approach let the vehicle turns by using one lane line and lane width information, and is given in Figure 18.

4.3.b.III Turn Right Scenario

Vehicle follows lane by estimating right lane line until awaring of traffic signs or signals. When the vehicle detects ‘saga_mecburi_yon’ sign, it passes Turn Right state, and follows the steps given in Figure 19 below.


Step 1:
Detects sign while driving in **Follow Lane** state. Then, it passes **Turn Right** state.
Step 2:
No need to change estimated line. So, passes **Follow Lane** state and keeps estimating according to right lane line.
Barriers
Barriers
Half of lane width
Lane Lines
Camera
Field of View
Lane Lines
Camera
Vehicle
Figure 19 : Turn Right State Illustration

4.3.b.IV Turn Left Scenario

Vehicle keeps lane by applying Follow Lane until detecting ‘sola_mecburi_yon’ sign. Then, it obeys the state rule given in Figure 20 in step by step.

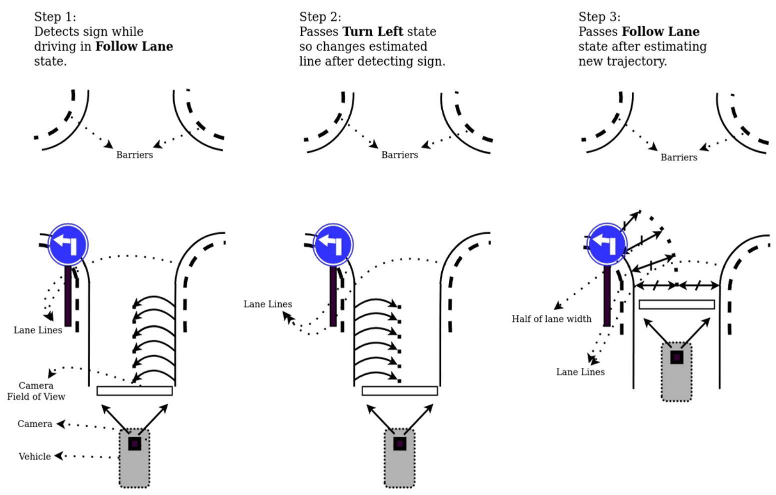

Step 1:
Detects sign while driving in **Follow Lane** state.
Step 2:
Passes **Turn Left** state so changes estimated line after detecting sign.
Step 3:
Passes **Follow Lane** state after estimating new trajectory.
Barriers
Barriers
Barriers
Lane Lines
Camera Field of View
Camera
Vehicle
Lane Lines
Half of lane width
Lane Lines

Figure 20 : Turn Left State Illustration

4.3.b.V Go Ahead (Move Forward) Scenario

Vehicle follows lane until detecting ‘sola_donulmez’, and ‘saga_donulmez’, at the same time as illustrated in Figure 21. After switching Go Ahead state, behavioral planner requests from visual perceiver to change lane estimation method, which was discussed in detail in visual perceiver subsystem. Then, it keeps lane by following right lane line.

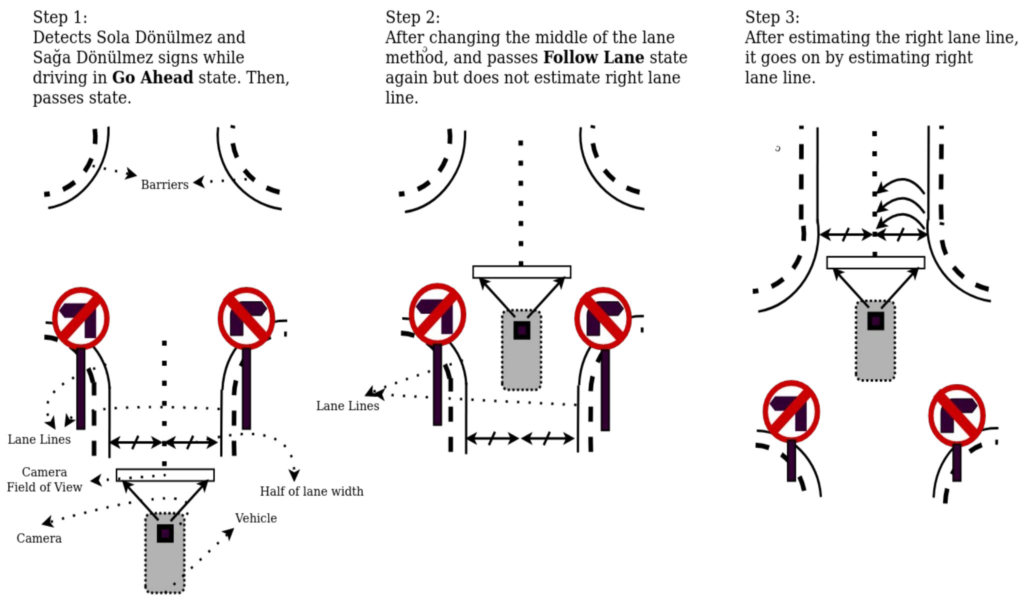
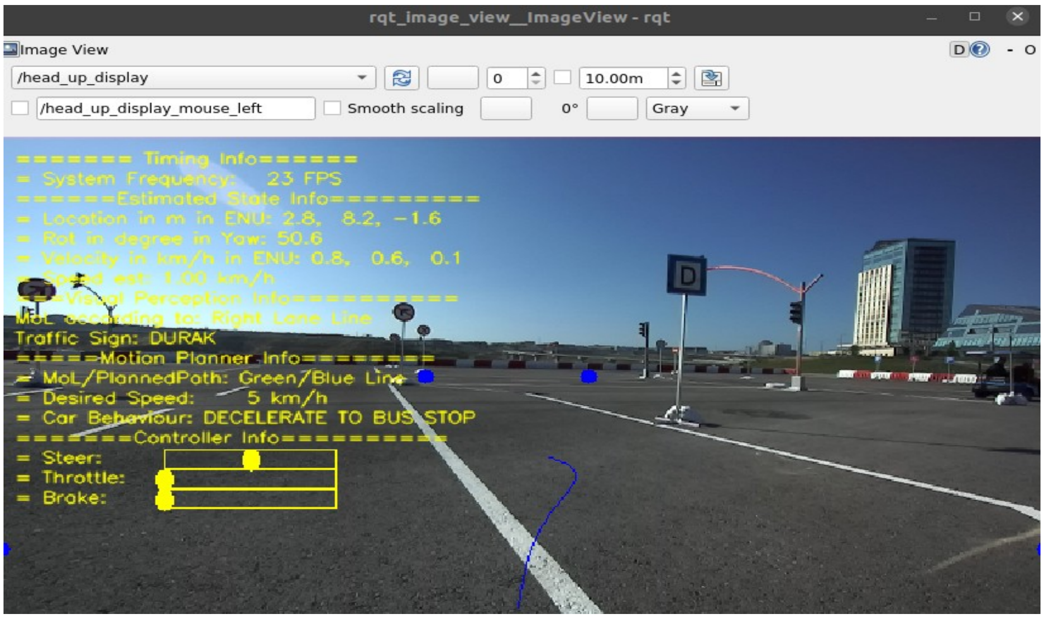

Step 1:
Detects Sola Dönülmez and Sağa Dönülmez signs while driving in **Go Ahead** state. Then, passes state.
Step 2:
After changing the middle of the lane method, and passes **Follow Lane** state again but does not estimate right lane line.
Step 3:
After estimating the right lane line, it goes on by estimating right lane line.
Barriers
Lane Lines
Camera Field of View
Camera
Half of lane width
Vehicle
Lane Lines

Figure 21 : Go Ahead State Illustration

6.2.b.VI Bus Stop Scenario

Visual perceiver subsystem detects ‘Durak’ sign while Bold Pilot drives the vehicle in Follow Lane state. It switches to Bus Stop state after detecting the sign, and keeps the sign’s location. The way of keeping location of a sing or signal was explained in detail in

