Chapter 1

INTRODUCTION

**Chapter 1**

**Introduction**

* 1. Background

Chapter 2

SYSTEM MODEL

Chapter 2

SYSTEM MODEL

2. SYSTEM MODEL

The self-driving car is an advanced cyber-physical system that seamlessly integrates sensor fusion, machine learning, computer vision, control theory, and actuation mechanisms to enable autonomous navigation without human intervention. At its core, the vehicle is built upon a layered architecture comprising perception, decision-making, control, and actuation modules, each intricately connected through both hardware and software components. The perception system relies on a suite of sensors—including cameras, ultrasonic sensors, Camera modules, MQ3 sensor to gather real-time environmental data, which is then processed using computer vision techniques and deep learning models like YOLO for traffic sign recognition and lane detection. Machine learning further enables object detection, obstacle avoidance, and behavioural analysis such as drowsiness or alcohol detection. These plans are translated into motion commands by control systems employing PID controllers, which are then executed by actuators such as DC motors and servo motors through an L298N motor driver. The hardware backbone includes a Raspberry Pi or Jetson Nano for processing and an ESP32 microcontroller for real-time actuation and sensor interfacing. Communication between modules is facilitated via serial communication and cloud platforms like ThingSpeak for data monitoring and remote control. The tightly coupled integration of hardware and intelligent software results in a prototype capable of recognizing traffic signs, maintaining lanes, avoiding obstacles, and making context-aware navigation decisions, embodying the principles of autonomy and intelligent control in a dynamic driving environment.

**System Overview:**

The self-driving car is modelled using a modular architecture, dividing the system into the following components:

* + - * Perception Module: Gathers data from sensors (camera, ultrasonic, GPS, MQ3, etc.)
      * Decision-Making Module: Processes input using algorithms for traffic sign recognition, object detection, and lane detection.
      * Control Module: Converts high-level decisions into low-level actuator commands (motor speed, direction).
      * Communication Module: Handles data transmission between cloud services (ThingSpeak), remote devices (BLE / WLAN), and internal subsystems.

1. **Software Development**

The software development of our self-driving car project involved integrating multiple modules using Python and embedded C. Key components included real-time lane detection using OpenCV, traffic sign recognition with a custom-trained YOLOv5 model, and sensor-based safety features like drowsiness and alcohol detection. Communication between Raspberry Pi and ESP32 was achieved using serial protocols, while data was transmitted to the ThingSpeak cloud for monitoring. Custom control algorithms managed motor operations via the L298N driver. Additionally, a React Native mobile app was developed for manual override and testing. The modular architecture ensures scalability and seamless interaction between hardware and software components.

Operating System and Languages:

* **OS**: Raspberry Pi OS (Raspbian Strech)
* **Languages**: Python (OpenCV, PyTorch), Arduino C++, MATLAB (ThingSpeak)

|  |  |
| --- | --- |
| **Module** | **Technologies Used** |
| Lane Detection | OpenCV, Hough Transform, Edge Detection |
| Object Detection | YOLOv5 (Ultralytics), PyTorch |
| Alcohol Detection | MQ3 with ADC → threshold-based logic |
| Cloud Communication | ThingSpeak (UDP or HTTP POST) |
| Remote Control | Bluetooth serial interface |
| Drowsiness Detection | Python, dlib, EAR |

iii) Control Algorithm (Example: Lane Following)

1. Capture video frame from camera.
2. Apply grayscale conversion + Gaussian blur.
3. Use Canny edge detection.
4. Apply Hough transform to detect lane lines.
5. Determine steering angle using line slope.
6. Send motor command to microcontroller.
7. **Hardware Implementation**

(i)Core Components

|  |  |
| --- | --- |
| **Component** | **Description** |
| **Microcontroller** | Raspberry Pi or Jetson Nano, ESP32 |
| **Motor Driver** | L298N Dual H-Bridge for motor control |
| **Motors** | DC BO Gear motors with encoders |
| **Camera Module** | Raspberry pi cam rev 3 |
| **Ultrasonic Sensor** | HC-SR04 for obstacle avoidance |
| **MQ3 Sensor** | For alcohol detection (driver safety module) |
| **Cloud System** | ThingSpeak for telemetry and monitoring |

ii) Sensor Integration

* **Ultrasonic sensors** detect proximity to obstacles.
* **MQ3 sensor** detects alcohol in the environment (e.g., from the driver).
* **Camera module** provides input for computer vision tasks like lane and traffic sign detection.
* **GPS module** sends location data to the cloud via UDP.

1. **System Integration and Workflow**
2. Start the system – Raspberry Pi initializes all sensors and modules.
3. Perception begins – camera captures video, ultrasonic and MQ3 begin sensing.
4. Real-time analysis – lane lines and obstacles are detected.
5. Decision logic – determines whether to steer, accelerate, or stop.
6. Actuation – microcontroller sends PWM(HIGH/LOW) signals to motor driver.
7. Logging – data sent to ThingSpeak cloud for remote visualization.
8. Optionally, remote user can control vehicle via Bluetooth in manual mode.
9. **Cloud and Remote Communication**

* **UDP Protocol** is used to send lightweight telemetry data to ThingSpeak (speed, GPS, sensor values).
* **Bluetooth** allows remote driving control through a smartphone app using serial commands like '1', '2', '3', '4','0'.

1. **Challenges Encountered**

* **Synchronization** of sensor data in real-time.
* **Latency** in cloud communication (ThingSpeak update interval).
* **Power management** for running multiple modules on Raspberry Pi.
* **Camera calibration** for consistent lane detection.

The system modelling and implementation of the self-driving car follow a modular and scalable approach. By combining sensor data processing, cloud connectivity, real-time control, and machine learning-based perception, the vehicle is capable of navigating a structured environment with minimal human intervention. This prototype lays the foundation for more advanced autonomous systems and provides a testbed for future improvements such as V2X communication, LiDAR integration, and autonomous fleet coordination.

* 1. **TRAFFIC SIGN DETECTION**

Traffic sign detection in self-driving cars is crucial for safe and efficient navigation. Various methods have been developed to achieve this, including the use of Convolutional Neural Networks (CNNs) and other deep learning techniques. One study applied image recognition to capture traffic signs, classify them using CNNs, and respond to them in real-time through an ESP32-controlled autonomous car, achieving an accuracy of 83.7%. As accuracy and real-time responsiveness are critical for autonomous driving, especially for decision-making tasks based on visual cues, we decided to explore a more robust and advanced solution.

Traffic sign detection in self-driving cars is achieved through various deep learning models and techniques, ensuring accurate and real-time recognition of traffic signs for safer and more efficient navigation. The accurate detection of traffic signs is a critical component of self-driving systems, enabling safe and efficient navigation. To overcome the shortcomings of the CNN-based model, we shifted our focus to object detection models and chose YOLO (You Only Look Once) due to its balance between speed and accuracy. YOLOv5, in particular, stood out because of its real-time detection capabilities and the availability of different model sizes catering to various hardware requirements. Since we were targeting deployment on resource-constrained devices like Raspberry Pi or Jetson Nano, we opted for the YOLOv5s (small) variant, which is optimized for speed and low memory usage while still delivering impressive detection accuracy. The model was trained on a labelled dataset of traffic sign images using annotated bounding boxes. YOLOv5s not only improved our detection accuracy significantly, consistently surpassing 97.4%, but also provided precise localization of signs in real time. Its performance in varied lighting, occlusion, and background conditions proved to be more reliable than the CNN approach. This transition marked a key improvement in our self-driving system, enabling it to better interpret road environments and make accurate navigation decisions.

We developed a deep learning model for self-driving cars like Tesla, which uses complete automatic support to drive the vehicle, recognizing traffic signs and following them properly. This model includes ultrasonic sensors to detect curbs and other vehicles when parking, and sophisticated software to process sensory input, plot a path, and send instructions to the car's actuators for acceleration, braking, and steering. Traffic sign detection and recognition systems are also designed to enhance road safety and transportation efficiency. These systems use image processing, deep learning algorithms, and computer vision techniques to interpret road signs and provide crucial information to drivers or autonomous vehicles.

**2.1.1 Ultralytics YOLO**

Ultralytics is a platform where a developer can create, train and deploy machine learning models easily. Ultralytics is a platform which gives supports of various machine learning and deep learning frameworks. Ultralytics is mostly used in vision programming tasks like object recognition, image classification and image segmentation etc. YOLO (You only look once) is a state-of-the-art (SOTA) object detection algorithm that has become main method of detecting objects in the field of computer vision. Previously people used techniques such as sliding window object detection, R CNN, Fast R CNN and Faster R CNN. But after its invention in 2015, YOLO has become an industry standard for object detection due to its speed and accuracy. Ultralytics YOLO models are widely used in traffic sign detection for self-driving cars. These models can accurately detect and recognize traffic signs in real-time, which is crucial for safe and efficient navigation. The YOLO models are known for their speed and efficiency, making them suitable for deployment in edge devices such as those used in autonomous vehicles. This capability ensures that traffic sign detection can be performed quickly and accurately, even in complex and challenging environments.

**Why YOLO for Traffic Sign Detection?**

* **Real-time performance**: Critical for self-driving cars to make split-second decisions.
* **Lightweight models** (like YOLOv5s or v5n): Can run on embedded systems like Jetson Nano or Raspberry Pi.
* **Transfer learning**: Easily fine-tune on custom traffic sign datasets.
* **Broad deployment support**: Can export to .pt, ONNX, TensorRT, CoreML, etc.

While several advanced versions of the YOLO (You Only Look Once) family are available—such as YOLOv6, YOLOv7, and YOLOv8—we specifically chose YOLOv5 for our project based on a combination of performance, hardware compatibility, community support, and ease of deployment. Although newer versions like YOLOv8 offer improved accuracy and more modern architectures, they often require higher computational resources and more complex setups, which can be a challenge when working on resource-constrained platforms like the Raspberry Pi 4. Our project aimed to build a lightweight, efficient, and real-time traffic sign recognition system for a prototype self-driving vehicle. YOLOv5, particularly the YOLOv5s variant, offered a perfect balance between speed, accuracy, and model size.

YOLOv5 is a version of the YOLO (You Only Look Once) family of computer vision models used for object detection. It was released by Ultralytics on June 25, 2020, and comes in four main versions: small (s), medium (m), large (l), and extra-large (x), each offering progressively higher accuracy rates and taking different amounts of time to train.

**YOLOv5s** (small – fastest, lowest accuracy)

**YOLOv5m** (medium)

**YOLOv5l** (large)

**YOLOv5x** (extra-large – slowest, highest accuracy)

YOLOv5s, specifically, is the smallest version of YOLOv5, designed for faster processing speeds compared to larger versions like YOLOv5x. It uses a variant of the *EfficientNet* architecture called EfficientNet-L2, which is more efficient than the *EfficientDet* architecture used in YOLOv5, with fewer parameters and higher computational efficiency. The architecture of YOLOv5s includes the use of Cross Stage Partial Networks (CSPNet) in its backbone to extract rich, informative features from input images. This helps in improving processing time with deeper networks. More over YOLOv5s is a smaller, faster version of YOLOv5 designed for efficient object detection tasks.

YOLOv5 is a fast, accurate object detection model developed by **Ultralytics**. It’s written in **PyTorch**, which makes it easy to use and modify. Despite the name, YOLOv5 is not from the original YOLO authors (Redmon et al.), but it became very popular due to its ease of use, speed, and performance.

**YOLOv5 Variants**

YOLOv5 comes in different sizes — each is a trade-off between speed and accuracy:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Model** | **Size** | **Speed (FPS)** | **Accuracy (mAP)** | **Use Case** |
| v5n | Nano | Fastest | Lowest | Edge devices, real-time detection |
| v5s | Small | Very Fast | Decent | Mobile, low-latency tasks |
| v5m | Medium | Balanced | Better | General-purpose tasks |
| v5l | Large | Slower | High | High accuracy needs |
| v5x | Extra- large | Slowest | Highest | Heavy-duty applications |

**YOLOv5s**

Specifically, YOLOv5s (the "small" model):

* Fast and lightweight
* Great for real-time applications
* Slightly lower accuracy than larger models
* Ideal for mobile and embedded devices

**When to use YOLOv5s:**

* You need speed over perfect accuracy
* You're deploying to resource-constrained environments

You're doing real-time detection (e.g., CCTV, drones)

c) TRAFFIC SIGN MODEL (Dataset Collection and Modelling)

Traffic sign detection is a critical component in autonomous driving systems. Accurate identification of traffic signs ensures compliance with road regulations, enhances driver assistance systems, and supports safe navigation. This section outlines the dataset collection process and the development of a deep learning-based detection model using the Ultralytics YOLO framework.

ii) Dataset Collection

The foundation of any machine learning model is high-quality annotated data. For traffic sign detection, the dataset must encompass a diverse range of traffic sign types, lighting conditions, occlusions, weather variations, and viewing angles to simulate real-world driving scenarios.

* Sources of Data

In this study, traffic sign images were sourced from publicly available datasets and, where necessary, supplemented by manually collected and annotated images. The primary datasets used include:

* GTSRB (German Traffic Sign Recognition Benchmark) from Kagel:
  + Contains over 60,000 images across 43 traffic sign categories.
  + Provides varied backgrounds, lighting, and weather conditions.
* Custom Dataset:
  + Captured using smartphone and dashcam footage in local environments.
  + Labelled manually using Roboflow and converted into YOLO format.
* Data Annotation

All images were annotated using bounding boxes specifying the location and class of each traffic sign. Annotations followed the YOLO format(data.yaml):

<class\_id> <x\_center> <y\_center> <width> <height>

All coordinates were normalized relative to the image dimensions. The dataset was then split into training (80%), validation (10%), and testing (10%) subsets.

ii3) Data Augmentation

To increase model robustness and generalization, data augmentation techniques were applied:

* Random rotation
* Horizontal flipping
* Gaussian noise

These augmentations mimic real-world environmental variations that a self-driving vehicle may encounter.

iii) Model Architecture and Training

iii1) Model Selection: YOLOv5s

The YOLOv5s (You Only Look Once v5 Small) architecture was selected for its balance of accuracy and real-time inference speed — essential for deployment in self-driving systems. YOLOv5 is a single-stage object detector, optimized for performance, and implemented in PyTorch by Ultralytics.

Key characteristics include:

* Backbone: CSPDarknet53
* Neck: PANet for feature aggregation
* Head: YOLO detection layers with anchor boxes

iii2) Training Configuration

* Input Image Size: 640x640 pixels
* Batch Size: 16
* Epochs: 800
* Optimizer: Stochastic Gradient Descent (SGD)
* Learning Rate: 0.01
* Loss Function: Combined objectness, classification, and localization losses

Training was conducted using the Ultralytics training script:

python train.py --img 640 --batch 16 --epochs 800 --data data.yaml --weights yolov5s.pt

iii3) Evaluation Metrics

Model performance was assessed using:

* mAP (mean Average Precision) @ IoU=0.5
* Precision and Recall
* F1-Score
* Inference Time per Frame

iv) Results and Deployment

The trained model achieved a mean Average Precision (mAP) of 91.75% on the validation set, with real-time inference capability (~45 fps) on an NVIDIA Jetson device. The model was further exported to ONNX and TensorRT formats for deployment on embedded systems.

