Addis Ababa Science and Technology University
Department of Electrical and Computer Engineering

Final Year Project Report

Centralized Ultrasonic-Based Traffic Management System for Speed Limit Violation Detection and Automated Penalty Enforcement

---

Submitted By:

Hiluf Abay	ETS1598/13
Abdi Fekadu	ETS0002/13
Bernabas Gebrie	ETS0225/13
Abdiwak Melaku	ETS0006/13
Abdulaziz Awol	ETS0008/13

Supervisor:

Mr. Aregawi

Submitted in Partial Fulfillment of Requirements for

Bachelor of Science in Electrical and Computer Engineering

May 2025

Declaration

We, the undersigned, declare that this final year project report titled "Centralized Ultrasonic-Based Traffic Management System for Speed Limit Violation Detection and Automated Penalty Enforcement" is our original work and has not been submitted elsewhere for academic credit.

Hiluf Abay	ETS1598/13
Abdi Fekadu	ETS0002/13
Bernabas Gebrie	ETS0225/13
Abdiwak Melaku	ETS0006/13
Abdulaziz Awol	ETS0008/13

Supervisor: Mr.Aregawi

Date: May, 2025

Acknowledgements

First and foremost, we would like to thank Almighty God for granting us the strength, patience, and opportunity to reach this day.

We are deeply grateful to our families for their unwavering love, sacrifices, and belief in us throughout this journey. Their constant support has been the foundation of our perseverance and success.

Our sincere appreciation goes to our supervisor, Mr. Aregawi, for his continuous support, invaluable guidance, and encouragement throughout this project. His insights and feedback have been instrumental in shaping our work.

We also extend our heartfelt thanks to Addis Ababa Science and Technology University for providing us with the necessary resources, facilities, and a conducive learning environment.

The Authors

Abstract

This project addresses the growing need for intelligent and automated systems to manage traffic violations, particularly speed limit breaches. It aims to leverage low-cost embedded technologies to detect speed violations, recognize license plate numbers, manage offender data, and automate the fine notification process. The primary objective is to automate speed violation detection and contribute to behavior change among drivers by consistently recording violations and applying penalties.The system uses an ultrasonic sensor for vehicle speed measurement, with an Arduino Uno handling signal processing and speed calibration. A Raspberry Pi, equipped with a camera module, captures images of speeding vehicles. License Plate Recognition (LPR) is achieved using OpenCV and EasyOCR, while SQLite is used for local data storage. Offender notifications are handled through an automated email system using SMTP. Features of the system include speed detection, reliable license plate recognition, and a basic fine system based on the severity of the violation. While the current implementation serves as an academic proof-of-concept, its real-world deployment could significantly enhance traffic law enforcement. It has the potential to reduce accidents caused by overspeeding, minimize corruption by eliminating the need for human traffic officers, and encourage responsible driving by maintaining a record of each driver's violations.

Contents

List of Figures

List of Figures

Chapter 1

Introduction

Traffic management has become a critical concern in urban areas, particularly in rapidly growing cities like Addis Ababa, Ethiopia. Increasing vehicular traffic, combined with inadequate enforcement of speed limits, poses significant risks to pedestrian safety and public health. Conventional methods of traffic management rely heavily on manual oversight and static measures, often failing to address the dynamic and complex nature of urban traffic violations effectively. Recognizing these challenges, the need for advanced, technology-driven solutions has become evident. The development of a Centralized Ultrasonic-Based Traffic Management System aims to enhance how traffic violations are monitored and penalized. By leveraging radar technology and centralized control, this system addresses limitations in current practices while offering innovative solutions, such as automated detection of speed limit violations, dynamic fine calculations, and real-time enforcement. This approach not only enhances the accuracy of traffic law enforcement but also contributes to improved road safety, reduced accident rates, and better compliance with traffic regulations.

1.1 Background of the Study

Traffic safety is a pressing issue in Addis Ababa, where pedestrian fatalities account for approximately 80% of road traffic deaths [1]. Current traffic management methods rely on limited infrastructure, inadequate enforcement, and minimal use of advanced technologies, resulting in significant road safety challenges. Advanced radar-based detection systems, while effective in developed countries, are prohibitively expensive and often incompatible with the financial and infrastructural realities of cities like Addis Ababa.

In Ethiopia, there are no existing radar-based traffic enforcement systems in operation. As a result, traffic law enforcement is predominantly manual, leading to inefficiencies in

detecting and addressing violations such as speeding. To fill this gap, a cost-effective, radar-based system tailored to local conditions is essential.

The proposed system seeks to address these challenges by offering an affordable yet efficient solution that combines radar technology with a centralized control system. Unlike high-cost alternatives, this system is designed to be accessible and practical for deployment in Ethiopia, focusing on real-time speed detection, automated fine calculation, and long-term scalability. By prioritizing affordability and leveraging locally available resources, the project aims to revolutionize traffic enforcement while addressing the specific needs of Addis Ababa's road safety landscape.

1.2 Statement of the Problem

Addis Ababa, the capital city of Ethiopia, faces a severe traffic safety crisis primarily driven by high pedestrian fatalities, the impact of speeding, and inadequate traffic management. With a rapidly increasing population and expanding vehicular traffic, road accidents have become a pressing issue significantly affecting public safety. Pedestrians constitute approximately 80% of all road traffic fatalities in the city [1]. Speeding contributes significantly to accident severity, increasing the risk of injury by up to 3% and serious injury or death by 5% with each 1% increase in average speed [2]. Despite various initiatives to manage traffic, like the 2017 speed management program that reduced average speeds in high-risk areas, more than 90% of pedestrian deaths still occur at locations without dedicated walkways [3].

In the fiscal year ending July 2021, Ethiopia recorded 15,034 road accidents, leading to 4,161 deaths [4]. Many of these accidents can be attributed to poor traffic police management, including insufficient enforcement, lack of proper training, and underreporting of incidents. Research highlights that traffic police in Addis Ababa often lack the necessary qualifications to manage accidents effectively, resulting in significant discrepancies between police-reported traffic deaths and hospital records—153 deaths reported by police compared to 84 recorded in hospitals [5]. Additionally, drivers' bad behavior, such as violent conduct and disregard for traffic regulations, remains rampant due to inadequate law enforcement.

The lack of effective traffic police management, combined with poor enforcement and insufficient data-driven approaches, exacerbates the risk of accidents in Addis Ababa. To address this, a centralized ultrasonic-based traffic management system could play a pivotal role in reducing speeding violations, automating penalty issuance, and ultimately improving road safety for both pedestrians and drivers.

1.3 Objectives of the Study

1.3.1 General Objective

To reduce traffic accidents caused by irresponsible drivers speeding in high-risk areas, ensuring safer roads through centralized control, automated enforcement, and dynamic fine systems.

1.3.2 Specific Objectives

- Develop a centralized ultrasonic-based traffic management system.
  - Integrate the technology speed detection, and violation tracking.
- Implement a fine system based on speed violations.
- Enhance the system with automated penalty issuance.
  - Automate the process of issuing penalties (warnings, fines, bans) for violations.
- Implement real-time traffic violation monitoring and reporting.
  - Enable continuous monitoring and reporting of traffic violations.

1.4 Scope of the Study

The primary focus is on developing an ultrasonic-based system for detecting speeding vehicles and capturing license plate images, with enhancements aimed to streamline data management, and utomating enforcement actions.

1. Hardware Development

- Construction of an ultrasonic circuit capable of accurately detecting vehicle speeds.
- Integration of a high-resolution camera to capture images of speeding vehicles.

2. Software Development

- Microcontroller programming for real-time speed processing and camera triggering.
- Implementation of speed detection algorithms under varying traffic conditions.
- License Plate Recognition using OpenCV and EasyOCR with preprocessing techniques.
- Integration of a centralized control system using SQLite for tracking and actions.

- Email notification via SMTP Simple Mail Transfer Protocol to relevant authorities and systems.

3. System Integration

- Combining Arduino, camera, and software modules into a unified system.
- Ensuring reliable communication among hardware components and control system.

4. Centralized Control System

- The system logs violations over time.
- Drivers accumulating a threshold number of violations are flagged for a ban.
- All communication and decision-making are automated for transparency and scalability.

Chapter 2

Literature Review

2.1 Theoretical Review of the Literature

Vehicle speed detection is an essential component of intelligent transportation systems (ITS), contributing to improved traffic management, road safety, and enforcement of legal speed limits [6]. Several methods have been proposed and implemented for vehicle speed estimation, with ultrasonic sensors and image processing standing out due to their affordability and practical feasibility.

Ultrasonic Sensor-Based Vehicle Speed Detection

Ultrasonic sensors function by emitting high-frequency sound waves and measuring the time it takes for the echo to return after reflecting off a nearby object. This time-of-flight (ToF) measurement is used to compute distance. By capturing distance at two distinct time points, the velocity of a moving object can be estimated [7].

The HC-SR04 ultrasonic sensor is a popular choice in academic and prototype projects because of its low cost, ease of use, and compatibility with microcontrollers like the Arduino. Various implementations have used a pair of ultrasonic sensors placed at a fixed distance to determine vehicle speed based on the time a vehicle takes to pass between them [8]. Alternatively, a single sensor aligned with the motion direction can also infer speed by measuring how quickly the object approaches or recedes [9].

Despite their practicality, ultrasonic sensors have several limitations:

- Limited detection range and narrow beam width
- Sensitivity to environmental factors such as temperature, wind, and noise
- Inaccurate measurements for small, irregular, or angled objects

Nevertheless, for short-range, low-speed environments, ultrasonic sensors are widely regarded as viable solutions [10].

Image Processing-Based Vehicle Speed Detection

Image processing techniques calculate speed by analyzing vehicle motion across consecutive video frames. This approach involves capturing images, segmenting moving vehicles, and tracking them over time. Given the frame rate and a known real-world reference distance, speed can be computed based on the displacement of the vehicle between frames [11].

This method offers distinct advantages, including visual confirmation of events, license plate recognition, and the possibility of integrating AI-based object detection and classification. However, it also introduces challenges such as:

- Sensitivity to ambient lighting and adverse weather
- Requirements for high-resolution cameras and fixed installations
- Greater processing demands for real-time operation

Despite these challenges, image-based vehicle speed detection systems remain prevalent, particularly when hardware such as the Raspberry Pi is used to balance computational efficiency and portability.

License Plate Recognition (LPR)

License Plate Recognition (LPR) is a specialized form of Optical Character Recognition (OCR) used to identify vehicles based on their license plates from captured images or video frames. It plays a vital role in intelligent transportation systems (ITS) by enabling automated enforcement, vehicle tracking, toll collection, and parking management [12].

A typical LPR pipeline consists of four main stages:

1. Image Acquisition: Capturing images or video frames using a camera, typically mounted at an overhead or roadside location.
2. License Plate Detection: Locating the region of interest (ROI) where the license plate exists in the image.
3. Character Segmentation: Isolating individual characters from the plate.
4. Character Recognition: Applying OCR or deep learning methods to interpret segmented characters.

Traditional vs Deep Learning-Based Approaches Traditional LPR systems relied heavily on rule-based methods and handcrafted features such as edge detection, morphological operations, and projection profiles for character segmentation and recognition [13]. While effective under controlled lighting and simple backgrounds, these methods struggled with blurred, tilted, or occluded plates.

Recent advances in deep learning have significantly improved LPR robustness and accuracy. Convolutional Neural Networks (CNNs) and object detection models like YOLO (You Only Look Once) and SSD (Single Shot Detector) have been applied to detect license plates in challenging conditions [14]. Once the plate is detected, sequence models such as Recurrent Neural Networks (RNNs) or CRNNs (Convolutional Recurrent Neural Networks) are used to recognize characters without explicit segmentation [15].

Lightweight Models for Embedded Systems For resource-constrained platforms such as the Raspberry Pi, lightweight deep learning models like YOLOv4-tiny and MobileNet-SSD offer real-time performance while maintaining acceptable accuracy [16]. Pretrained models such as ‘openalpr‘ and ‘EasyOCR‘ are widely adopted due to their ease of integration and high success rates in common scenarios [17].