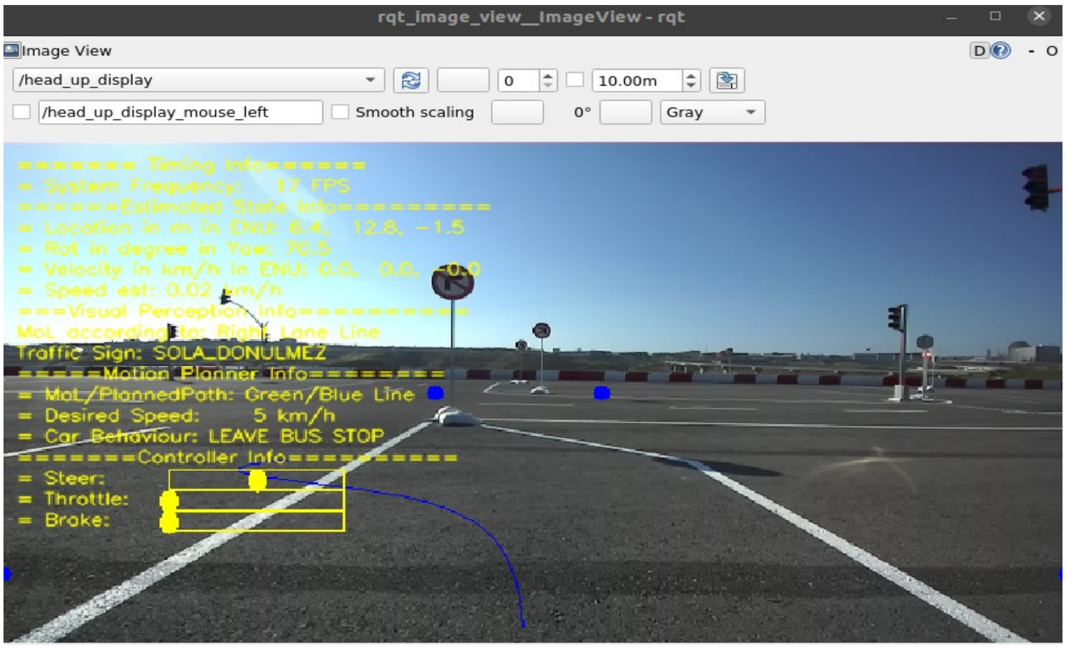
Object Detection section under *Visual Perceiver* topic. Then, it passes to Decelerate to Bus Stop state to decelerate and stop at the bus stop location as can be seen in Figure 26.



```
======== Timing Info=======
= System Frequency:   23 FPS
=======Estimated State Info==========
= Location in m in ENU: 2.8,  8.2, −1.6
= Rot in degree in Yaw: 50.6
= Velocity in km/h in ENU: 0.8,  0.6,  0.1
= Speed est: 1.00 km/h
===Visual Perception Info===========
MoL according to: Right Lane Line
Traffic Sign: DURAK
======Motion Planner Info=========
= MoL/PlannedPath: Green/Blue Line
= Desired Speed:     5 km/h
= Car Behaviour: DECELERATE TO BUS STOP
========Controller Info===========
= Steer:
= Throttle:
= Brake:
```

Figure 26 : Decelerating to Bus Stop State Illustration

To be able to switch to Stay Stopped state, vehicle’s distance to the location of the ‘Durak’ sign has to be less than one meter according to Bold Pilot. This threshold is an empirically chosen value dependent to state estimation subsystem performance. This is because, in the calculation of the distance, the information of the current fused position of the vehicle comes from state estimation subsystem. After decelerating process stops, vehicle waits for 30 seconds at the bus stop. Then, the planner switches to ‘Leave Bus Stop’ state to continue to apply proper lane keeping. All the scenario was illustrated in Figure 27 below.

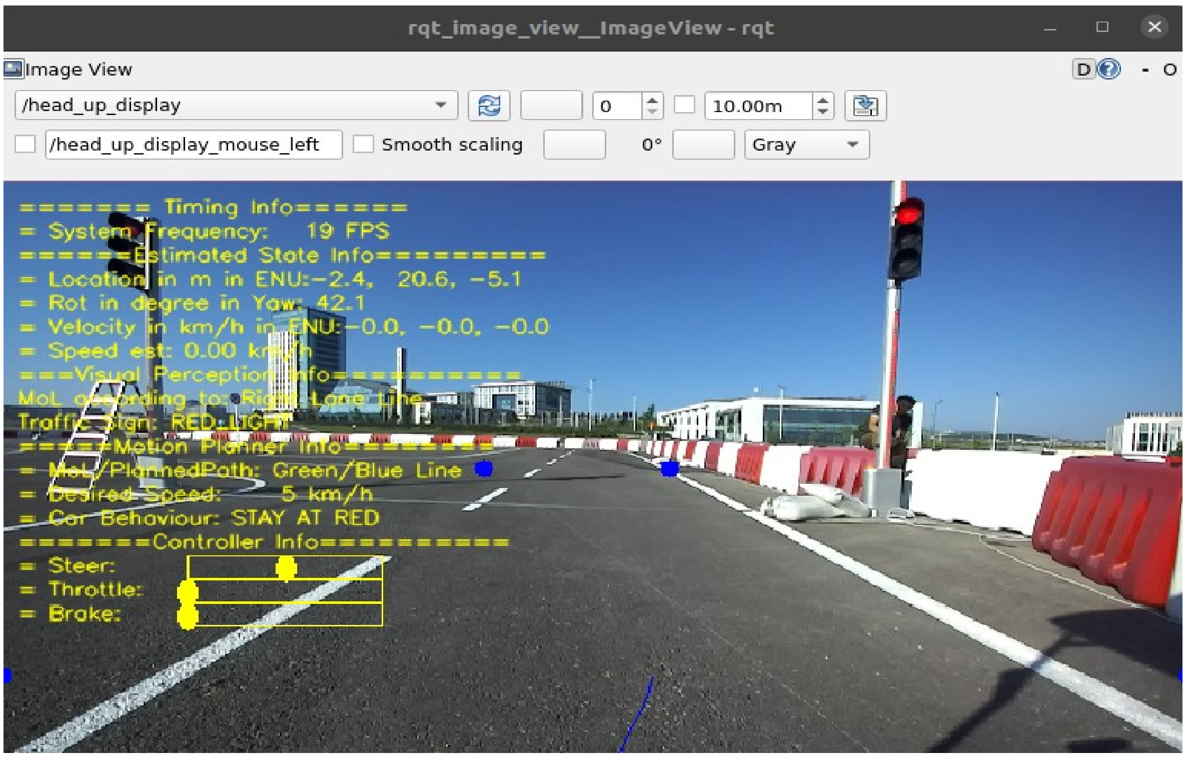


```
======== Timing Info=======
= System Frequency:   17 FPS
=======Estimated State Info==========
= Location in m in ENU: 6.4,  12.8, −1.5
= Rot in degree in Yaw: 70.5
= Velocity in km/h in ENU: 0.0,  0.0, -0.0
= Speed est: 0.02 km/h
===Visual Perception Info===========
MoL according to: Right Lane Line
Traffic Sign: SOLA_DONULMEZ
======Motion Planner Info==========
= MoL/PlannedPath: Green/Blue Line
= Desired Speed:     5 km/h
= Car Behaviour: LEAVE BUS STOP
========Controller Info===========
= Steer:
= Throttle:
= Brake:
```

