Chapter 3

Software Modules

3.1 Project Architecture

Most of the car accidents are caused by human error, The main cause to these errors is that, the driver may fall into a fainting state for any medical reason. Also, the driver may fall asleep due to lack of sleep. Accordingly, these accidents cause traffic congestion for a long time and sometimes people pay their lives for it. The proposed solution is based on building a hybrid system composed of Self-Driving technologies for vehicles and Remotely Driving through the Control Centers of the Department of Traffic Emergency Situations. Specifically, in the event of an emergency, the proposed system does one of two scenarios; the ***first scenario*** is that, the self-driving system takes over driving the vehicle based on the decisions taken by the main control unit. The control unit relies on AI and the IoT in analyzing the driver's vital indicators data such as pulse, pressure, sugar, etc. to know the driver's health condition on which his ability to drive depends, as well as analyzing his facial features to determine if the driver lost consciousness or fell asleep.

The ***second scenario*** is that the remote driving system takes over driving the vehicle through the command-and-control centers of the Department of Traffic Emergency Situations. This as a result of the presence of circumstances that prevented the self-driving system from carrying out its mission, such as the lack of the necessary infrastructure for the self-driving system, which is common on most roads and hubs in Egypt. In this case, the car can be controlled remotely by sending a live broadcast from the car to the web page of the remote central control. Through this direct broadcast, the traffic officer responsible for this emergency situation can drive the car until it reaches the nearest safe point and avoid any accidents which can be occurred due to this emergency situation.

Experiments conducted using the initial model of the project showed that the proposed solution is highly efficient and effective in solving the problem of road accidents caused by human error. Figure 3.1 shows the complete architecture of the proposed solution. The system is an experimental prototype, and all hardware used are experimental and cannot be used in real applications. However, the concept and functionality of the hardware are almost the same as the hardware used in the real environment. The IDRS start up by initializing an interior camera in the car that monitors the driver's facial expressions in real-time and a smartwatch which serves the purpose of gathering additional vital measurements to the automated driving system for emergency situations. These measurements provide a more accurate picture of the driver's condition and guide the IDRS to take the appropriate actions.

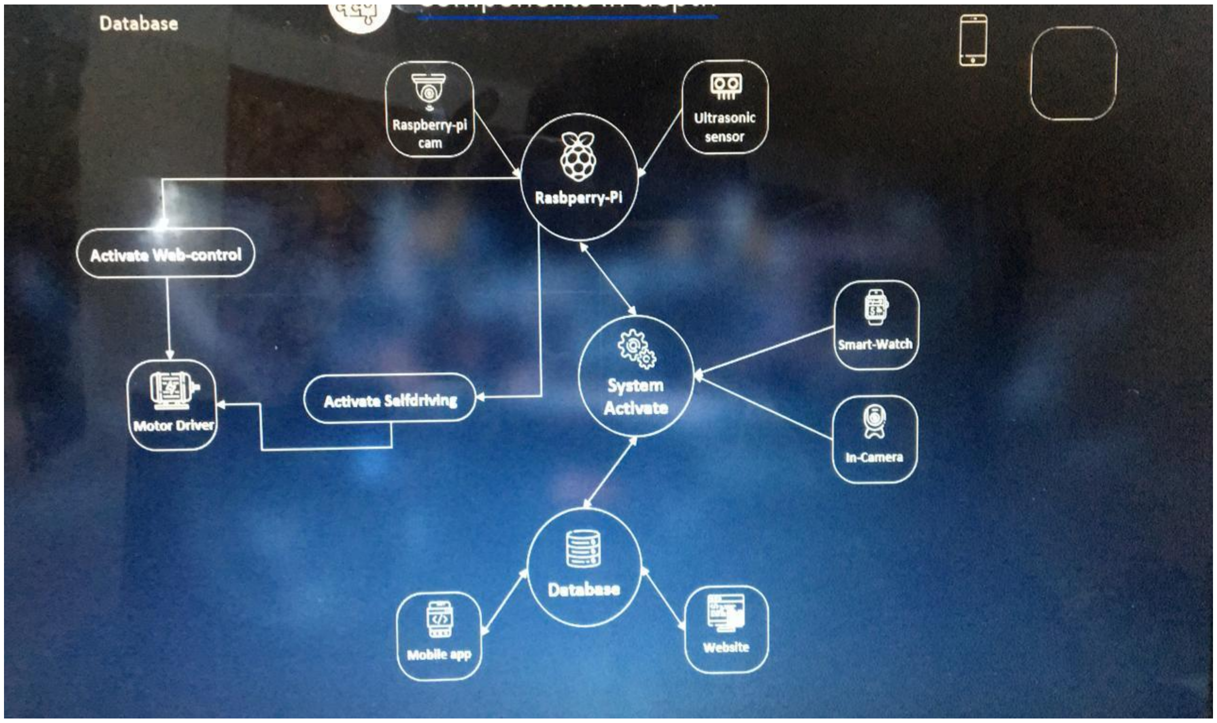


Fig. 3.1: IDRS Architecture

Overall, the system works by integrating the information gathered from the interior camera and the smartwatch to assess the driver's condition and take appropriate measures to ensure their safety and transport them to the nearest medical point as quickly as possible. Once triggered, the system automatically takes control of the vehicle using programmed mechanisms such as line and object detection algorithms. This is achieved by utilizing an interior camera to ensure the car remains in a safe state while moving towards the nearest safe point. Subsequently, the system sends the driver's vital indicators and the car's location to an integrated web system, which, in turn, dispatches the car's location to the nearest emergency point for assistance to reach the driver promptly. In the event of a malfunction in the vehicle's automated system, there is an alternative integrated control system. This system allows a person to remotely control the car using programmed buttons connected to the car's steering wheel through signals on a web page within the web system. Additionally, there is a process to add the driver's information and car number to the database of the overall system.

3.2 System Modules

The implementation of the software of the architecture of fig. 3.1 are coded in three main software modules as shown in figure 3.2.

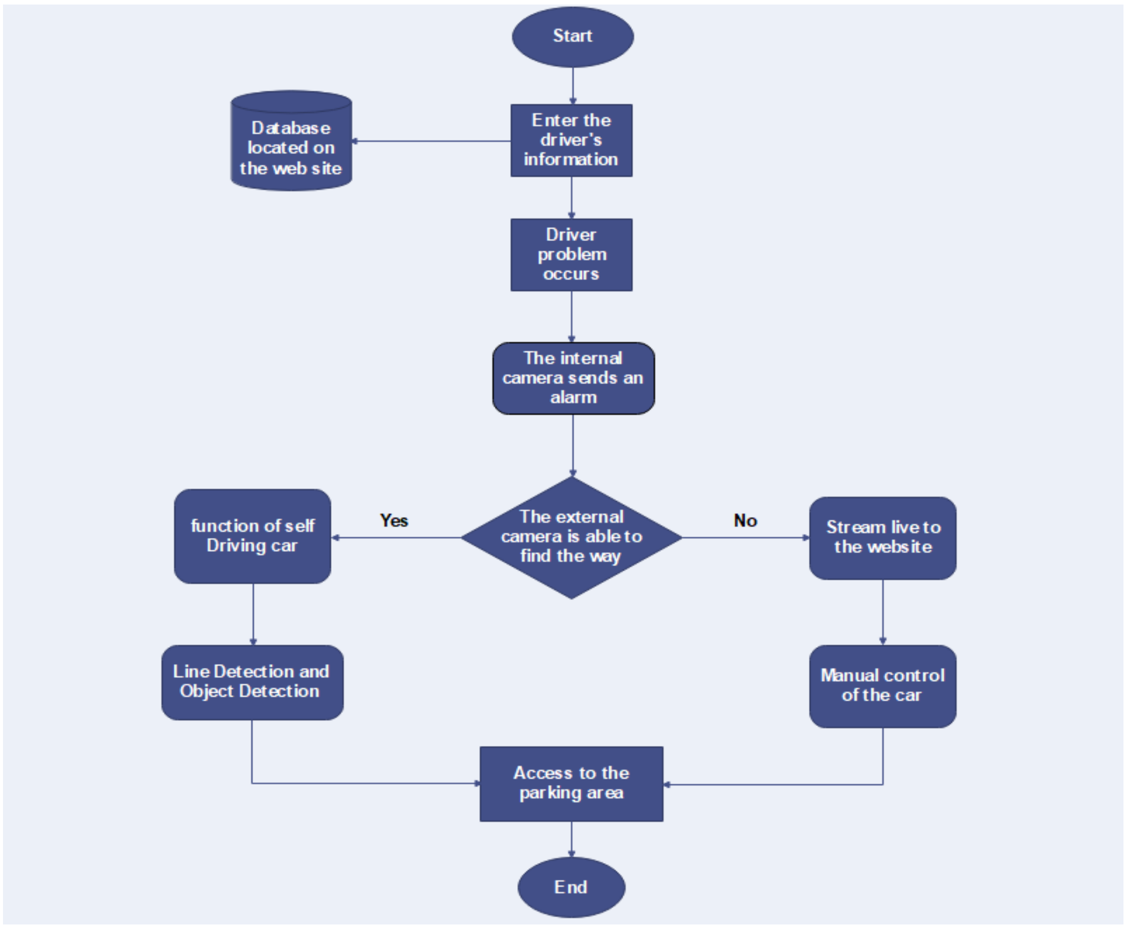

Start
Enter the driver's information
Database located on the web site
Driver problem occurs
The internal camera sends an alarm
The external camera is able to find the way
Yes
No
function of self Driving car
Stream live to the website
Line Detection and Object Detection
Manual control of the car
Access to the parking area
End

Fig.3.2: System Central Diagram

The rescue action begins when the interior camera detects a problem with the driver and sends an alert to the web application. Then, a comparison is made between the two decisions: self-driving or remote driving. Either the self-driving is activated or remote control is initiated, the car will be driven to reach the nearest safe point. Emergency services are contacted, and the car's location is determined using GPS.

According to this scenario there are four main modules constructing the system as follows;

1. **Drivers Database Setup Module:** We are working to provide data for all drivers and information about the vehicle
2. **Vision and Computing Module:** This module will be responsible for capturing the image using image processing algorithms, and extracting the image features to perform both modules of face detection algorithm and road detection algorithm.
3. **Hardware Module:** This module will be responsible for managing the control motors using the motor driver.
4. **User Control Module:** This module will be responsible for performing the complete control of the system.

These modules work together to perform the whole project scenarios.

3.2.1 Drivers Database Setup Module

At the beginning of the project, the client enters their data by downloading the mobile application on their own mobile device. They input their personal information, and then the new data is saved in the associated database between the mobile application and the website as shown in figure 3.3.

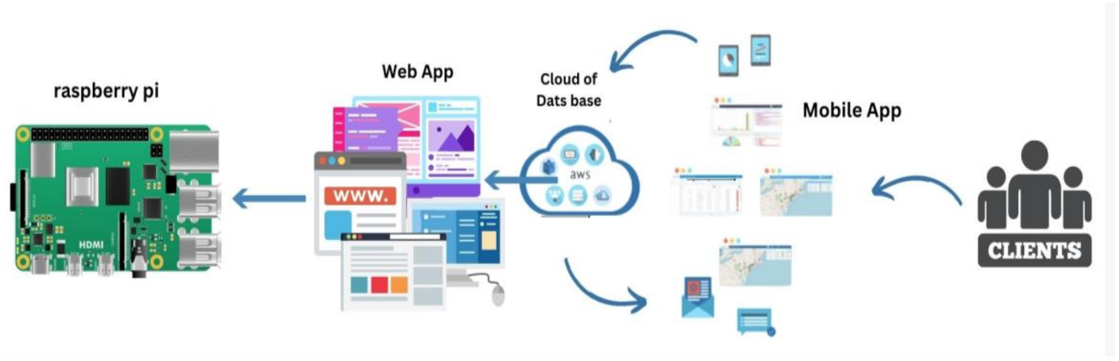


Fig.3.3: Data Collection System

In this module, the data is collected on a local network and it has its own protection measures to safeguard the client's information and ensure privacy. The local network also works to maintain a backup copy of the information.