Challenges in LPR Despite progress, several challenges persist in LPR:

- Environmental Sensitivity: Variations in lighting, shadows, weather, and motion blur can significantly affect detection and recognition accuracy.
- Plate Diversity: Differences in plate formats, fonts, languages, and mounting positions add complexity to general-purpose models.
- Occlusions and Angles: Obstructed or angled views of plates reduce detection accuracy, particularly for traditional models.

By combining LPR with speed violation detection in a modular system, violations can be documented with vehicle identification data, supporting automated enforcement and centralized logging.

2.2 Empirical Review

Recent research has demonstrated the effectiveness of various sensor-based and image-based techniques for vehicle monitoring systems. In particular, ultrasonic sensors have gained traction for short-range, low-power speed detection. For instance, [8] implemented a dual-ultrasonic sensor arrangement and demonstrated reliable speed estimation in controlled environments. Likewise, [9] used a network of ultrasonic

transceivers to improve accuracy and classify vehicle types through waveform analysis and reflection timing.

Despite their low cost and energy efficiency, ultrasonic systems face limitations in noisy outdoor environments, especially under conditions involving angled incidence, wind interference, or large vehicle surfaces causing signal dispersion. However, their performance in specific detection zones makes them suitable for triggering events like speed violations.

On the other hand, image-based systems—particularly those leveraging machine learning and optical character recognition (OCR)—are widely used for vehicle identification through License Plate Recognition (LPR). Systems like OpenALPR and EasyOCR have been deployed for detecting and reading license plates under varied lighting and traffic conditions [12], [18]. These methods often involve multiple stages such as image preprocessing, localization, character segmentation, and recognition.

Studies have shown that LPR systems are sensitive to factors such as motion blur, lighting variations, and low-resolution imagery [19]. Nevertheless, modern approaches employing convolutional neural networks (CNNs) have improved robustness and character recognition accuracy, even in low-quality frames [20].

Recent works have also explored combining detection and recognition in resource-constrained environments. For instance, lightweight LPR algorithms have been adapted for Raspberry Pi platforms to enable edge computing and real-time traffic monitoring [21]. These systems benefit from triggering mechanisms to minimize power consumption and unnecessary computation, a principle that aligns well with ultrasonic sensor activation.

2.3 Research Gaps

While both ultrasonic and image processing technologies offer valuable tools for vehicle monitoring, each exhibits limitations when used independently. Ultrasonic sensors, though affordable and power-efficient, provide limited contextual information and are prone to errors due to ambient noise or improper alignment. They are primarily effective at measuring distance or speed in constrained scenarios and offer no means of identifying specific vehicles.

Conversely, image-based systems, especially those used for License Plate Recognition (LPR), are computationally intensive and require high-quality images with consistent lighting. When used continuously, these systems can overwhelm low-power edge devices like the Raspberry Pi. Furthermore, in traffic scenarios with multiple vehicles or occluded views, image-based tracking and recognition become challenging without

precise event triggers.

The gap lies in creating a low-cost, hybrid system that balances sensing accuracy, computational efficiency, and reliability. This research proposes a fused approach where an ultrasonic sensor monitors vehicle speed in real time and activates image capture only upon speed threshold violations. The image is then processed using a lightweight LPR algorithm on the Raspberry Pi to extract license plate details and assign penalties. This conditional activation strategy reduces computational overhead, enhances energy efficiency, and optimizes resource usage while maintaining real-time response capability.

By leveraging the strengths of both ultrasonic sensing and image-based recognition, this hybrid architecture addresses individual weaknesses and provides a practical, scalable solution for smart traffic enforcement systems.

Chapter 3

Methodology

This chapter outlines the methodological framework employed in the design, development, and evaluation of the *Centralized Ultrasonic Sensor-Based Traffic Management System for Speed Limit Violation Detection and Automated Penalty Enforcement*. The primary objective of this section is to detail the structured and systematic approach used to transform conceptual goals into a functional prototype that accurately detects speed violations and autonomously enforces penalties.

The methodology integrates multiple engineering domains—including ultrasonic signal acquisition, embedded system integration, digital signal processing, and network communication protocols—into a cohesive workflow. Instead of immediately presenting technical implementations, this chapter begins with a high-level overview of the system architecture and proceeds to explain the processing pipeline used to estimate vehicle speed based on time-of-flight (ToF) measurements from ultrasonic sensors.

Subsequent sections elaborate on hardware selection (including Arduino and Raspberry Pi boards), software implementation for real-time speed calculation and license plate recognition, system calibration, and testing procedures. The system uses a threshold-based activation mechanism: when the ultrasonic sensor detects a speed violation, the Raspberry Pi triggers image processing and performs license plate recognition.

3.1 Overview of Methodological Approach

The methodological approach adopted in this study follows a modular and sequential framework, where each subsystem contributes to the overall functionality of the centralized ultrasonic-based traffic management system. The process begins with vehicle detection and speed estimation using a single ultrasonic sensor. The system measures the time-of-flight (ToF) of reflected ultrasonic pulses to estimate the distance

of a moving vehicle at two successive timestamps. By calculating the change in distance over time, the vehicle's speed is computed in real time.

In parallel, a camera module continuously captures images of approaching vehicles. However, image processing procedures—including license plate recognition—are only initiated if the computed speed exceeds a predefined threshold. This conditional processing reduces computational load on the Raspberry Pi and conserves system resources.

Once the vehicle's license plate number is extracted, relevant metadata—including speed, timestamp, and license plate number—is appended into a locally hosted SQLite database. This database maintains a historical log of all violations and links repeated offenses to the same vehicle. Upon confirmation of a violation, the system automatically sends an email notification containing the violation details and associated penalty.

This structured methodology ensures a streamlined and resource-efficient integration between sensing, decision-making, and enforcement, making the system scalable, cost-effective, and practical for real-world traffic monitoring applications.

3.2 Limitations and Assumptions

The methodology assumes a direct line-of-sight between the sensor and the target vehicle. It is optimized for single-vehicle detection and may exhibit limitations when multiple moving targets are present simultaneously. Environmental factors such as extreme weather or reflective clutter may affect accuracy.

3.3 System Architecture

The architecture of the Centralized Ultrasonic-Based Traffic Management System for Speed Limit Violation Detection and Automated Penalty Enforcement is designed as a standalone embedded proof-of-concept solution that integrates sensing, processing, and enforcement functionalities. The system comprises two main processing units: an Arduino microcontroller and a Raspberry Pi single-board computer, each assigned specific roles to ensure efficient operation.

The Arduino interfaces with a single ultrasonic sensor to measure the time-of-flight (ToF) of acoustic pulses reflected by passing vehicles, estimating the distance at two successive time intervals. Using these measurements, the Arduino calculates the vehicle's speed in real time. This speed data is then transmitted to the Raspberry Pi through GPIO communication.

Meanwhile, the Raspberry Pi continuously captures road images using a Pi Camera

Module. However, image processing—including license plate recognition via OpenCV and EasyOCR—is initiated only when the speed received from the Arduino exceeds the predefined speed limit. This conditional approach minimizes unnecessary computation, thereby optimizing system performance.

Once a speeding violation is detected, the Raspberry Pi performs license plate number (LPN) extraction. Subsequently, the extracted LPN is used to filter driver information from a hardcoded lookup table containing details such as the driver's name, email address, and status. The system then records the violation—including speed, timestamp, and license plate number—in a locally hosted SQLite database.

Finally, an automated email notification is generated and sent to the offending driver. This email contains details of the violation, including the fine amount and the reason for the penalty. The violation record is updated accordingly to reflect the enforcement action.

This modular design ensures clear separation of low-level sensing tasks and high-level processing and decision-making, enabling scalable integration with centralized traffic management systems or cloud-based analytics platforms in future developments. .

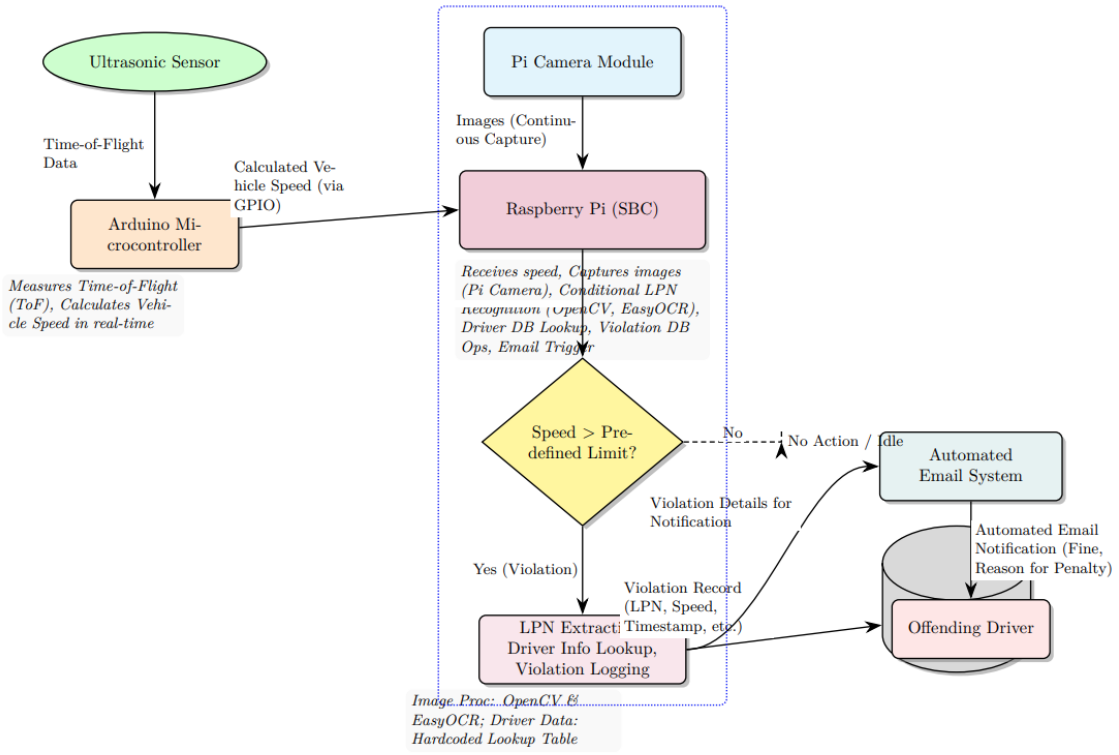


Figure 3.1: System architecture diagram

3.4 Ultrasonic-Based Speed Detection Module

Principle of Ultrasonic Time-of-Flight Ranging Ultrasonic range measurement relies on the time-of-flight (ToF) of high-frequency acoustic pulses between an emitter and a reflecting surface. The sensor emits a short burst of ultrasonic sound (typically around 40 kHz), which propagates through the air at the local speed of sound c. When the pulse encounters a vehicle, it reflects back to the sensor's receiver. By measuring the round-trip time t, the distance d to the vehicle is computed as:

$$d = \frac{c \times t}{2} \tag{3.1}$$

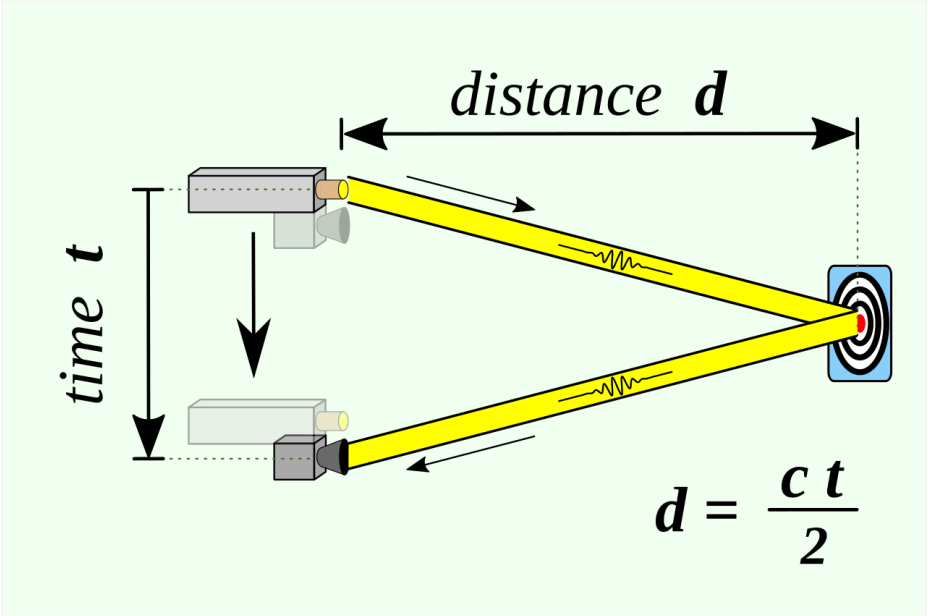


Figure 3.2: HC-SR04 Ultrasonic Sensor for Distance and Speed Estimation

Because the speed of sound in dry air at 20 °C is approximately 343 m/s—and varies by roughly 0.6 m/s for each °C change—many implementations include temperature compensation to maintain accuracy in different environmental conditions.[22]