Figure 27 : Leave Bus Stop State Illustration

6.2.b.VII Red/Green Signal Scenarios

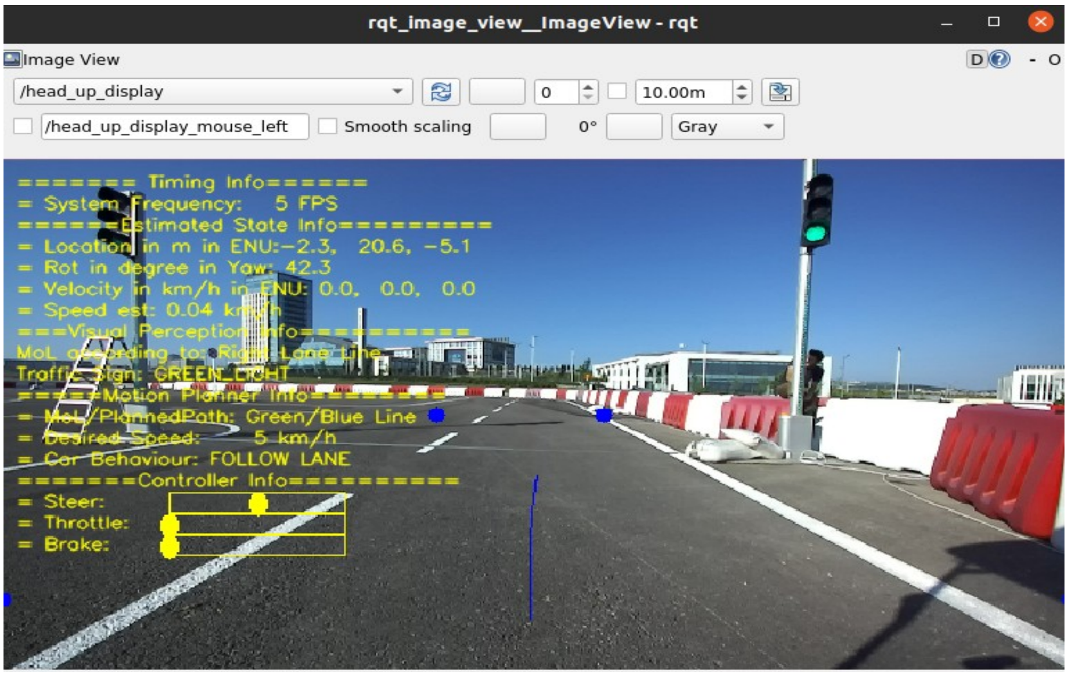
The way of keeping location of a sing or signal was explained in detail in *Object Detection* section under *Visual Perceiver* topic. After determining the target waypoint, planner switches to 'Decelerate to Red Stop' and try to stop at the target point by one more switching to 'Stay at Red' state as can be seen in Figure 29.



```
======= Timing Info======
= System Frequency:    19 FPS
======Estimated State Info=========
= Location in m in ENU:-2.4,  20.6, -5.1
= Rot in degree in Yaw: 42.1
= Velocity in km/h in ENU: -0.0, -0.0, -0.0
= Speed est: 0.00 km/h
===Visual Perception Info==========
MoL according to: Right Lane Line
Traffic Sign: RED LIGHT
=====Motion Planner Info=======
= MoL/PlannedPath: Green/Blue Line
= Desired Speed:       5 km/h
= Car Behaviour: STAY AT RED
=======Controller Info===========
= Steer:
= Throttle:
= Brake:
```

Figure 29 : Stay at Red State Illustration

Then, Bold Pilot follows the steps given in Figure 28. Besides, Bold Pilot stays in Follow Lane state as long as it detects GREEN signal. Therefore, there is no need to design a distinct state for this case.



```
======= Timing Info=======
= System Frequency:   5 FPS
======Estimated State Info==========
= Location in m in ENU:-2.3,  20.6, -5.1
= Rot in degree in Yaw: 42.3
= Velocity in km/h in ENU: 0.0,  0.0,  0.0
= Speed est: 0.04 km/h
===Visual Perception Info===========
MoL according to: Right Lane Line
Traffic Sign: GREEN_LIGHT
=====Motion Planner Info==========
= MoL/PlannedPath: Green/Blue Line
= Desired Speed:     5 km/h
= Car Behaviour: FOLLOW LANE
=======Controller Info============
= Steer:
= Throttle:
= Brake:
```

Figure 30 : Green Signal State Illustration

6.2.b.VIII Some Combined Scenarios

If we think about the scenarios could be possible with the different usage of the traffic signs, the ones discussed above are not all but enough to represent fundamental tasks. There are some other scenarios which Bold Pilot could face through racetrack in Figure 28. At this time, the 4-way intersection was represented in more different way than the ones discussed earlier. This perspective represents the view from camera on the vehicle. There are three signs detected and filtered at the same time. The two bigger signs are the closest ones and come before the intersection, and the other smaller sign, 'giris_olmayan_yol', is the one comes after the intersection. These scenarios were made similar to Go Ahead, Turn Left and Turn Right states to make the decisions easier for Bold Pilot.

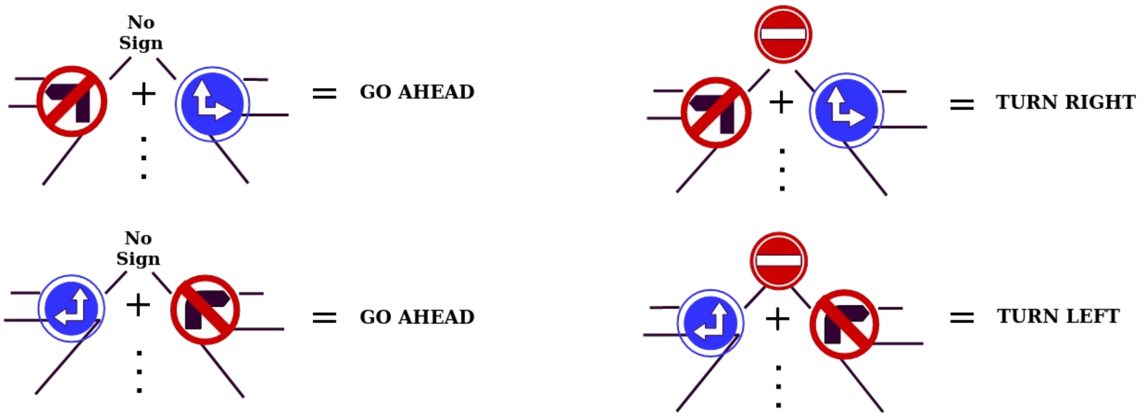
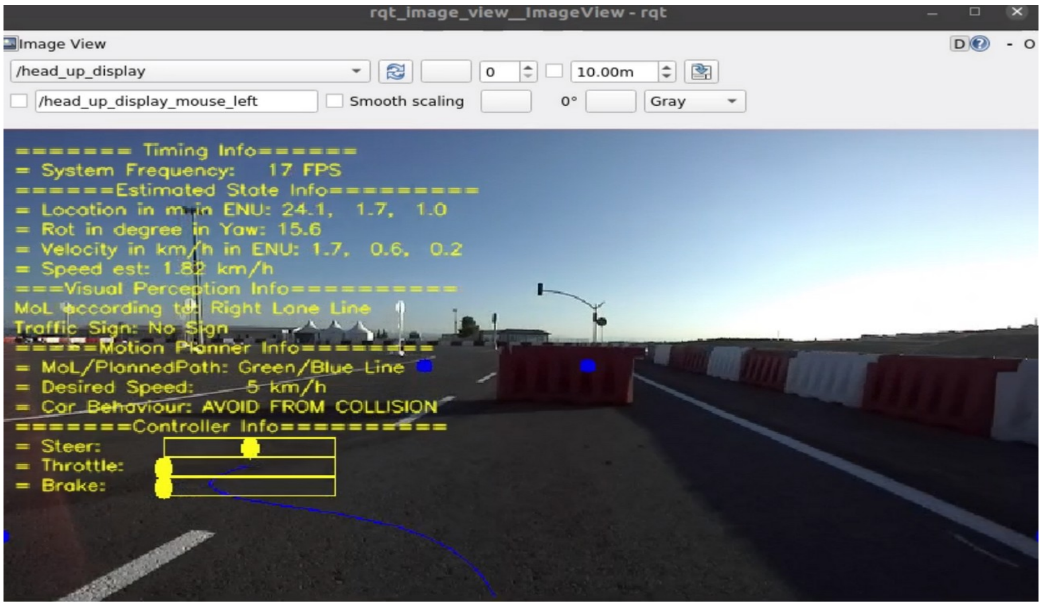