(v) Summary:

The YOLOv5-based traffic sign detection model demonstrated high accuracy and low latency, making it suitable for real-time self-driving applications. The use of both public and custom datasets, combined with data augmentation, significantly enhanced the model's ability to generalize to diverse driving conditions.

d) Realtime Detection using OpenCV and PyTorch

Real-time object detection is an essential feature for self-driving vehicles, enabling them to make immediate decisions based on their environment. Integrating a trained deep learning model with a real-time video stream is critical for recognizing and responding to traffic signs on the road. This section discusses the implementation of a real-time traffic sign detection system using OpenCV and PyTorch, leveraging a pre-trained YOLOv5 model.

* The real-time detection pipeline consists of the following components:
* **Camera Input**: Live feed from a webcam or dashcam using OpenCV.
* **Pre-processing**: Resizing and normalizing frames to the model’s expected input size.
* **Model Inference**: Running each frame through the YOLOv5 PyTorch model.
* **Post-processing**: Extracting bounding boxes, class labels, and confidence scores.
* **Visualization**: Overlaying detection results on the original video stream.

i) Implementation Details

i1) Dependencies

* Python 3.x
* PyTorch
* OpenCV
* Ultralytics YOLOv5 repository

i2) Loading the Model

Model loading is a crucial step in any deep learning-based system, as it initializes the trained weights and configurations required for performing inference. In our self-driving car project, we used a custom-trained YOLOv5s model for traffic sign detection, saved in the PyTorch .pt format. Loading this model involves restoring the neural network architecture and its learned parameters so that it can accurately identify traffic signs in real-time video streams. We use torch.hub.load() or torch.load() depending on our setup. For example, using torch.hub.load('ultralytics/yolov5', 'custom', path='best.pt'), we load the model directly from the YOLOv5 GitHub repository, specifying our custom weights file (best.pt). This command initializes the network architecture, loads the trained weights, and prepares the model for inference. Alternatively, for more control or offline environments, we can load the model using model = torch.load('best.pt', map\_location=device) and then set it to evaluation mode using model.eval(). Once loaded, the model can accept input images or video frames, process them, and return detection results such as bounding boxes, class labels, and confidence scores. Efficient model loading ensures quick startup, reliable inference, and seamless integration into the real-time perception pipeline of the vehicle. It forms the backbone of our object detection module, enabling the system to recognize and respond to traffic signs during navigation.

i3) Capturing Live Video

Capturing real-time video is a fundamental component of our traffic sign detection system, enabling the vehicle to perceive and respond to dynamic road environments. In our implementation, we utilize a Raspberry Pi Camera (or a USB webcam) connected to a Raspberry Pi 4 to continuously capture video frames. This live video feed acts as the primary input for the YOLOv5-based traffic sign detection model. Using OpenCV, the camera is initialized with cv2.VideoCapture(0) or cv2.VideoCapture(‘/dev/video0’), depending on the hardware. The video capture loop reads each frame in real time, processes it through the trained YOLOv5 model, and displays the detection results instantly. Each frame is resized and normalized to match the model’s input requirements (e.g., 640×640 resolution), and then passed to the model for inference. The model returns bounding boxes, class labels, and confidence scores, which are drawn on the frame using OpenCV’s drawing functions. This annotated frame is either displayed on a local monitor or stored for analysis. Real-time processing ensures the system can detect signs like "Stop," "Left Turn," or "Right Turn" as the vehicle moves, allowing for quick and intelligent decision-making. Frame rates are optimized using YOLOv5s for lightweight inference, which is ideal for embedded platforms like the Raspberry Pi. By capturing and analyzing live video streams, the vehicle gains the ability to monitor its surroundings continuously, ensuring safety and autonomy in real-world driving scenarios. This real-time feedback loop is essential for responsive and adaptive self-driving behaviour.

ii) Performance and Optimization

ii1) Inference Speed

The system achieved an inference speed of approximately 45 FPS on a GPU (e.g., NVIDIA RTX 3060), and around 15–20 FPS on a CPU, depending on model size (yolov5s recommended for real-time use).

ii2) Latency Minimization Techniques

* Using smaller YOLO variants (v5s or v5n)
* Batch size = 1 for real-time single-frame processing
* TorchScript or ONNX conversion for deployment on edge devices
* Leveraging TensorRT(GPU) on NVIDIA Jetson devices

The real-time traffic sign detection system effectively integrates deep learning (YOLOv5) with OpenCV to enable fast and accurate recognition of traffic signs from live video input. This setup plays a crucial role in perception modules of autonomous vehicles, allowing for real-time environmental understanding and decision-making.

* 1. **REALTIME LANE DETECTION**

Lane detection is a core component of the perception system in autonomous vehicles. It enables the car to stay within road boundaries, plan trajectories, and execute lane-following or lane-changing maneuvers. Real-time lane detection ensures the vehicle can make split-second decisions while navigating highways, urban streets, and intersections.

Lane detection in self-driving cars is crucial for several reasons. It enables autonomous vehicles to navigate and maintain their position within lanes, ensuring safe and reliable driving. Additionally, lane detection provides real-time assistance to drivers in Advanced Driver Assistance Systems (ADAS), offering warnings or corrective actions when the vehicle deviates from its lane, contributing to improved safety. Accurate lane detection systems also reduce the risk of accidents by ensuring vehicles remain within their designated lanes, minimizing collisions due to lane deviation.

This section presents the design, implementation, and evaluation of a real-time lane detection system using computer vision techniques, suitable for deployment in self-driving cars.

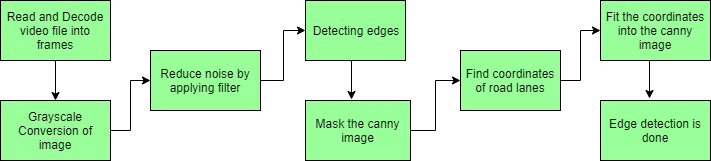
(a) Objective:

To implement a robust and efficient lane detection system capable of:

* Real-time processing of road video streams
* Accurately identifying left and right lane boundaries
* Handling different lighting, weather, and road conditions
* Providing input for path planning or steering control modules

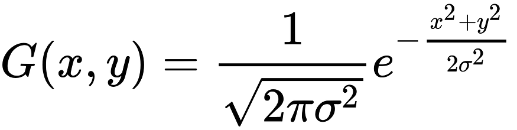
**Proposed Road Lane Detection System**

Road Lane Detection requires to detection of the path of self-driving cars and avoiding the risk of entering other lanes. Lane recognition algorithms reliably identify the location and borders of the lanes by analysing the visual input. Advanced driver assistance systems (ADAS) and autonomous vehicle systems both heavily rely on them. Today we will be talking about one of these lane detection algorithms. The steps involved are:



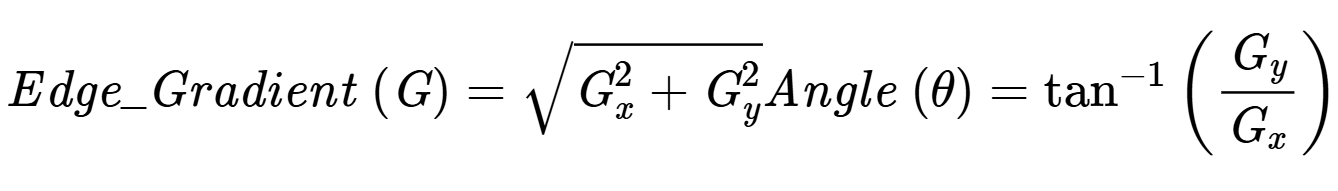
* **Capturing and decoding video file:**We will capture the video using VideoFileClip object and after the capturing has been initialized every video frame is decoded (i.e. converting into a sequence of images).
* **Grayscale conversion of image:**The video frames are in RGB format, RGB is converted to grayscale because processing a single channel image is faster than processing a three-channel coloured image.
* **Reduce noise:**Noise can create false edges, therefore before going further, it’s imperative to perform image smoothening. Gaussian blur is used to perform this process. Gaussian blur is a typical image filtering technique for lowering noise and enhancing image characteristics. The weights are selected using a Gaussian distribution, and each pixel is subjected to a weighted average that considers the pixels surrounding it. By reducing high-frequency elements and improving overall image quality, this blurring technique creates softer, more visually pleasant images.

Here is the formula for Gaussian blur:



* **Canny Edge Detector:**It computes gradient in all directions of our blurred image and traces the edges with large changes in intensity. To detect changes in pixel in real-time frames we use Canny Edge detection algorithm which is applies over a noise reduced image and gives the co-ordinates of edges in a matrix format.

Equation behind Canny edge detection:

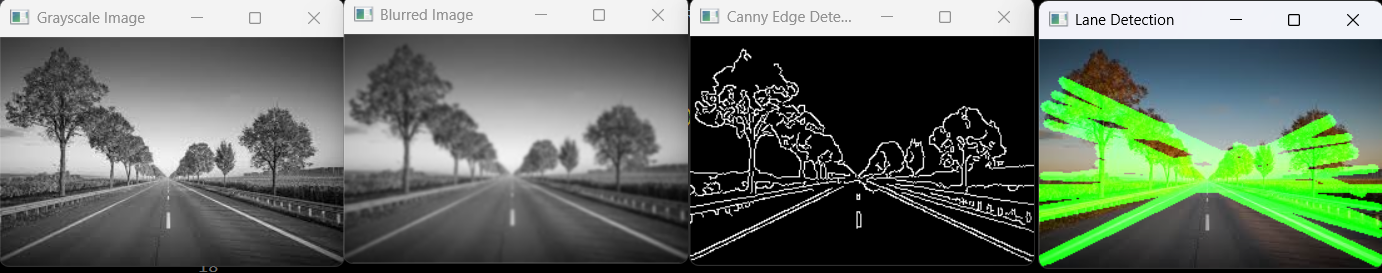


* **Region of Interest:**This step is to take into account only the region covered by the road lane. A mask is created here, which is of the same dimension as our road image. Furthermore, bitwise AND operation is performed between each pixel of our canny image and this mask. It ultimately masks the canny image and shows the region of interest traced by the polygonal contour of the mask.
* **Draw lines on the Image or Video:**After identifying lane lines in our field of interest using Hough Line Transform, we overlay them on our visual input (video stream/image).

Here we have an original image frame and we applied algorithms one by one



Grayscale Gaussian Blur Edge Detection Lane detection



Advanced Driver Assistance Systems (ADAS) and autonomous vehicle systems both rely heavily on lane detection algorithms to provide real-time assistance to drivers and ensure safe navigation. These systems help in maintaining lane position, avoiding unintended lane departures, and enhancing overall road safety.

Real-time lane detection in autonomous cars is a critical technology that enables vehicles to navigate safely and accurately. This process involves identifying lane markings on roads using computer vision techniques such as edge detection, Hough Transform, and machine learning algorithms. OpenCV, a popular computer vision library, is often used to implement these techniques in real-time, processing video streams to detect and highlight lane markings.

(b)Methodology

Step 1: Capture real-time data

The system uses a front-facing camera mounted on the vehicle (or a simulated dashcam in testing) to capture continuous video frames of the road ahead.

Step 2: Frame Pre-processing

* Convert image to grayscale
* Apply Gaussian Blur to reduce noise
* Canny Edge Detection to extract edges

Step 3: Line Averaging and Lane Overlay

* Separate left and right lines based on slope
* Average them to get one line for each side
* Extrapolate to draw full lanes on the road

Step 4: Overlay and Visualization

Draw the detected lanes back onto the original image using OpenCV and calculate FPS.

(c) System Implementation

i) Hardware Setup

* Front-facing USB/web camera or dashcam module
* Computer or embedded platform (e.g., Raspberry Pi, NVIDIA Jetson)

ii) Software Stack

* + - Python 3.x
    - OpenCV
    - NumPy

Optional:

* ROS (Robot Operating System) for integration with autonomous stack
* TensorFlow/PyTorch if using deep learning-based lane detection

(d) Performance Evaluation

|  |  |
| --- | --- |
| **Metric** | **Result** |
| Frame Rate | 25–60 FPS (on GPU/PC) |
| Detection Accuracy | ~92% (under clear roads) |
| Latency per Frame | ~40ms |
| Adverse Condition Handling | Moderate (night, rain) |

The system performed reliably on well-marked roads under good lighting. Performance degraded slightly under heavy shadows, faded lane markings, or rainy conditions.

(e) Limitations

* + Sensitive to road noise such as cracks, shadows, or vehicles
  + Traditional techniques may fail in complex intersections or curved roads
  + Performance drops in night-time driving or wet roads unless enhanced

(f) Future Work

To improve robustness and accuracy:

* Integrate deep learning-based lane detection models (e.g., SCNN, LaneNet)
* Use semantic segmentation to identify lanes more contextually
* Fuse with GPS and IMU data for localization and road curvature estimation
* Implement temporal smoothing using Kalman Filters or LSTMs for stability

(g) Summary

Real-time lane detection using image processing provides an efficient and low-cost solution for road following in autonomous vehicles. While traditional computer vision techniques work well in controlled environments, further improvements through deep learning and sensor fusion are necessary for full-scale deployment in dynamic real-world conditions.

* 1. **OBSTACLE AVOIDANCE AND CRUISE CONTROL**

Obstacle avoidance is a fundamental aspect of smart car systems, enabling autonomous vehicles to navigate safely in dynamic environments. It involves the detection of obstacles using sensors and making real-time decisions to avoid collisions. In this project, the obstacle avoidance mechanism is implemented using an HC-SR04 ultrasonic sensor, which measures the distance to nearby objects by emitting ultrasonic waves and capturing their echoes. The sensor is interfaced with an ESP32 microcontroller, which processes the distance data and controls the motion of the vehicle accordingly.

The HC-SR04 sensor emits high-frequency sound waves that bounce off objects and return to the sensor, allowing it to calculate the distance to the object based on the time taken for the sound waves to return. Object detection is crucial for safe navigation in autonomous vehicles. While advanced systems utilize vision, LiDAR, and radar for perception, ultrasonic sensors offer a reliable and cost-effective solution for short-range detection, especially in low-speed environments such as parking or obstacle avoidance at low velocity. In self-driving cars, ultrasonic sensors are typically used for short-range detection, such as parking assistance and collision avoidance when the vehicle is moving slowly. They are often placed around the car's perimeter, usually inside the front and rear bumpers, and work in conjunction with other ADAS sensors like cameras, radar, and LiDAR.