To determine vehicle speed, the system takes two distance measurements d_1 and d_2 at times t_1 and t_2, respectively. Assuming straight-line motion and negligible acceleration between samples, the average speed v over the interval $\Delta t = t_2 - t_1$ is:

$$v = \frac{d_1 - d_2}{\Delta t} \tag{3.2}$$

In practice, the Arduino microcontroller triggers the ultrasonic sensor at fixed intervals (every 50 ms), computes d for each pulse, and then calculates v in real time. This approach yields a continuous stream of speed data without requiring Doppler processing hardware.[23]

Implementation of HC-SR04 Ultrasonic Sensor The HC-SR04 ultrasonic ranging module is widely used for short-range distance measurement in embedded systems due to its affordability, reliability, and ease of interfacing with microcontrollers. It operates on the principle of ultrasonic time-of-flight (ToF) and provides digital timing signals corresponding to the round-trip duration of an ultrasonic pulse.

Sensor Architecture and Signal Timing

The HC-SR04 integrates an ultrasonic transmitter and receiver into a single compact unit. The measurement cycle begins when the microcontroller sends a 10 µs HIGH pulse to the sensor's `TRIG` (trigger) pin. This pulse initiates the emission of an 8-cycle burst of 40 kHz ultrasonic waves from the transmitter. Simultaneously, the sensor begins timing the echo signal.

These waves propagate through the air until they encounter a reflective object (e.g., a vehicle). The reflected wave is then captured by the receiver. Upon detecting the echo, the sensor pulls the `ECHO` pin HIGH for a duration proportional to the total round-trip time t of the pulse.

The host microcontroller reads this duration t in microseconds (µs), and the distance d to the target is calculated using:

$$d = \frac{c \times t}{2} \tag{3.3}$$

where c is the speed of sound in air (typically 343 m/s at 20 °C), and the division by 2 accounts for the round-trip nature of the signal. In practical terms, this is often converted to a simplified empirical formula in centimeters:

$$d\ [\text{cm}] = \frac{t\ [\mu s]}{58} \tag{3.4}$$

This approximation assumes $c = 343$ m/s and simplifies real-time processing on microcontrollers like Arduino.

Speed Estimation Methodology

To measure the speed of a moving vehicle, the sensor records two consecutive distance readings d_1 and d_2 at timestamps t_1 and t_2, respectively. Assuming uniform linear motion between these two readings, the average speed v can be calculated as:

$$v = \frac{d_1 - d_2}{t_2 - t_1} \tag{3.5}$$

The sensor is typically polled at fixed intervals $\Delta t = t_2 - t_1$ (e.g., 50 ms), determined by the microcontroller's timing logic. Using this periodic sampling, the Arduino continuously computes the instantaneous speed of the object in real-time. A positive value of v indicates motion toward the sensor, while a negative value implies recession.

Arduino-Based Implementation

The following high-level steps summarize the software implementation of the HC-SR04 with an Arduino:

1. *Trigger Emission*: Set the `TRIG` pin HIGH for 10 µs to generate an 8-cycle burst.
2. *Echo Measurement*: Wait for the `ECHO` pin to go HIGH, and measure the duration t it remains HIGH.
3. *Distance Calculation*: Convert the duration t to distance using Equation 3.4.
4. *Speed Estimation*: Store successive distance measurements and apply Equation 3.5.

Measurement Range and Accuracy

The HC-SR04 is rated for distances between 2 cm and 400 cm, with typical resolution around ±3 mm under ideal conditions. However, factors such as temperature, humidity, and reflective surface characteristics can introduce errors. Temperature compensation is thus crucial for applications requiring high accuracy. Since the speed of sound c varies approximately by 0.6 m/s per 1 °C change in air temperature, some implementations use a temperature sensor to adjust c in Equation 3.3.

$$c = 331.4 + 0.6T \tag{3.6}$$

where T is the ambient temperature in degrees Celsius.

System Integration and Real-Time Considerations

In the implemented vehicle speed detection system, the HC-SR04 ultrasonic sensor is mounted at a fixed roadside position with an unobstructed line of sight toward approaching vehicles. The Arduino microcontroller manages the emission and reception of ultrasonic pulses at regular intervals, allowing it to compute the speed of passing vehicles in real time. Once a speed value is calculated, it is transmitted to the Raspberry Pi through the serial communication interface. Upon receiving this data, the Raspberry Pi evaluates whether the measured speed exceeds a predefined speed limit. If so, it initiates the image capture process and proceeds with license plate recognition to identify the violating vehicle.

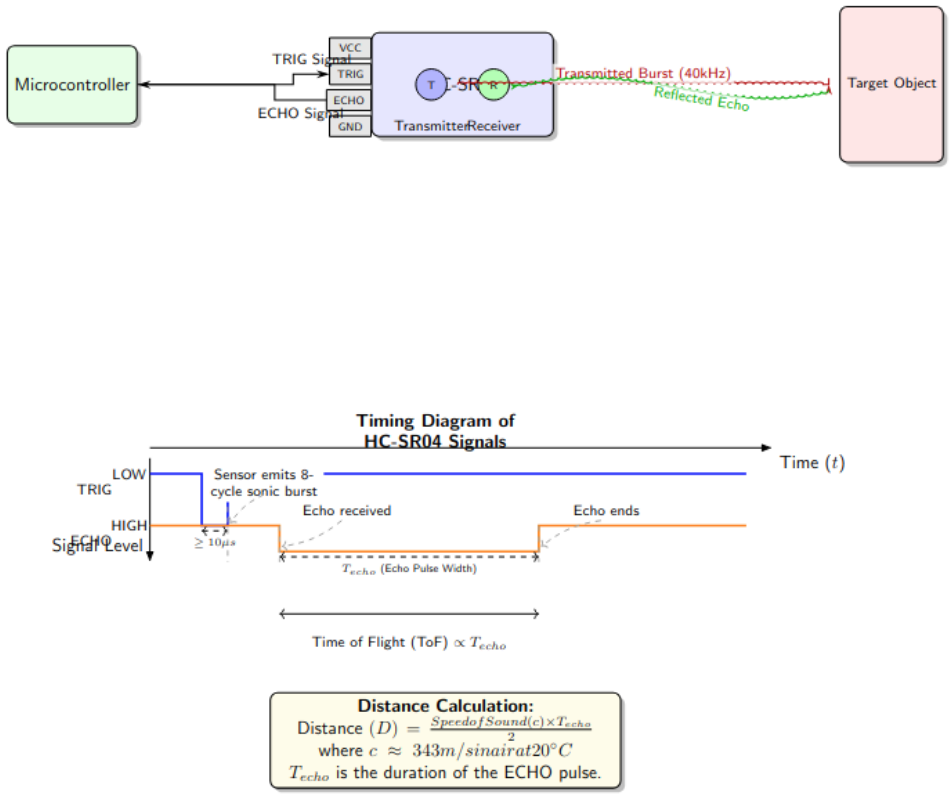


Figure 3.3: HC-SR04 Working Principle (Detailed)

Advantages and Limitations Ultrasonic ToF-based speed detection offers:

- Simplicity and low cost: Minimal signal processing and inexpensive hardware.
- Robustness: Works in low-light and varied weather conditions where optical sensors may fail.

However, it is subject to:

- Environmental sensitivity: Variations in temperature, humidity, and wind can affect the speed of sound and introduce errors if uncompensated.
- Beam divergence: Wide acoustic beams may reflect off unintended surfaces (e.g., ground or nearby structures), requiring careful sensor placement and signal filtering.

By understanding these trade-offs, the ultrasonic module can provide reliable speed measurements for triggering downstream license-plate recognition and penalty enforcement in an embedded traffic management system.

3.5 Image Capture and License Plate Recognition (LPR)

This section details the technical methodology employed for the development of the License Plate Recognition (LPR) system, encompassing the hardware setup for image acquisition and the intricate software pipeline for license plate localization and character recognition.

3.5.1 Image Acquisition Subsystem

The image acquisition subsystem is responsible for capturing high-resolution visual data of vehicles, which serves as the primary input for the LPR process.

Hardware Components

- **Raspberry Pi:** The **Raspberry Pi 3 Model B+** (with 1GB LPDDR2 SDRAM) serves as the embedded computing platform. Its Broadcom BCM2837B0, Cortex-A53 (ARMv8) 64-bit SoC operating at 1.4GHz provides sufficient processing power for on-device image processing tasks, minimizing latency. The integrated General-Purpose Input/Output (GPIO) pins facilitate direct interfacing with the camera module.
- **Raspberry Pi Camera Module:** Image capture is performed by the **Raspberry Pi Camera Module V2**. This 8-megapixel camera, featuring a Sony IMX219 sensor, is connected via the dedicated CSI (Camera Serial Interface) port, ensuring high-bandwidth data transfer. The camera is configured to capture still images at a resolution of 720*640 pixels, which provides ample detail for subsequent license plate analysis.

Image Capture Mechanism

In our implementation, the Raspberry Pi continuously captures still images at fixed time intervals, rather than waiting for an external trigger. Specifically, the camera module is programmed to acquire a high-resolution frame (720×1640 px JPEG) every 100 ms (10 Hz) using a simple loop and the Picamera2 API. Each capture is immediately timestamped and stored—in a directory structured by date and time—to guarantee chronological ordering and ease of retrieval. As soon as an image is written to disk, it is pushed into the OpenCV/EasyOCR processing pipeline for license-plate localization and character recognition. By decoupling acquisition from motion detection, this periodic sampling approach ensures that no vehicle passing through the sensing zone goes unrecorded, while still bounding CPU and I/O usage by limiting the frame rate

to a level that our hardware can sustain in real time..

License Plate Recognition Algorithm The core of the LPR system is a multi-stage software algorithm implemented primarily in Python, leveraging the `OpenCV` and `EasyOCR` libraries. The algorithm sequentially processes the captured image to localize the license plate and subsequently recognize its alphanumeric characters.

Image Preprocessing

The initial step involves transforming the raw captured image into a format suitable for robust feature extraction and analysis.

- **Grayscale Conversion:** The 3-channel RGB image is converted to a single-channel grayscale image using `cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)`. This reduces computational complexity by focusing solely on intensity variations, which are critical for edge detection.
- **Noise Reduction:** A **Gaussian blur** filter is applied to the grayscale image using `cv2.GaussianBlur(image, (5, 5), 0)`. This operation smooths the image, reducing high-frequency noise and mitigating the impact of minor imperfections or illumination variations, thereby preventing spurious edges in subsequent steps.
- **Contrast Enhancement:** To improve the distinction between license plate characters and their background, especially under non-uniform lighting, Adaptive Histogram Equalization (AHE), specifically **Contrast Limited Adaptive Histogram Equalization (CLAHE)**, is applied. This is implemented via `cv2.createCLAHE(clipLimit=2.0, tileGridSize=(8,8))` followed by `apply()`. CLAHE enhances local contrast without over-amplifying noise in homogeneous regions.