Figure 31 : The Scenarios Made Similar to Turn Left/Right

6.2.b.X Collision Avoidance Scenario

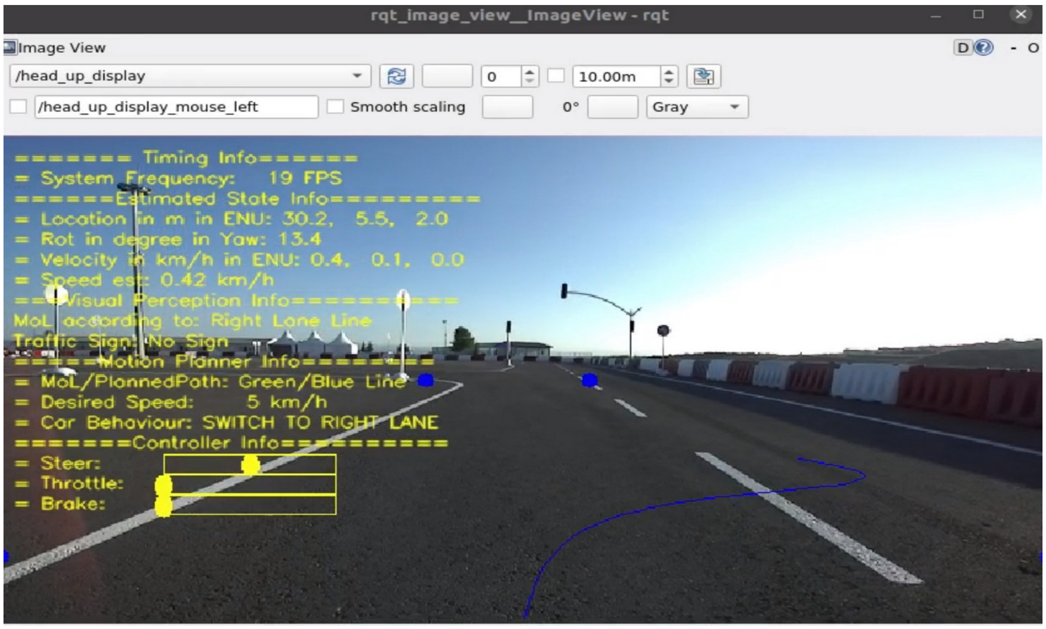
In this scenario, while Bold Pilot continuously checking out its depth map, if it founds out consistent joint pixels close to the vehicle on its following lane, then the state of the car passes immediately to ‘Collision Detected’. After that Bold Pilot prepares itself for collision avoidance by passing its state to ‘Avoid From Collision’ as can be seen in Figure 33.



```
======== Timing Info=======
= System Frequency:   17 FPS
=======Estimated State Info==========
= Location in m in ENU: 24.1,  1.7,  1.0
= Rot in degree in Yaw: 15.6
= Velocity in km/h in ENU: 1.7,  0.6,  0.2
= Speed est: 1.82 km/h
===Visual Perception Info============
MoL according to: Right Lane Line
Traffic Sign: No Sign
=====Motion Planner Info==========
= MoL/PlannedPath: Green/Blue Line
= Desired Speed:     5 km/h
= Car Behaviour: AVOID FROM COLLISION
========Controller Info===========
= Steer:
= Throttle:
= Brake:
```

Figure 33 : Avoid From Collision Scenario

After passing the obstacle by switching to the collision-free lane, Bold Pilot prepares itself for switching back to its previous lane. In order to do this, Bold Pilot takes the advantage of the local position of the obstacle, adds some safe margin to this position information and use it as a new goal state to switch to its previous lane as can be seen in Figure 34.



```
======== Timing Info=======
= System Frequency:   19 FPS
=======Estimated State Info==========
= Location in m in ENU: 30.2,  5.5,  2.0
= Rot in degree in Yaw: 13.4
= Velocity in km/h in ENU: 0.4,  0.1,  0.0
= Speed est: 0.42 km/h
===Visual Perception Info============
MoL according to: Right Lane Line
Traffic Sign: No Sign
=====Motion Planner Info==========
= MoL/PlannedPath: Green/Blue Line
= Desired Speed:     5 km/h
= Car Behaviour: SWITCH TO RIGHT LANE
========Controller Info===========
= Steer:
= Throttle:
= Brake:
```

Figure 34 : Switching to the right after passing the obstacle

6.2.d Smooth Local Planer Solution

6.2.d.I Conformal Lattice Planner for Path Generation

In the lateral control section, it was stated that path can be generated as straight line segments, waypoints, and parametric curves. Parametric curves is the best option to choose because a curve is more similar to a real-life path generation when we consider a human driving style. There were two options for parametric curves such as quintic splines and cubic spirals. Cubic spirals were preferred because of the discontinuity in derivative of path curvature of quintic splines. The path curvature κ is highly important kinematic constraint to take into account. It is represented as

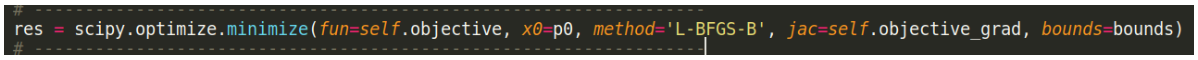
$$\kappa = 1 / R \tag{15}$$

where R is the radius of the circle that vehicle will try to turn around. Therefore, the lower the radius means higher the path curvature. Namely, the vehicle tries to turn around more sharper corner. Cubic spirals obtains proper yaw for the vehicle by integrating κ over path length. Then, x and y locations are obtained by integrating yaw over path length by using Fresnel Integrals. By using Simpson's Rule, these integrals are evaluated numerically. After the related theory we have such function to be optimized that,

$$\begin{gathered} min\, f_{be}\left(a_0, a_1, a_2, a_3, s_f\right) + \alpha\left(x_s\left(p_4\right) - x_f\right) + \beta\left(y_s\left(p_4\right) - y_f\right) + \gamma\left(\theta_s\left(p_4\right) - \theta_f\right) \\ s.t.\ \left|p_1\right| \le \kappa_{max}\ and\ \left|p_2\right| \le \kappa_{max} \end{gathered} \tag{16}$$

where α, β, and γ are softening constraints; x_f, y_f, and θ_f are the final locations and orientation, respectively[20]. More detailed information about Fresnel Integrals and softening constraints, and the reason to use Simpson's Rule were provided under *Appendix* topic.