3.2.2 Vision and Computing Module

Drowsy driving is one of the common causes of road accidents resulting in injuries, even death, and significant economic losses to drivers, road users, families, and society. There have been many studies carried out in an attempt to detect drowsiness for alert systems**.** The US National Highway Traffic Safety Administration (NHTSA) reports that drowsy driving is the cause of an estimated 40,000 injuries and 1,550 car accident deaths annually. So, there are a great need to deal with this issue.

This Vision and Computing will be responsible for capturing the image using image processing algorithms, and extracting the image features to perform both modules of face detection algorithm and road detection algorithm. The face and eye detection algorithms are used to determine the drowsy state of the driver as follows.

A) Face Detection

Face detection is a computer vision technique used to identify and locate human faces in images or videos. It involves analyzing the visual patterns and features of an image to determine the presence and position of faces. Many information can be written on the face that is being read, as shown in figure 3.4. Its algorithms typically utilize machine learning and artificial intelligence algorithms to detect facial features such as eyes, nose, and mouth.

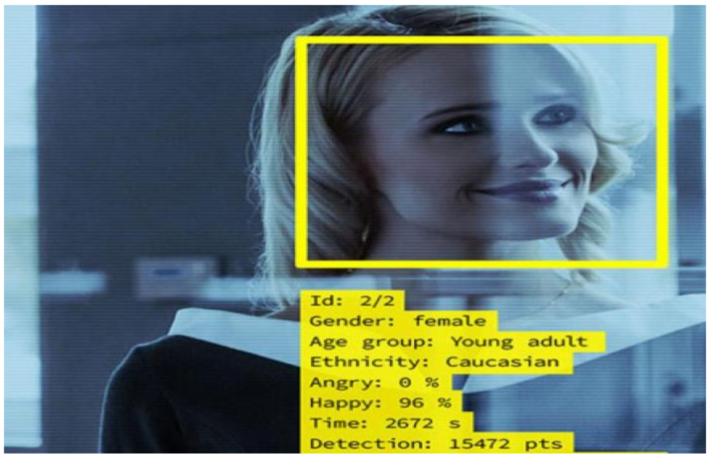


Fig.3.4: Face Detection

The process of face detection involves the following steps:

1. **Preprocessing:** The input image is prepared by converting it into a suitable format and size for analysis.
2. **Feature Extraction:** The algorithm analyzes the image to identify potential areas that could contain faces. It examines the patterns and characteristics of the image to identify regions that are likely to be faces.
3. **Classification:** The identified regions are further analyzed to determine if they indeed contain faces or not. This step involves using a trained classifier that distinguishes between face and non-face regions.
4. **Localization:** Once a face is detected, the algorithm calculates the location and size of the face within the image. This information is usually represented as a bounding box that surrounds the detected face.
5. **Post-processing:** Additional steps may be applied to refine the results, remove false positives, or improve the accuracy of the face detection.

B) Eyes Detection

Eyes detection is a specific subset of face detection that focuses on detecting and locating the eyes within a face region as shown in the Figure 3.5. It involves using computer vision techniques to identify and extract the eyes' features in images or videos.

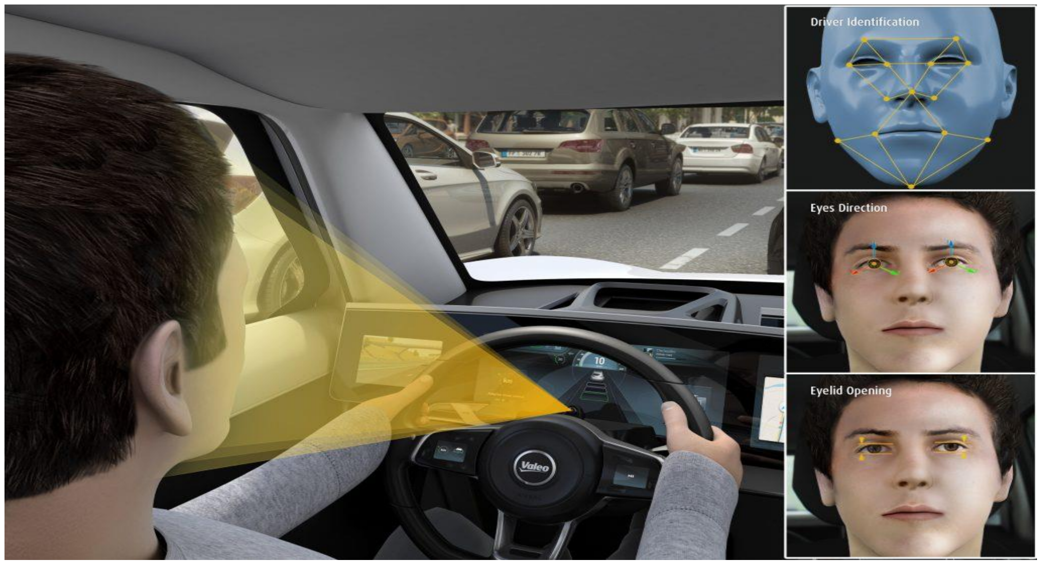


Fig.3.5: Eyes Detection

The process of eyes detection generally follows these steps:

1. **Face Detection:** Initially, a face detection algorithm is applied to locate and extract the face regions within an image or video frame. This step helps narrow down the search area for eyes detection.
2. **Region of Interest (ROI) Extraction:** Once the face regions are detected, a region of interest is defined around each face to focus specifically on the eye regions.
3. **Preprocessing:** The ROI is preprocessed to enhance the eye features and reduce noise or artifacts. Common preprocessing techniques include image resizing, grayscale conversion, and normalization.
4. **Eye Feature Extraction:** Various computer vision techniques can be employed to extract eye features, such as shape, texture, or color information. These features can help differentiate eyes from other facial components and background elements.
5. **Classification or Detection:** A classification or detection algorithm is applied to the extracted eye features to determine whether they represent eyes or not. This step involves utilizing machine learning algorithms or trained models to classify the extracted features as eyes or non-eyes.
6. **Localization:** If eyes are detected, their locations within the face region are determined. This information is typically represented as bounding boxes or key-points around the eyes.

Eyes detection plays a crucial role in various applications, including gaze tracking, driver drowsiness detection, emotion analysis, and facial expression recognition. By accurately detecting and tracking the eyes, these applications can extract valuable information about the person's attention, emotions, or cognitive states.

The face and eyes detection technology are used to analyze the face and eyes using artificial intelligence and deep learning algorithms. The process involves the following steps:

1- The face in the image is detected using advanced algorithms that search for distinctive facial features such as the eyes, nose, mouth, and cheeks.

2- The location of the eyes in the face is determined using facial feature detection technology, which analyzes the entire face to determine the position of the eyes.

3- Eye detection technology is used to determine whether the eyes are open or closed, using various techniques to analyze the movement, size, and frame angle of the eyes.
4- This information is analyzed by computer algorithms to determine whether the person is awake or asleep. For example, when a person is asleep, the size of the eyes is smaller and they are more closed, whereas when a person is awake, the size of the eyes is larger and they are more open.

In this way, face and eye detection technology can be used to determine whether a person is awake or asleep. Initially, we placed a camera inside the car directly facing the driver to recognize the driver's face and specifically perform eye detection, as shown in Figure 3.6. If the driver closes their eyes for a duration of ten seconds, the camera issues an alert to wake up the driver. If the driver fails to wake up, it immediately sends a warning to the traffic signal center, which in turn sends a notification to our website. The self-driving software of the car then takes over directly.

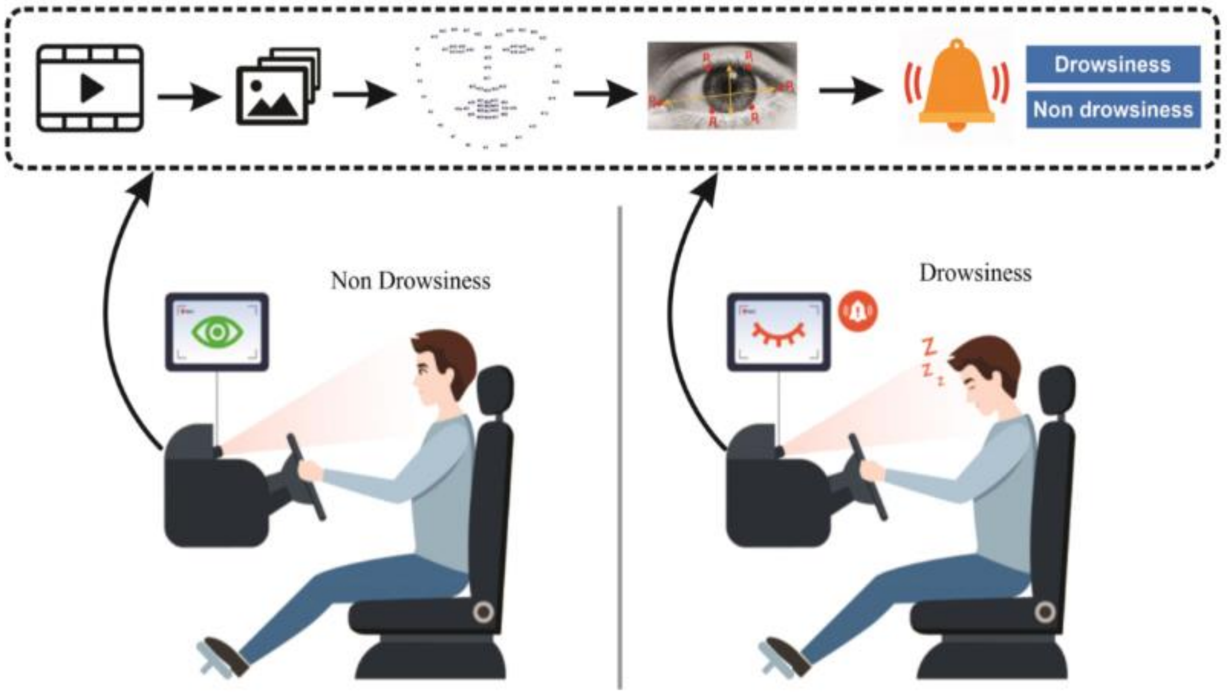


Fig. 3.6: The drivers status

The face and eyes detection technology are used to analyze the face and eyes using artificial intelligence and deep learning algorithms. As an example, figure 3.7 represents awake person (one of our project team) his eyes were opened, and his name and status were displayed on the screen.

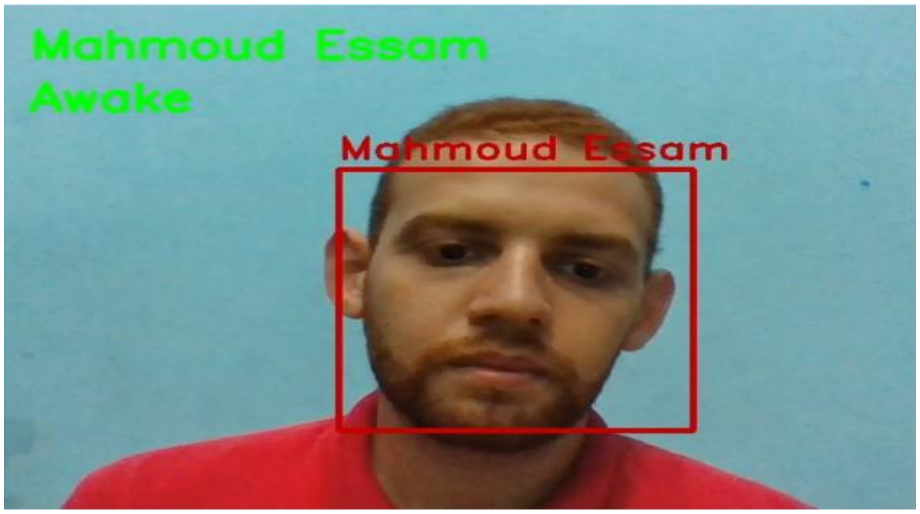


Fig.3.7: The Person is Awake

It seems that there are no issues with the person's appearance, and there are no signs indicating his poor condition. Additionally, the indicators suggest that everything is alright. Based on figure 3.8, When the person starts to close his eyes, a timer appears on the screen along with his name and status, which indicates that he is still awake.