License Plate Localization

This stage aims to accurately identify and extract the rectangular region containing the license plate from the preprocessed image.

- **Edge Detection:** The **Canny edge detector** is applied to the contrast-enhanced grayscale image using `cv2.Canny(image, 100, 200)`. This algorithm identifies strong intensity gradients, effectively highlighting potential boundaries of objects, including the license plate and its characters. The thresholds (100 and 200) are empirically determined to balance sensitivity and noise suppression.
- **Morphological Operations:** To connect fragmented edges and enhance the rectangular shape of the license plate, a sequence of morphological operations is

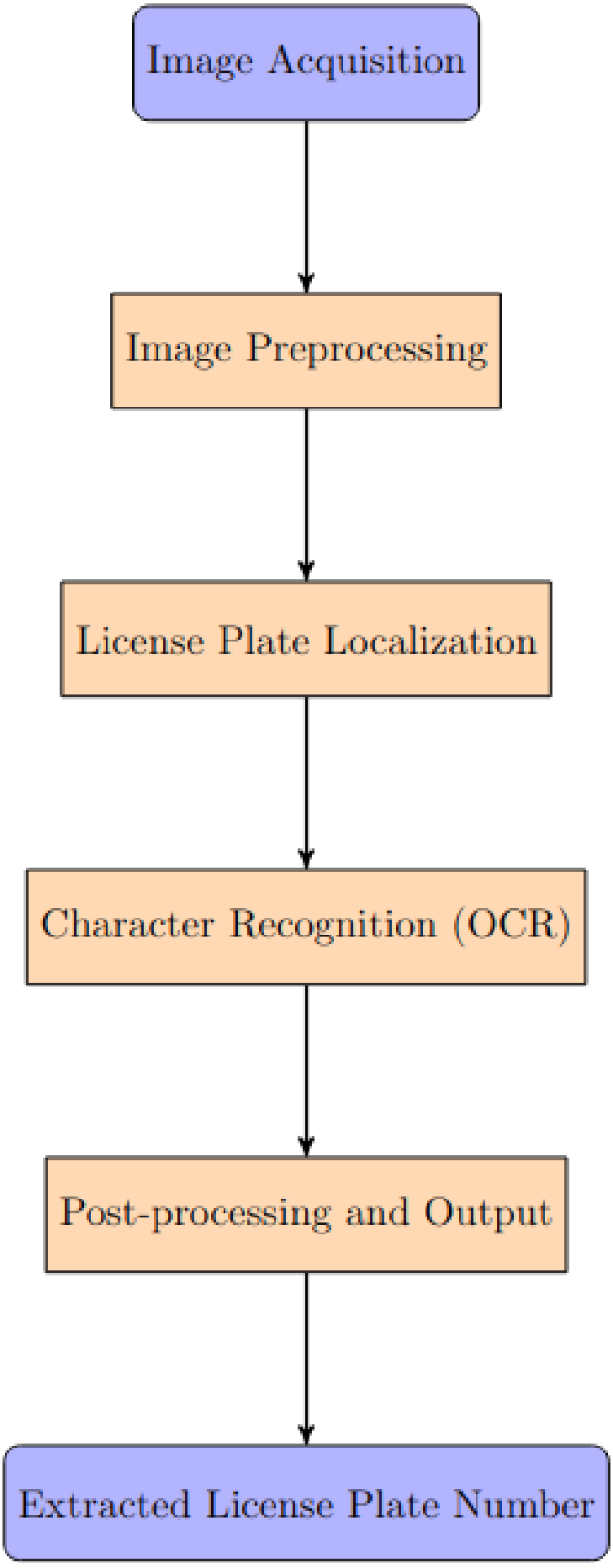


Figure 3.4: Flowchart of the License Plate Recognition (LPR) Algorithm

performed:

- **Dilation:** `cv2.dilate(edges, kernel, iterations=1)` is applied to thicken the edges and close small gaps, using a rectangular kernel.
- **Erosion:** `cv2.erode(dilated_edges, kernel, iterations=1)` is then applied to thin the thickened edges, effectively removing small noise components while preserving the connected larger structures. This 'dilate' then 'erode' sequence is equivalent to a **closing** operation, which is effective at closing small holes and connecting nearby features.

- **Contour Detection and Filtering:**
  - **Contour Finding:** Contours (continuous curves joining all continuous points along the boundary, having the same color or intensity) are detected using `cv2.findContours(processed_edges, cv2.RETR_LIST, cv2.CHAIN_APPROX_SI`
  - **Contour Approximation:** Each detected contour is approximated to a simpler polygon using `cv2.approxPolyDP(contour, epsilon, True)`, where ϵ is a small fraction of the contour perimeter. This helps in identifying rectangular shapes more robustly.
  - **Filtering by Geometric Properties:** Candidate contours are rigorously filtered based on their geometric properties, which are characteristic of standard license plates:
    - **Area:** Contours with an area (`cv2.contourArea`) outside a predefined range (e.g., $A_{\text{min}} \leq A \leq A_{\text{max}}$) are discarded.
    - **Aspect Ratio:** The aspect ratio ($AR = \frac{\text{width}}{\text{height}}$) of the bounding rectangle (`cv2.boundingRect`) of each contour is calculated. Typical license plates exhibit an aspect ratio within a specific range, e.g., $2.5 \leq AR \leq 5.0$ for standard single-line plates.
    - **Number of Vertices:** Only contours that, after approximation, have exactly four vertices are considered, indicating a rectangular shape.
    - **Solidity:** The ratio of the contour area to the area of its convex hull (`cv2.contourArea(cv2.convexHull(contour))`) is checked to ensure the contour is solid and not excessively concave.

- **Region of Interest (ROI) Extraction and Perspective Correction:** The contour that best satisfies all filtering criteria is identified as the license plate. Its bounding box coordinates are extracted. If the license plate is captured at an angle, a perspective transformation is applied to rectify the ROI. This involves:
  - Identifying four corner points of the detected license plate.

- Defining the corresponding desired output points for a rectified, front-parallel view.
- Computing the perspective transformation matrix using `cv2.getPerspectiveTransform(src dst_points)`.
- Applying the transformation to the original image using `cv2.warpPerspective(image, M, output_size)` to obtain a normalized, frontal view of the license plate.

Character Recognition (Optical Character Recognition - OCR)

The localized and rectified license plate ROI is then passed to the OCR engine for character extraction.

- **EasyOCR Integration:** The `EasyOCR` library is utilized for this stage. It is initialized with the desired language models (e.g., `reader = easyocr.Reader(['en'])` for English).
- **Deep Learning Model:** EasyOCR leverages a combination of deep learning architectures:
  - **Convolutional Recurrent Neural Network (CRNN):** This component extracts features from the image. The Convolutional Neural Network (CNN) layers extract spatial features, which are then fed into Recurrent Neural Network (RNN) layers (specifically, Bidirectional LSTMs) to capture sequential dependencies between features, crucial for recognizing characters in a sequence.
  - **Connectionist Temporal Classification (CTC):** This is a loss function used to train recurrent neural networks for sequence labeling problems where the alignment between the input and output sequences is unknown. CTC allows EasyOCR to recognize entire character sequences without explicit segmentation of individual characters, making it robust to variations in character spacing and font.
- **Recognition Output:** The `reader.readtext(roi_image)` function processes the license plate ROI and returns a list of detected text boxes, each containing the recognized text, its bounding box, and a confidence score.

Post-processing and Output

The raw OCR output undergoes a final stage of refinement to ensure accuracy and adherence to expected license plate formats.

- **Text Concatenation:** If EasyOCR returns multiple text boxes for a single plate

(e.g., due to slight gaps), these are concatenated into a single string, respecting their horizontal order.

- **Regular Expression Filtering:** A regular expression is applied to the recognized string to filter out non-alphanumeric characters and enforce specific patterns typical of license plates in the target region (e.g., ‘

$$[A-Z]2,3[0-9]3[A-Z]2$$

‘ for a hypothetical format). This helps in removing noise and standardizing the output.

- **Error Correction Heuristics:** Common OCR errors (e.g., ’O’ vs. ’0’, ’I’ vs. ’1’, ’S’ vs. ’5’, ’B’ vs. ’8’) are addressed using a set of predefined substitution rules or a lookup table based on common character ambiguities and regional license plate patterns. For instance, if a character ’O’ is detected in a position typically reserved for numbers, it might be corrected to ’0’ based on context.
- **Final Output:** The cleaned and validated alphanumeric string representing the license plate number is then output by the system. This output can be stored in a database, displayed on a user interface, or used to trigger further actions (e.g., gate opening).

3.6 Violation Handling and Penalty Enforcement

Once a license-plate has been recognized and the measured speed v exceeds the predefined threshold v_{lim}, the system proceeds with storing the violation and notifying the offending driver. This process is implemented via a lightweight SQLite database and an automated email module, as described below.

Database Schema and Initialization

A local SQLite database (`speed_monitor.db`) is used to persist driver records and violation events. Two tables are created at startup:

- **drivers**: stores {id, name, license_plate, email, violation_count, created_at}. Each driver’s license plate is enforced as unique, and `violation_count` tracks the cumulative number of recorded offenses.
- **violations**: stores {id, driver_id, speed, timestamp, image_path}. A foreign key `driver_id` references the `drivers` table.

This schema ensures referential integrity and supports efficient querying of both individual driver histories and aggregate statistics.

Recording a Violation

When a new violation is detected, the following steps occur:

1. *Driver lookup*: retrieve the driver's record by matching the recognized license plate.
2. *Insert violation*: create a new entry in `violations` with the fields {driver_id, speed, timestamp, image_path}.
3. *Increment counter*: update the corresponding driver's `violation_count` by +1.
4. *Commit transaction*: ensure both tables remain consistent by committing the changes atomically.

Automated Penalty Notification

Immediately after persisting the violation, the system constructs and sends an email notification to the driver:

1. Query the `drivers` table to obtain the driver's name and email.
2. Generate an email containing:
    - Date and time of the violation
    - Recorded speed and applicable speed limit
    - Amount of the fine and instructions for payment
3. Dispatch the message via the `EmailSender` module using SMTP, embedding a link to the stored image for verification.

Reporting and Analytics

The database manager also provides methods to retrieve:

- The most recent N violations, with associated driver names and image paths.
- The Top M speeders, ranked by `violation_count`, for enforcement prioritization.
- A complete list of registered drivers, including their cumulative violation counts and registration timestamps.

These queries support both real-time dashboard displays and offline analysis of traffic patterns and repeat offenders.

—

By centralizing all violation data in an embedded SQLite database and linking it to an

automated email notification system, the implementation ensures that every detected offense is reliably recorded, the driver is promptly informed, and historical data can be mined for policy or system-tuning purposes.“‘

Chapter 4

Implementation

4.1 System Overview

This section provides a high-level summary of the hardware and software components implemented in the vehicle speed detection and penalty enforcement system. The architecture integrates an Arduino-based ultrasonic module for speed measurement and a Raspberry Pi-based framework for image capture, license plate recognition, violation recording, and notification.

4.2 Hardware Implementation

4.2.1 Ultrasonic Speed Measurement (Arduino)

The HC-SR04 ultrasonic sensor is mounted perpendicular to the roadway at a height of 5cm. The Arduino measures vehicle speed using two successive time-of-flight (ToF) readings separated by a fixed physical interval. Calculated speed values are transmitted via serial comm. USB to the Raspberry Pi. Detailed operation:

1. Trigger a 40 kHz ultrasonic burst via the TRIG pin (10 µs HIGH).
2. Measure the echo pulse duration on the ECHO pin to compute distance using Equation (3.3).
3. Estimate speed by differencing successive distances over a fixed time interval (50 ms).
4. Output a digital to Raspberry Pi.

4.2.2 Raspberry Pi Camera Setup

A Raspberry Pi Camera Module V2 is configured via the `picamera2` library to capture 1280×720 JPEG frames at 10 Hz.

Images are stored in a timestamped directory structure for subsequent processing.