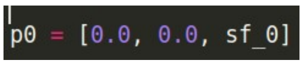
All the optimizations above are done by a single Python function in SciPy library. The minimization function evaluates the result by using 'L-BFGS-B' method and passing the required boundary as given in Figure 35.



```
# ----------------------------------------------------------------------------------
res = scipy.optimize.minimize(fun=self.objective, x0=p0, method='L-BFGS-B', jac=self.objective_grad, bounds=bounds)
# ----------------------------------------------------------------------------------
```

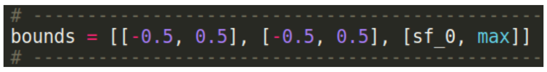
Figure 35 : Application of Path Optimization via Scipy

There is no possibility to parse the *objective* and *objective_grad* functions of optimization. However, we can introduce the other inputs. Initial parameters p0 was preferred such that 0.0 for p0, 0.0 for p4 and sf_0 for path length, given in Figure 36. Bounds for optimization was constructed by preferring κ to be minimum -0.5 and maximum 0.5, given in Figure 37. The lower bound input sf_0 for optimization is a straight line distance calculated in code, and higher bound is system max size because there is no limit for maximum.



```
p0 = [0.0, 0.0, sf_0]
```

Figure 36 : Initial Parameters for Optimization

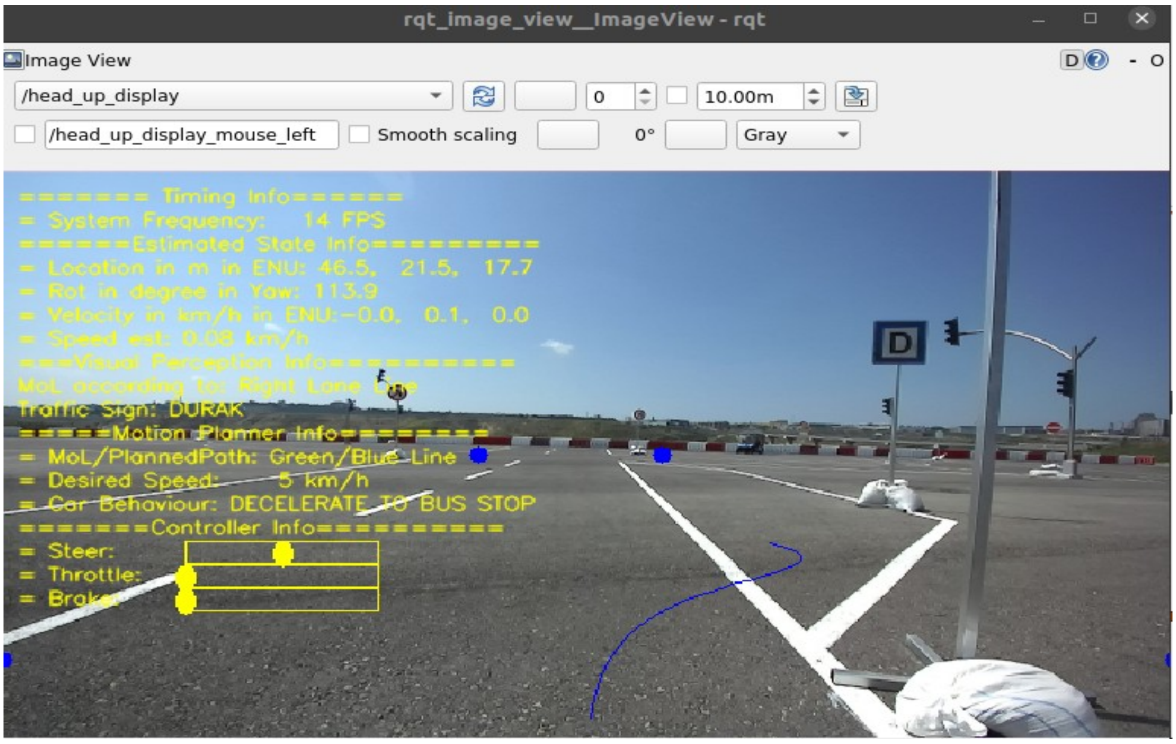


```
# ----------------------------------------------
bounds = [[-0.5, 0.5], [-0.5, 0.5], [sf_0, max]]
# ----------------------------------------------
```

Figure 37 : Bounds Array for Path Optimization

Conformal lattice planner have a goal horizon to generate the spirals. It determines a goal point in look ahead direction, then evaluate many points laterally offsetted with the first goal point, which is called goal set. This goal point is a specific distance far away from the vehicle, and determined before starting the system. This distance is also called as lookahead distance, 7 meter in Bold Pilot 2.5. In other words, The system is able to plan 7 meter in front of itself in every timestamp. Then, it generates the spirals due to the first and last point of the goal set. After generating spirals taking the path curvature into account, the planner converts optimization variables back into the spiral parameters. This is because, the next step is sampling points along spiral using the spiral coefficients. Sampling points is done by another numerical approximation method, that is trapezoidal rule integration. This evaluation is not proper to evaluate with Simpson's rule because of its hard calculation process. The trapezoidal method is more efficient because each subsequent point along the curve can be constructed from the previous one. Figure 38 below shows the constructed curves.

The number of curves will be a parameter in the code of Bold Pilot 2.5 so it will be set to 1 to generate only one path. Also, Figure 39 shows the constructed curve with blue line in pixels.



```
======= Timing Info=======
= System Frequency:   14 FPS
======Estimated State Info==========
= Location in m in ENU: 46.5,  21.5,  17.7
= Rot in degree in Yaw: 113.9
= Velocity in km/h in ENU:-0.0,  0.1,  0.0
= Speed est: 0.08 km/h
===Visual Perception Info===========
MoL according to: Right Lane Line
Traffic Sign: DURAK
=====Motion Planner Info=========
= MoL/PlannedPath: Green/Blue Line
= Desired Speed:       5 km/h
= Car Behaviour: DECELERATE TO BUS STOP
=======Controller Info===========
= Steer:
= Throttle:
= Brake:
```

Figure 39 : Illustration of Cubic Spiral with Blue Line in Pixel Coordinates (20)

7. Software Security Precautions

Bold Pilot 2.5 autonomous driving system was designed regarding autonomy levels beginning from 0 to 5. The first target of the system was to maintain longitudinal and lateral control of the vehicle which is known as level 2 autonomy. The system firstly processes the road image in front of the vehicle. Then if the perception subsystem is successful to estimate the middle of the lane that the vehicle follows, Bold Pilot 2.5 is activated. Furthermore, even if Bold Pilot 2.5 is active for a long time but lane lines are distorted in a section of road for a while, the system is deactivated because there is no middle line to track. It means that the system is active as long as it estimates the middle of the lane, and this case provides security. However, behavior planner will be extended for this case to set the desired speed to 0 km/h when there is no estimated line.

Motion planner subsystem includes a velocity planner that has a trapezoidal profile. It means that desired speed is reached by using one linear, one flat, and one linear acceleration or deceleration behavior, respectively. This characteristic prevents the vehicle from instantaneous acceleration jumps. After the integration of object detection pipeline, required security precautions will be evaluated. Bold Pilot 2.5 is a scalable software system so it can adapt to requirements of new tasks. Every subsystem is responsible for it’s robustness of outputs, and each will be easily manipulated by the designers because Bold Pilot 2.5 is assertive about individuality.

Autonomous-ready vehicle is coming with a built-in security solution. The wireless controller coming with the vehicle is more superior than Xavier in CAN network so that any unexpected behaviour of vehicle can be blocked by the controller. The controller has an emergency brake button to stop the vehicle by using motor brake.

Bold Pilot 2.5 uses another wireless keyboard to run system both in manual and autopilot modes.
The button p in keyboard helps us handle the vehicle control when autopilot is not able to manage an event. Related keyboard inputs are:

- p to switch between manual control and Bold Pilot 2.5
- w to throttle in manual control
- s to brake in manual control
- d to turn right in manual control

- a to turn left in manual control

8. Simulation

In the simulation environment, the data to be produced by the vehicle's sensors in the real world was simulated. First, the model of the vehicle had to be obtained. The vehicle model used last year was suitable for us and some sensors were already installed on it. The shared model was a model written in URDF format. With this format, the x, y, z positions of the sensors in space, roll-pitch-yaw values, standard deviations of the incoming data, noises and many similar parameters could be arranged by adhering to certain references. The parameters used by the sensors are explained below. Positioning the vehicle's sensors correctly is important for success in real-world tests. For this reason, we have assigned parameters according to the measurements made last year.

IMU
The role of the IMU sensor has been explained in previous sections. Its position relative to the chassis was set as x = -0.2 , y = 0 , z = 1.82 in meters. Roll , pitch and yaw values are positioned to be 0 in radian.