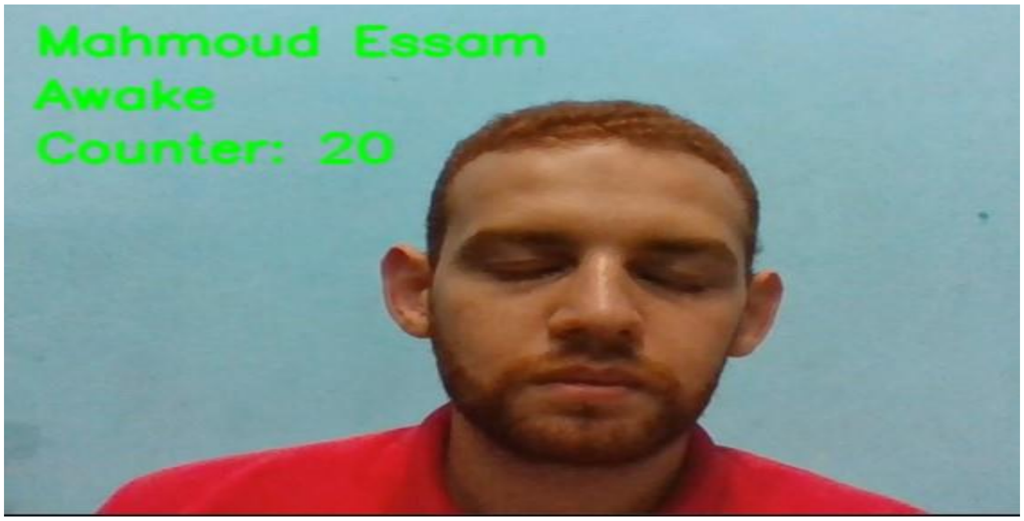


Fig. 3.8: The person started to close his eyes

This is normal because the person may blink or close his eyes for a short period of time for reasons other than sleeping. However, if the timer continues to count and reaches 20 seconds, the person's status changes from awake to sleeping. Based on the information provided earlier, which shows the person sleeping. At this point, a new phase begins for entering the control part.

Code samples:

Let's now explain a simple explanation of how the code works for face and eyes detection:

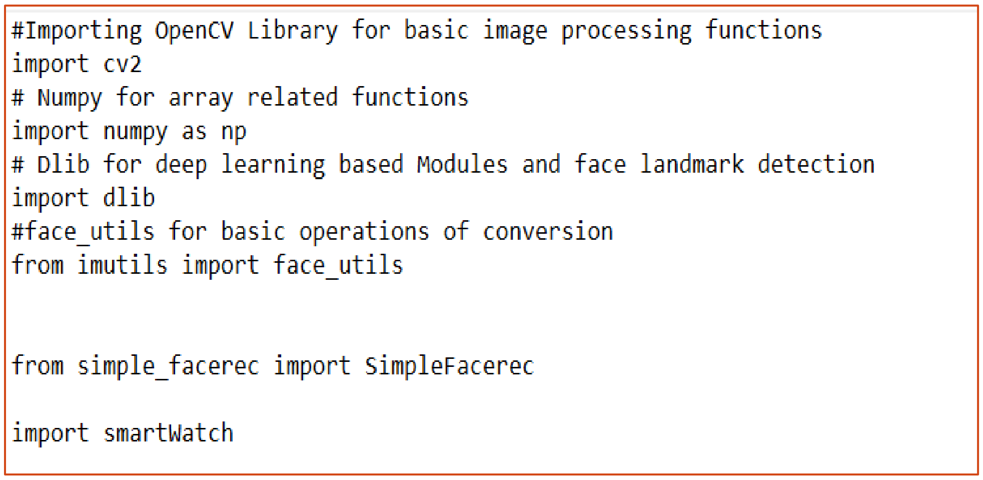
1- The face in the image is detected using advanced algorithms that search for distinctive facial features such as the eyes, nose, mouth, and cheeks.

2- The location of the eyes in the face is determined using facial feature detection technology, which analyzes the entire face to determine the position of the eyes.

3- Eye detection technology is used to determine whether the eyes are open or closed, using various techniques to analyze the movement, size, and frame angle of the eyes.

4- This information is analyzed by computer algorithms to determine whether the person is awake or asleep. For example, when a person is asleep, the size of the eyes is smaller and they are more closed, whereas when a person is awake, the size of the eyes is larger and they are more open.

In this way, face and eye detection technology can be used to determine whether a person is awake or asleep. Fig. 3.9 illustrates what we explain in the previous points.



```
#Importing OpenCV Library for basic image processing functions
import cv2
# Numpy for array related functions
import numpy as np
# Dlib for deep learning based Modules and face landmark detection
import dlib
#face_utils for basic operations of conversion
from imutils import face_utils


from simple_facerec import SimpleFacerec

import smartWatch
```

Fig.3.9: Used Libraries

In figure 3.9 the following libraries are appeared;

Open CV; it was used for Face recognition and photo and video processing

Numpy; it was used for working with arrays perform mathematical operations on arrays.

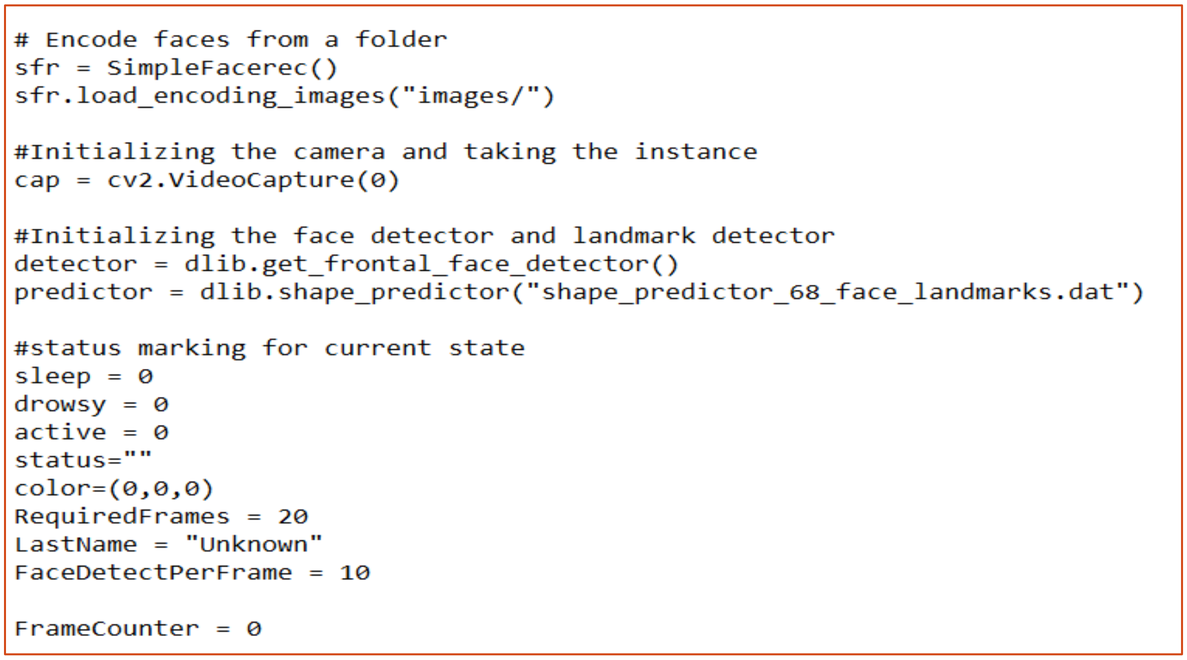
Dlib (Deep Learning Image Processing)**:** is a Python library for deploying a deep learning models in image processing tasks.

face_utils: is part of the dlib library provides a set of utility functions for working with facial landmarks detection and manipulation.

SimpleFacerec: functionality using the OpenCV and Dlib libraries. It allows you to perform face detection and recognition tasks on images and videos.

smartwatch: programming library for developers to create apps or customize functionality.

In Figure 3.10: The **SimpleFacerec()** object is initialized and trained on the images presented in the **"images/"** folder using the **load_encoding_images()** method.

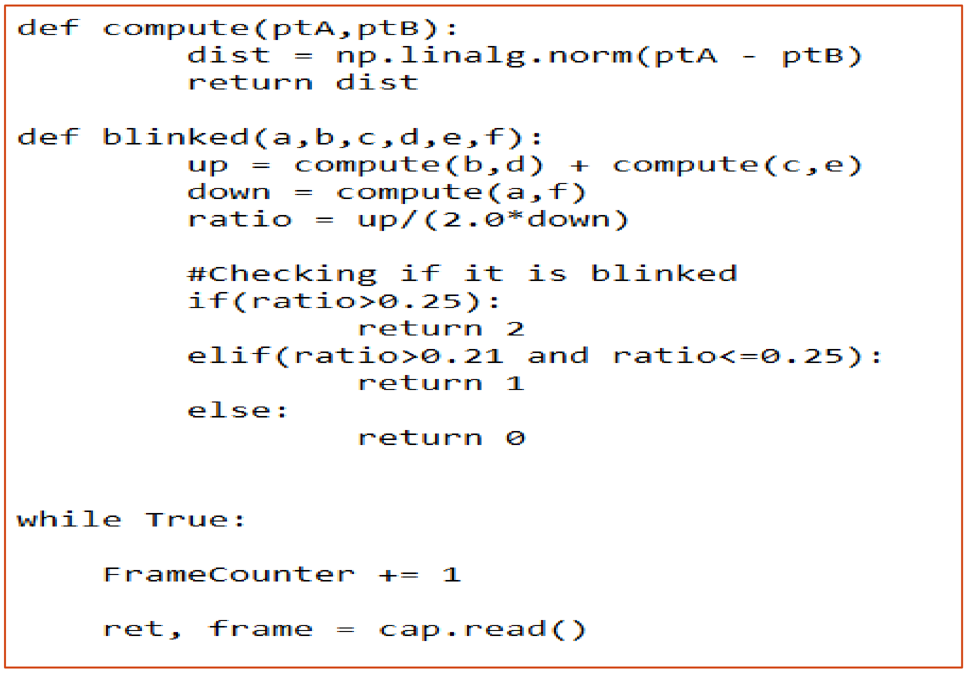


```
# Encode faces from a folder
sfr = SimpleFacerec()
sfr.load_encoding_images("images/")

#Initializing the camera and taking the instance
cap = cv2.VideoCapture(0)

#Initializing the face detector and landmark detector
detector = dlib.get_frontal_face_detector()
predictor = dlib.shape_predictor("shape_predictor_68_face_landmarks.dat")

#status marking for current state
sleep = 0
drowsy = 0
active = 0
status=""
color=(0,0,0)
RequiredFrames = 20
LastName = "Unknown"
FaceDetectPerFrame = 10

FrameCounter = 0
```

Fig.3.10: Camera initialization code.

After that, the camera object is initialized with **cv2.VideoCapture(0)** where the argument "0" specifies the default camera. The **dlib.get_frontal_face_detector()** method initializes the face detector, while **dlib.shape_predictor** ("shape_predictor_68_face_landmarks.dat") initializes the landmark detector which is required to locate specific points on the face. The **sleep**, **drowsy**, and **active** variables are initialized to track the drowsiness level of the person in front of the camera. The **status** variable is used to indicate whether the person is "Sleeping", "Drowsy", or "Active" and the **color** variable is used to indicate the color of the status text. The **RequiredFrames** variable is set to **20**, which specifies the number of consecutive frames in which the person should be drowsy or sleeping before being classified as "Sleeping". The **LastName** variable is initialized to "Unknown", which will be updated with the name of the recognized person. The **FaceDetectPerFrame** variable is set to **10**, which specifies the number of frames after which face detection will be performed again. Finally, the **FrameCounter** variable is initialized to **0**, which will be used to count the number of frames processed by the program.