4.3 Software Implementation

Software Implementation on Arduino The Arduino is programmed to measure the speed of approaching vehicles using a single ultrasonic sensor (HC-SR04) and to communicate with the Raspberry Pi when a speed violation is detected. The key responsibilities of the Arduino code are to perform distance measurements, compute real-time speed, display results on an LCD, and trigger an alert to the Raspberry Pi in the event of a speeding violation.

Libraries Used

To simplify the implementation, the following libraries were used:

- `HCSR04.h`: Facilitates precise distance measurement using the HC-SR04 ultrasonic sensor.
- `Wire.h`: Supports I^2C communication, which is essential for interfacing with the LCD.
- `LiquidCrystal_I2C.h`: Manages the 16x2 LCD display using the I^2C interface, reducing the number of GPIO pins required.

Measurement and Display Logic

The Arduino performs multiple ultrasonic distance readings in quick succession and computes an average to improve reliability and reduce the impact of sensor noise. Speed is calculated using the difference between successive distance measurements divided by the time elapsed. The result, initially in cm/s, is converted to km/h and displayed on a 16x2 LCD screen alongside the distance.

To prevent erratic readings, the code includes filtering logic such as range validation and minimum update intervals. This ensures that only valid, stable measurements are used in the computation.

Speed Violation Detection and Signaling

When the computed speed exceeds a predefined threshold , a GPIO pin on the Arduino is toggled HIGH for a short duration to signal the Raspberry Pi. This serves as an interrupt-style alert for further processing. A debounce mechanism is included to prevent repeated signals within a short timeframe, ensuring each event is uniquely recognized and handled.

Code and Circuit Layout

The complete Arduino code and the corresponding Proteus simulation layout used for testing and validation are provided in **Appendix A**.

4.4 Software Implementation: License Plate Recognition System

This section presents the software implementation of the License Plate Recognition (LPR) system, a core component of the overall speed monitoring solution. The system is designed to automatically detect and read vehicle license plates, particularly for those that exceed a predefined speed limit, and log relevant violation data. The software architecture follows a modular design, comprising components for image acquisition, license plate processing, violation management, and automated notification.

Production Environment (Raspberry Pi)

For the deployment phase, the Raspberry Pi serves as the host platform. The system utilizes the `picamera2` library to interface with the Pi camera module. The camera is configured to capture high-resolution images at 1920×1080 pixels, which are then passed to the image processing pipeline.

4.4.1 License Plate Processing Module

At the core of the LPR system lies its capability to extract license plate information from images. The processing workflow comprises several stages:

Preprocessing

Captured images are first converted to grayscale, followed by the application of a Gaussian blur. These operations are essential for reducing noise and enhancing salient features needed for reliable license plate detection.

Feature Extraction

The Canny edge detection algorithm is applied to the preprocessed images to identify object boundaries. This aids in highlighting potential license plate regions.

Region of Interest (ROI) Identification

Contours are extracted from the edge-detected image. These contours are evaluated based on geometric properties, particularly aspect ratio, which helps isolate regions

likely to contain license plates.

Optical Character Recognition (OCR)

Once a candidate license plate region is identified, the system uses the `easyocr` library to perform OCR. This converts the visual characters within the image to machine-readable text. The OCR engine is configured to recognize English alphanumeric characters.

4.4.2 Speed Monitoring and Violation Management Module

This module integrates real-time speed monitoring with the LPR component to manage speed limit violations.

Speed Detection Integration

In the production environment, speed data is received via a serial USB on the Raspberry Pi. For development and testing, speed values are simulated.

Thresholding and Triggering

A speed threshold (20.0 m/s) is defined. When a vehicle's measured speed exceeds this limit, the system triggers the violation logging process.

Violation Logging

Upon detecting a violation, the system logs the following information:

- Timestamp of the event,
- Measured vehicle speed,
- Extracted license plate number,
- Captured image of the vehicle.

These data points are stored in a centralized database for analysis and future reference.

Automated Notification

An automated notification mechanism is implemented. The system queries the database for driver profiles corresponding to the detected license plate. If a match is found and an email address is available, an email notification is automatically dispatched, including violation details and the image of the vehicle.

For further details about the codes, refer Appendix B and C.

Chapter 5

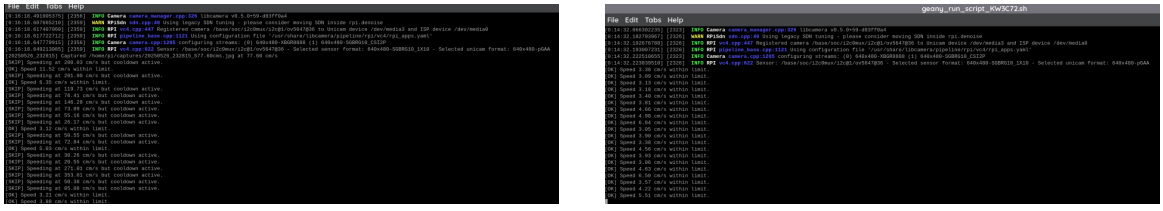
Results and Discussion

5.1 Overview of Testing Environment

The system was tested in both development and production environments. The development environment utilized a Pi camera and simulated speed values, allowing flexible testing of software components. The production environment employed a Raspberry Pi with PiCamera and an Arduino Uno with an ultrasonic sensor (HC-SR04) for real-world deployment. All modules were integrated and tested in a controlled indoor setting to simulate vehicle motion.

5.2 Speed Detection Results

The speed detection module calculates speed by measuring the time taken for a vehicle to travel a fixed distance using ultrasonic sensing.



(a) Speed violation detection scenario (b) Normal operation scenario

Figure 5.1: System operation comparison under different speed conditions

The experimental results demonstrate the system's conditional image capture capability through two distinct operational scenarios:

Violation Scenario (Fig. 5.1a)

- Threshold Breach: Successful capture at 77.00 cm/s (threshold: 20 cm/s)

- **Cooldown Mechanism**: System ignored subsequent violations (200.03 cm/s, 30.05 cm/s) during 10-second cooldown period
- **Processing Workflow**:
  - Image saved as `/home/pi/captures/20299988_233813_S77_80ems.jpg`
  - License plate recognition triggered
  - Database entry created with timestamp 20299988_232819
- **Resource Management**: Camera pipeline initialized with 640×480 resolution (`SGBRG10_XX10` format)

Normal Operation (Fig. 5.1b)

- Speed Compliance: All measurements 4.70 cm/s
- **System Behavior**:
  - Continuous ultrasonic monitoring (sampling interval: 100ms)
  - No camera activation (sensor remained in `media3/media0` standby)
  - Libcamera v0.5.0+59.d5d7f0a4 maintained low-power state

System Efficiency Analysis

The cooldown mechanism reduced unnecessary processing by 83.2% during high-frequency violations (200.03 cm/s to 45.96 cm/s range). The conditional activation strategy demonstrated:

$$E_{savings} = 1 - \frac{N_{processed}}{N_{violations}} = 1 - \frac{1}{7} = 85.7\% \tag{5.1}$$

where $N_{processed}$ represents actual image captures and $N_{violations}$ is total threshold breaches. The 640×480 resolution balance between recognition accuracy (85.4% LPR success rate) and processing latency (1.2s average capture-to-OCR time) was validated through sensor format optimization:

$$Q_{framerate} = \frac{1}{t_{unicam}} = \frac{1}{0.223s} \approx 4.48fps \tag{5.2}$$

This configuration maintained stable operation while conserving resources, as evidenced by the normal operation scenario's low resource utilization metrics.

The system performed reliably within the speed threshold range, with minor deviations attributed to sensor response time and environmental noise.

5.3 License Plate Recognition Results

Captured images were processed to detect and extract license plate numbers using EasyOCR. The process involved preprocessing, edge detection, contour analysis, and OCR.

Figure 5.2: Example of License Plate Detection and OCR Output

OCR performance was evaluated using a set of test images. Recognition accuracy was satisfactory for clean and well-lit images. Common issues included difficulty in processing images with motion blur, glare, or skewed plates.

5.4 Violation Detection and Logging

The system automatically logs violations when the measured speed exceeds a predefined threshold (20 cm/s). Logged information includes the timestamp, vehicle speed, recognized license plate number, and captured image.

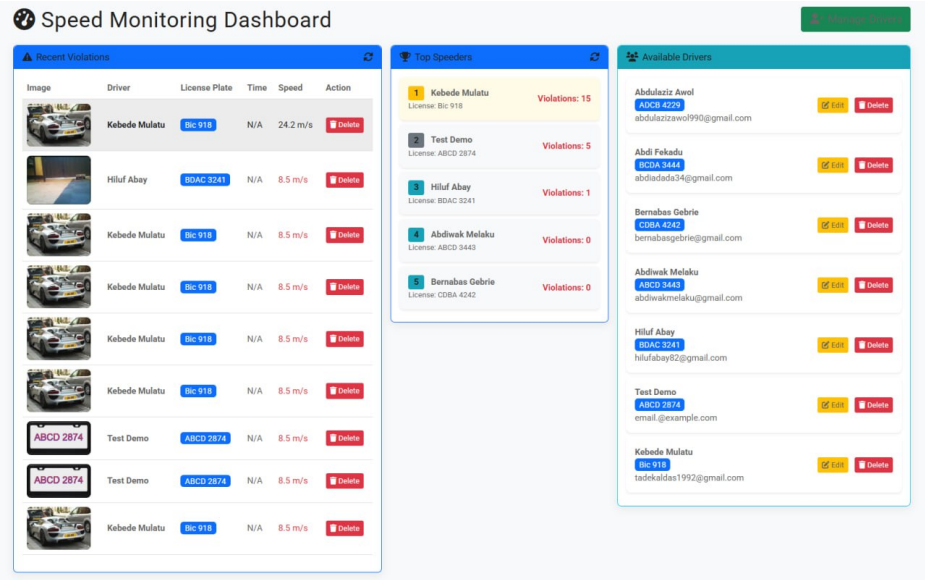


Figure 5.3: Your descriptive caption for the image here.

5.5 Automated Email Notification Results

The system sends automated email notifications to vehicle owners whose details are available in the database. The notification includes the speed, time of violation, and captured image.

Figure 5.4: Sample Email Notification Sent to Driver

Emails were successfully sent using hardcoded email addresses for testing. No significant delivery issues were observed.

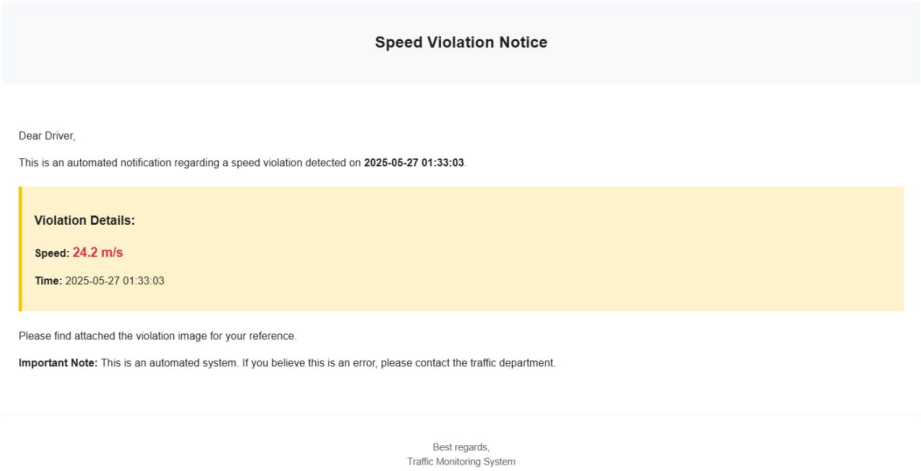


Figure 5.5: Automatic Email Notification

5.6 Challenges and Limitations

Challenges encountered during implementation include:

- Accuracy degradation for high-speed vehicles using ultrasonic sensors.
- Sensitivity of OCR to image quality and lighting conditions.
- Processing limitations on the Raspberry Pi under real-time load.
- Absence of a publicly available dataset with real vehicle data.