GPS
The position of the GPS on the vehicle was positioned as x = -0.2, y = 0, z = 1.83 relative to the chassis, to a point very close to the IMU so that there would be no problem during localization.

Stereo Camera
An open source plugin developed by OpenNI was used for the camera sensor. Through this plugin, we have installed both a depth camera and an RGB camera on the vehicle. In the first version of the URDF code, the camera was located where the driver was. This camera was positioned to be inclined towards the front of the vehicle and towards the road. Updated to x = 0.92 , y = 0 and z = 0.87 in meters relative to the chassis, and the pitch value to be -0.122173, yaw and roll 0 in radian to make it look sloping to the road. While calculating the pitch value, since the parameter is in radian, the assignment was made according to the radian equivalent of 7 degrees. Camera parameters are adjusted to be suitable for ZED2 camera, camera resolution is 1280 * 720 and R8G8B8 format. The noise standard deviation of about 0.007 was assigned. Parameters were assigned on camera distortion, but assuming the cameras would be calibrated later, they were commented out to avoid recalibration. For the Depth camera, the cutoff value of the point cloud

data was adjusted to be min 0.5 and max 50.0. Again, camera distortion parameters have been added, but they have been commented out.

The noise values and position values of the sensors were defined via URDF. We defined the topics where the data was published in the URDF file. The model was run in a gazebo environment via ROS.

10. System Integration

In software development process, it is critical that the code base is *maintainable*, *reproducible* and *portable*. Maintainability means that any developer in the team can integrate their changes without getting stuck on the complexities that the previous developer left behind. We achieve this by using version control systems and Github. Since our system is designed with modularity in mind, this was trivial to set up. Having the benefit of controlling and synchronizing our code base, we implemented a reproducible system by writing documentation in our Github repository. This enabled every team member to setup their environments without wasting time. Lastly and most importantly, a code base has to be portable. Especially in an autonomous car project in which the system and code dependencies are complex and prone to conflicts with other system packages and libraries. For this we containerized our application and its environment with Docker.

When working with a team of developers, a version control system is needed to synchronize and integrate changes in the code. Git works a distributed version control system where the system fully mirrors the entire history of the repository. This allows the team members to have the entire development history in just one folder. A basic diagram of how distributed version control systems work can be seen in the figure below

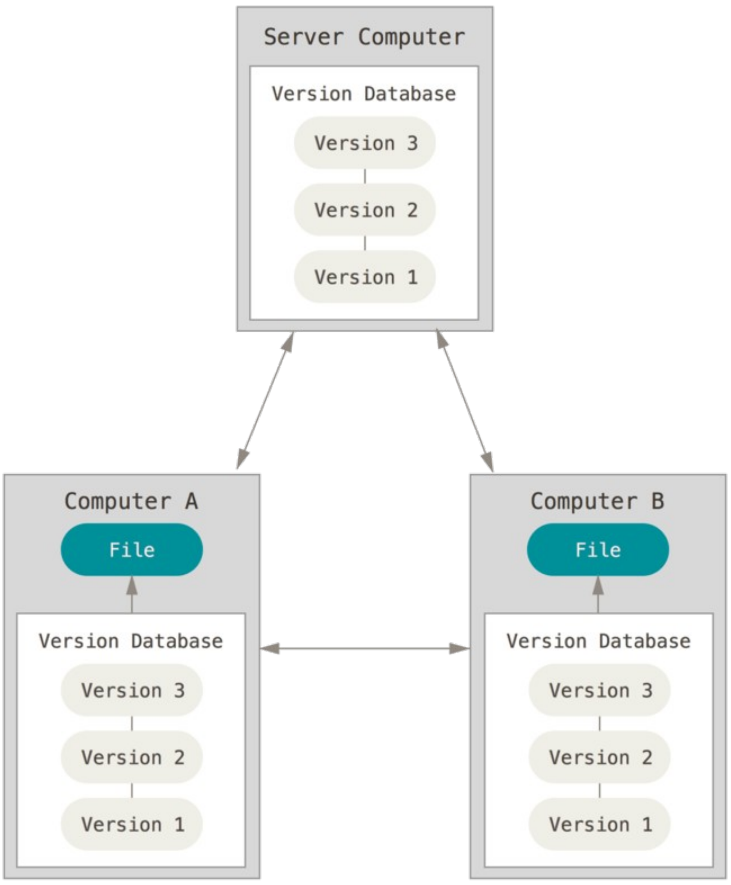


Figure 3 Git distributed version control system diagram

Another challenge was to reduce the time spent to set up a development environment for our project. When anyone in our team has had problems with their environment, it usually took hours, if not days to resolve. We quickly realized a need to make our code base portable and reproducible. Extending the process that began with our version control system, we created documentation to help team members quickly resolve their environment problems. Then we packaged or rather containerized our application, BOLD PILOT. This was achieved with Docker. The way Docker works is by OS-level virtualization to package an application and its dependencies in a container. This allows BOLD PILOT to work on any Linux, Windows and macOS machine. Thus, lowering the complexity to setup a local development environment for our team members.