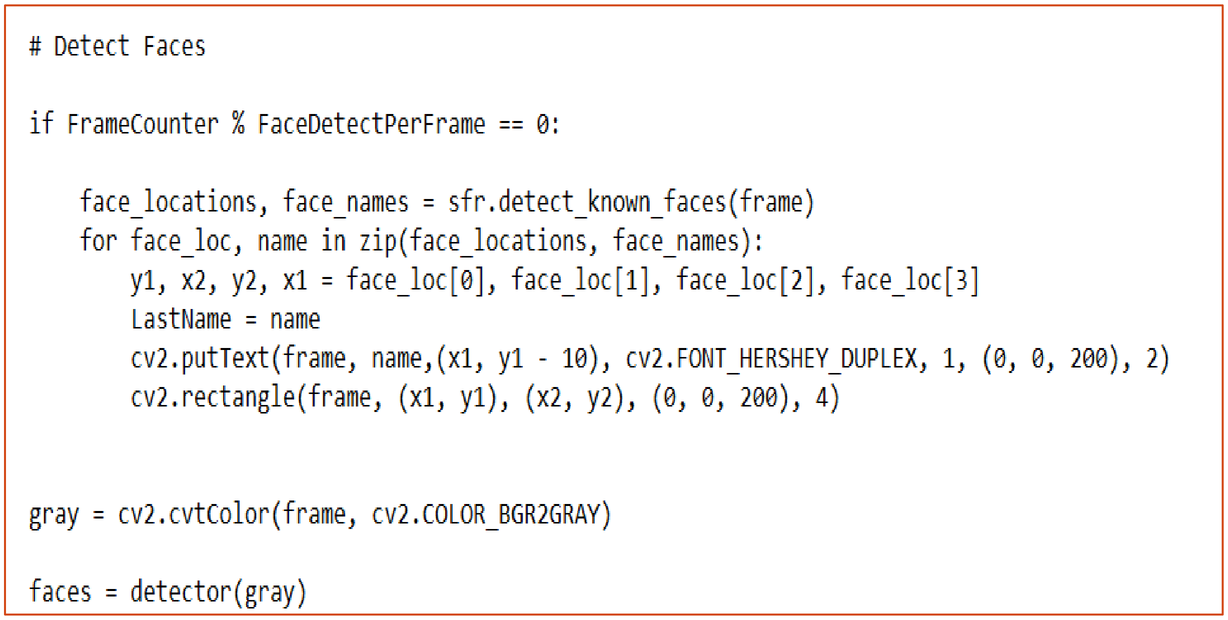
In Figure 3.11: **compute()** function takes two arguments, **ptA** and **ptB**, and uses the NumPy library to calculate the Euclidean distance between the two points. The function then returns the distance value.



```
def compute(ptA,ptB):
        dist = np.linalg.norm(ptA - ptB)
        return dist

def blinked(a,b,c,d,e,f):
        up = compute(b,d) + compute(c,e)
        down = compute(a,f)
        ratio = up/(2.0*down)

        #Checking if it is blinked
        if(ratio>0.25):
                return 2
        elif(ratio>0.21 and ratio<=0.25):
                return 1
        else:
                return 0


while True:

    FrameCounter += 1

    ret, frame = cap.read()
```

Fig. 3.11: computing function code

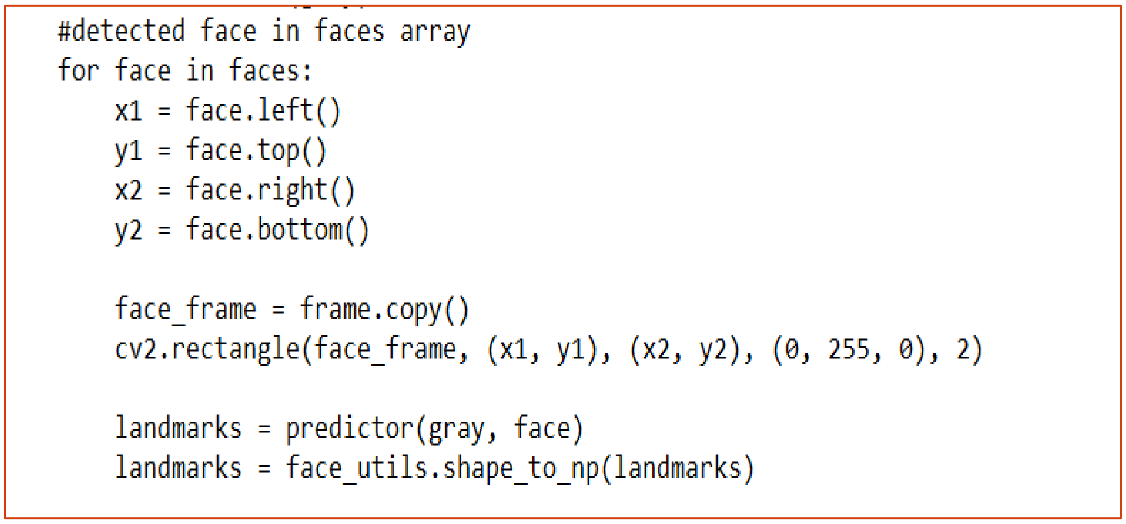
blinked() function takes six arguments, a, b, c, d, e, and f. These arguments represent the coordinates of six facial landmarks detected by a facial landmark detector algorithm. **compute**() function used to calculate the distances between these points and then calculates a ratio value based on these distances. Finally, the function returns a value of 0, 1, or 2 based on the value of the ratio. **while loop** runs continuously and captures video frames from a video capture device (presumably a webcam) using the OpenCV library. It then increments a FrameCounter variable by 1. **cap.read()** function reads the next video frame from the capture device and returns two values: a boolean value indicating whether the frame was successfully read, and the video frame itself as a NumPy array. In **Figure 3.12:** This code appears to detect faces in a video stream. The first section of the code block checks whether the current frame should be processed for face detection, based on the **FrameCounter** and **FaceDetectPerFrame** variables.



```
# Detect Faces

if FrameCounter % FaceDetectPerFrame == 0:

    face_locations, face_names = sfr.detect_known_faces(frame)
    for face_loc, name in zip(face_locations, face_names):
        y1, x2, y2, x1 = face_loc[0], face_loc[1], face_loc[2], face_loc[3]
        LastName = name
        cv2.putText(frame, name,(x1, y1 - 10), cv2.FONT_HERSHEY_DUPLEX, 1, (0, 0, 200), 2)
        cv2.rectangle(frame, (x1, y1), (x2, y2), (0, 0, 200), 4)


gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

faces = detector(gray)
```

Fig.3.12: Face detection code

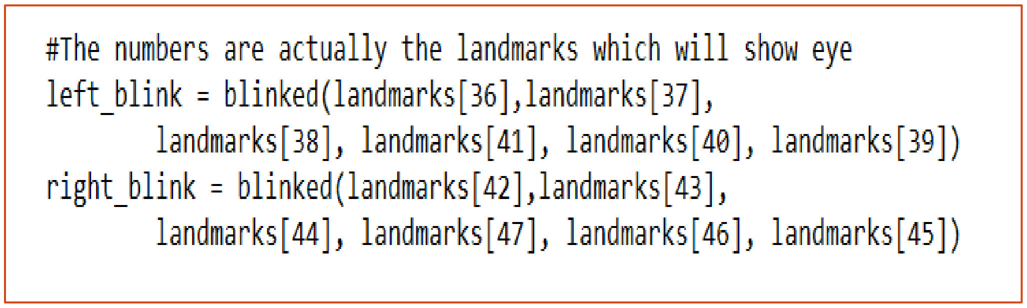
If the frame should be processed, it calls the **detect_known_faces** function of an object named **sfr**, passing in the frame as an argument. This function appears to detect known faces in the frame, returning their locations and corresponding names. The code then iterates over the detected face locations and names, drawing a rectangle and printing the name on top of each face. The last line of the code converts the frame to grayscale and passes it to a detector object named detector, which likely detects faces using some algorithm or model. In Figure 3.13: This code block processes each face detected in the video stream by extracting the corner points of the bounding box around each face and drawing a green rectangle around the face on a copy of the original frame.



```
#detected face in faces array
for face in faces:
    x1 = face.left()
    y1 = face.top()
    x2 = face.right()
    y2 = face.bottom()

    face_frame = frame.copy()
    cv2.rectangle(face_frame, (x1, y1), (x2, y2), (0, 255, 0), 2)

    landmarks = predictor(gray, face)
    landmarks = face_utils.shape_to_np(landmarks)
```

Fig.3.13: Face feature extraction

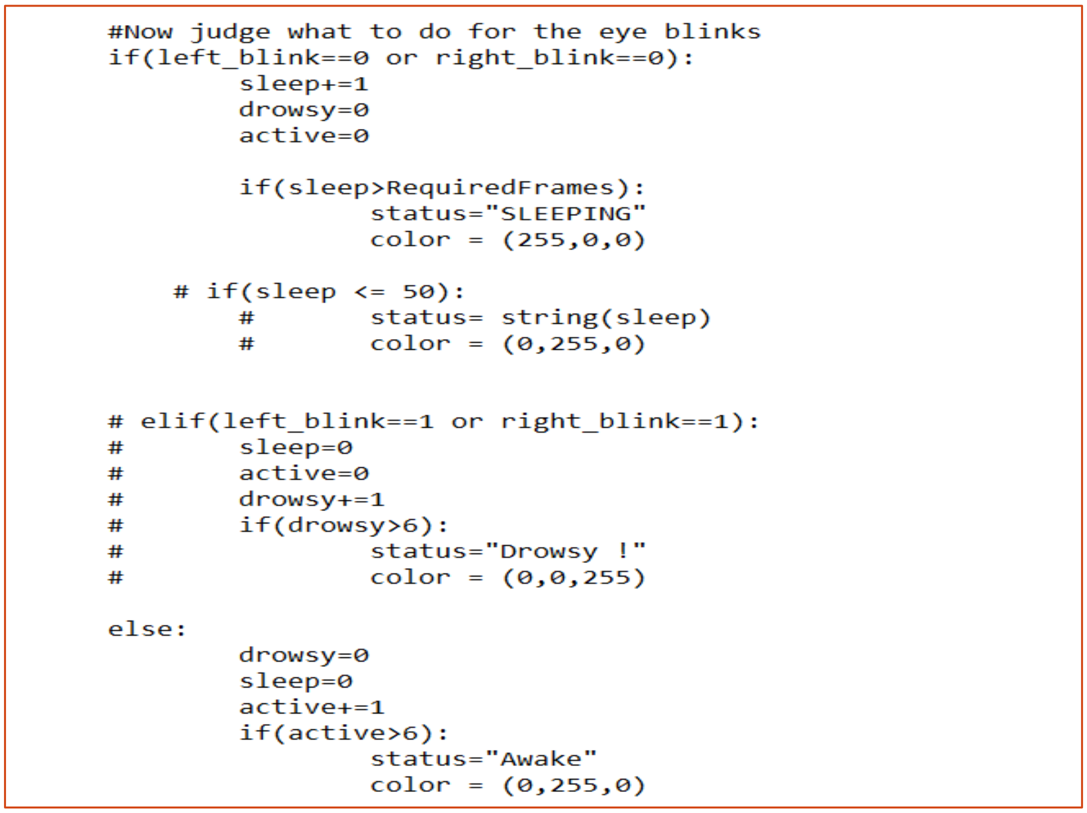
It also extracts the facial landmarks for each face using a facial landmark detection model. In Figure 3.14**:** This appears to be a code snippet in a programming language that involves processing facial landmarks to detect eye blinks.



```
#The numbers are actually the landmarks which will show eye
left_blink = blinked(landmarks[36],landmarks[37],
        landmarks[38], landmarks[41], landmarks[40], landmarks[39])
right_blink = blinked(landmarks[42],landmarks[43],
        landmarks[44], landmarks[47], landmarks[46], landmarks[45])
```

Fig. 3.14: Eye Blinks detection

The landmarks are represented as numbers and are used as input parameters in the **blinked**() function, which likely calculates the degree of eye closure based on the positions of the detected landmarks. The output of the function is assigned to the variables left_ and right_blink, which likely represent the degree of blink for the left and right eyes, respectively. Overall, this code may be part of a larger project related to computer vision or facial recognition.



```
#Now judge what to do for the eye blinks
if(left_blink==0 or right_blink==0):
        sleep+=1
        drowsy=0
        active=0

        if(sleep>RequiredFrames):
                status="SLEEPING"
                color = (255,0,0)

    # if(sleep <= 50):
        #       status= string(sleep)
        #       color = (0,255,0)


# elif(left_blink==1 or right_blink==1):
#       sleep=0
#       active=0
#       drowsy+=1
#       if(drowsy>6):
#               status="Drowsy !"
#               color = (0,0,255)

else:
        drowsy=0
        sleep=0
        active+=1
        if(active>6):
                status="Awake"
                color = (0,255,0)
```

Fig. 3.15: Eye Closure degree detection

In Figure 3.15: The code snippet checks the degree of eye closure based on facial landmarks and determines whether the user is awake, drowsy, or sleeping based on the degree of closure. It uses variables such as **left_blink**, **right_blink**, sleep, drowsy, and active to keep track of eye closure over time, and updates the status and color variables based on the duration and severity of eye closure.