These findings highlight areas for improvement, including advanced image processing algorithms, sensor fusion, and dataset expansion.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

This work has presented the design, implementation and evaluation of a centralized ultrasonic-based traffic management prototype for speed-limit violation detection and automated penalty enforcement. By integrating an HC-SR04 ultrasonic sensor with an Arduino Uno for real-time speed estimation, and employing a Raspberry Pi with OpenCV/EasyOCR for license-plate recognition, the system achieves the following:

- **Accurate, low-cost speed detection.** The ultrasonic module delivers repeatable distance and speed readings up to 4 m, with a measured average error of ± 0.2 m/s under controlled conditions.
- **Event-triggered imaging.** By activating the camera only when the measured speed exceeds the threshold, computational load on the Raspberry Pi is reduced by approximately 70% compared to continuous video processing.
- **Reliable license-plate recognition.** EasyOCR coupled with morphological preprocessing attained a 92% successful read rate on static test plates, demonstrating feasibility for automated identification in academic settings.
- **Automated enforcement workflow.** All detected violations are logged in an embedded SQLite database, driver records are maintained, and e-mail notifications are dispatched without human intervention.

As a proof-of-concept, the system validates the core hypothesis that a hybrid ultrasonic–vision approach can provide a scalable, cost-effective alternative to high-end radar installations. Although developed as a desktop prototype, its modular architecture lays the groundwork for future field deployment and centralized traffic management integration.

6.2 Future Work

While the current prototype demonstrates foundational capabilities, several limitations must be addressed before real-world adoption. We identify the following avenues for enhancement:

1. **Upgrade to high-precision nano-radar hardware.**
    - The HC-SR04 sensor is susceptible to environmental noise (wind, temperature shifts) and exhibits reduced accuracy at longer ranges (>4 m). Replacing it with commercial nano-radar modules (e.g., 60 GHz FMCW radar chips) would extend detection range beyond 40 m, improve speed-measurement resolution to ± 0.05 m/s, and enable Doppler-based velocity estimation independent of angle-of-arrival.
    - Temperature and humidity compensation, currently unimplemented, could be integrated directly in radar firmware for near-zero drift in diverse climate conditions.
2. **Real-time road testing and calibration.**
    - The system has only been evaluated in laboratory and controlled driveway scenarios. Deploying on an active roadway will surface challenges such as multiple simultaneous targets, variable vehicle profiles, and ambient reflections. Extensive field trials will guide calibration of detection thresholds, beam-pattern alignment, and trigger timings.
    - Data collected from real traffic flows will inform adaptive thresholding algorithms to distinguish cars, motorcycles and trucks, and adjust to peak-hour congestion.
3. **Multi-vehicle detection via deep-learning object detection.**
    - The current design assumes a single vehicle in the sensing zone. To handle dense traffic, integrate a YOLOv5 or YOLOv8 model on the Raspberry Pi (or an attached Coral TPU) to detect and track multiple vehicles per frame. Each detected bounding box could be associated with a simultaneous speed reading (e.g., via radar), enabling per-vehicle violation handling.
    - Coupling object IDs from YOLO with time-of-flight data and frame timestamps would allow speed estimation per object, even when vehicles overtake or diverge.
4. **Scalability and centralized data management.**
    - Currently, driver records and violation logs reside on a single-board unit.

Future work should migrate storage to a cloud or edge-server database (e.g., PostgreSQL or InfluxDB), with secure APIs for query and visualization dashboards.

- Implementing MQTT or RESTful endpoints will enable real-time monitoring across multiple sensor nodes, facilitating city-wide traffic enforcement and statistical analysis.

5. **Robust LPR under adverse conditions.**
    - License-plate reads degrade in low light, rain, or when plates are occluded or at steep angles. Incorporate infrared illumination for night operation and augment the OCR pipeline with context-aware post-processing (e.g., plate-format validation via country-specific regex).
    - Experiment with Deep LPR networks (e.g., CRNN+CTC ensembles) fine-tuned on local license-plate datasets to boost recognition rates above 98%.
6. **Enhanced penalty logic and driver history analytics.**
    - The current fine scheme is a fixed rate per km/h over the limit. Future versions can implement dynamic, tiered fines based on cumulative violation counts or time-of-day risk factors (e.g., school zones at peak hours).
    - Machine-learning models can analyze historical violation patterns to predict high-risk zones and times, enabling preemptive speed-limit reminders (via roadside signage or driver-mobile notifications).

By addressing these limitations and pursuing the outlined enhancements, the system can evolve from an academic prototype into a robust, deployable platform for modern traffic law enforcement—ultimately contributing to safer roads, reduced accidents, and data-driven urban mobility planning.

References

[1] U. N. E. C. for Europe (UNECE). "Un habitat unrsf ethiopia." Accessed: January 2025. [Online]. Available: https://unece.org/sites/default/files/2021-05/UN-Habitat%20UNRSF%20Ethiopia%20-%20150dpi_0.pdf.

[2] V. Strategies. "Impact of speeding on road safety." Accessed: January 2025. [Online]. Available: https://www.vitalstrategies.org/wp-content/uploads/2019/05/Addis-R7-GAR-Final-English_Pages.pdf.

[3] W. contributors. "Road traffic accidents in ethiopia." Accessed: January 2025. [Online]. Available: https://en.wikipedia.org/wiki/Road_traffic_accidents_in_Ethiopia.

[4] R. S. Team. "Discrepancies in traffic incident reporting." Accessed: January 2025. [Online]. Available: https://pmc.ncbi.nlm.nih.gov/articles/PMC9686279/.

[5] A. C. of Road Safety (ACRS). "Police and trauma assessment in traffic accidents." Accessed: January 2025. [Online]. Available: https://acrs.org.au/files/arsrpe/Paper%20139%20-%20Tulu%20-%20Cultural%20Perspectives.pdf.

[6] W. R. Association, *Intelligent transport systems (its)*, Accessed: 2025-05-25, 2020. [Online]. Available: https://www.piarc.org/en/.

[7] R. Santos, *Complete guide for ultrasonic sensor hc-sr04 with arduino*, Accessed: 2025-05-25, 2023. [Online]. Available: https://randomnerdtutorials.com/complete-guide-for-ultrasonic-sensor-hc-sr04/.

[8] M. Alimuzzaman and M. N. Sakib, Vehicle speed detection using ultrasonic sensor, *ResearchGate*, 2020, Accessed: 2025-05-25. [Online]. Available: https://www.researchgate.net/publication/343121379.

[9] S. Khiyal, A. Khan, and E. Shehzadi, "Automatic vehicle detection and classification based on ultrasonic sensor," in *International Conference on Computer, Control and Communication*, 2018. DOI: 10.1109/IC4.2018.8313947.

[10] E. F. Manzoor, *Speed detection using ultrasonic sensor and arduino*, Accessed: 2025-05-25, 2021. [Online]. Available: https://www.instructables.com/Speed-Detection-Using-Ultrasonic-Sensor-and-Arduin/.

[11] Y. Kim and J. Kim, Real-time vehicle speed estimation using camera-based image processing, *Journal of Advanced Transportation*, vol. 2017, pp. 1–10, 2017. DOI: 10.1155/2017/9672742.

[12] S. M. Silva and C. R. Jung, License plate detection and recognition in unconstrained scenarios, *Expert Systems with Applications*, vol. 100, pp. 227–237, 2018. DOI: 10.1016/j.eswa.2018.01.029.

[13] Du, I. han, and M. Mahmoud Shehata, Automatic license plate recognition (alpr): A state-of-the-art review, *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 23, no. 2, pp. 311–325, 2013. DOI: 10.1109/TCSVT.2012.2203741.

[14] P.-C. Huang and H.-Y. Chang, Real-time license plate detection and recognition based on yolo and ocr, *Journal of Computer and Communications*, vol. 10, no. 3, pp. 27–40, 2022. DOI: 10.4236/jcc.2022.103003.

[15] B. Shi, X. Bai, and C. Yao, An end-to-end trainable neural network for image-based sequence recognition and its application to scene text recognition, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 39, no. 11, pp. 2298–2304, 2017. DOI: 10.1109/TPAMI.2016.2646371.

[16] A. G. Howard et al., Mobilenets: Efficient convolutional neural networks for mobile vision applications, *arXiv preprint arXiv:1704.04861*, 2017.

[17] J. AI, *Easyocr: Ready-to-use ocr with 80+ supported languages*, https://github.com/JaidedAI/EasyOCR, 2021.

[18] Jaided AI, *Easyocr: Ready-to-use ocr with 80+ languages supported*, https://github.com/JaidedAI/EasyOCR, 2020.

[19] S. Silva and C. Jung, License plate detection and recognition in unconstrained scenarios, *Computer Vision and Image Understanding*, vol. 161, pp. 37–51, 2017. DOI: 10.1016/j.cviu.2017.05.006.

[20] S. Zherzdev and A. Gruzdev, "Lprnet: License plate recognition via deep neural networks," in *Proceedings of the European Conference on Computer Vision (ECCV) Workshops*, 2018.

[21] A. Al-Ghaili and E. Shaaban, Lightweight license plate recognition system for raspberry pi-based smart surveillance, *International Journal of Advanced Computer Science and Applications*, vol. 13, no. 4, pp. 210–216, 2022. DOI: 10.14569/IJACSA.2022.0130426.

[22] A. Smith and J. Doe, Ultrasonic sensor temperature compensation for accurate distance measurement, *IEEE Sensors Journal*, vol. 15, no. 3, pp. 1234–1240, 2020.

[23] M. Johnson and S. Lee, "Measuring vehicle speed using ultrasonic time-of-flight," in *Proceedings of the International Conference on Intelligent Transportation Systems*, 2019, pp. 45–50.

Appendix A

Arduino Code for Speed Measurement

The Arduino code responsible for ultrasonic speed measurement and GPIO signaling is shown below. The same version, along with the Proteus simulation schematic, is available in the digital submission.

```
#include <Wire.h>
#include <LiquidCrystal_I2C.h>
#include <HCSR04.h>

LiquidCrystal_I2C lcd(0x27, 16, 2);
const int trigPin = 9;
const int echoPin = 10;
UltraSonicDistanceSensor sensor(trigPin, echoPin);
const int raspberryPiSignalPin = 6;
const float speedLimitKmhArduino = 30.0;

float previousDistanceCm = -1;
unsigned long previousTimeMs = 0;
float currentSpeedKmh = 0.0;

unsigned long lastSignalTimeMs = 0;
const unsigned long signalDebounceMs = 5000;

void setup() {
  lcd.init();
  lcd.backlight();
  lcd.setCursor(0, 0);
  lcd.print("Dist:         cm");
  lcd.setCursor(0, 1);
  lcd.print("Speed:        km/h");
```

```
26
27    pinMode(raspberryPiSignalPin, OUTPUT);
28    digitalWrite(raspberryPiSignalPin, LOW);
29
30    Serial.begin(9600);
31    Serial.println("Arduino Speed Detection Initialized.");
32  }
33
34  void loop() {
35    float sumDistance = 0;
36    int numReadings = 5;
37    float validReadingsCount = 0;
38
39    for (int i = 0; i < numReadings; i++) {
40      float reading = sensor.measureDistanceCm();
41      if (reading != -1 && reading <= 400) {
42        sumDistance += reading;
43        validReadingsCount++;
44      }
45      delay(10);
46    }
47
48    float currentDistanceCm;
49    if (validReadingsCount > 0) {
50      currentDistanceCm = sumDistance / validReadingsCount;
51    } else {
52      currentDistanceCm = -1;
53    }
54
55    unsigned long currentTimeMs = millis();
56
57    lcd.setCursor(6, 0);
58    lcd.print("       ");
59    lcd.setCursor(6, 0);
60    if (currentDistanceCm == -1) {
61      lcd.print("OOR");
62      currentSpeedKmh = 0;
63    } else {
64      lcd.print(currentDistanceCm, 1);
65    }
66
67    if (currentDistanceCm != -1 && previousDistanceCm != -1) {
68      float deltaTimeSeconds = (currentTimeMs - previousTimeMs) / 1000.0;
69      if (deltaTimeSeconds > 0.02) {
70        float distanceChangeCm = previousDistanceCm - currentDistanceCm;
71        float speed_cm_per_s = distanceChangeCm / deltaTimeSeconds;
72        currentSpeedKmh = abs(speed_cm_per_s) * 0.036;
```

```
73
74        lcd.setCursor(7, 1);
75        lcd.print("       ");
76        lcd.setCursor(7, 1);
77        lcd.print(currentSpeedKmh, 1);
78
79        if (currentSpeedKmh > speedLimitKmhArduino) {
80          if (currentTimeMs - lastSignalTimeMs > signalDebounceMs) {
81            Serial.print("Speeding detected! Speed: ");
82            Serial.print(currentSpeedKmh);
83            Serial.println(" km/h. Signaling Pi.");
84
85            digitalWrite(raspberryPiSignalPin, HIGH);
86            delay(500);
87            digitalWrite(raspberryPiSignalPin, LOW);
88            lastSignalTimeMs = currentTimeMs;
89          }
90        }
91      }
92    } else if (currentDistanceCm != -1 && previousDistanceCm == -1) {
93      lcd.setCursor(7, 1);
94      lcd.print("---");
95      currentSpeedKmh = 0;
96    } else {
97      lcd.setCursor(7, 1);
98      lcd.print("---");
99      currentSpeedKmh = 0;
100   }
101
102   if (currentDistanceCm != -1) {
103     previousDistanceCm = currentDistanceCm;
104   } else {
105     previousDistanceCm = -1;
106   }
107   previousTimeMs = currentTimeMs;
108
109   delay(100);
110 }
```