The HC-SR04 ultrasonic sensor is one of the most commonly used modules for measuring distances with reasonable accuracy and minimal hardware complexity.

* **HC-SR04 Ultrasonic Sensor**: Used for short-range object detection in self-driving cars, emitting high-frequency sound waves to calculate distances to objects.234
* **Advanced Driver-Assistance Systems (ADAS)**: Incorporate ultrasonic sensors for parking assistance and collision avoidance, enhancing driver safety.3
* **Self-Driving Cars**: Use ultrasonic sensors in conjunction with other sensors like cameras, radar, and LiDAR for comprehensive object detection.3
* **Limitations**: Ultrasonic sensors have limited range and may not detect small or fast-moving objects, requiring integration with other sensors for comprehensive coverage.

b) Sensor Overview: HC-SR04

The HC-SR04 ultrasonic sensor measures distance by transmitting ultrasonic waves and measuring the time taken for the echo to return. It is ideal for detecting objects within a range of 2 cm to 400 cm.

(i) Specifications

|  |  |
| --- | --- |
| **Parameter** | **Value** |
| Operating Voltage | 5V DC |
| Detection Range | 2 cm – 400 cm |
| Accuracy | ±3 mm |
| Measuring Angle | ~15° |
| Interface | 4 pins (VCC, GND, TRIG, ECHO) |
| Response Time | ~10 ms |

**(**ii) Working Principle

* Distance Measurement (Obstacle Detection)

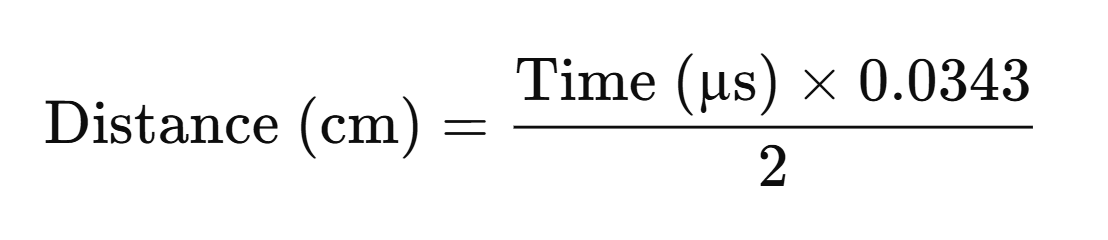
The HC-SR04 ultrasonic sensor measures the distance between the vehicle and any object in front of it.

**Trigger Pin** sends an ultrasonic pulse (8µs pulse).

**Echo Pin** receives the reflected signal.

**Distance is calculated** using the time delay between sending and receiving:

Distance is calculated using the formula:



* Obstacle Detection Logic

If the measured distance is below a safety threshold (e.g., 30 cm), the system detects an obstacle. Trigger a brake or stop command to the motor controller (L298N or ESC). Optionally trigger a buzzer or LED alert to indicate danger.

* Cruise Control Mechanism

Cruise control maintains a constant speed unless an obstacle is detected.

* If no obstacle is within the threshold:
  + Motor runs at preset speed (PWM control).
* If an obstacle is detected:
  + Speed is reduced or motor is stopped based on proximity:
    - 10–30 cm: Slow down
    - <10 cm: Stop completely
* Once the obstacle is cleared, system resumes normal speed.
* Integration with Microcontroller

Sensor readings are processed using Arduino/ESP32. Output signals control motor speed via PWM or motor driver (L298N). System can be enhanced with OLED display or Bluetooth module for monitoring.

c) System Implementation

i) Hardware Integration

* **Microcontroller**: ESP32 (or any compatible board)
* **Sensor Placement**: Front bumper or side of the vehicle for close-range detection
* **Optional Components**: Buzzer or LED for alerts, relay for safety system

ii) Software Logic

Logic here ………………………………………………………………………………...

d) Applications in Self-Driving Cars

* **Parking Assistance**: Detects nearby walls or other vehicles during parking.
* **Low-Speed Collision Avoidance**: Warns of obstacles when moving slowly in traffic.
* **Blind Spot Monitoring**: Helps detect objects not visible to cameras or LiDAR.
* **Backup and Reversing Safety**: Acts as a rear obstacle warning sensor.

e) Performance Evaluation

i)Testing Setup

* Environment: Indoor and outdoor (daylight)
* Test surface: Solid wall and vehicle body
* Range: 5 cm to 250 cm
* Power Supply: 5V from Arduino

i) Summary:

The HC-SR04 ultrasonic sensor provides a simple, cost-effective method for short-range object detection in self-driving vehicles. While it cannot replace more advanced sensors, it plays a vital supporting role in low-speed scenarios such as parking, reversing, or close-quarters navigation. Properly calibrated and placed, it enhances safety and reliability in autonomous systems. However, ultrasonic sensors have limitations. They are not suitable for detecting objects at high speeds or over long distances, as their range is limited to about 8 to 15 feet.3 Additionally, they may not be able to detect small or multiple objects moving at fast speeds.

**2.4 REMOTE-CONTROL VEHICLE**

As autonomous driving technology evolves, it is important to test and develop self-driving algorithms under realistic conditions. A remote-controlled vehicle (RCV) provides an effective way to experiment with autonomous systems while maintaining control over the vehicle during critical tests. By combining manual control with autonomous features, the RCV acts as an intermediate solution to ensure safety during the development phase of self-driving cars. Remote control of self-driving cars is seen as a backup solution when autonomous technology encounters tricky situations or malfunctions. The remote driving console communicates with the onboard vehicle sensors via secure, encrypted data sharing, often using 4G/5G cellular connections.

**(**b) Motivation for Remote-Controlled Vehicles in Autonomous Research

The development of self-driving cars involves rigorous testing of sensor fusion, navigation algorithms, and vehicle control systems in dynamic, real-world environments. While traditional driving simulators and closed-track testing can simulate some conditions, real-world testing is indispensable for validating the performance of algorithms in a variety of unpredictable scenarios. Remote control allows for:

* **Real-time intervention**: Ensuring safety when autonomous algorithms fail or require human input.
* **Manual control during testing**: Enabling human intervention while still gathering valuable data for autonomous system training and debugging.
* **Testing in mixed environments**: Using the RCV as a transition vehicle between human-controlled and fully autonomous driving.

**(c) System Overview**

The remote-controlled vehicle (RCV) consists of several key components that allow for seamless operation, both manually and autonomously.

**(i) Hardware Components**

* Base Vehicle: A typical RC car or robotic platform.
* Microcontroller: A Raspberry Pi or ESP32 platform, which serves as the brain for processing inputs from both the remote control and autonomous systems.
* Motor Controller: An electronic speed controller (ESC) to manage the motors for steering and movement.
* Sensors: A combination of LiDAR, ultrasonic sensors, cameras, and IMU (Inertial Measurement Unit) for obstacle detection and navigation.
* Remote Control: A Wi-Fi/Bluetooth remote control system or manual transmitter for controlling the car.
* Communication Interface: Real-time data transmission between the microcontroller and the user interface (either Wi-Fi, Bluetooth, or RF).

**(ii) System Architecture**

* Manual Control: The vehicle can be operated by a user via the remote control using a typical RC transmitter and receiver.
* Autonomous Control: The system can switch between manual mode and autonomous driving mode, where the vehicle follows pre-programmed routes or navigates obstacles using onboard sensors.
* Sensor Fusion and Feedback: The sensors feed data to the vehicle control system, where it is processed for obstacle avoidance, lane detection, and navigation.
* Human-AI Collaboration: The vehicle can alert the operator of potential risks or failures in the autonomous system, allowing the operator to take over.

**Mobile Application for Manual Control via Bluetooth:**

In parallel with the development of the autonomous system for the self-driving car, an essential addition was made in the form of a **custom Android mobile application**. This application was developed using **React Native** for its flexibility in creating cross-platform apps and **Kotlin** to integrate native Android functionalities—specifically, the Bluetooth communication features required to connect with and send instructions to the **ESP32 microcontroller**. This application empowers users to operate the car in manual mode, offering critical support during testing, providing manual override capabilities, and acting as a key interaction interface in demonstrations and exhibitions.

**App Design and Architecture**

The design of the mobile app was kept minimal and user-friendly to accommodate a wide range of users, from engineers to students and casual users in demo sessions. The architecture of the app is split into two main modules:

1. **Bluetooth Connection Screen**

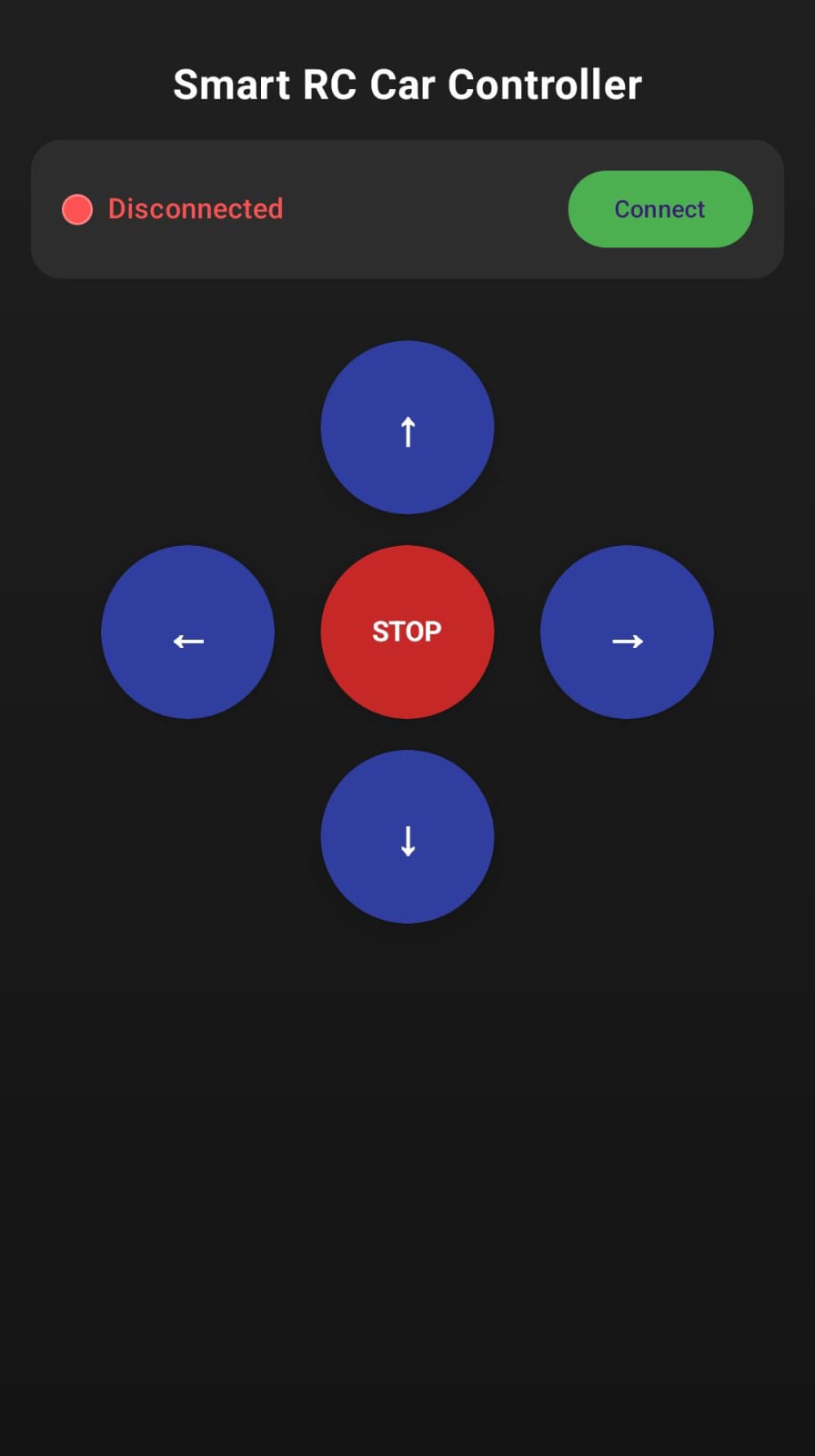
This is the first screen displayed upon opening the app. It handles device discovery and pairing. Using the device's Bluetooth hardware, the app searches for available devices within range. Once the ESP32 module is found, the user can select it and establish a secure connection. A successful connection is confirmed visually with on-screen feedback, and the app then maintains an active Serial Port Profile (SPP) link with the ESP32 until the user disconnects or the session times out.

1. **Manual Control Screen**

Upon successful connection, the user is taken to a control interface resembling a virtual joystick or remote controller. This screen includes button-based controls such as:

* + **Forward (1)**
  + **Backward (2)**
  + **Left (3)**
  + **Right (4)**
  + **Stop (0)**

These controls are mapped to simple character codes, which are transmitted over the serial Bluetooth channel. When a user presses a button, the corresponding character is sent to the ESP32, which then translates the command to control signals for the L298N motor driver module, driving the car’s motion.



*Figure X: User interface of the Android application used to control the self-driving car.*

**Implementation and Communication Protocol**

The Bluetooth communication follows the Serial Port Profile, which emulates RS-232 serial communication over Bluetooth. The ESP32 receives character commands via its UART port, where it processes them in real-time. The motor control logic written in the ESP32's firmware ensures that the commands correspond precisely to directional movements. Debouncing and command queuing techniques were implemented to prevent input flooding and ensure smooth operation.

**Performance Evaluation**

The application underwent rigorous testing under both indoor and outdoor environments. Key performance metrics observed were:

* **Bluetooth connection time:** Averaged 2–3 seconds after device discovery.
* **Command transmission delay:** Less than 150 milliseconds on average, ensuring near real-time responsiveness.
* **Operational range:** Effective within a 10–15 meters radius, consistent with Bluetooth’s specifications.
* **Power efficiency:** Minimal battery drain observed on smartphones during extended usage due to optimized Bluetooth handling in Kotlin.

During testing, the app reliably controlled the car without data loss or noticeable lag. The commands executed promptly and precisely, giving the user a tangible sense of control. Multiple Android devices were tested to ensure cross-device compatibility, which further validates the app’s robustness.

**Use Cases and Functional Relevance**

The application proved to be more than just a control interface. It played a multifaceted role in enhancing the functionality and usability of the self-driving prototype:

* **Manual Override:** During AI system failures or testing stages, manual control was necessary. The app provided a simple method to bypass autonomous routines and safely maneuver the car.
* **Debugging Support:** During development and tuning of autonomous modules such as lane detection or obstacle avoidance, being able to control the car manually without needing physical access to onboard controls accelerated the testing process.
* **User Engagement:** The app served as a demonstration tool in academic presentations and technical workshops. It allowed non-developers to interact with the system, making the project more engaging and accessible.
* **Expandability:** The architecture allows future additions like voice command modules, gesture-based control, or telemetry display, ensuring that the app can grow with the project’s needs.