C) Computer vision

Computer vision is a field of study that focuses on enabling computers to acquire, process, analyze, and understand visual information from images or videos. It involves developing algorithms and techniques to enable machines to interpret and make sense of visual data, similar to how humans perceive and understand the visual world. Computer vision finds applications in various domains, including object recognition, image classification, video surveillance, autonomous vehicles, medical imaging, and augmented reality, among others.

It plays a crucial role in tasks such as object detection, image segmentation, facial recognition, and scene understanding. Firstly, the system relied heavily on road boundary detection. The lane detection and object detection functions were responsible for determining the road boundaries using the Raspberry Pi's camera. These images were then sent to the Raspberry Pi for processing, where the road boundaries were drawn. If the camera captured something else, such as another car or a pedestrian, the object detection function would pause and wait for the person to pass or the car ahead to move, as illustrated in Figures 3.16, 3.17.

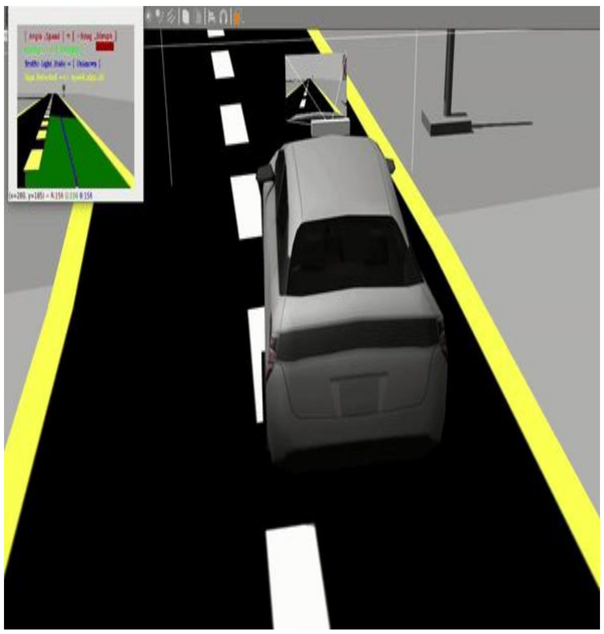


Fig.3.16: Line Detection in Gazebo

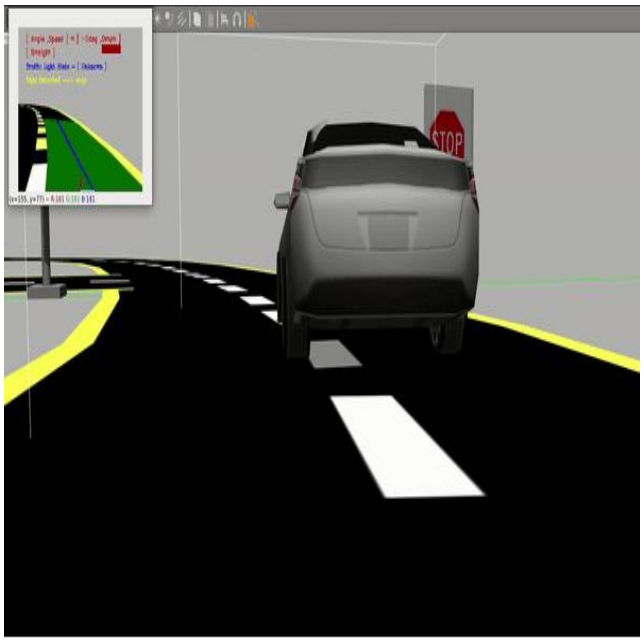


Fig.3.17: Object Detection in Gazebo

We conducted simulations using the Gazebo software to simulate the car in a real-world environment and ensure its efficiency. We tested the simulation before finalizing the function, ensuring that it achieved the desired efficiency and accuracy. In the line detection and object detection functions, the process of detection and image analysis is performed using a set of operations, which are illustrated in the **Figure 3.18**

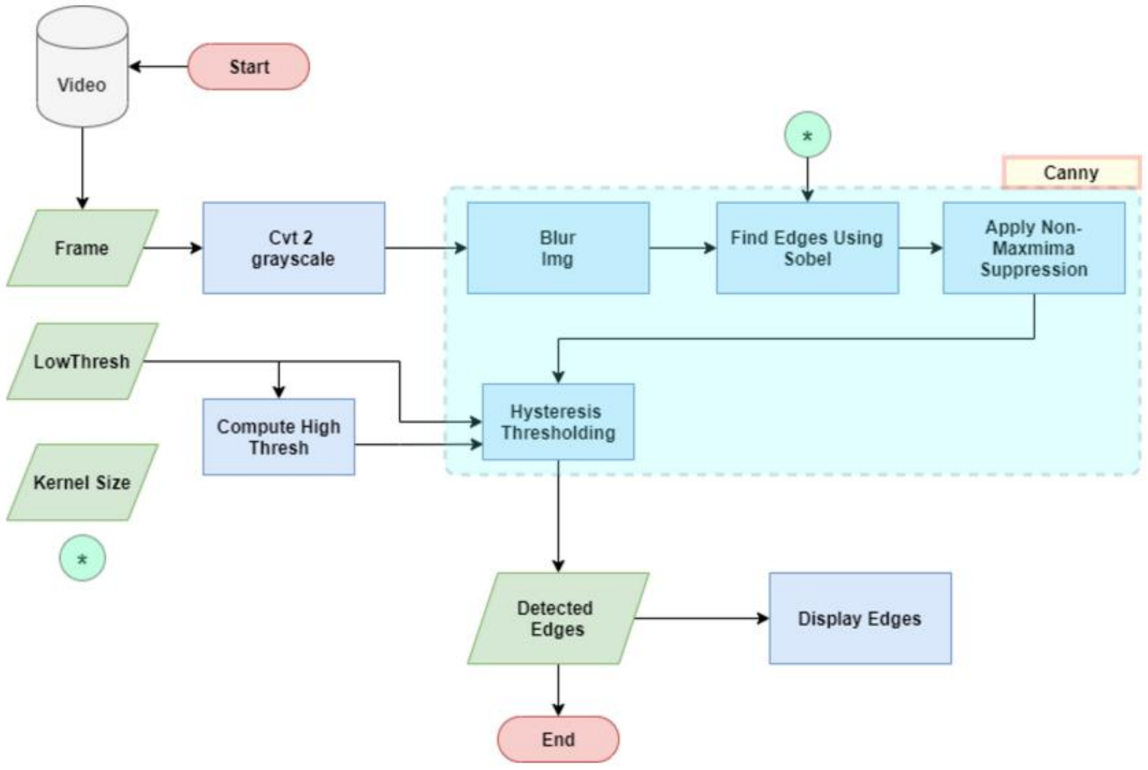


Fig.3.18: Image analysis steps.

Here are the steps involved in video analysis using computer vision (CV):

1- Video Input: Obtain the video footage that you want to analyze. This can be from a camera, recorded video file, or live video stream.

2- Frame Extraction: Extract individual frames from the video. Each frame represents a single image.

3- Preprocessing: Apply preprocessing techniques to enhance the quality of the frames. This may include resizing, cropping, color correction, denoising, or image stabilization.

4- Object Detection/Tracking: Use object detection algorithms to identify and locate specific objects or regions of interest within each frame. Object tracking algorithms can be employed to track the movement of objects across multiple frames.

5- Feature Extraction: Extract relevant features from the detected objects, such as shape, color, texture, or motion information. These features will be used for further analysis and recognition.

6- Action Recognition: Analyze the sequence of frames to recognize specific actions or activities taking place in the video. This can involve techniques such as gesture recognition, activity recognition, or behavior analysis.

7- Event Detection: Identify specific events or occurrences within the video, such as accidents, anomalies, or specific patterns of interest.
8- Data Analysis: Perform statistical analysis or machine learning algorithms on the extracted features and recognized patterns to gain insights, make predictions, or classify the video content.
9- Post-processing: Apply any necessary post-processing techniques, such as filtering, smoothing, or visualization, to improve the results or present them in a meaningful way.
10- Output: Present the analyzed results, which can include visual overlays, annotations, reports, or real-time alerts.

It's important to note that the specific steps and techniques used in video analysis can vary depending on the application and desired outcomes.

3.2.3 Control Module

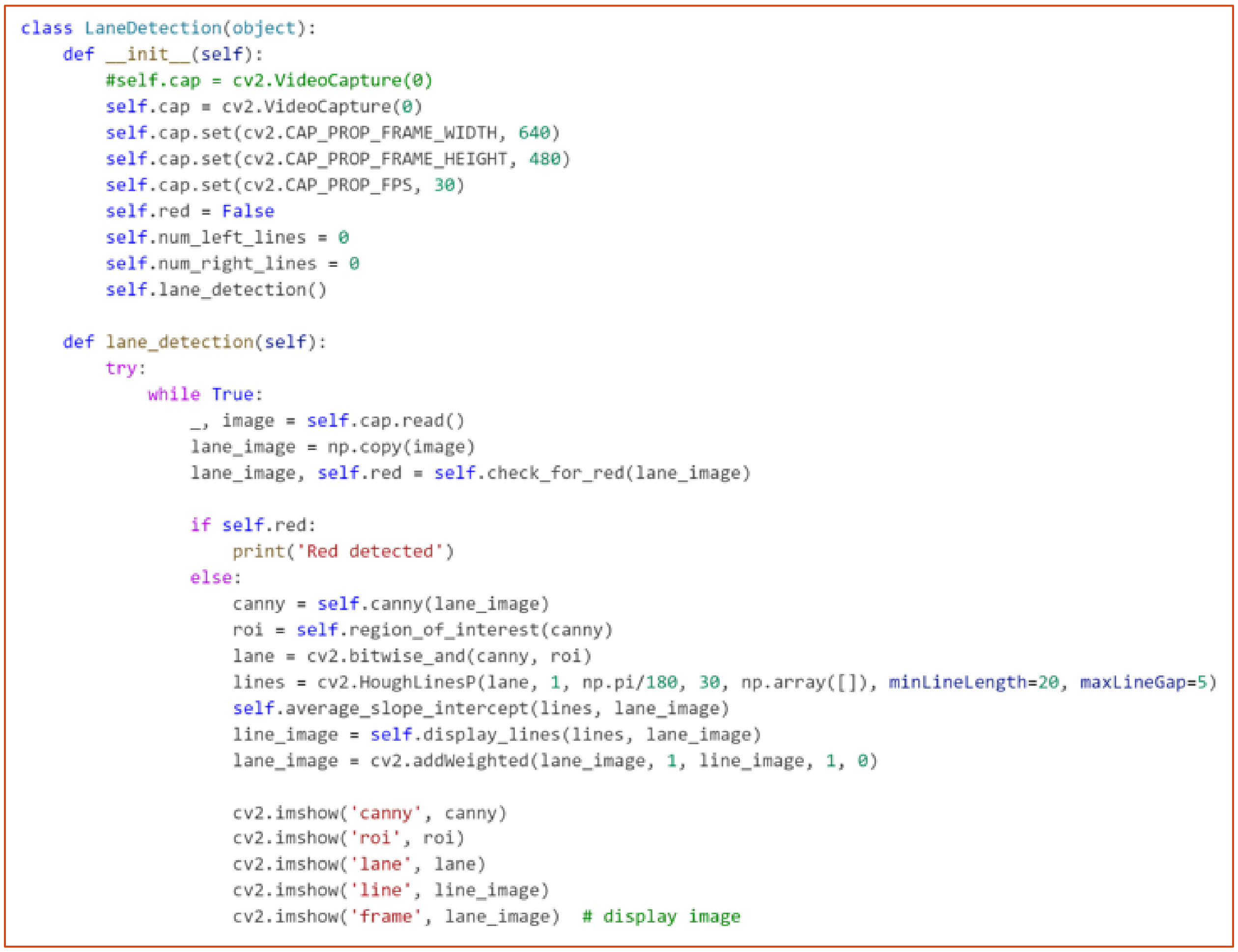
In this unit, we have two components for control that can be used either to drive the car itself or through a web page:

A. Self-Driving

The self-driving system is divided into two parts:

- **Lane detection:** which is responsible for identifying the road boundaries using various functions such as cv2 and it analyzing video stream
- **Object detection:** Which is used for detecting nearby vehicles and avoiding collisions with them using ultrasonic sensors. A function is used to calculate the distance between the car and the surrounding objects using 3 ultrasonic sensors. These codes are available on the Raspberry Pi.