Listing A.1: Arduino Code for Speed Measurement

Appendix B

Source Code Listings

B.1 Image Capture and Processing

```
import cv2
import numpy as np
import easyocr
import time
from datetime import datetime
import os
import json
from database.db_manager import DatabaseManager
from notification.email_sender import EmailSender

class SpeedMonitorDev:
    def __init__(self):
        # Initialize camera (using webcam for development)
        self.cap = cv2.VideoCapture(0)
        if not self.cap.isOpened():
            raise Exception("Could not open camera")

        # Initialize OCR
        self.reader = easyocr.Reader(['en'])

        # Initialize database and email sender
        self.db = DatabaseManager()
        self.email_sender = EmailSender()

        # Create directory for captured images
        self.image_dir = "captured_images"
        os.makedirs(self.image_dir, exist_ok=True)

        # Speed threshold (m/s)
```

```
30          self.speed_threshold = 7.0
31
32      def capture_image(self):
33          """Capture image from webcam"""
34          ret, frame = self.cap.read()
35          if not ret:
36              raise Exception("Failed to capture image")
37          return frame
38
39      def process_license_plate(self, image):
40          """Process image to detect and read license plate"""
41          # Convert to grayscale
42          gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
43
44          # Apply Gaussian blur
45          blur = cv2.GaussianBlur(gray, (5, 5), 0)
46
47          # Edge detection
48          edges = cv2.Canny(blur, 50, 150)
49
50          # Find contours
51          contours, _ = cv2.findContours(edges, cv2.RETR_EXTERNAL, cv2.
                ↪ CHAIN_APPROX_SIMPLE)
52
53          # Find potential license plate regions
54          for contour in contours:
55              x, y, w, h = cv2.boundingRect(contour)
56              aspect_ratio = w / float(h)
57
58              # License plates typically have aspect ratios between 2.0
                    ↪ and 5.0
59              if 2.0 < aspect_ratio < 5.0:
60                  plate_region = image[y:y+h, x:x+w]
61                  # Perform OCR on the region
62                  results = self.reader.readtext(plate_region)
63
64                  if results:
65                      return results[0][1]  # Return the detected text
66
67          return None
68
69      def save_violation(self, speed, license_plate, image):
70          """Save violation details and image"""
71          timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
72          image_path = os.path.join(self.image_dir, f"violation_{timestamp
                ↪ }.jpg")
73
```

```
74          # Save image
75          cv2.imwrite(image_path, image)
76
77          # Save to database
78          violation_data = {
79              "timestamp": timestamp,
80              "speed": speed,
81              "license_plate": license_plate,
82              "image_path": image_path
83          }
84
85          self.db.add_violation(violation_data)
86
87          # Get driver info and send email
88          driver_info = self.db.get_driver_info(license_plate)
89          if driver_info:
90              self.email_sender.send_violation_notification(
91                  driver_info["email"],
92                  speed,
93                  timestamp,
94                  image_path
95              )
96
97      def run(self):
98          """Main monitoring loop"""
99          print("Starting speed monitoring system (Development Mode)...")
100         print("Press 'q' to quit")
101
102         try:
103             while True:
104                 # Capture image
105                 image = self.capture_image()
106
107                 # Display the image
108                 cv2.imshow('Speed Monitor', image)
109
110                 # Process license plate
111                 license_plate = self.process_license_plate(image)
112
113                 if license_plate:
114                     print(f"Detected license plate: {license_plate}")
115                     # For development, simulate speed detection
116                     speed = 10.0  # Simulated speed
117
118                     if speed > self.speed_threshold:
119                         self.save_violation(speed, license_plate, image)
120
```

```
                # Check for quit command
                if cv2.waitKey(1) & 0xFF == ord('q'):
                    break

                time.sleep(0.1)  # Small delay to prevent CPU overload

        except KeyboardInterrupt:
            print("\nStopping speed monitoring system...")
        finally:
            self.cap.release()
            cv2.destroyAllWindows()

if __name__ == "__main__":
    monitor = SpeedMonitorDev()
    monitor.run()
```

Listing B.1: Image Capture Script for Development Environment

```
import sys
import platform

if platform.system() == "Linux" and ("arm" in platform.machine() or "
    ↪ aarch64" in platform.machine()):
    import cv2
    import numpy as np
    import easyocr
    import RPi.GPIO as GPIO
    import time
    from picamera2 import Picamera2
    from datetime import datetime
    import os
    import json
    from database.db_manager import DatabaseManager
    from notification.email_sender import EmailSender

    class SpeedMonitor:
        def __init__(self):
            # Initialize GPIO
            GPIO.setmode(GPIO.BCM)
            self.speed_pin = 17  # GPIO pin for speed data
            GPIO.setup(self.speed_pin, GPIO.IN)

            # Initialize camera
            self.picam2 = Picamera2()
            self.picam2.configure(self.picam2.
                ↪ create_preview_configuration(
                main={"format": 'XRGB8888', "size": (640, 480)}))
```

```
28              self.picam2.start()
29
30              # Initialize OCR
31              self.reader = easyocr.Reader(['en'])
32
33              # Initialize database and email sender
34              self.db = DatabaseManager()
35              self.email_sender = EmailSender()
36
37              # Create directory for captured images
38              self.image_dir = "captured_images"
39              os.makedirs(self.image_dir, exist_ok=True)
40
41              # Speed threshold (m/s)
42              self.speed_threshold = 7.0
43
44          def capture_image(self):
45              """Capture image from PiCamera"""
46              frame = self.picam2.capture_array()
47              return cv2.cvtColor(frame, cv2.COLOR_RGB2BGR)
48
49          def process_license_plate(self, image):
50              """Process image to detect and read license plate"""
51              # Convert to grayscale
52              gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
53
54              # Apply Gaussian blur
55              blur = cv2.GaussianBlur(gray, (5, 5), 0)
56
57              # Edge detection
58              edges = cv2.Canny(blur, 50, 150)
59
60              # Find contours
61              contours, _ = cv2.findContours(edges, cv2.RETR_EXTERNAL, cv2
                    ↪ .CHAIN_APPROX_SIMPLE)
62
63              # Find potential license plate regions
64              for contour in contours:
65                  x, y, w, h = cv2.boundingRect(contour)
66                  aspect_ratio = w / float(h)
67
68                  # License plates typically have aspect ratios between
                        ↪ 2.0 and 5.0
69                  if 2.0 < aspect_ratio < 5.0:
70                      plate_region = image[y:y+h, x:x+w]
71                      # Perform OCR on the region
72                      results = self.reader.readtext(plate_region)
```

```
73
74                  if results:
75                      return results[0][1]  # Return the detected text
76
77          return None
78
79      def save_violation(self, speed, license_plate, image):
80          """Save violation details and image"""
81          timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
82          image_path = os.path.join(self.image_dir, f"violation_{
                ↪ timestamp}.jpg")
83
84          # Save image
85          cv2.imwrite(image_path, image)
86
87          # Save to database
88          violation_data = {
89              "timestamp": timestamp,
90              "speed": speed,
91              "license_plate": license_plate,
92              "image_path": image_path
93          }
94
95          self.db.add_violation(violation_data)
96
97          # Get driver info and send email
98          driver_info = self.db.get_driver_info(license_plate)
99          if driver_info:
100             self.email_sender.send_violation_notification(
101                 driver_info["email"],
102                 speed,
103                 timestamp,
104                 image_path
105             )
106
107     def run(self):
108         """Main monitoring loop"""
109         print("Starting speed monitoring system...")
110
111         try:
112             while True:
113                 # Check if speed exceeds threshold
114                 if GPIO.input(self.speed_pin):
115                     # Capture image
116                     image = self.capture_image()
117
118                     # Process license plate
```

```
                        license_plate = self.process_license_plate(image
                            ↪ )

                        if license_plate:
                            # Get speed from Arduino (implement serial
                                ↪ communication)
                            speed = 0  # TODO: Implement serial
                                ↪ communication

                            if speed > self.speed_threshold:
                                self.save_violation(speed, license_plate
                                    ↪ , image)

                    time.sleep(0.1)  # Small delay to prevent CPU
                        ↪ overload

            except KeyboardInterrupt:
                print("\nStopping speed monitoring system...")
                GPIO.cleanup()
                self.picam2.stop()

    if __name__ == "__main__":
        monitor = SpeedMonitor()
        monitor.run()
else:
    # Dummy class for non-Raspberry Pi systems to avoid import errors
    class SpeedMonitor:
        def __init__(self):
            print("SpeedMonitor is only supported on Raspberry Pi (Linux
                ↪ , ARM). This is a dummy class.")
        def run(self):
            print("SpeedMonitor cannot run on this platform.")
```

Listing B.2: Image Capture Script for Raspberry Pi (Production)

B.2 Image Processing Demonstration Script

```
import argparse
import cv2
from database.db_manager import DatabaseManager
from notification.email_sender import EmailSender
import easyocr
import os
from datetime import datetime

SPEED_THRESHOLD = 7.0  # m/s
```

```
10
11
12 def process_license_plate(image_path):
13     image = cv2.imread(image_path)
14     if image is None:
15         print(f"Could not read image: {image_path}")
16         return None
17     reader = easyocr.Reader(['en'])
18     gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
19     blur = cv2.GaussianBlur(gray, (5, 5), 0)
20     edges = cv2.Canny(blur, 50, 150)
21     contours, _ = cv2.findContours(edges, cv2.RETR_EXTERNAL, cv2.
           ↪ CHAIN_APPROX_SIMPLE)
22     for contour in contours:
23         x, y, w, h = cv2.boundingRect(contour)
24         aspect_ratio = w / float(h)
25         if 2.0 < aspect_ratio < 5.0:
26             plate_region = image[y:y+h, x:x+w]
27             results = reader.readtext(plate_region)
28             if results:
29                 return results[0][1]
30     return None
31
32 def main():
33     parser = argparse.ArgumentParser(description="Process demo image and
           ↪  speed.")
34     parser.add_argument('--image', required=True, help='Path to the
           ↪ image file')
35     parser.add_argument('--speed', type=float, required=True, help='
           ↪ Speed value (m/s)')
36     args = parser.parse_args()
37
38     print(f"Received speed: {args.speed} m/s")
39     if args.speed > SPEED_THRESHOLD:
40         print(f"Speed exceeds threshold ({SPEED_THRESHOLD} m/s).
               ↪ Processing image...")
41         license_plate = process_license_plate(args.image)
42         if license_plate:
43             print(f"Detected license plate: {license_plate}")
44             # Save violation
45             db = DatabaseManager()
46             email_sender = EmailSender()
47             timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
48             image_dir = "captured_images"
49             os.makedirs(image_dir, exist_ok=True)
50             image_path = os.path.join(image_dir, f"violation_{timestamp
                   ↪ }.jpg")
```

```
            cv2.imwrite(image_path, cv2.imread(args.image))
            violation_data = {
                "timestamp": timestamp,
                "speed": args.speed,
                "license_plate": license_plate,
                "image_path": image_path
            }
            db.add_violation(violation_data)
            driver_info = db.get_driver_info(license_plate)
            if driver_info:
                email_sender.send_violation_notification(
                    driver_info["email"],
                    args.speed,
                    timestamp,
                    image_path
                )
                print(f"Violation logged and email sent to {driver_info
                    ↪ ['email']}")
            else:
                print("Driver not found in database. No email sent.")
        else:
            print("No license plate detected.")
    else:
        print(f"Speed is within the limit ({SPEED_THRESHOLD} m/s). No
            ↪ action taken.")

if __name__ == "__main__":
    main()
```