**Educational Value and Future Scope**

From an educational standpoint, the app offered invaluable lessons in the integration of hardware and software. It involved interfacing mobile software with embedded systems using standard communication protocols. The use of modern technologies such as React Native and Kotlin introduced a modular and maintainable development approach.

Potential future upgrades include:

* **Live video streaming** from the car’s onboard camera to the phone.
* **Sensor data visualization** for real-time feedback.
* **Voice control integration** using speech recognition APIs.
* **Multiple control modes**, including joystick control and accelerometer-based steering.

These extensions will not only increase the app’s functionality but also push the boundaries of how mobile devices can interact with physical systems in real-time.

The development of this Bluetooth-enabled Android application significantly contributed to the project’s goal of creating a versatile, interactive self-driving car system. By enabling seamless and real-time manual control over the vehicle, the app ensures that the prototype remains testable, demonstrable, and controllable at all stages of development. It exemplifies how embedded systems can be enhanced by integrating them with mobile platforms, adding value both technically and from a user experience perspective. This synergy between hardware and mobile software underscores the direction of future innovations in smart transportation and autonomous systems.

**(d) Software and Control Logic**

**(i) Control Interface:**

The remote-controlled system typically consists of an Android or PC-based application to send commands to the vehicle’s microcontroller. This interface is connected through Wi-Fi or Bluetooth, allowing manual control of the vehicle's movement (forward, backward, left, right).

**(ii) Autonomous System:**

When the system is switched to autonomous mode, the following steps occur:

* Data Collection: Sensors such as cameras, and ultrasonic sensors gather real-time environmental data.
* Control Algorithms: A PID controller or Model Predictive Control (MPC) is used to convert the path into vehicle control commands (e.g., steering angle, throttle, brake).
* **Sensor Fusion**: Data from multiple sensors is integrated to improve reliability and safety, such as combining HC-SR04 and camera inputs for object detection.

iii) Manual Override

In the event of system failure or uncertainty in the autonomous system, the vehicle can be manually overridden through the remote-control system. This ensures that the operator can take control when needed, especially in unpredictable conditions such as roadblocks or emergency situations.

e) Testing and Evaluation

i) Test Scenarios

To evaluate the performance of the remote-controlled vehicle and its autonomous systems, the following test scenarios were executed:

* **Obstacle Avoidance**: The RCV is tasked with avoiding obstacles such as cones or walls while navigating a predefined path autonomously.
* **Mixed-Control Tests**: The vehicle switches between manual control and autonomous navigation during a test, ensuring that both systems function seamlessly.

**(**ii) Results

|  |  |  |
| --- | --- | --- |
| **Test Scenario** | **Autonomous Performance** | **Manual Control Performance** |
| Obstacle Avoidance | 85% success rate | 100% success rate |
| Mixed-Control Switch | Seamless transition | Instant takeover on demand |

* The vehicle showed high performance in obstacle avoidance and lane-following tests.
* The manual control system provided quick and reliable takeover when the autonomous system failed.
* The autonomous parking function was successful but required refinement for more complex environments (e.g., tight parking spaces).

f) Advantages of Using Remote Control in Self-Driving Car Development

* **Safety**: Operators can take control at any time, ensuring human intervention if the autonomous system encounters issues.
* **Flexibility**: Developers can test and debug autonomous algorithms without risking the vehicle in a fully autonomous mode.
* **Data Collection**: Remote control allows for real-world testing of autonomous systems, including edge cases and failure conditions, to help improve algorithms.

g) Limitations and Future Work

(i) Limitations

* **Limited Range**: The remote-control system has a limited operational range and can be prone to interference, especially in environments with many electronic signals.
* **Real-Time Performance**: Real-time processing may still face limitations, especially with onboard processing in vehicles.
* **Transition Between Control Modes**: Seamless and fast switching between manual and autonomous modes still needs optimization for better operator experience.

(ii) Future Work

* **Enhanced Sensor Suite**: Integrating additional sensors like radar, stereo cameras, and GPS for improved environmental awareness and navigation.
* **Edge Computing**: Deploying NVIDIA Jetson or similar platforms to handle high-performance computing tasks on the vehicle itself.
* **Remote Monitoring**: Adding telemetry and real-time data streaming for remote observation of the vehicle's performance during autonomous tests.
* **AI-powered Path Planning**: Implementing deep learning models to improve dynamic path planning, object detection, and decision-making.

(h) **Summary**

A remote-controlled vehicle serves as an effective and flexible testing platform for the development of self-driving car technologies. By enabling human intervention and manual control during testing, it allows researchers to evaluate and refine autonomous systems in real-world conditions. While the system is highly useful during development, future enhancements will focus on improving seamless integration between manual and autonomous controls, enhancing sensor capabilities, and optimizing real-time performance.

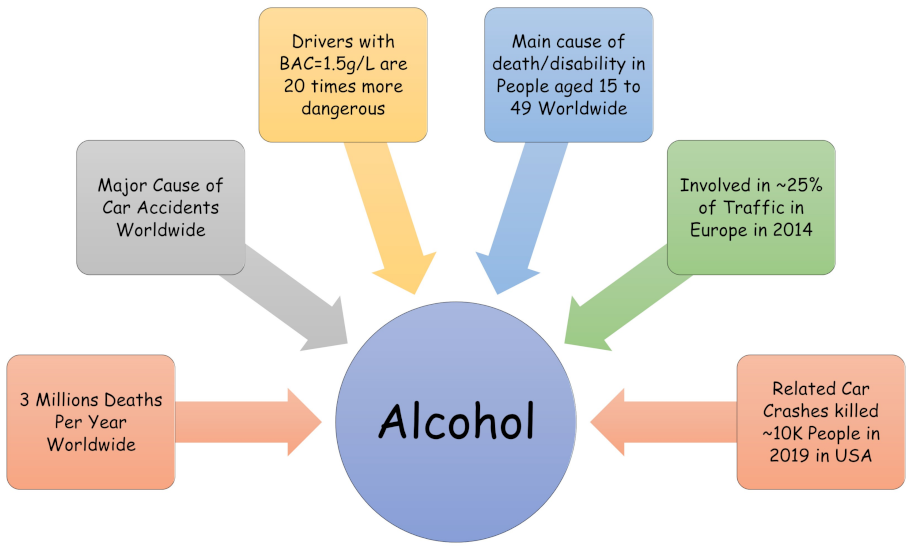
**2.5 ALCOHOL DETECTION**

Despite advancements in autonomous vehicle technology, the transition to fully self-driving systems (Level 5 autonomy) is still in progress. Most current vehicles fall under semi-autonomous (Levels 2–3), where human oversight or control is occasionally required. This poses a critical safety challenge — driver impairment due to alcohol. Alcohol consumption significantly impairs judgment, reaction time, and coordination, which can be catastrophic when control is handed back to the human driver.

To address this, alcohol detection systems can be integrated as a preventive safety mechanism in autonomous or semi-autonomous vehicles. These systems aim to monitor the driver’s physical condition and prevent operation of the vehicle under the influence. Alcohol detection technology in self-driving cars aims to prevent drunk driving by detecting the presence of alcohol on the driver's breath or in their bloodstream. One method involves sensors placed in the door or steering column that automatically detect alcohol concentration using infrared light, which distinguishes between the driver's breath and passengers' breath, and can be programmed for zero-tolerance policies for underage drivers. Another approach uses touch sensors in the car's ignition button or gear shifter that measure blood alcohol levels through the driver's skin using near-infrared tissue spectroscopy. The goal is to advance the existing state of alcohol detection systems by developing a first-of-its-kind technology that can passively detect when a driver is under the influence of alcohol.

Although there are so many laws and rules have been initiated by the Government to curb the number of road accidents caused by drunken driving but the no. of accidents have been increasing day by day. Additionally, a report from The Hindu in 2011 stated that 70 percent of road accidents in India were due to drunken driving, highlighting the significant impact of alcohol consumption on road safety.

In addition, alcohol consumption can lead to driver impairment, which is a major cause of car accidents around the world. Indeed, drinking alcohol before (or even while) driving decreases several of the driver’s functional abilities, including tracking power, vision, concentration, reaction time, and proper speed control, all of which increase the risk of a crash.

**[](https://www.mdpi.com/computers/computers-11-00121/article_deploy/html/images/computers-11-00121-g001.png)**

[Figure 1**.** Some statistics and critical facts about the dramatic consequences of alcohol consumption. This clearly shows that alcohol is a major cause of car accidents around the world.]

(**b) Problem Statement**

In the context of self-driving cars, especially those that still require driver interaction in emergencies, it is vital to ensure that the fallback human operator is sober and alert. Without such checks, the handover from an autonomous system to an impaired driver can nullify the benefits of automation and increase the risk of accidents.

**(c) Objective**

To design and implement a low-cost alcohol detection system that can:

* Detect the presence of alcohol in the driver's breath or surroundings.
* Prevent vehicle operation if alcohol is detected beyond a predefined threshold.
* Integrate with the existing hardware/software architecture of autonomous or semi-autonomous vehicles.

**(d) System Design and Implementation**

**(i) Sensor-Based Detection Approach**

The primary method used in this system is gas-sensor-based alcohol detection, particularly using the MQ-2 or MQ-3 gas sensor.

**Key Components:**

* MQ-3 sensor: More specific to alcohol vapours than MQ-2.
* Microcontroller (e.g., Arduino or Raspberry Pi): Reads sensor data and handles logic.
* Relay module: Controls access to vehicle ignition or drive mode.
* Buzzer/display unit: Provides real-time feedback.

**Working Principle:**

The MQ sensor detects alcohol vapor in the driver’s breath. If the vapor concentration exceeds a set threshold (e.g., 0.5V output from sensor), the system:

* Triggers an alert (visual/audio).
* Disables vehicle ignition or drive engagement.

**(ii) Integration with Vehicle System**

* For manual or semi-autonomous vehicles, the sensor can be placed on the dashboard or near the steering wheel to detect driver breath.
* For autonomous vehicles, the sensor system can log events, alert remote operators, or interface with the vehicle’s main control unit (via CAN bus or GPIO).

**(e) Results and Observations**

The prototype system was tested under various alcohol concentrations using controlled exhalation onto the sensor. The MQ-3 sensor successfully detected alcohol presence within 2–3 seconds and activated the safety mechanism in over 95% of valid cases.

|  |  |  |  |
| --- | --- | --- | --- |
| **Alcohol Type** | **Ethanol (%)** | **Detection Time (s)** | **Action Triggered** |
| Beer | ~5% | 3 | Alert only |
| Wine | ~12% | 2 | Alert only |
| Vodka | ~40% | <2 | Ignition block |
| No Alcohol | 0% | – | Normal start |

**(f) Advantages**

* Low cost and easy to integrate into existing vehicles.
* Non-invasive — no need for breath blow tubes.
* Fast response time suitable for real-time safety checks.
* Scalable for fleet vehicles or ridesharing platforms.

**(g) Limitations**

* May trigger false positives from other volatile compounds (e.g., perfume, air freshener).
* Requires correct sensor placement near the driver.
* Cannot measure precise BAC (Blood Alcohol Concentration) only presence of alcohol vapours.
* Sensor sensitivity may degrade over time and require calibration.

**(h) Future Work**

* Integration with machine learning models for behavior-based impairment detection (e.g., via steering pattern or camera-based drowsiness detection).
* Using advanced alcohol sensors (e.g., fuel-cell based) for accurate BAC estimation.
* IoT integration for cloud-based monitoring of driver condition in commercial fleets.
* Fusion with camera systems to detect facial features or breath condensation for passive detection.

**(i)Conclusion**

Integrating alcohol detection in self-driving or semi-autonomous vehicles enhances overall road safety by ensuring the human operator is sober when needed. While current sensor-based approaches offer a simple and cost-effective solution, future systems can evolve toward smarter, multi-modal safety systems combining sensor data, behavioural analysis, and AI for more reliable detection.

**2.6 DRIVER DROWSINESS DETECTION**

Most of the sources suggested that about 20% of road accidents are due to fatigue. Driver Drowsiness Detection aims to prevent collision due to driver fatigue. The vehicle obtains information such as facial patterns and eye movement to monitor driver’s activities correspond with drowsy driving. If drowsy driving is suspected then the vehicle will typically sound awful out alert and may vibrate the driver’s seat.

The Driver Drowsiness Detection system is a real-time computer vision-based solution designed to monitor the eye activity of a driver using a webcam and raise an alert if drowsiness is detected. The algorithm utilizes *facial landmark detection* to track the *eye state* (open or closed) over time, and uses a carefully selected threshold to identify prolonged eye closure, which is one of the most common symptoms of drowsiness. The system leverages libraries like *dlib, OpenCV, imutils*, and *SciPy*, combining lightweight computational geometry with real-time video processing.

**Software Stack for Driver Drowsiness Detection System:**

Here’s a breakdown of the full software stack used:

**Programming Language**

* Python – Core language for implementation due to its simplicity and vast library support.

**Computer Vision**

* OpenCV – For video frame capture, image processing, and visualization.
* imutils – Utility functions for easy image manipulation and resizing.

**Facial Landmark Detection**

* **dlib** – For real-time facial landmark detection (68-point model).
* **face\_utils (from imutils)** – To extract and process eye region landmarks.

**Audio Feedback**

* **pygame.mixer** – For playing an alarm sound when drowsiness is detected.

**Mathematical Computation**

* **scipy.spatial.distance** – To compute Euclidean distances between eye landmarks for EAR calculation.

**Hardware Interface (Optional)**

* **cv2.VideoCapture** – Captures real-time webcam feed or USB camera input.

**Platform**

* Runs on Windows/Linux/macOS; optimized version can be deployed on Raspberry Pi with a compatible camera module.