The lane detection class initializes a video capture object and sets its properties. It then runs the lane detection function which reads frames from the video capture object, checks for the presence of red color (which indicates a stop signal), applies Canny edge detection and Hough transform techniques as shown in Figure 3.19 to extract the road lines, and displays the detected lines on the image.

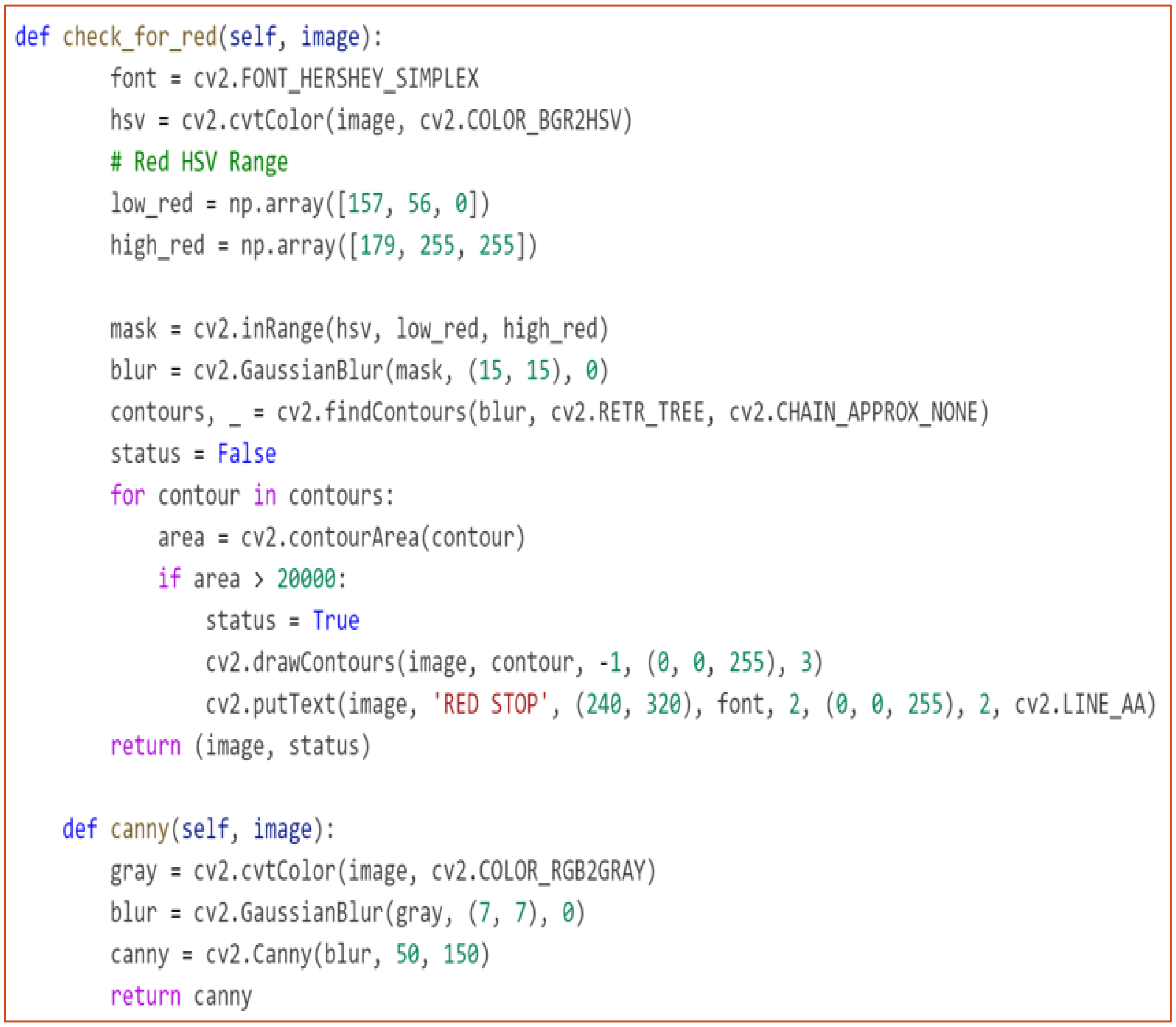


```
class LaneDetection(object):
    def __init__(self):
        #self.cap = cv2.VideoCapture(0)
        self.cap = cv2.VideoCapture(0)
        self.cap.set(cv2.CAP_PROP_FRAME_WIDTH, 640)
        self.cap.set(cv2.CAP_PROP_FRAME_HEIGHT, 480)
        self.cap.set(cv2.CAP_PROP_FPS, 30)
        self.red = False
        self.num_left_lines = 0
        self.num_right_lines = 0
        self.lane_detection()

    def lane_detection(self):
        try:
            while True:
                _, image = self.cap.read()
                lane_image = np.copy(image)
                lane_image, self.red = self.check_for_red(lane_image)

                if self.red:
                    print('Red detected')
                else:
                    canny = self.canny(lane_image)
                    roi = self.region_of_interest(canny)
                    lane = cv2.bitwise_and(canny, roi)
                    lines = cv2.HoughLinesP(lane, 1, np.pi/180, 30, np.array([]), minLineLength=20, maxLineGap=5)
                    self.average_slope_intercept(lines, lane_image)
                    line_image = self.display_lines(lines, lane_image)
                    lane_image = cv2.addWeighted(lane_image, 1, line_image, 1, 0)

                    cv2.imshow('canny', canny)
                    cv2.imshow('roi', roi)
                    cv2.imshow('lane', lane)
                    cv2.imshow('line', line_image)
                    cv2.imshow('frame', lane_image)  # display image
```

Fig. 3.19: Canny edge detection

The code then reads the distances measured by the ultrasonic sensors, and based on that, it steers the car to the left or right or stops it completely if the distance in front of the car is less than the allowed limit.

The check for red function checks for the presence of red color in the image using HSV color space, and draws a contour around the detected red area as shown in **Figure 3.20**. The average slope intercept function calculates the average slope and intercept for the left and right lines detected using linear regression.

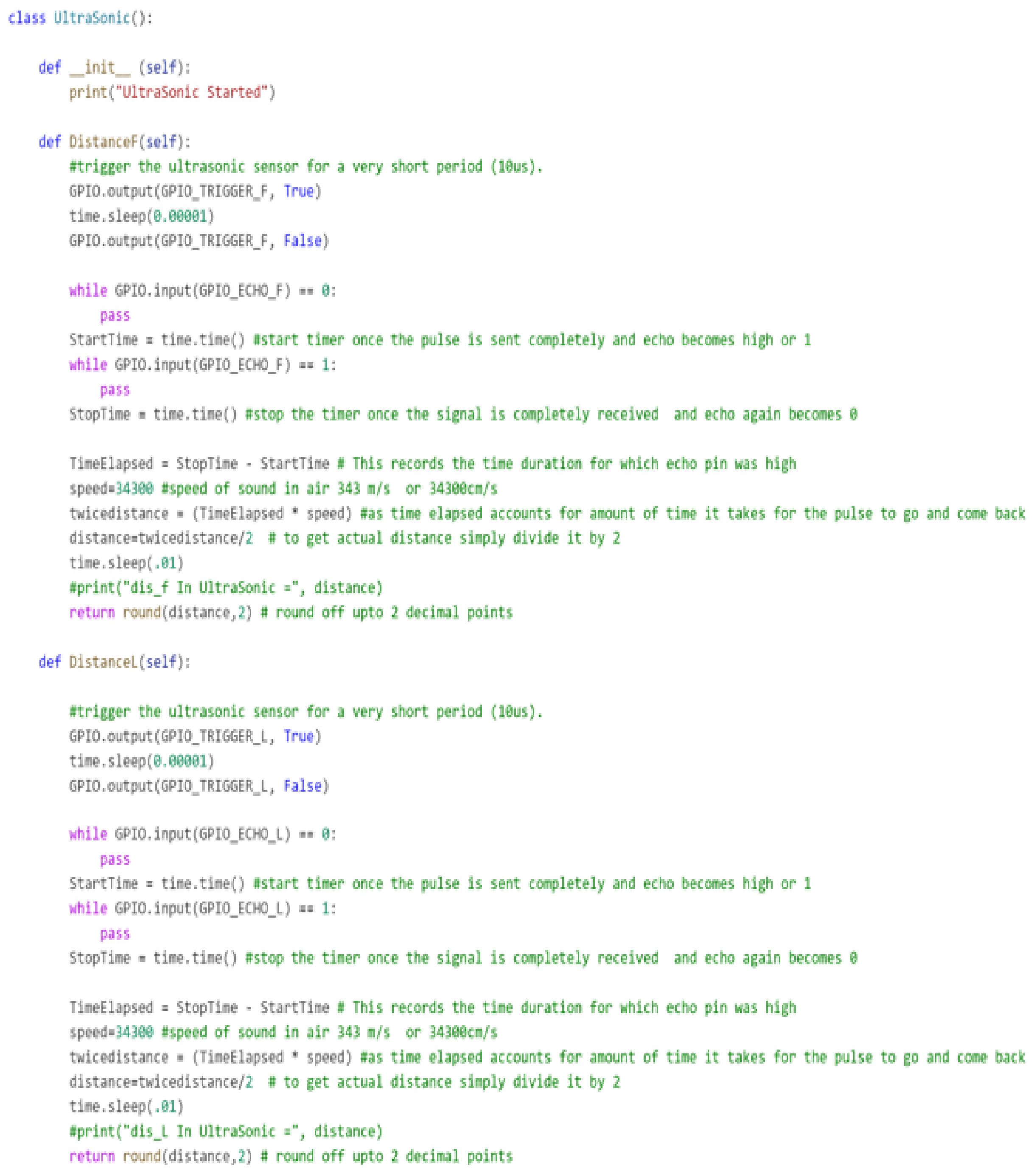


```
def check_for_red(self, image):
        font = cv2.FONT_HERSHEY_SIMPLEX
        hsv = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)
        # Red HSV Range
        low_red = np.array([157, 56, 0])
        high_red = np.array([179, 255, 255])

        mask = cv2.inRange(hsv, low_red, high_red)
        blur = cv2.GaussianBlur(mask, (15, 15), 0)
        contours, _ = cv2.findContours(blur, cv2.RETR_TREE, cv2.CHAIN_APPROX_NONE)
        status = False
        for contour in contours:
            area = cv2.contourArea(contour)
            if area > 20000:
                status = True
                cv2.drawContours(image, contour, -1, (0, 0, 255), 3)
                cv2.putText(image, 'RED STOP', (240, 320), font, 2, (0, 0, 255), 2, cv2.LINE_AA)
        return (image, status)

    def canny(self, image):
        gray = cv2.cvtColor(image, cv2.COLOR_RGB2GRAY)
        blur = cv2.GaussianBlur(gray, (7, 7), 0)
        canny = cv2.Canny(blur, 50, 150)
        return canny
```

Fig. 3.20: Average Slope Intercept function

The canny function converts the image to grayscale, applies Gaussian blur to remove noise, and then applies Canny edge detection to detect edges. The region of interest function defines a polygon-shaped mask to focus on the region of the image where the road lines are expected to be. The display line function draws the detected road lines on a black image and returns it.