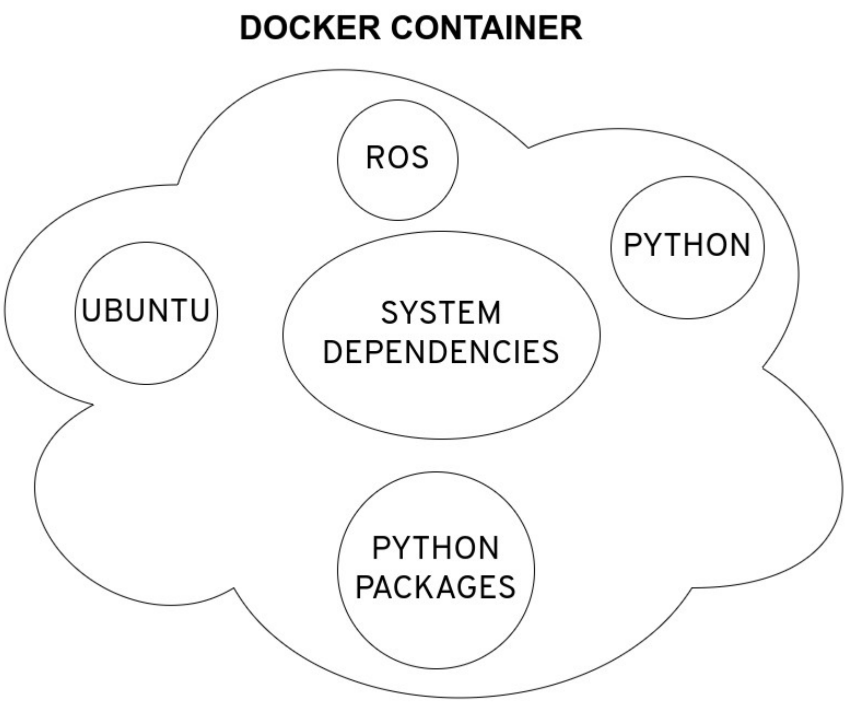
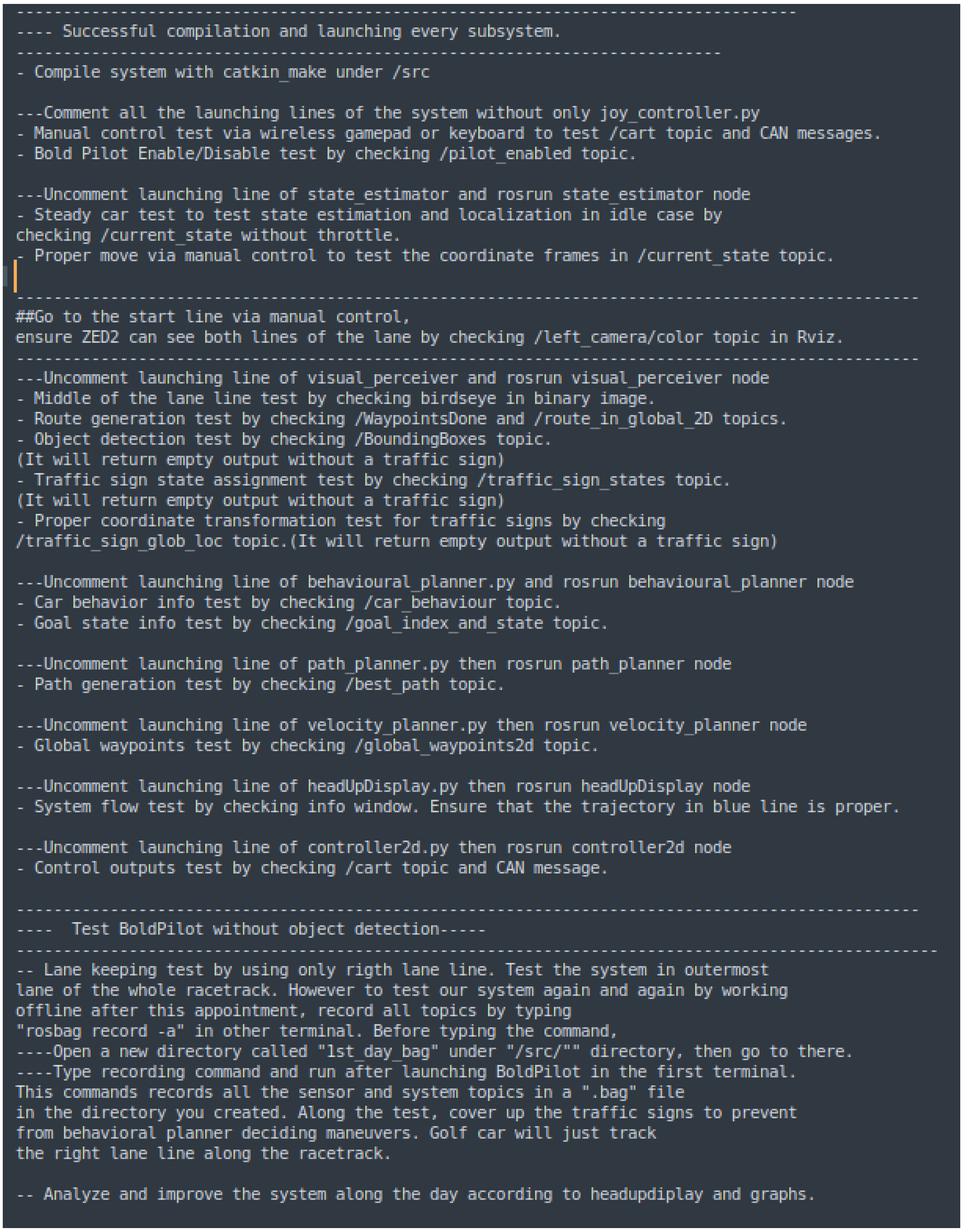


Figure 4: Applications and their dependencies in a docker container

The technical specification was taken as a basis while designing the software. A software design was made to meet the requirements here. Many constraints had to be considered while designing. The most basic problem is that the software behaves like a taxi in the city. However, the team had to work regularly because the software should work in a simulation environment before working in a real-world vehicle, it was a complex system and it would be a long project process. At this point, we applied design considerations in our software design. The software had to be extensible because we couldn't write the whole system at once. It was planned to expand the project and turn it into a large system. Modularity itself was applied from the smallest part to the largest part of the system. The modularity of the codes increased both extensibility and portability. The modules created, for example ROS packages, were shared easily and worked in other environments without the need for additional configuration. Since the project is a real-time system, its performance had to be high. For this reason, care was taken to keep the time complexity of the code low. In addition, the code has been written maintainable in order to easily support the problems and bugs that occur during the development process. Complying with all these design considerations allowed us to control the process comfortably. Through parametric data, debug made easy. The fact that the code was moved to different environments and that it was understandable helped the team members.

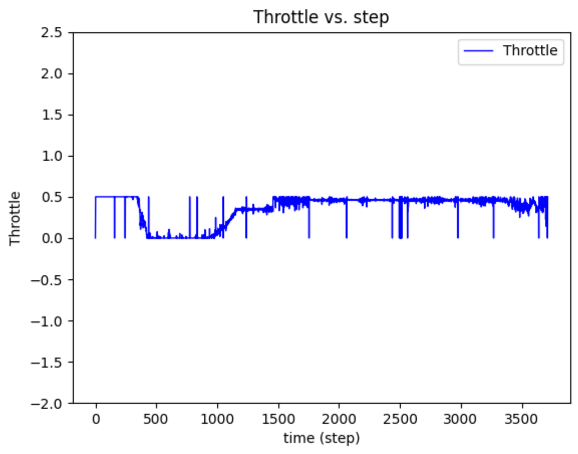
We used this portability and interoperability characteristics of the Bold Pilot to build whole system while we were handling rebooting problems of Xavier and bad internet connections in racetrack. We prebuilt the containerized system in our home so compilation process in test appointment day just took us 2 hours while debugging some known issues. Eventually, we successfully compiled a software system that is prepared in ROS Noetic, in Ubuntu 18.04.
We had prepared a test guideline that tracks the design given in Figure 1. It helps us to avoid waste of time. The guideline is given below.