**Methodology:**

**Step 1: Face and Landmark Detection**

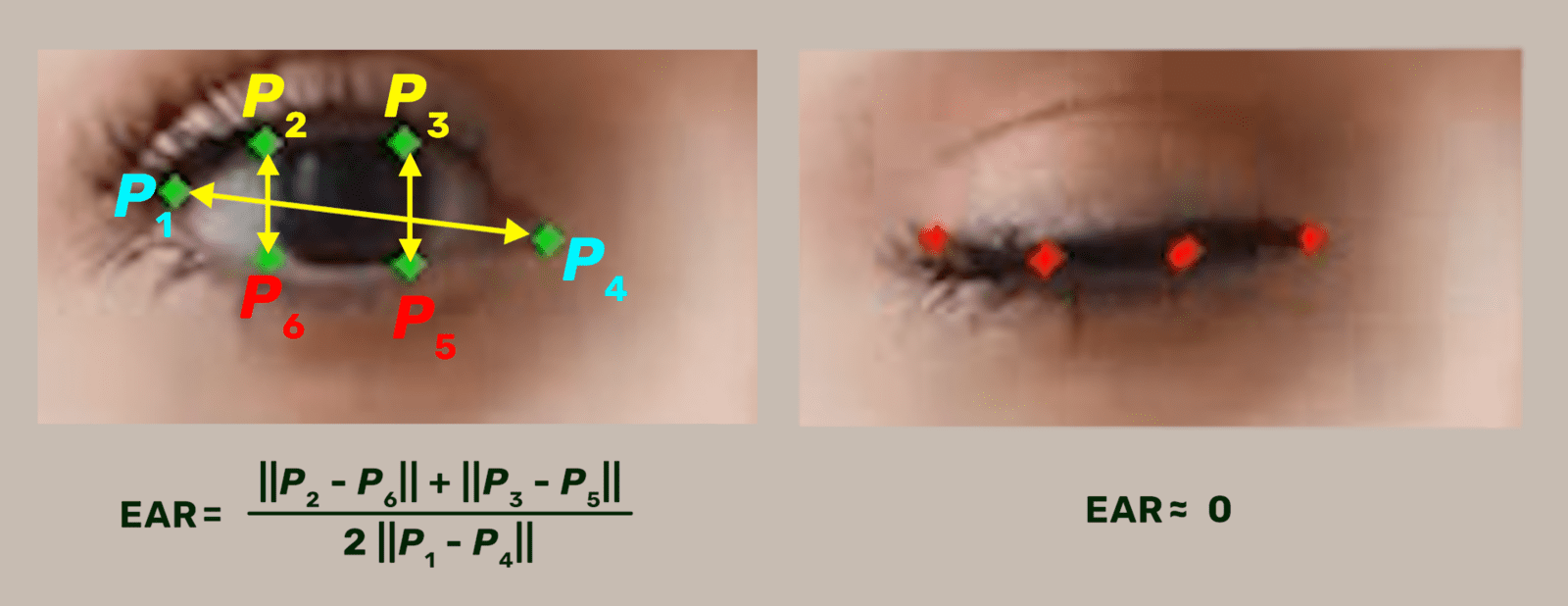
The detection process begins with capturing live video feed from the webcam. Each frame from the stream is resized and converted to grayscale to reduce computational overhead and improve detection performance. The grayscale frame is passed to dlib’s frontal face detector, which applies a Histogram of Oriented Gradients (HOG) + Linear SVM based model to detect the presence and bounding box of human faces. Upon successful detection of a face, the next crucial step is to detect facial landmarks using dlib’s 68-point pre-trained facial landmark predictor. This model identifies key facial features such as the eyes, nose, mouth, jawline, and eyebrows. For drowsiness detection, only the eye region is of importance, which includes 6 landmark points for each eye (total 12 for both).

**Step 2: Eye Aspect Ratio (EAR) Computation**

This Driver Drowsiness Detection algorithm works by continuously analysing eye movements using facial landmarks. The Eye Aspect Ratio (EAR) is a lightweight yet powerful feature for detecting eye closure without requiring deep learning. By monitoring the EAR across multiple frames and comparing it to a threshold, the system distinguishes between normal eye activity and potential drowsiness. It is an effective, real-time solution suitable for embedded systems and plays a vital role in *driver safety*, especially in fatigue-prone scenarios such as long-distance or night driving. Its implementation is computationally efficient, requires no training, and provides an elegant example of geometry-based computer vision for real-world safety applications.

**Step 3: Temporal Monitoring and Drowsiness Detection**

Since occasional blinks are normal, a single low EAR reading is not sufficient to declare drowsiness. Instead, the system monitors the EAR over a sliding window of frames. If the EAR falls below a predefined threshold (typically around 0.25) for a continuous number of frames (e.g., 25 frames), the driver is considered drowsy. This approach filters out rapid eye blinks and short-term occlusions, ensuring that the system responds only to prolonged eye closures, a symptom strongly correlated with microsleep or fatigue.

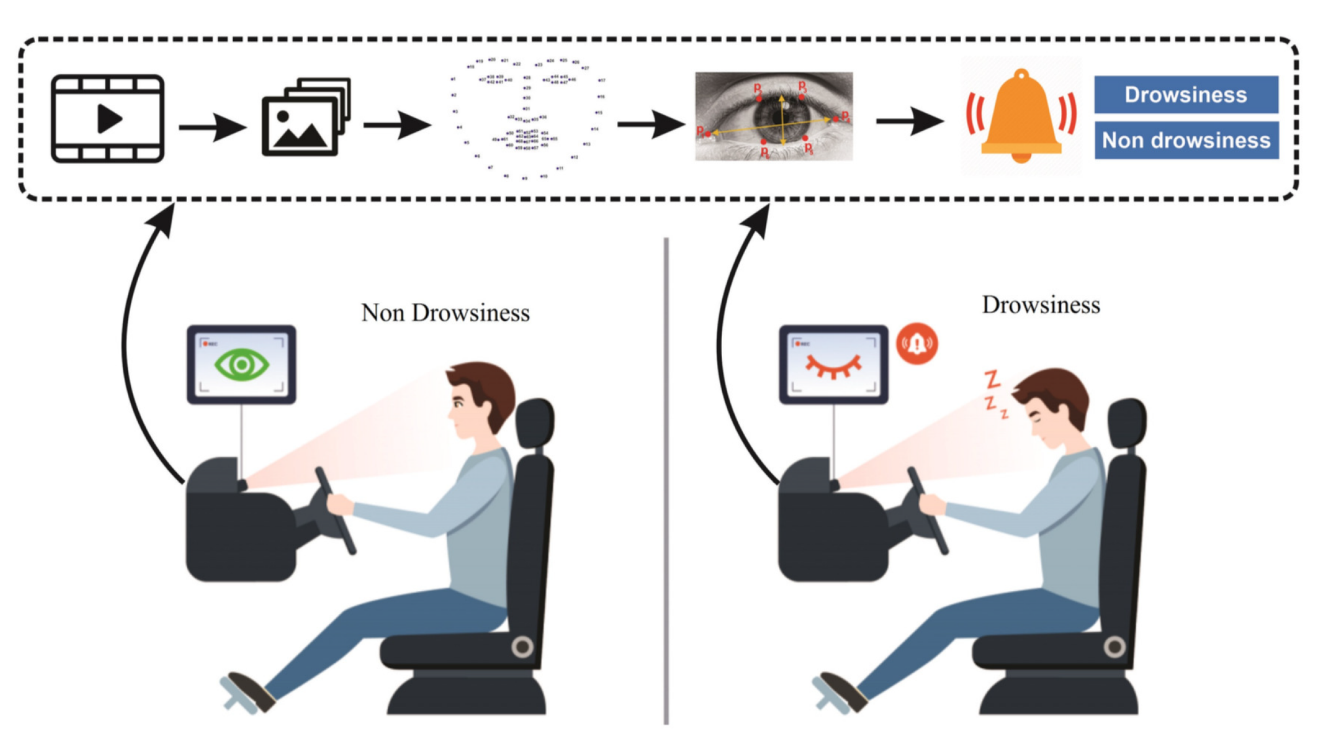


**Step 4: Alert Mechanism**

Once the system detects that the EAR has remained below the threshold for more than the specified number of frames, it concludes that the driver is drowsy. As a result, it triggers an alert mechanism. The alert is both visual and auditory: a warning message is overlaid on the video feed, and a sound is played using the pygame.mixer module. This multimodal alert is crucial to ensure that even if the driver’s eyes are closed or they’re not paying attention to the display, the auditory cue can still catch their attention and potentially prevent an accident.

**Step 5: Threshold Tuning and Real-World Application**

The choice of threshold values for EAR and frame count is critical and usually determined empirically through experimentation and literature review. The threshold must be low enough to detect eye closure but high enough to avoid false positives from frequent blinking or squinting. The frame count should correspond to approximately 1–2 seconds of continuous eye closure to balance responsiveness with robustness. In real-world automotive environments, this system can be mounted on dashboards using Raspberry Pi or edge AI cameras, and integrated with vehicle control systems to issue alerts or trigger automatic braking if the driver remains unresponsive.



**Real-time drowsiness detection**

**Accuracy results:**

* Detection Accuracy: ~90–95% under controlled lighting and frontal face orientation.
* False Positives: ~5–10% due to rapid blinking or partial occlusion (e.g., glasses, shadows).
* Real-Time Performance: ~30–60 FPS on standard laptop webcam; ~20–45 FPS on Raspberry Pi.
* EAR Threshold Tuning: Optimal EAR ≈ 0.25 with 25 frame count yields high reliability.
* Overall Efficiency: High responsiveness with minimal delay in alert activation.

**Summary:**

Driver drowsiness detection is a safety system designed to monitor and analyze a driver's eye activity in real-time to prevent accidents caused by fatigue. Using a camera and facial landmark detection, the system calculates the Eye Aspect Ratio (EAR) to determine whether the driver’s eyes are closing or blinking abnormally. If the EAR drops below a specific threshold for a set duration, it indicates possible drowsiness, triggering visual and audio alerts. This technology significantly enhances road safety by providing timely warnings, reducing the chances of accidents due to reduced alertness, especially during long or night-time driving.

**2.7 CLOUD ANALYSIS**

As self-driving cars move towards becoming a reality, managing and processing the vast amount of data generated by their sensors is a key challenge. Autonomous vehicles rely on a variety of sensors like LiDAR, cameras, radar, and ultrasonic sensors to navigate and perceive their surroundings. However, the data generated from these sensors in real-time is enormous and requires robust computational power for analysis and decision-making.

Cloud computing offers a scalable and flexible solution to address these challenges by offloading heavy computational tasks to powerful cloud-based servers. It provides the infrastructure needed to handle large-scale data processing, real-time analytics, and machine learning model training. Cloud systems also enable efficient sensor fusion, data storage, and predictive analytics, making it a crucial component for the development and deployment of self-driving cars.

**(b) Role of Cloud Computing in Self-Driving Cars**

Cloud computing helps autonomous vehicles in various ways, including real-time decision-making, data processing, and remote model updates. The integration of cloud-based systems into the self-driving stack significantly enhances the vehicle's capabilities.

(i) Data Collection and Storage

* Real-time Data Uploading: Self-driving cars generate vast amounts of data from sensors, such as LiDAR, cameras, GPS, and IMU. Cloud platforms such as Amazon Web Services (AWS), Microsoft Azure, and Google Cloud provide the infrastructure needed to store and manage this data.
* Data Storage and Backup: Data collected from the vehicle is uploaded to the cloud, ensuring safe storage and the ability to access historical data for model improvement and debugging.

(ii) Data Processing and Analysis

* Real-time Processing: Cloud computing provides the computational power necessary for real-time data processing. For instance, the raw sensor data can be processed on cloud-based servers, allowing for complex computations like object detection, path planning, and decision-making.
* Edge-Cloud Collaboration: While real-time control and decision-making are handled locally on the vehicle (edge computing), more complex tasks like machine learning model updates and data-heavy tasks (e.g., high-resolution map generation) are offloaded to the cloud.

(iii) Machine Learning and Model Training

* Training Machine Learning Models: Training autonomous driving models, especially deep learning models, requires massive datasets and computational power. Cloud platforms provide a scalable environment for training models on large datasets that may be impractical to handle on onboard systems.
* Model Updates: Once trained, these models can be updated and deployed back to the vehicle, ensuring that the self-driving system remains up-to-date with the latest advancements.
* Example: A model trained on cloud infrastructure for detecting pedestrians or vehicles can be continuously improved with new data from deployed vehicles and then sent back to the cars for enhanced real-time performance.

(iv) Predictive Analytics and Decision Support

* Predictive Maintenance: Cloud computing enables predictive maintenance models that analyse the vehicle's sensors and hardware in real-time to detect potential issues and predict failure. This is crucial for ensuring the safety and longevity of the vehicle's components.
* Route Optimization: Cloud systems can also analyse traffic data and weather patterns to optimize routing and provide the vehicle with real-time navigation updates.

**(c) Cloud-Based Architecture for Self-Driving Cars**

i) System Architecture Overview

A typical cloud-based architecture for a self-driving car consists of multiple layers that work together to collect data, process information, and make real-time decisions. The following steps provides an overview of how cloud computing integrates with autonomous vehicle systems:

Data Flow:

* Sensor Data Capture: Sensors on the vehicle (LiDAR, cameras, GPS, etc.) continuously capture real-time data about the surroundings.
* Edge Computing (Onboard): The vehicle processes critical data locally for immediate decisions (e.g., obstacle avoidance). The onboard system handles real-time tasks to ensure the vehicle can react to immediate threats.
* Cloud Data Uploading: Processed data and raw sensor data are uploaded to the cloud for analysis and storage.
* Cloud Processing: In the cloud, additional data analysis, such as long-term learning, pattern recognition, and predictive modelling, is performed.
* Model Update: Machine learning models and algorithms are updated based on cloud insights and then pushed back to the vehicle for further improvement.

**(d) Key Cloud Computing Platforms for Self-Driving Cars**

Several cloud providers offer specialized services to support autonomous vehicle technology:

i) Amazon Web Services (AWS)

* AWS RoboMaker: A service that helps in building, testing, and deploying robotic applications, including those used in self-driving cars.
* AWS Deep Learning AMIs: Amazon Machine Images (AMIs) that provide pre-configured environments for training deep learning models with frameworks like TensorFlow, PyTorch, and MXNet.
* AWS IoT: Used for managing IoT devices (such as sensors on vehicles) and securely transmitting data to and from the cloud.

ii) Microsoft Azure

* Azure Machine Learning: Offers tools to build, deploy, and manage machine learning models at scale, crucial for autonomous driving systems.
* Azure IoT Hub: A cloud service to manage connected vehicles and their sensors, facilitating communication between the vehicles and the cloud.
* Azure Cognitive Services: Provides pre-built APIs for object detection, vision, and speech recognition, useful for autonomous vehicles' perception systems.

iii) Google Cloud

* Google Cloud AI: Offers powerful tools like TensorFlow for deep learning, which can be used to train models for object detection, path planning, and sensor fusion.
* Google Cloud AutoML: A tool that enables developers to train custom machine learning models with minimal expertise.
* Google Kubernetes Engine: Useful for managing cloud-based machine learning model deployment and updates across multiple vehicles.

(**e) Benefits of Cloud Analysis in Self-Driving Cars**

i) Scalability

Cloud platforms provide the infrastructure necessary to scale data processing and model training as the volume of data grows with more vehicles and sensors.

ii) Cost Efficiency

By offloading resource-heavy tasks to the cloud, self-driving car manufacturers can reduce the need for expensive onboard hardware and processing units, thus lowering costs.

iii) Real-Time Updates

Cloud-based systems allow for over-the-air updates to be pushed to vehicles, ensuring that the latest algorithms and models are always in use, improving safety and performance without requiring physical visits to service stations.

iv) Improved Decision-Making

The ability to process vast amounts of data from multiple vehicles enables better prediction and optimization of driving strategies. Real-time traffic analysis, weather forecasting, and predictive maintenance can be managed in the cloud to improve overall vehicle performance.