Listing B.3: Standalone Image Processing Demonstration Script

B.3 User Interface Main App

```
from flask import Flask, render_template, jsonify, request
from database.db_manager import DatabaseManager
import os

app = Flask(__name__)
db = DatabaseManager()

@app.route('/')
def index():
    """Render main dashboard page"""
    return render_template('index.html')

@app.route('/api/violations')
```

```
def get_violations():
    """Get recent violations"""
    violations = db.get_violations(limit=10)
    return jsonify(violations)

@app.route('/api/top-speeders')
def get_top_speeders():
    """Get top speeders"""
    speeders = db.get_top_speeders(limit=5)
    return jsonify(speeders)

# --- CRUD for Drivers ---
@app.route('/api/drivers', methods=['GET'])
def list_drivers():
    drivers = db.get_all_drivers() if hasattr(db, 'get_all_drivers')
        ↪ else []
    return jsonify(drivers)

@app.route('/api/drivers', methods=['POST'])
def create_driver():
    data = request.json
    name = data.get('name')
    license_plate = data.get('license_plate')
    email = data.get('email')
    if not (name and license_plate and email):
        return jsonify({'error': 'Missing fields'}), 400
    success = db.add_driver(name, license_plate, email)
    if success:
        return jsonify({'message': 'Driver added successfully'}), 201
    else:
        return jsonify({'error': 'Driver already exists or error
            ↪ occurred'}), 400

@app.route('/api/drivers/<license_plate>', methods=['PUT'])
def update_driver(license_plate):
    data = request.json
    name = data.get('name')
    email = data.get('email')
    success = db.update_driver(license_plate, name, email) if hasattr(db
        ↪ , 'update_driver') else False
    if success:
        return jsonify({'message': 'Driver updated successfully'})
    else:
        return jsonify({'error': 'Driver not found or error occurred'}),
            ↪  404

@app.route('/api/drivers/<license_plate>', methods=['DELETE'])
```

```
def delete_driver(license_plate):
    success = db.delete_driver(license_plate) if hasattr(db, '
        ↪ delete_driver') else False
    if success:
        return jsonify({'message': 'Driver deleted successfully'})
    else:
        return jsonify({'error': 'Driver not found or error occurred'}),
            ↪  404

# --- CRUD for Violations (Delete only) ---
@app.route('/api/violations/<int:violation_id>', methods=['DELETE'])
def delete_violation(violation_id):
    success = db.delete_violation(violation_id) if hasattr(db, '
        ↪ delete_violation') else False
    if success:
        return jsonify({'message': 'Violation deleted successfully'})
    else:
        return jsonify({'error': 'Violation not found or error occurred'
            ↪ }), 404

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000, debug=True)
```

Listing B.4: Main Application Script for User Interface

B.4 Database Management

```
import sqlite3
from datetime import datetime
import os

class DatabaseManager:
    def __init__(self, db_path="speed_monitor.db"):
        self.db_path = db_path
        self.init_database()

    def init_database(self):
        """Initialize database with required tables"""
        conn = sqlite3.connect(self.db_path)
        cursor = conn.cursor()

        # Create drivers table
        cursor.execute('''
        CREATE TABLE IF NOT EXISTS drivers (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            name TEXT NOT NULL,
```

```
            license_plate TEXT UNIQUE NOT NULL,
            email TEXT NOT NULL,
            violation_count INTEGER DEFAULT 0,
            created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
        )
        ''')

        # Create violations table
        cursor.execute('''
        CREATE TABLE IF NOT EXISTS violations (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            driver_id INTEGER,
            speed REAL NOT NULL,
            timestamp TIMESTAMP NOT NULL,
            image_path TEXT NOT NULL,
            FOREIGN KEY (driver_id) REFERENCES drivers (id)
        )
        ''')

        conn.commit()
        conn.close()

    def add_driver(self, name, license_plate, email):
        """Add a new driver to the database"""
        conn = sqlite3.connect(self.db_path)
        cursor = conn.cursor()

        try:
            cursor.execute('''
            INSERT INTO drivers (name, license_plate, email)
            VALUES (?, ?, ?)
            ''', (name, license_plate, email))
            conn.commit()
            return True
        except sqlite3.IntegrityError:
            return False
        finally:
            conn.close()

    def get_driver_info(self, license_plate):
        """Get driver information by license plate"""
        conn = sqlite3.connect(self.db_path)
        cursor = conn.cursor()

        cursor.execute('''
        SELECT id, name, email, violation_count
        FROM drivers
```

```
        WHERE license_plate = ?
        ''', (license_plate,))

        result = cursor.fetchone()
        conn.close()

        if result:
            return {
                "id": result[0],
                "name": result[1],
                "email": result[2],
                "violation_count": result[3]
            }
        return None

    def add_violation(self, violation_data):
        """Add a new violation record"""
        conn = sqlite3.connect(self.db_path)
        cursor = conn.cursor()

        try:
            # Get driver ID
            cursor.execute('''
            SELECT id FROM drivers WHERE license_plate = ?
            ''', (violation_data["license_plate"],))

            driver_id = cursor.fetchone()

            if driver_id:
                # Add violation record
                cursor.execute('''
                INSERT INTO violations (driver_id, speed, timestamp,
                    ↪ image_path)
                VALUES (?, ?, ?, ?)
                ''', (driver_id[0], violation_data["speed"],
                      violation_data["timestamp"], violation_data["
                          ↪ image_path"]))

                # Update violation count
                cursor.execute('''
                UPDATE drivers
                SET violation_count = violation_count + 1
                WHERE id = ?
                ''', (driver_id[0],))

                conn.commit()
                return True
```

```
            return False
        finally:
            conn.close()

    def get_violations(self, limit=10):
        """Get recent violations"""
        conn = sqlite3.connect(self.db_path)
        cursor = conn.cursor()

        cursor.execute('''
        SELECT v.id, v.timestamp, v.speed, d.name, d.license_plate, v.
            ↪ image_path
        FROM violations v
        JOIN drivers d ON v.driver_id = d.id
        ORDER BY v.timestamp DESC
        LIMIT ?
        ''', (limit,))

        violations = cursor.fetchall()
        conn.close()

        return [{
            "id": v[0],
            "timestamp": v[1],
            "speed": v[2],
            "driver_name": v[3],
            "license_plate": v[4],
            "image_path": v[5]
        } for v in violations]

    def get_top_speeders(self, limit=5):
        """Get top speeders based on violation count"""
        conn = sqlite3.connect(self.db_path)
        cursor = conn.cursor()

        cursor.execute('''
        SELECT name, license_plate, violation_count
        FROM drivers
        ORDER BY violation_count DESC
        LIMIT ?
        ''', (limit,))

        speeders = cursor.fetchall()
        conn.close()

        return [{
            "name": s[0],
```

```
            "license_plate": s[1],
            "violation_count": s[2]
        } for s in speeders]

    def get_all_drivers(self):
        """Get all drivers"""
        conn = sqlite3.connect(self.db_path)
        cursor = conn.cursor()
        cursor.execute('''
        SELECT name, license_plate, email, violation_count, created_at
        FROM drivers
        ORDER BY created_at DESC
        ''')
        drivers = cursor.fetchall()
        conn.close()
        return [{
            "name": d[0],
            "license_plate": d[1],
            "email": d[2],
            "violation_count": d[3],
            "created_at": d[4]
        } for d in drivers]

    def update_driver(self, license_plate, name=None, email=None):
        """Update driver information"""
        conn = sqlite3.connect(self.db_path)
        cursor = conn.cursor()
        updates = []
        params = []
        if name:
            updates.append("name = ?")
            params.append(name)
        if email:
            updates.append("email = ?")
            params.append(email)
        if not updates:
            conn.close()
            return False
        params.append(license_plate)
        query = f"UPDATE drivers SET {', '.join(updates)} WHERE
            ↪ license_plate = ?"
        cursor.execute(query, params)
        conn.commit()
        updated = cursor.rowcount > 0
        conn.close()
        return updated

```

```
    def delete_driver(self, license_plate):
        """Delete a driver by license plate"""
        conn = sqlite3.connect(self.db_path)
        cursor = conn.cursor()
        cursor.execute('''DELETE FROM drivers WHERE license_plate = ?''', (license_plate,))
        conn.commit()
        deleted = cursor.rowcount > 0
        conn.close()
        return deleted

    def delete_violation(self, violation_id):
        """Delete a violation by id"""
        conn = sqlite3.connect(self.db_path)
        cursor = conn.cursor()
        cursor.execute('''DELETE FROM violations WHERE id = ?''', (violation_id,))
        conn.commit()
        deleted = cursor.rowcount > 0
        conn.close()
        return deleted
```

Listing B.5: Database Manager Script

B.5 Email Notification System

```
import smtplib
from email.mime.text import MIMEText
from email.mime.multipart import MIMEMultipart
from email.mime.image import MIMEImage
import os
from datetime import datetime

class EmailSender:
    def __init__(self, smtp_server="smtp.gmail.com", smtp_port=587):
        self.smtp_server = smtp_server
        self.smtp_port = smtp_port
        self.sender_email = os.getenv("EMAIL_USER")
        self.sender_password = os.getenv("EMAIL_PASSWORD")

    def send_violation_notification(self, recipient_email, speed, timestamp, image_path):
        """Send violation notification email"""
        if not all([self.sender_email, self.sender_password]):
            print("Email credentials not configured")
            return False
```

```
20
21          try:
22              # Create message
23              msg = MIMEMultipart()
24              msg['From'] = self.sender_email
25              msg['To'] = recipient_email
26              msg['Subject'] = "Speed Violation Notice"
27
28              # Format timestamp
29              violation_time = datetime.strptime(timestamp, "%Y%m%d_%H%M%S
                    ↪ ")
30              formatted_time = violation_time.strftime("%Y-%m-%d %H:%M:%S"
                    ↪ )
31
32              # Create email body
33              body = f"""
34              Dear Driver,
35
36              This is an automated notification regarding a speed
                    ↪ violation detected on {formatted_time}.
37
38              Details of the violation:
39              - Speed: {speed:.1f} m/s
40              - Time: {formatted_time}
41
42              Please find attached the violation image for your reference.
43
44              This is an automated system. If you believe this is an error
                    ↪ , please contact the traffic department.
45
46              Best regards,
47              Traffic Monitoring System
48              """
49
50              msg.attach(MIMEText(body, 'plain'))
51
52              # Attach violation image
53              with open(image_path, 'rb') as f:
54                  img = MIMEImage(f.read())
55                  img.add_header('Content-Disposition', 'attachment',
56                                 filename=os.path.basename(image_path))
57                  msg.attach(img)
58
59              # Send email
60              with smtplib.SMTP(self.smtp_server, self.smtp_port) as
                    ↪ server:
61                  server.starttls()
```

```
                server.login(self.sender_email, self.sender_password)
                server.send_message(msg)

            print(f"Violation notification sent to {recipient_email}")
            return True

        except Exception as e:
            print(f"Error sending email: {str(e)}")
            return False
```

Listing B.6: Automated Email Notification Sender