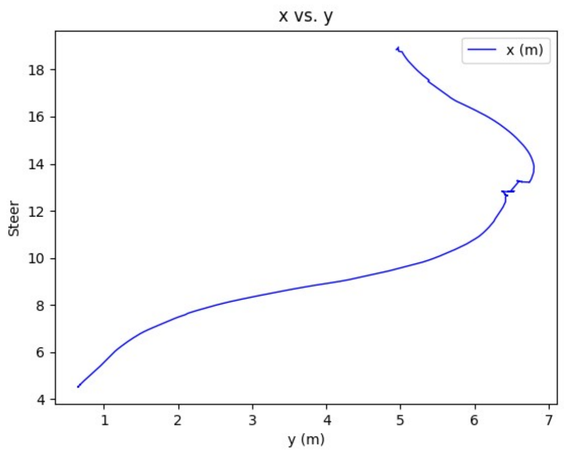


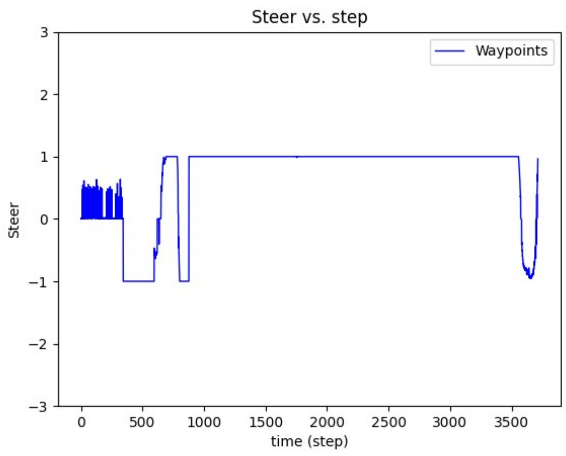
```
-----------------------------------------------------------------------------------
---- Successful compilation and launching every subsystem.
---------------------------------------------------------------------------
- Compile system with catkin_make under /src

---Comment all the launching lines of the system without only joy_controller.py
- Manual control test via wireless gamepad or keyboard to test /cart topic and CAN messages.
- Bold Pilot Enable/Disable test by checking /pilot_enabled topic.

---Uncomment launching line of state_estimator and rosrun state_estimator node
- Steady car test to test state estimation and localization in idle case by
checking /current_state without throttle.
- Proper move via manual control to test the coordinate frames in /current_state topic.

------------------------------------------------------------------------------------------------
##Go to the start line via manual control,
ensure ZED2 can see both lines of the lane by checking /left_camera/color topic in Rviz.
------------------------------------------------------------------------------------------------
---Uncomment launching line of visual_perceiver and rosrun visual_perceiver node
- Middle of the lane line test by checking birdseye in binary image.
- Route generation test by checking /WaypointsDone and /route_in_global_2D topics.
- Object detection test by checking /BoundingBoxes topic.
(It will return empty output without a traffic sign)
- Traffic sign state assignment test by checking /traffic_sign_states topic.
(It will return empty output without a traffic sign)
- Proper coordinate transformation test for traffic signs by checking
/traffic_sign_glob_loc topic.(It will return empty output without a traffic sign)

---Uncomment launching line of behavioural_planner.py and rosrun behavioural_planner node
- Car behavior info test by checking /car_behaviour topic.
- Goal state info test by checking /goal_index_and_state topic.

---Uncomment launching line of path_planner.py then rosrun path_planner node
- Path generation test by checking /best_path topic.

---Uncomment launching line of velocity_planner.py then rosrun velocity_planner node
- Global waypoints test by checking /global_waypoints2d topic.

---Uncomment launching line of headUpDisplay.py then rosrun headUpDisplay node
- System flow test by checking info window. Ensure that the trajectory in blue line is proper.

---Uncomment launching line of controller2d.py then rosrun controller2d node
- Control outputs test by checking /cart topic and CAN message.

------------------------------------------------------------------------------------------------
----  Test BoldPilot without object detection-----
--------------------------------------------------------------------------------------------------
-- Lane keeping test by using only rigth lane line. Test the system in outermost
lane of the whole racetrack. However to test our system again and again by working
offline after this appointment, record all topics by typing
"rosbag record -a" in other terminal. Before typing the command,
----Open a new directory called "1st_day_bag" under "/src/"" directory, then go to there.
----Type recording command and run after launching BoldPilot in the first terminal.
This commands records all the sensor and system topics in a ".bag" file
in the directory you created. Along the test, cover up the traffic signs to prevent
from behavioral planner deciding maneuvers. Golf car will just track
the right lane line along the racetrack.

-- Analyze and improve the system along the day according to headupdiplay and graphs.
```

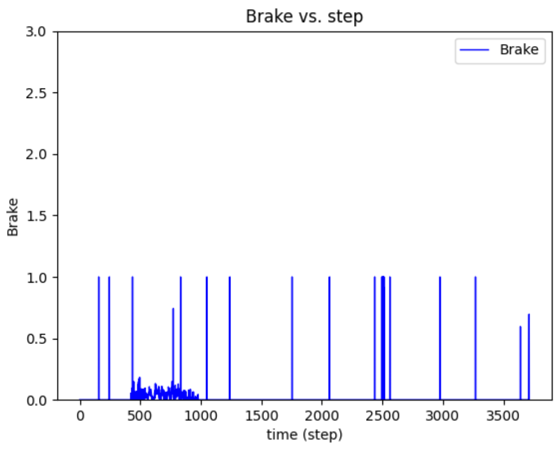
10. Test and Validation

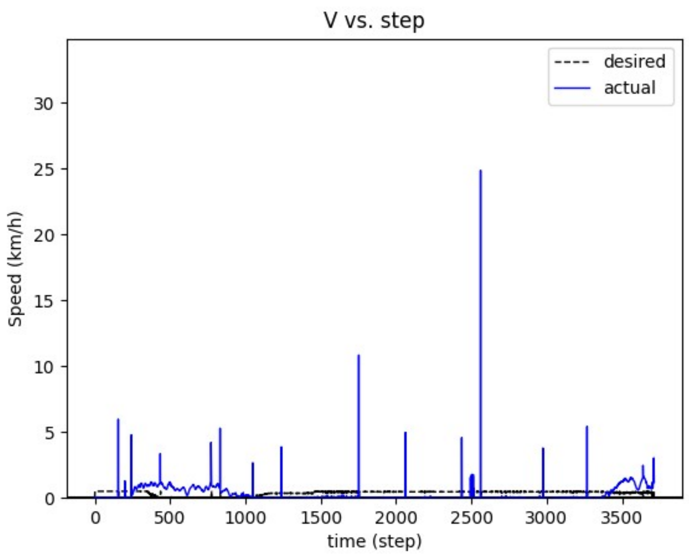
The throttle, east and north, steer and brake vs time graphics related to Decelerate to Bus Stop state, are given below.











11. References

[2] 262588213843476. (n.d.). A python class to convert GPS coordinates to a local enu coordinate system and vice versa. Gist. Retrieved April 1, 2022, from https://gist.github.com/MikeK4y/1d99b93f806e7d535021b15afd5bb04f
[3] https://www.stereolabs.com/docs/ros/zed-node/
[4] Finding Lane Lines—Simple Pipeline For Lane Detection. (2022). Retrieved 30 March 2022, from https://towardsdatascience.com/finding-lane-lines-simple-pipeline-for-lane-detection-d02b62e7572b
[5]Edge Detection - (2022). Retrieved 30 March 2022, from https://www.cs.auckland.ac.nz/compsci373s1c/PatricesLectures/Edge%20detection-Sobel_2up.pdf
[6] OpenCV: Arithmetic Operations on Images. (2022). Retrieved 30 March 2022, from https://docs.opencv.org/3.4/d0/d86/tutorial_py_image_arithmetics.html
[7] self-driving-car/project_4_advanced_lane_finding at master · ndrplz/self-driving-car. (2022). Retrieved 30 March 2022, from https://github.com/ndrplz/self-driving-car/tree/master/project_4_advanced_lane_finding
[8] G. Ballew, "galenballew/SDC-Lane-and-Vehicle-Detection-Tracking," *GitHub*. [Online]. Available:https://github.com/galenballew/SDC-Lane-and-Vehicle-Detection-Tracking/tree/master/ Part%20II%20-%20Adv%20Lane%20Detection%20and%20Road%20Features. [Accessed: 20-Mar-2021].
[9] S. Waslender, "Online Courses & Credentials From Top Educators. Join for Free," *Coursera*. [Online]. Available:https://www.coursera.org/learn/visual-perception-self-driving-cars/home/week/1. [Accessed: 05-Apr-2021].
[10] AlexeyAB. (n.d.). *AlexeyAB/darknet: YOLOv4 / SCALED-YOLOV4 / YOLO - neural networks for object Detection (Windows and Linux version of Darknet).* GitHub. https://github.com/AlexeyAB/darknet.
[11] "YOLOv4: Optimal Speed and Accuracy of Object Detection." [Online]. Available: https://arxiv.org/pdf/2004.10934.pdf. [Accessed: 08-May-2021].

[40]t1mkhuan9.(n.d.).t1mkhuan9*/yolov4-ros-noetic:*yolov4-ros-noetic.GitHub. https://github.com/t1mkhuan9/yolov4-ros-noetic.

[12] Ar-Ray-code, "Ar-Ray-code/darknet_ros_fp16," *GitHub*. [Online].Available: https://github.com/Ar-Ray-code/darknet_ros_fp16 [Accessed: 28-March-2022].

[13]ros2, "ros2/ros1_bridge," *GitHub*. [Online]. Available: https://github.com/ros2/ros1_bridge [Accessed: 28-March-2022].
[14] S. Waslender, "Motion Planning for Self-Driving Cars," *Coursera*. [Online]. Available: https://www.coursera.org/learn/motion-planning-self-driving-cars. [Accessed: 10-Jul-2020].
[15] R. Rajamani, "Chapter 2," in *Vehicle dynamics and control*, New York, US: Springer, 2012, pp. 15–26.
[16] G. M. Hoffmann, C. J. Tomlin, M. Montemerlo, and S. Thrun, "Autonomous Automobile Trajectory Tracking for Off-Road Driving: Controller Design, Experimental Validation and Racing," *2007 American Control Conference*, 2007.