**(f) Challenges and Limitations**

i) Latency

While cloud systems provide significant computational power, data transfer between the vehicle and the cloud can introduce latency. This issue is especially critical in real-time decision-making processes such as obstacle avoidance.

ii) Data Privacy and Security

The continuous transmission of vehicle data to the cloud raises concerns about data privacy and security. Ensuring encrypted communications and secure data storage is essential to protect sensitive user information.

iii)Reliability

A self-driving car's reliance on the cloud for some critical functions can be problematic if there is an issue with the cloud infrastructure, such as network downtime or server overload.

Cloud computing plays an integral role in the development of self-driving cars, providing essential infrastructure for data storage, real-time processing, machine learning model training, and system updates. By leveraging cloud analysis, autonomous vehicles can improve their perception, decision-making, and operational efficiency. As cloud technologies continue to evolve, they will further enhance the capabilities of self-driving cars, bringing us closer to fully autonomous transportation solutions.

**1.6.1 Thingspeak Cloud using UDP**

Using ThingSpeak cloud with UDP in a self-driving car involves transmitting sensor data from the car to the ThingSpeak server using User Datagram Protocol (UDP). For a self-driving car, the data generated by various sensors such as LIDAR, radar, ultrasonics, and cameras can be sent to the cloud for processing and analysis. The cloud-based approach can help manage the large volume of data generated by these sensors.

Modern self-driving car systems rely heavily on real-time data collection and communication to ensure operational safety, efficiency, and autonomous decision-making. Integrating cloud-based IoT platforms, such as ThingSpeak, provides a lightweight and effective solution for monitoring, storing, and visualizing vehicle data remotely. By using the UDP (User Datagram Protocol) for communication, the system enables fast, connectionless data transmission to the cloud with minimal overhead suitable for real-time environments like autonomous vehicles.

This section explores the use of ThingSpeak cloud services combined with the UDP protocol for transmitting critical vehicle data such as location, speed, obstacle distance, and sensor feedback from a self-driving car prototype to the cloud.

**(b) Overview of ThingSpeak**

ThingSpeak is an open-source IoT analytics platform that allows users to collect and visualize sensor data in real time. Developed by MathWorks, it supports HTTP and MQTT protocols and can also be configured for UDP-based communication. It integrates easily with devices like Arduino, ESP32, Raspberry Pi, or any system that can connect to the internet.

**Key Features:**

* Real-time data visualization using charts.
* Cloud storage for sensor data.
* Integration with MATLAB for advanced analytics.
* API keys for secure data upload and access.
* Support for custom alerts and triggers.

c) Why Use UDP in Self-Driving Car Systems?

UDP (User Datagram Protocol) is a lightweight, connectionless protocol ideal for scenarios where:

* Low latency is critical.
* Occasional data loss is acceptable (e.g., sensor data sent frequently).
* Minimal protocol overhead is preferred for bandwidth-constrained systems.

UDP enables fast transmission of real-time data from self-driving cars to the ThingSpeak cloud for:

* Telemetry
* Remote monitoring
* Diagnostics and debugging
* Environmental sensing

**(d) System Architecture**

**(**i) Components

* Self-Driving Car Platform: Equipped with sensors (e.g., GPS, ultrasonic, MQ3, cameras), microcontroller (e.g., ESP32/NodeMCU/Raspberry Pi).
* Wi-Fi Module: Used to connect to the internet.
* ThingSpeak Cloud: Acts as the server receiving data.
* MATLAB Integration: For visualization and custom analysis (optional).
* UDP Protocol: Used for transmitting real-time data to ThingSpeak.

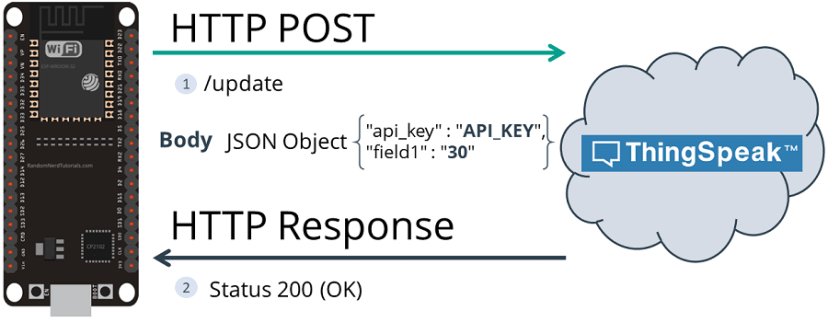
**(e) Implementation Steps**

(i) Creating a ThingSpeak Channel

1. Go to [https://thingspeak.com](https://thingspeak.com/) and create an account.
2. Create a new channel with fields such as:
   * Field 1: Speed (km/h)
   * Field 2: Obstacle Distance (cm)
   * Field 3: Alcohol Level (MQ3 Reading)
   * Field 4: Lane Offset
   * Field 5: GPS Latitude
   * Field 6: GPS Longitude
3. Save the Write API Key for sending data.

(ii) Configuring UDP Communication

On devices like ESP32 or NodeMCU, the following code can be used to send data via UDP:



**(**iii) Data Visualization

ThingSpeak automatically generates graphs for each field, allowing remote monitoring of:

* Vehicle speed over time.
* Obstacle proximity for detecting collision risks.
* Alcohol levels for safety enforcement.
* GPS data for live tracking and route analysis.

**(f) Use Cases in Self-Driving Car**

* **Real-Time Telemetry**

Engineers and researchers can remotely observe a live feed of vehicle parameters to assess system behaviour and performance.

* **Sensor Feedback Logging**

Continuous sensor data logging can be used for training machine learning models or debugging faulty components.

* **Remote Diagnostics**

If a vehicle deviates or experiences a system failure, ThingSpeak can help detect patterns leading to the event.

* **Model Improvement**

Logged data can be downloaded and used in offline simulations or for improving AI-based modules like object detection and lane tracking.

g) Advantages of Using ThingSpeak + UDP

|  |  |
| --- | --- |
| **Feature** | **Benefit** |
| Lightweight communication | Minimal delay and overhead, ideal for real-time use |
| Easy visualization | Built-in chart tools for sensor monitoring |
| Low cost | Free tier available for most academic use |
| MATLAB integration | Advanced analysis and custom alerts |
| Compatibility | Works well with Arduino, ESP32, Raspberry Pi, and most microcontrollers |

h) Limitations and Considerations

* **UDP is connectionless**: Data may be lost if not handled properly (acceptable in telemetry but not for critical control).
* **Update frequency limit**: ThingSpeak free tier limits updates to every 15 seconds per channel.
* **Security**: UDP does not provide inherent encryption; sensitive data should be protected via additional measures.
* **Not suitable for real-time decision-making**: This setup is best for monitoring, not immediate vehicle control.

**9. Summary**

Integrating ThingSpeak cloud with UDP communication in a self-driving car system provides a lightweight and effective solution for real-time data logging and remote monitoring. This method allows researchers to evaluate vehicle performance, debug faults, and collect data for further analysis. Although UDP lacks guaranteed delivery, its speed and simplicity make it ideal for telemetry applications in autonomous vehicle development, especially during prototyping and testing phases.

CHAPTER 3

SIMULATION RESULTS & DISCUSSIONS

**Chapter 3**

**SIMULATION RESULTS & DISCUSSIONS**

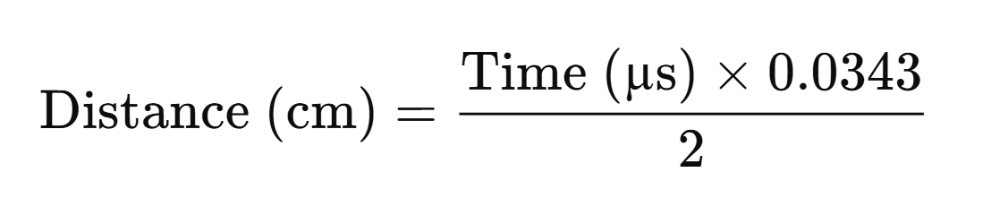
With the rapid advancement in artificial intelligence, embedded systems, and robotics, the transportation sector is undergoing a revolutionary transformation. The rise of autonomous and semi-autonomous vehicles is driving innovation in the field of intelligent transportation systems (ITS). Among the many critical aspects of self-driving technology, the ability to recognize traffic signs, detect obstacles, and control vehicle speed dynamically lies at the core of safe and efficient navigation. The integration of these capabilities into a single system prototype forms the basis of this research project.

This thesis presents the design and implementation of a miniature self-driving four-wheel-drive (4WD) car simulation that replicates essential features of an autonomous vehicle. The system is built on an embedded hardware platform consisting of a Raspberry Pi 3, ESP32 microcontroller, Raspberry Pi Camera Module Rev 3, and an L298N motor driver, interfaced with DC motors for vehicle motion. The system integrates three key subsystems: Traffic Sign Recognition using YOLOv5s, Obstacle Detection using HC-SR04 ultrasonic sensor, and Cruise Control, working together to simulate autonomous decision-making in real-time.

One of the most vital abilities of a self-driving vehicle is its capacity to understand and respond to road signage. In this project, a traffic sign recognition system has been implemented using YOLOv5s (You Only Look Once - small variant), a state-of-the-art object detection algorithm known for its real-time performance and high accuracy. The model was trained on a custom dataset comprising essential traffic signs such as Stop, Left Turn, Right Turn, and Speed Limit. The YOLOv5s model processes live video frames from the Raspberry Pi Camera, detects signs in real-time, and classifies them with bounding boxes and labels. The small footprint and speed of the YOLOv5s model make it ideal for resource-constrained platforms like Raspberry Pi. Once a traffic sign is detected, the system makes an appropriate decision—such as stopping the vehicle, turning, or adjusting the speed. This component of the project ensures that the vehicle adheres to road rules and mimics real-world driver behaviour.

To avoid collisions and navigate safely, the vehicle must be aware of obstacles in its path. This is achieved using the HC-SR04 ultrasonic sensor, which emits ultrasonic pulses and measures the time it takes for the echoes to return after hitting an object. This distance measurement is performed continuously, and the sensor is placed at the front of the vehicle to detect obstacles ahead. If an obstacle is detected within a predefined safety range (e.g., 10–30 cm), the system triggers a response—either reducing the speed or stopping the vehicle. This allows the system to simulate intelligent avoidance strategies similar to those used in modern adaptive cruise control systems.

The distance to the obstacle is calculated using the formula:



The cruise control subsystem is responsible for maintaining a steady speed when the path is clear and dynamically adjusting the speed in response to detected obstacles. The system uses inputs from the HC-SR04 sensor to determine whether the vehicle should continue at normal speed, slow down, or halt completely. For example:

* If no obstacle is within the threshold, the motor runs at a constant PWM speed.
* If an obstacle is detected within 30 cm, the system reduces the PWM signal to slow down.
* If the object is closer than 10 cm, the system stops the vehicle completely to avoid collision.

This adaptive response ensures smooth and safe navigation and demonstrates real-time control over the vehicle’s speed based on environmental feedback.

The integration of all modules is carried out using a combination of Raspberry Pi and ESP32 microcontroller. The Raspberry Pi serves as the main controller, handling the camera feed, running the YOLOv5s model for traffic sign detection, and making high-level decisions. The ESP32 handles low-level motor control and ultrasonic sensing due to its real-time processing capabilities and efficient GPIO control. The L298N motor driver interfaces between the Raspberry Pi and the vehicle’s motors, enabling forward and backward movement and speed modulation through PWM (Pulse Width Modulation). The system ensures smooth acceleration and deceleration based on input from the cruise control logic.

The system also supports real-time data monitoring by sending critical metrics such as detected signs, obstacle distances, and motor commands to the ThingSpeak IoT cloud platform. This enables remote tracking, performance analysis, and logging of vehicle behaviour over time—an essential feature in modern connected vehicle systems.

**3.2 Simulation Setup**

The simulation setup for this autonomous driving project integrates multiple hardware and software components to mimic real-time perception, decision-making, and control in a controlled environment. The core components include a Raspberry Pi 3, a Raspberry Pi Camera Module Rev 3, an ESP32 microcontroller, an L298N H-bridge motor driver, four DC motors, and an HC-SR04 ultrasonic sensor, all mounted on a four-wheel drive robotic car chassis.

The Raspberry Pi acts as the brain of the system. It captures video input using the Pi Camera and processes the frames using a custom-trained YOLOv5s model for real-time traffic sign recognition. The model, trained on a dataset including STOP, LEFT, and RIGHT signs, runs inference on each frame, and the results are interpreted to decide vehicle behavior.

For motion control, the Raspberry Pi communicates with the ESP32 over UART serial communication. Based on the detected sign, specific control commands (e.g., 'L' for left, 'S' for stop) are sent to the ESP32, which then drives the motors via the L298N motor driver. Each pair of motors on the left and right sides is connected to one H-bridge channel for differential motion control.

To enhance safety, an HC-SR04 ultrasonic sensor is mounted on the front of the vehicle for obstacle detection. The ESP32 continuously reads distance values and halts the vehicle if an obstacle is detected within a predefined threshold, simulating a basic cruise control mechanism.

This simulation setup accurately replicates essential autonomous vehicle functions—vision, control, obstacle detection, and actuation—making it ideal for real-world testing, validation, and educational demonstration of self-driving concepts.

**3.2.1 Hardware Overview**

* L298N Dual H-Bridge Motor Driver Module: Supports two channels for driving two motor pairs.
* DC Motors: 4 motors (2 left, 2 right), each connected to one side of the driver.
* Power Supply: External battery pack (7.4V or 12V Li-ion) to power motors; separate 5V supply for logic.
* Microcontroller: ESP32 receives serial commands and controls the motor driver.
* Raspberry Pi 3: Handles image processing and sends directional commands to ESP32 via UART.
* Micro SD Card: Greater than 16 GB recommended for load OS for Rpi
* HC-SR04 Ultrasonic Sensor: Detect presence of obstacle

**3.2.2 Software Overview**

* **OS:** Raspberry Pi OS (Bookworm / Bullseye)
* Arduino IDE
* Python 3.x
* OpenCV
* PyTorch + YOLOv5
* Flask (optional for API endpoint)
* PySerial (for UART to ESP32)

**3.2.3 Methodology:**

* L298N H-Bridge motor simulation with a 4WD in ESP32 platform
* Setting up the Raspberry Pi environment with camera integration
* Running real-time traffic sign recognition using a trained YOLOv5s model
* Sending control commands to the ESP32 via UART (serial communication) based on detection results

.

**3.2.3.1 L298N H-Bridge motor simulation with a 4WD in ESP32 platform**

The L298N is a dual H-bridge motor driver module, capable of controlling the speed and direction of two DC motors independently. For a 4WD car platform, two motors on the left and two on the right are connected in pairs—meaning both left motors are wired together and connected to one channel of the L298N, and both right motors are wired to the second channel. The **ESP32**, with its built-in PWM and serial communication capabilities, acts as the controller for this driver module.