Finally, the code moves the car using the function based on the detected road lines and distances measured by the ultrasonic sensors as shown in Figure 3.21. If the red color is detected or the distance in front of the car is less than the allowed limit, the car is stopped.



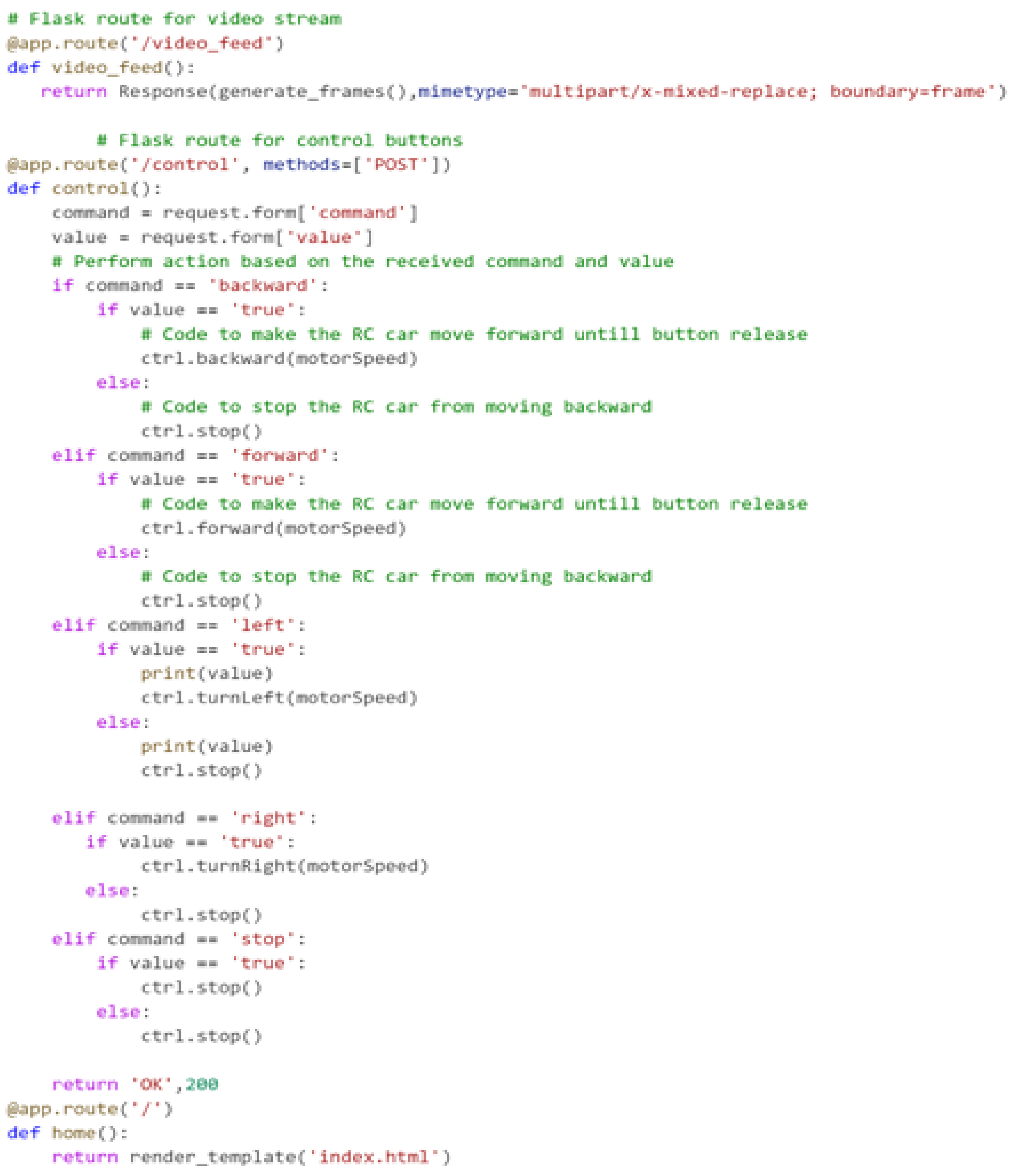
```
class UltraSonic():

    def __init__ (self):
        print("UltraSonic Started")

    def DistanceF(self):
        #trigger the ultrasonic sensor for a very short period (10us).
        GPIO.output(GPIO_TRIGGER_F, True)
        time.sleep(0.00001)
        GPIO.output(GPIO_TRIGGER_F, False)

        while GPIO.input(GPIO_ECHO_F) == 0:
            pass
        StartTime = time.time() #start timer once the pulse is sent completely and echo becomes high or 1
        while GPIO.input(GPIO_ECHO_F) == 1:
            pass
        StopTime = time.time() #stop the timer once the signal is completely received  and echo again becomes 0

        TimeElapsed = StopTime - StartTime # This records the time duration for which echo pin was high
        speed=34300 #speed of sound in air 343 m/s  or 34300cm/s
        twicedistance = (TimeElapsed * speed) #as time elapsed accounts for amount of time it takes for the pulse to go and come back
        distance=twicedistance/2  # to get actual distance simply divide it by 2
        time.sleep(.01)
        #print("dis_f In UltraSonic =", distance)
        return round(distance,2) # round off upto 2 decimal points

    def DistanceL(self):

        #trigger the ultrasonic sensor for a very short period (10us).
        GPIO.output(GPIO_TRIGGER_L, True)
        time.sleep(0.00001)
        GPIO.output(GPIO_TRIGGER_L, False)

        while GPIO.input(GPIO_ECHO_L) == 0:
            pass
        StartTime = time.time() #start timer once the pulse is sent completely and echo becomes high or 1
        while GPIO.input(GPIO_ECHO_L) == 1:
            pass
        StopTime = time.time() #stop the timer once the signal is completely received  and echo again becomes 0

        TimeElapsed = StopTime - StartTime # This records the time duration for which echo pin was high
        speed=34300 #speed of sound in air 343 m/s  or 34300cm/s
        twicedistance = (TimeElapsed * speed) #as time elapsed accounts for amount of time it takes for the pulse to go and come back
        distance=twicedistance/2  # to get actual distance simply divide it by 2
        time.sleep(.01)
        #print("dis_L In UltraSonic =", distance)
        return round(distance,2) # round off upto 2 decimal points
```

Fig. 3.21: Ultrasonic sensors function

This code is for ultrasonic sensors to calculate the distance between the car and the surrounding objects and send it to self-driving function to use it in object detection.

B. Web Application

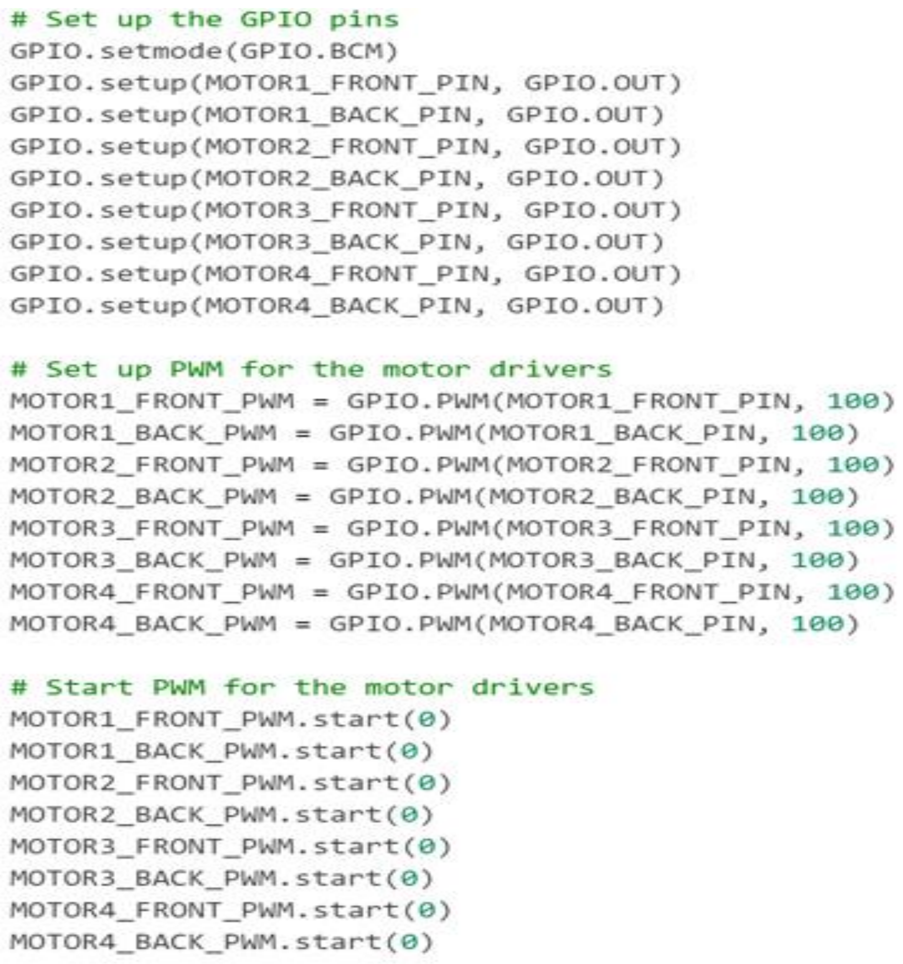
You are setting up a connection between Raspberry Pi and the web using Flask code, which is used to link between Python and Raspberry Pi as shown in **Figure 3.22.** Raspberry Pi loads an HTML code that generates a web page containing the control interface.



```
# Flask route for video stream
@app.route('/video_feed')
def video_feed():
   return Response(generate_frames(),mimetype='multipart/x-mixed-replace; boundary=frame')

        # Flask route for control buttons
@app.route('/control', methods=['POST'])
def control():
    command = request.form['command']
    value = request.form['value']
    # Perform action based on the received command and value
    if command == 'backward':
        if value == 'true':
            # Code to make the RC car move forward untill button release
            ctrl.backward(motorSpeed)
        else:
            # Code to stop the RC car from moving backward
            ctrl.stop()
    elif command == 'forward':
        if value == 'true':
            # Code to make the RC car move forward untill button release
            ctrl.forward(motorSpeed)
        else:
            # Code to stop the RC car from moving backward
            ctrl.stop()
    elif command == 'left':
        if value == 'true':
            print(value)
            ctrl.turnLeft(motorSpeed)
        else:
            print(value)
            ctrl.stop()

    elif command == 'right':
       if value == 'true':
            ctrl.turnRight(motorSpeed)
       else:
            ctrl.stop()
    elif command == 'stop':
        if value == 'true':
            ctrl.stop()
        else:
            ctrl.stop()

    return 'OK',200
@app.route('/')
def home():
    return render_template('index.html')
```

Fig. 3.22: Flask code

This interface includes a video stream and five buttons for forward, backward, right, left, and stop. Additionally, the web page contains a frame that displays the video stream from the front camera, as well as information about the location and distance. The control button is also included. Code is used in self -driving and flask that is responsible for controlling the wheels' motors, where it receives "forward", "backward", "left", "right", and "stop" commands from Flask and self-driving modules as shown in **Figure 3.23**, and outputs high and low voltage signals to the motor pins.



```
# Set up the GPIO pins
GPIO.setmode(GPIO.BCM)
GPIO.setup(MOTOR1_FRONT_PIN, GPIO.OUT)
GPIO.setup(MOTOR1_BACK_PIN, GPIO.OUT)
GPIO.setup(MOTOR2_FRONT_PIN, GPIO.OUT)
GPIO.setup(MOTOR2_BACK_PIN, GPIO.OUT)
GPIO.setup(MOTOR3_FRONT_PIN, GPIO.OUT)
GPIO.setup(MOTOR3_BACK_PIN, GPIO.OUT)
GPIO.setup(MOTOR4_FRONT_PIN, GPIO.OUT)
GPIO.setup(MOTOR4_BACK_PIN, GPIO.OUT)

# Set up PWM for the motor drivers
MOTOR1_FRONT_PWM = GPIO.PWM(MOTOR1_FRONT_PIN, 100)
MOTOR1_BACK_PWM = GPIO.PWM(MOTOR1_BACK_PIN, 100)
MOTOR2_FRONT_PWM = GPIO.PWM(MOTOR2_FRONT_PIN, 100)
MOTOR2_BACK_PWM = GPIO.PWM(MOTOR2_BACK_PIN, 100)
MOTOR3_FRONT_PWM = GPIO.PWM(MOTOR3_FRONT_PIN, 100)
MOTOR3_BACK_PWM = GPIO.PWM(MOTOR3_BACK_PIN, 100)
MOTOR4_FRONT_PWM = GPIO.PWM(MOTOR4_FRONT_PIN, 100)
MOTOR4_BACK_PWM = GPIO.PWM(MOTOR4_BACK_PIN, 100)

# Start PWM for the motor drivers
MOTOR1_FRONT_PWM.start(0)
MOTOR1_BACK_PWM.start(0)
MOTOR2_FRONT_PWM.start(0)
MOTOR2_BACK_PWM.start(0)
MOTOR3_FRONT_PWM.start(0)
MOTOR3_BACK_PWM.start(0)
MOTOR4_FRONT_PWM.start(0)
MOTOR4_BACK_PWM.start(0)
```

Fig. 3.23: Control Signals.