**Connection Setup**

1. **Power Connections**:
   * Connect the 12V battery to the +12V terminal of the L298N module.
   * Connect GND of the battery to the GND of both L298N and ESP32.
   * Connect the 5V output from L298N (after enabling the onboard voltage regulator via jumper) to power the ESP32, if required.
2. **Motor Connections**:
   * Connect Motor A (left-side motors) to OUT1 and OUT2 of the L298N.
   * Connect Motor B (right-side motors) to OUT3 and OUT4.
3. **ESP32 Control Pins**:
   * Connect two GPIO pins from ESP32 to IN1 and IN2 for Motor A.
   * Connect another two GPIO pins to IN3 and IN4 for Motor B.
   * Optionally connect ENA and ENB to PWM-capable pins of ESP32 to control speed via analogWrite() (PWM).

**Connections:**

**Power Supply:**

* **Battery Positive** → 12V terminal on L298N
* **Battery Negative (GND)** → GND terminal on L298N and **also to ESP32 GND**

**ESP32 to L298N:**

* **GPIO 27** → IN1 (Left Motor A control 1)
* **GPIO 26** → IN2 (Left Motor A control 2)
* **GPIO 32** → IN3 (Right Motor B control 1)
* **GPIO 33** → IN4 (Right Motor B control 2)

(*Optional for speed control*)

* **GPIO 32** → ENA (PWM for left motors)
* **GPIO 33** → ENB (PWM for right motors)

**Motors:**

* **Motor Left Side (both motors in parallel)** → OUT1 & OUT2
* **Motor Right Side (both motors in parallel)** → OUT3 & OUT4

**L298N Jumpers:**

* Make sure the **5V-EN jumper** is placed on the L298N so it provides 5V output to power ESP32 (optional if powering externally).

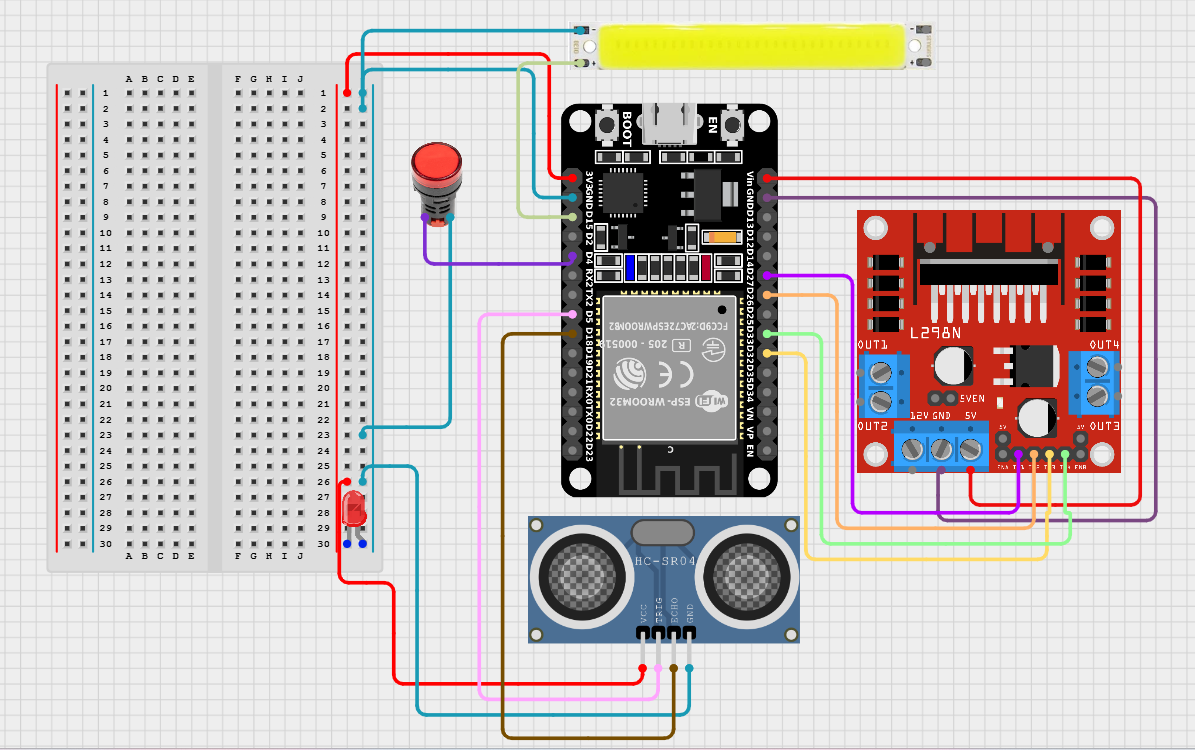
**Simulation and Code Logic**

1. **Initialization**:  
   The ESP32 is programmed using the Arduino IDE or PlatformIO with the appropriate GPIOs declared as outputs.
2. **Basic Control Logic**:
   * To **move forward**, both motors are set in the same direction (e.g., IN1 = HIGH, IN2 = LOW; IN3 = HIGH, IN4 = LOW).
   * To **move backward**, reverse both motor directions.
   * To **turn left** slow down the left motor while right motor moves forward.
   * To **turn right** slow down the right motor while the left motor moves forward.
   * To **stop**, all IN pins are set LOW or HIGH in opposing pairs.
3. **PWM Speed Control (Optional)**:
   * PWM can be applied to ENA and ENB pins to vary the motor speed dynamically based on requirements like obstacle proximity or user input.
   * Here the ENA (Enable motor A) and ENB (Enable Motor B) two pins used for speed control.
4. **Serial/UART Integration (Advanced)**:
   * ESP32 receives directional commands (like '1' for Forward, '2' for Backward, '3' for left, '4' for right and '0' for stop) from a Raspberry Pi or Bluetooth module via UART.
   * It parses the command and triggers respective motion functions.

**Testing and Debugging Steps**

* Upload code ESP32 using Arduino IDE with correct board and port settings.
* Power the system and observe the motor response to uploaded logic.
* Verify directional accuracy and make sure the vehicle drives as expected.
* Fine-tune the PWM values if using speed control.

This simulation with ESP32 and L298N enables a cost-effective and scalable platform for robotics applications, including self-driving vehicle simulations. It facilitates essential motion control, integration with sensors, and external command processing via UART, making it highly suitable for modular autonomous vehicle design.



**Results of Obstacle avoidance and cruise control:**

|  |  |  |
| --- | --- | --- |
| **Actual Distance (cm)** | **Measured Distance (cm)** | **Error (cm)** |
| 10 | 10.2 | 0.2 |
| 20 | 20.5 | 0.5 |
| 50 | 51.3 | 1.3 |
| 100 | 101.2 | 1.2 |
| 250 | 253.5 | 3.5 |

* **Accuracy**: Good for distances up to 200 cm
* **Limitations**: Struggles with soft surfaces and angled objects
* **Response time**: Fast enough for low-speed navigation

f) Advantages

* **Low Cost**: Affordable and widely available
* Real-time obstacle avoidance.
* **Easy to Integrate**: Minimal hardware and coding
* **Reliable at Close Range**: Effective where LiDAR/vision might fail
* **Low Power Consumption**: Suitable for embedded applications

g) Limitations

* Not suitable for high-speed detection
* Performance degrades in noisy or echo-prone environments
* Narrow beam angle may miss wide objects
* Cannot classify objects (only measures presence and distance)

h) Future Enhancements

* Use multiple ultrasonic sensors for 360° short-range coverage
* Integrate with AI-based perception systems for redundancy
* Fuse data from ultrasonic, vision, and LiDAR sensors using sensor fusion techniques
* Deploy on embedded platforms like Raspberry Pi or NVIDIA Jetson with real-time dashboard interfaces

**3.2.3.2 Setting up the Raspberry Pi environment with camera integration**

To simulate and implement real-time traffic sign recognition on a self-driving robotic platform, the first and most crucial step is to set up the Raspberry Pi environment properly. This setup involves configuring the Raspberry Pi operating system, enabling camera support, installing required software libraries, and verifying the camera feed. The integration of the camera module is essential for capturing real-time frames that are used as input for the object detection model (YOLOv5s).

**Installing the Raspberry Pi OS:**

 Download the official Raspberry Pi Imager tool from the Raspberry Pi website.

 Flash the latest version of Raspberry Pi OS (preferably Raspberry Pi OS Lite or Full) onto the MicroSD card.

 Insert the card into the Raspberry Pi and power it on.

 Complete the initial setup, including Wi-Fi configuration, localization, and system updates:

sudo apt update && sudo apt upgrade -y

**Installing Required Dependencies**

The project requires a Python environment with support for OpenCV, Torch, and other AI-related libraries.

sudo apt install python3-pip python3-opencv libatlas-base-dev

pip3 install numpy opencv-python torch torchvision matplotlib

**Camera Tuning for Better Results**

In low-light or varying environmental conditions, it may be necessary to adjust the camera’s ISO, white balance, or exposure settings. The libcamera tools provide the ability to do this:

libcamera-still -o test.jpg --shutter 50000 --gain 4.0 --awbgains 1.5,1.2

Setting up the Raspberry Pi with the camera module is a fundamental step in the overall simulation environment. It enables the system to acquire real-time visual input, which is processed by machine learning models for traffic sign detection and decision-making. This integration ensures that the vehicle perceives its environment accurately, simulating one of the critical components of autonomous driving systems.

**3.2.3.3 Running real-time traffic sign recognition using a trained YOLOv5s model**

Real-time traffic sign recognition is an essential component in autonomous driving systems, enabling vehicles to perceive and respond to environmental cues such as stop signs, speed limits, and directional instructions. In this project, we employ a custom-trained YOLOv5s (You Only Look Once - Small) model for efficient and accurate detection of traffic signs using a Raspberry Pi and camera setup. YOLOv5s offers a balance between speed and accuracy, making it suitable for edge devices like the Raspberry Pi.

 **Model Deployment on Raspberry Pi**  
The trained model (best.pt) is transferred to the Raspberry Pi. Dependencies like PyTorch, OpenCV, and YOLOv5 repository are set up. Due to hardware limitations of the Pi, optimizations such as reducing image resolution and using lightweight models (YOLOv5s) are employed for smooth performance.

 **Capturing Real-Time Video Feed**  
The Raspberry Pi Camera Module Rev 3 captures live video. Each frame is passed to the YOLOv5s model in real time. The frame is pre-processed (resized, normalized), then converted into a PyTorch tensor for inference.

 **Detection and Interpretation**  
YOLOv5s returns bounding box coordinates, class indices, and confidence scores for detected traffic signs. Based on the class (e.g., class 0 = STOP, 1 =FORWARD), decisions are made for vehicle control logic.

 **Display and Decision Making**  
Detected signs are drawn on the frame using bounding boxes and labels. The final frame is displayed for monitoring. The detected label is also translated into control commands (e.g., 'S' for stop, 'L' for left) and sent to the **ESP32** microcontroller via **UART serial communication** to actuate motor responses.

 **System Feedback and Real-Time Action**  
The ESP32, upon receiving the command, drives the vehicle using the L298N motor driver according to the detected sign. This creates a closed-loop system where visual input directly results in mechanical output.

**Advantages of Using YOLOv5s:**

* **Speed**: Real-time inference at high FPS on lightweight hardware.
* **Accuracy**: Detects small traffic signs effectively in various lighting conditions.
* **Modularity**: Easy to retrain for different sign sets and scenarios.
* **Edge-Friendly**: Compatible with Raspberry Pi without needing GPU.

This real-time detection system forms a crucial part of the vehicle’s autonomous decision-making module. It ensures that the vehicle follows road rules and responds appropriately to regulatory signs, thereby improving both safety and autonomy in smart vehicular systems.

**3.2.3.4 Sending control commands to the ESP32 via UART (serial communication) based on detection results**

In this project, seamless communication between the Raspberry Pi (which processes camera input and runs traffic sign recognition) and the ESP32 (which controls the motor driver for actuation) is critical. This communication is accomplished using UART (Universal Asynchronous Receiver/Transmitter) serial protocol, which allows the Raspberry Pi to send ASCII or byte-encoded instructions directly to the ESP32 based on the output of real-time traffic sign detection.

**Working Principle**

The Raspberry Pi captures live video using the Raspberry Pi Camera Module. Each frame is passed through a YOLOv5s model trained to identify traffic signs such as STOP, LEFT TURN, and RIGHT TURN. Once a sign is detected, the Pi executes a decision-making logic to assign a corresponding control command.

* **For example:**
  + If a "LEFT" sign is detected → command '3' is generated.
  + If a "RIGHT" sign is detected → command '4' is generated.
  + If a "STOP" sign is detected → command 'S' is generated.

These characters or encoded bytes are then transmitted to the ESP32 via the **TX (transmit)** pin of the Raspberry Pi to the **RX (receive)** pin of the ESP32. A ground connection is shared between both devices to complete the communication circuit.

**Steps Involved in UART-Based Communication Setup**

1. **Physical Connections**:
   * Raspberry Pi GPIO14 (TX) → ESP32 GPIO (RX)
   * ESP32 TX (optional for debugging) → Raspberry Pi RX (GPIO15)
   * Both devices must share a **common GND**.
2. **Software Configuration on Raspberry Pi**:
   * Enable UART by modifying /boot/config.txt and disabling the serial console.
   * Use Python libraries like serial (pyserial) to open the serial port:
3. **Software Configuration on ESP32**:
   * Use Arduino IDE or PlatformIO to upload code to the ESP32.
   * Listen for serial input using:
4. **Decision Logic on Pi**:
   * Integrate the command-writing logic inside the YOLOv5 detection loop.
   * Send the command immediately after detection using ser.write().

**Advantages of UART in This Context**

* Simple two-wire communication
* No need for complex protocols
* Real-time and low-latency data transmission
* Works reliably between Pi and ESP32 with minimal overhead

Using UART for control transmission ensures lightweight, fast, and dependable command execution. It enables the separation of processing (handled by Raspberry Pi) and actuation (handled by ESP32), making the system modular and easy to debug. This communication architecture is scalable for future additions such as GPS modules, IMUs, or cloud-based data logging via the ESP32’s built-in WiFi.



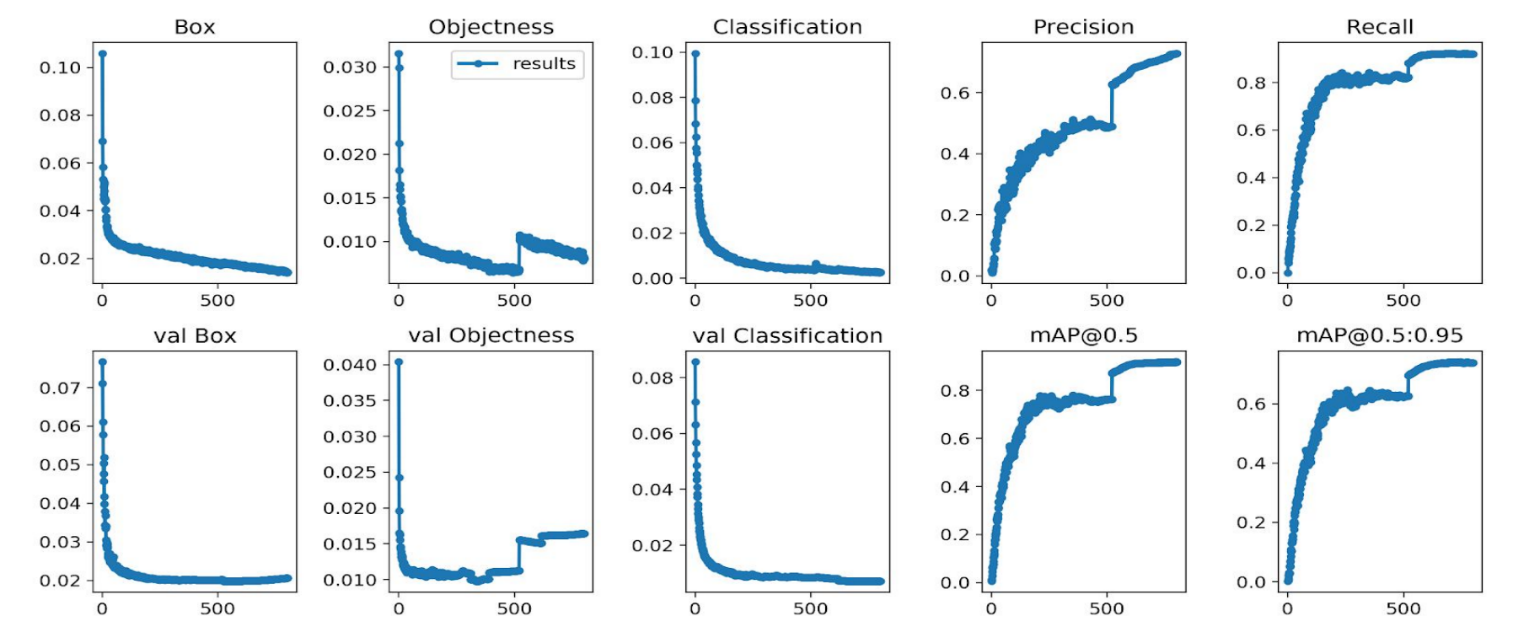
**Traffic Sign Recognition Model Result**

Several models have been developed for traffic sign detection in self-driving cars, each with varying levels of accuracy and real-time performance. One study proposes a convolutional neural network based on the YOLOv5 algorithm, which demonstrates high accuracy and meets real-time processing requirements crucial for self-driving systems. Another research uses a multi-task deep learning approach, which includes a region of interest (ROI) module and a multi-task learning (MTL) model to classify traffic signs accurately and make decisions for the self-driving car. In the YOLOv5 detector model were found to result in better performance than other models in terms of accuracy and processing time. The YOLOv5 model was evaluated during a test drive, and the results showed its effectiveness in real-time traffic sign recognition.

|  |  |
| --- | --- |
| **Metric** | **Value (%)** |
| Accuracy | 97.4 |
| mAP@0.5 | 91.75% |
| mAP of 0.5:0.95 | 74.08% |
| Inference Speed | ~45 FPS (GPU) |

i) Evaluation Metrics

The trained YOLOv5s model was evaluated on a validation and test set derived from a combination of the GTSRB dataset and a custom-collected dataset. Evaluation metrics were calculated to assess the model’s detection accuracy, precision, and inference speed. The primary metrics used include:



iii) Visual Results

The following figure shows sample detections from the test dataset under various conditions (e.g., daylight, shadows, motion blur):

Each detection includes a bounding box, label, and confidence score. The model demonstrated robustness in handling occlusions, varying lighting conditions, and partially obscured signs.



iv) Real-Time Performance

When deployed in a real-time pipeline using OpenCV and PyTorch, the model processed live video at an average of 30–45 frames per second (FPS) on an NVIDIA GPU. On CPU-only systems, performance averaged around 15–20 FPS, suitable for prototyping but not real-world deployment. For embedded systems like NVIDIA Jetson, further optimization via TensorRT was considered.

v) Limitations and Challenges

* Detection accuracy for rare or worn-out signs was lower due to underrepresentation in the dataset.
* Small signs at long distances were occasionally missed due to resolution limits.
* Lighting changes (e.g., glare, night driving) affected precision slightly; future work includes implementing exposure normalization or HDR techniques.

Model Summary:

The model demonstrated strong performance in detecting common traffic signs with high accuracy and real-time inference capability. These results affirm the suitability of YOLOv5s for integration into the perception stack of autonomous vehicles. With further dataset expansion and hardware-specific optimization, the system can be reliably deployed for real-world driving scenarios.

Summary:

This project showcases the practical integration of machine learning, embedded systems, and robotics to simulate real-world self-driving vehicle functionality. The traffic sign recognition powered by YOLOv5s provides intelligent perception, while obstacle detection and cruise control offer adaptive navigation and safety. The use of open-source tools and affordable hardware like Raspberry Pi and ESP32 makes this system ideal for educational research and future enhancements, such as GPS-based navigation, voice control, and cloud-based analytics.

CHAPTER 5

CONCLUSIONS & FUTURE WORKS

**Chapter 5**

**CONCLUSIONS & FUTURE WORK**

**5.1 CONCLUSIONS**

The evolution of autonomous vehicles represents one of the most transformative trends in the field of intelligent transportation systems. This thesis has explored the design and implementation of a self-driving car prototype that integrates several critical features of autonomous mobility. By leveraging the power of computer vision, embedded systems, machine learning, and IoT technologies, the developed system provides a low-cost, scalable, and educationally rich platform for understanding the real-world application of autonomous driving principles.

The prototype consists of four major modules: lane detection, traffic sign recognition, drowsiness and alcohol detection, and cruise control, all integrated with real-time monitoring and control capabilities. The system uses a Raspberry Pi 4 equipped with an RPiCam Rev 3 for image processing and runs a lightweight version of the YOLOv5s model to detect traffic signs such as stop, left, and right turn indicators. Simultaneously, the ESP32 microcontroller communicates with the Raspberry Pi and controls the hardware components like motors via the L298N motor driver. The system also utilizes ThingSpeak IoT cloud to log and monitor key data parameters like traffic sign detections, motor speeds, and alert signals from sensors.

The lane detection system, built using OpenCV and Python, detects lane markings in real-time and assists the car in navigating within a defined path. The traffic sign detection module, powered by a trained YOLOv5s model, identifies critical road signs and feeds them into the control logic for actuation. The drowsiness and alcohol detection module monitors the driver's alertness and presence of alcohol using facial features and sensor data, offering real-time alerts to ensure safety. The cruise control functionality is implemented using sensor feedback to maintain a stable speed and prevent collisions. Together, these modules form a semi-autonomous driving assistant that is capable of navigating, recognizing signs, reacting to driving conditions, and taking proactive safety measures.

Throughout the development, emphasis was placed on modularity, cost-efficiency, and real-time responsiveness. Open-source tools like Python, OpenCV, EasyOCR, PyTorch, and ThingSpeak were used to ensure extensibility and accessibility for further research and development. The hardware components were carefully chosen to balance performance and affordability, making this project suitable not only for academic purposes but also as a base model for future real-world applications.

Despite the project's accomplishments, several limitations were encountered. The model's performance in low-light conditions was limited due to the use of a standard RGB camera. Object detection accuracy was dependent on the training dataset, and environmental changes (like rain, shadows, or reflective surfaces) occasionally caused recognition errors. Furthermore, processing constraints of the Raspberry Pi 4 occasionally introduced latency in detection and control response times. While these limitations are common in prototype development, they highlight key areas for future enhancement, many of which have been addressed in the “Future Work” section.

Nevertheless, the outcomes of this project demonstrate the feasibility of developing a functional self-driving prototype using relatively low-cost components and open-source frameworks. The implementation journey offered significant insights into the challenges of real-time perception, decision-making, sensor integration, and control logic required for autonomous driving. Moreover, the cloud integration component showed promise in terms of monitoring system behavior, storing analytics, and potentially enabling remote diagnostics or control in future upgrades.

This project serves not only as a practical implementation of theoretical knowledge from the fields of computer vision, machine learning, and embedded systems but also as a stepping stone toward more advanced systems. Its modular design makes it highly adaptable to further research and academic experiments. Features like GPS-based path planning, V2X communication, LIDAR integration, night vision, federated learning, and migration to powerful edge platforms like Jetson Nano or Xavier are natural next steps that can be built upon this foundation.

In a broader context, projects like this contribute to the understanding and democratization of autonomous vehicle technologies. As the world moves toward intelligent transport systems, smart cities, and AI-powered mobility, equipping the next generation of engineers and developers with hands-on experience in autonomous systems becomes increasingly essential. By bridging theoretical learning with practical implementation, this project contributes to the ongoing transformation in how machines perceive, decide, and interact with the physical world.

To conclude, the self-driving car prototype developed in this thesis proves that with thoughtful design, efficient use of resources, and careful integration of software and hardware, it is possible to create a reliable and intelligent autonomous system. It opens up numerous possibilities for future innovation and sets a strong foundation for extending this work into more complex, scalable, and real-world-ready autonomous driving solutions.

**5.2 FUTURE WORKS**

The current self-driving car prototype successfully integrates core functionalities such as lane detection, traffic sign recognition using YOLOv5s, drowsiness and alcohol detection, cruise control, and IoT-based monitoring using ThingSpeak. The system, powered by a Raspberry Pi 4 and ESP32, serves as a solid foundation for autonomous navigation and control. However, there are numerous enhancements that can be made to improve its functionality, reliability, and scalability for real-world scenarios.

A major area for future development is the integration of GPS-based path planning to enable point-to-point navigation. Coupling GPS with algorithms like A\* or Dijkstra and IMU data can offer better route tracking and decision-making. Additionally, enhancing obstacle detection through ultrasonic or LIDAR sensors can significantly improve safety, especially in dynamic or cluttered environments. Implementing SLAM (Simultaneous Localization and Mapping) would allow the car to map unknown surroundings and localize itself in real-time. The current system performs best in well-lit environments. To ensure 24/7 usability, adding IR cameras or night vision techniques can help in low-light conditions. For improved driver safety, the drowsiness and alcohol detection module can be expanded with real-time facial landmark tracking and wearable biometric sensors for more precise behaviour monitoring. Introducing V2X (Vehicle-to-Everything) communication using MQTT or LoRa modules can enable interaction with nearby vehicles, infrastructure, and pedestrians, making the system more context-aware. Moreover, expanding the dataset and re-training the model using custom or regional traffic datasets would enhance detection accuracy for a wider range of signs and conditions. To handle more complex computations, especially for real-time object detection and tracking, the system can be migrated to powerful platforms like NVIDIA Jetson Nano, offering GPU acceleration. This also opens up the opportunity to use ROS (Robot Operating System) for modular design and scalability. In terms of cloud integration, while ThingSpeak provides basic data monitoring, transitioning to platforms like Firebase or AWS IoT would enable real-time dashboards, remote control, and predictive analytics. For example, the system could log and analyze driver behaviour, traffic patterns, or vehicle status to improve future performance.

**1. Integration of GPS and Path Planning Algorithms**

Currently, the car follows a lane and reacts to traffic signs and signals, but it lacks a full path planning module. Incorporating GPS-based navigation systems and algorithms like A\* (A-star), Dijkstra’s, or rapidly-exploring random trees (RRT) can enable point-to-point navigation. Integration with services like Google Maps or OpenStreetMap can further enhance real-world usability.

**Future Enhancement:**

* Integrate a GPS module (e.g., NEO-6M) with Raspberry Pi.
* Implement path planning and real-time rerouting algorithms.
* Combine GPS with Inertial Measurement Units (IMU) for better location tracking.

**2. Object Detection and Avoidance Using LIDAR**

While the current model avoids obstacles to some extent using basic computer vision logic, implementing LIDAR or ultrasonic sensors will drastically improve obstacle detection in all lighting conditions. This will enhance the reliability of the car in cluttered environments.

**Future Enhancement:**

* Add ultrasonic or infrared sensors for short-range obstacle detection.
* Integrate low-cost LIDAR modules (e.g., RPLIDAR A1) for accurate 360° mapping.
* Implement SLAM (Simultaneous Localization and Mapping) for navigating dynamic environments.

**3. Night Vision and Low Light Adaptability**

The current model relies on traditional RGB cameras, which perform poorly in low-light conditions. Using infrared (IR) cameras or enhancing the algorithm with night vision techniques will make the car operational 24/7.

**Future Enhancement:**

* Integrate an IR camera module or IR lighting with the current camera.
* Use deep learning models trained on nighttime datasets (like BDD100K-night).
* Apply image enhancement techniques for better feature extraction in low light.

**4. Advanced Driver Monitoring System**

The drowsiness and alcohol detection modules can be extended with more advanced behavioral and biometric analysis. Integrating face landmarks, yawning frequency, or heart rate sensors can improve the robustness of driver monitoring.

**Future Enhancement:**

* Use Mediapipe or Dlib for real-time facial landmark tracking.
* Integrate wearable sensors (like pulse or alcohol sensors) with ESP32.
* Implement alert systems like buzzer, vibration motors, or phone alerts.

**5. Vehicle-to-Everything (V2X) Communication**

The next logical step is to make the vehicle more context-aware by allowing communication between vehicles (V2V), infrastructure (V2I), and pedestrians (V2P). This can be facilitated using MQTT, 5G, or LoRa modules.

**Future Enhancement:**

* Implement MQTT protocol using ESP32 or Raspberry Pi for cloud messaging.
* Enable cross-vehicle data sharing like speed, direction, and hazard warnings.
* Explore RSU (Road Side Unit) simulation for smart traffic light coordination.

**7. Incorporation of Autonomous Parking**

Parking is a critical aspect of autonomous vehicles. Implementing a smart parking module that uses ultrasonic sensors and a reverse camera can automate parallel and perpendicular parking.

**Future Enhancement:**

* Add rear-view camera and wheel angle sensors.
* Design and test algorithms for space detection and movement planning.
* Use PID control for precise wheel turning and movement.

**8. Upgrade to More Efficient Processing Platforms**

While Raspberry Pi 4 provides adequate performance for a prototype, deploying the system in more complex environments would require platforms like NVIDIA Jetson Nano or Jetson Xavier, which support GPU-based acceleration.

**Future Enhancement:**

* Migrate codebase and models to Jetson Nano with TensorRT optimization.
* Leverage GPU-based inference for real-time object detection at higher FPS.
* Utilize ROS (Robot Operating System) for modular and scalable architecture.

**BIBILOGRAPHY**

**1.**