Script is written in Python and it controls the GPIO pins on a Raspberry Pi to control a four-wheeled car. The car has four motors, one for each wheel, and each motor has three pins: an enable pin for PWM control and two direction pins for forward and backward movement.

3.2.4 Simulation

One of the biggest challenges we faced was how to test and verify the efficiency of our code. It was essential for us to find a simulator that would help us visualize how our car behaves and reacts while driving on the road. That is why we turned to the Robot Operating System (ROS) framework. You can create a realistic three-dimensional (3D) simulated environment with ROS. You can create a 3D model of the car, map the obstacles, mark the route and simulate the movement of the car on this environment. With this, you can test your code and analyze how the vehicle behaves and reacts to surrounding conditions and obstacles. With this simulated environment in ROS, you can run your code, experience it in real life, and observe the behavior of the autonomous vehicle in a safe and certified simulated environment. You can analyze system performance and evaluate its efficiency and response to various challenges one of the reasons and advantages for our choice of the ROS simulation system and other features as will be discussed in the following topics.

A. **Visualization and Simulation**

ROS offers visualization tools that help in monitoring and debugging robot behavior. It includes graphical tools for visualizing sensor data, robot models, and trajectory planning. ROS also integrates well with simulation environments like Gazebo, allowing developers to test and validate their robot applications in realistic simulated environments.

B. Trajectory Planning:

ROS provides tools for visualizing and planning robot trajectories. Developers can visualize and manipulate robot paths to ensure smooth and collision-free motion. This is particularly useful for tasks such as path planning, obstacle avoidance, and motion control.

C. Integration with Gazebo

Gazebo is a popular physics-based robot simulator that integrates seamlessly with ROS. It provides a realistic simulation environment where developers can test and validate their robot applications before deploying them on physical robots. Gazebo supports accurate simulation of robot dynamics, sensor interactions, and environmental factors, enabling comprehensive testing and evaluation as shown in the following **figure 3.24**

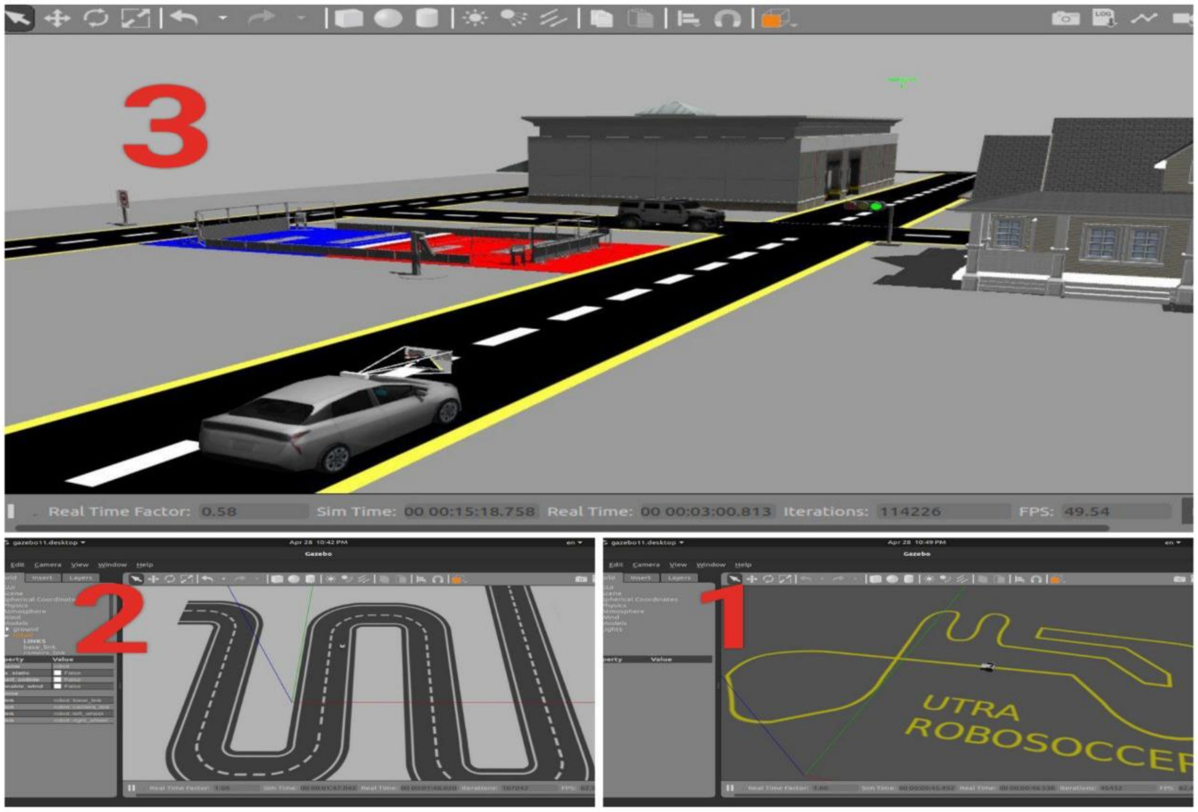


Fig.3.24: The virtual world is divided into three stages

As for the project, we have divided the virtual world into 3 parts, and then development in each part until we move to the goal to be reached.

- **The first part:**
  The virtual world was very simple, and it is a one-way road in which there is no object, and the goal here is that the car is driving on the road and taking orders automatically without any intervention from anyone, as the external camera captures the borders of the road and if the direction of the road is to North, a signal is sent to the motors to move to the north.
- **The second part:**
  After the completion of the first part, and we see here that the road is divided into two lanes and has become a road **similar** to reality, and here the camera determines the borders of the road and walks parallel to the borders of the road, in addition to that if there is a barrier or another car in front of us, the car slows down and stops until it is gone Obstacle.

- **The third and final part**

  Which is a road completely similar to the real world, and as we can see, we have added camera activities, and there have become many obstacles, as there are houses and other cars that meet our car. According to the color of the traffic light and we trained the camera to make a detection of this part dozens of times until we reach a good and satisfactory result for us.

- Robot **Models**

  ROS enables the creation and visualization of robot models. Developers can define the physical structure, kinematics, and dynamics of the robot using URDF (Unified Robot Description Format). The robot model can be visualized in 3D, allowing developers to assess the robot's configuration and movement as shown in figure 3.25.

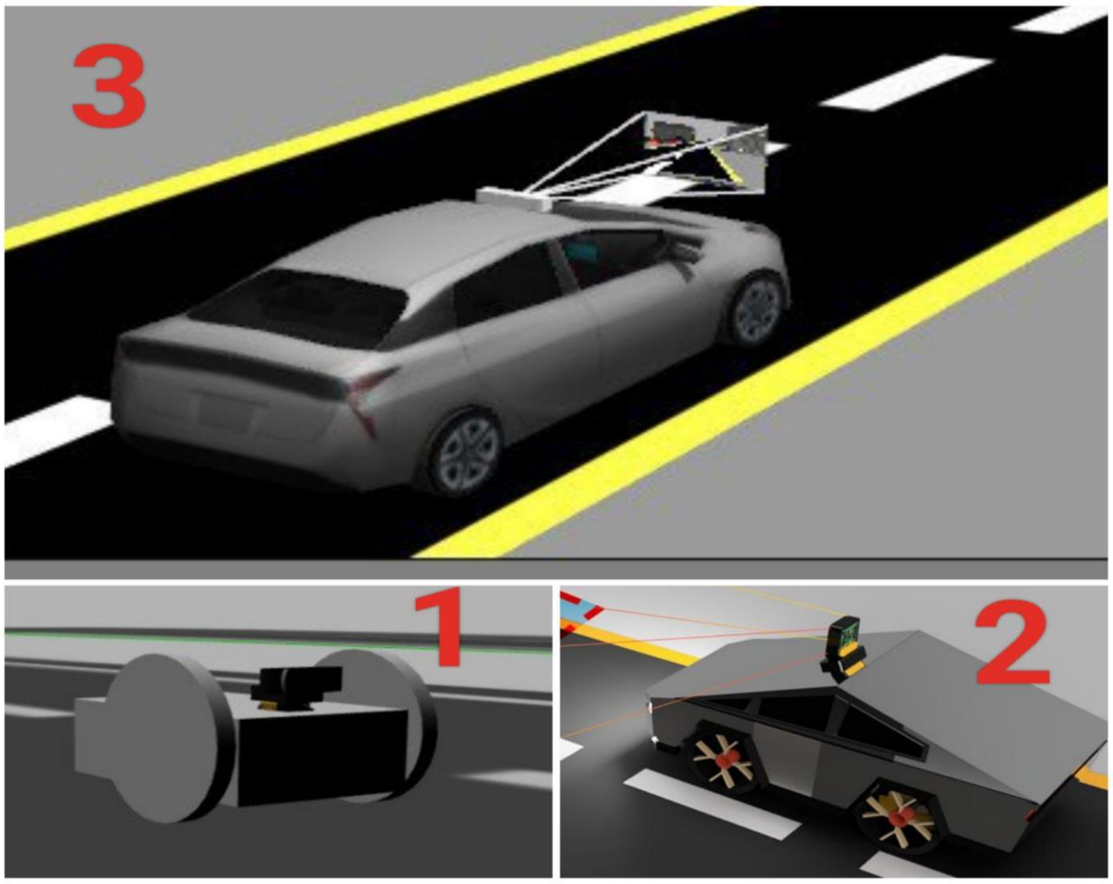


Fig. 3.25: The Robot stages

Summary

This chapter presented a detailed description of the proposed architecture and presented an experimental prototype that incorporates various hardware components to create an automated driving system for emergency situations. It utilizes an interior camera and a smartwatch to monitor the driver's condition in real-time and gather vital measurements. By integrating this information, the system can assess the driver's condition and take appropriate actions. It employs line and object detection algorithms to control the vehicle's movement and ensure it stays on a safe path. In case of system failure, there is a backup control system that allows remote control of the car through a web interface. The system also includes a mechanism to transmit the driver's vital indicators and location to a web-based platform, enabling prompt assistance from nearby emergency services. Overall, the system aims to enhance driver safety and provide efficient emergency response in critical situations.

The project focused on developing a self-driving car using a combination of functions and capabilities commonly found in autonomous vehicles. These functions include Sensing and Perception, Mapping and Localization, Decision-Making and Planning, Control and Execution, Real-Time Monitoring and Redundancy, and Connectivity and Communication. To verify the efficiency of the code, a simulation environment using the ROS framework was employed. The simulation allowed for testing and observing the behavior of the self-driving car in a realistic virtual environment. Visualization tools and integration with Gazebo, a physics-based robot simulator, facilitated the monitoring, debugging, and evaluation of the autonomous vehicle.