ELEPHANT DETECTION AND ALERT SYSTEM

MINI PROJECT REPORT

Submitted in partial fulfillment of the requirements for the award of the Degree of ***Bachelor of Technology*** *in* ***Electronics & Communication Engineering*** *of* ***APJ Abdul Kalam Technological University***

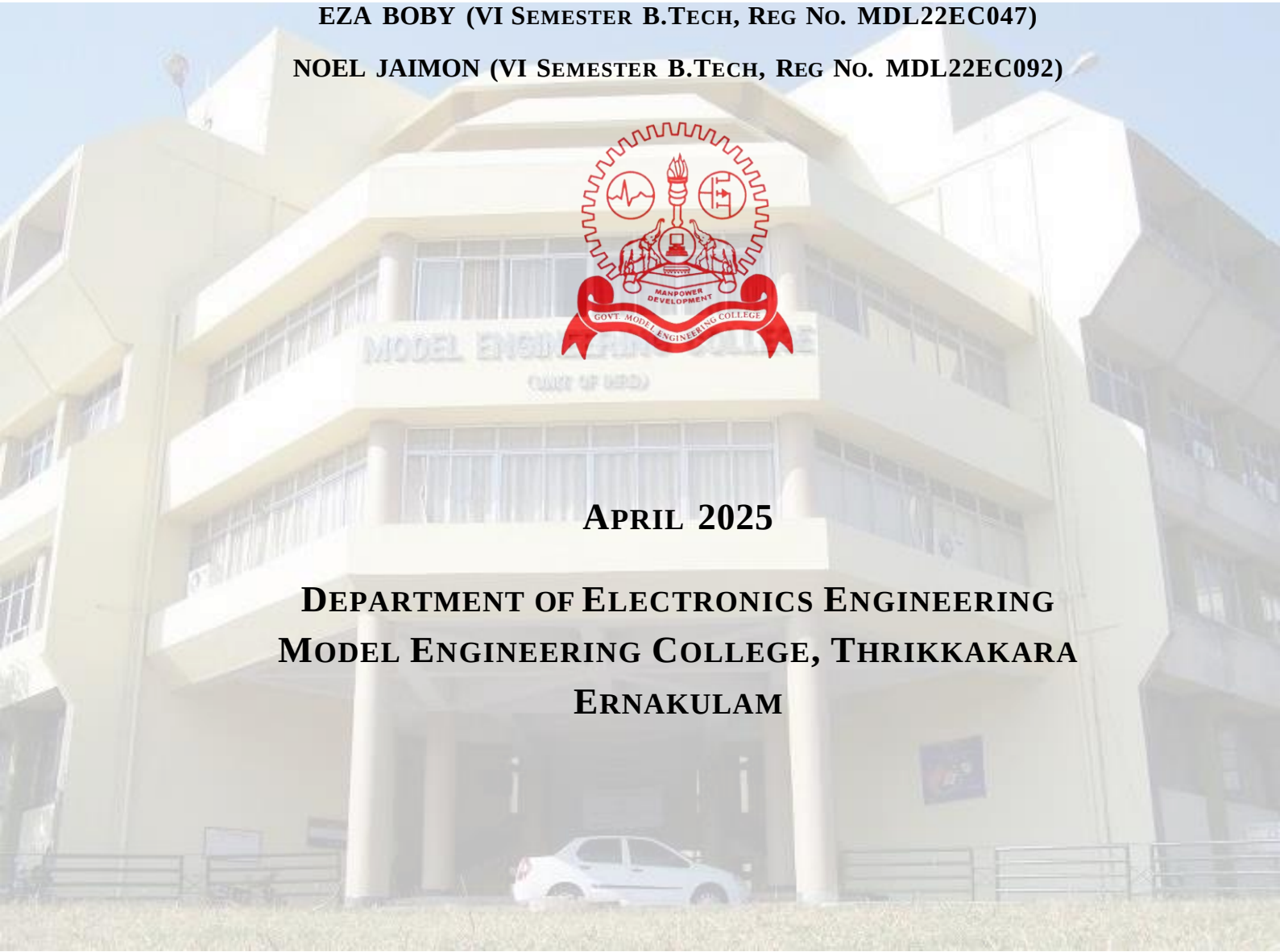
By

ASWIN AJITH (VI SEMESTER B.TECH, REG NO. MDL22EC036)

BENJAMIN JOSHY (VI SEMESTER B.TECH, REG NO. MDL22EC039)

EZA BOBY (VI SEMESTER B.TECH, REG NO. MDL22EC047)

NOEL JAIMON (VI SEMESTER B.TECH, REG NO. MDL22EC092)



APRIL 2025

DEPARTMENT OF ELECTRONICS ENGINEERING

MODEL ENGINEERING COLLEGE, THRIKKAKARA

ERNAKULAM

MODEL ENGINEERING COLLEGE, THRIKKAKARA

DEPARTMENT OF ELECTRONICS ENGINEERING

CERTIFICATE

This is to certify that the project report entitled **"ELEPHANT DETECTION AND ALERT SYSTEM"** *is a bonafide record of the project work done by* **ASWIN AJITH** *(VI Semester B.Tech, Reg No. MDL22EC036)* **BENJAMIN JOSHY** *(VI Semester B.Tech, Reg No. MDL22EC039)* **EZA BOBY** *(VI Semester B.Tech, Reg No. MDL22EC047)* **NOEL JAIMON** *(VI Semester B.Tech, Reg No. MDL22EC092) towards the partial fulfillment of the requirements for the award of the Degree of Bachelor of Technology in Electronics & Communication Engineering of APJ Abdul Kalam Technological University during the year 2025.*

PROJECT CO-ORDINATOR

MANOJKUMAR P
ASST. PROFESSOR
DEPT. OF ELECTRONICS

PROJECT GUIDE

DR. SUMITHA MATHEW
ASSOCIATE PROFESSOR
DEPT. OF ELECTRONICS

HEAD OF DEPARTMENT

PRADEEP M
DEPT. OF ELECTRONICS ENGINEERING

ACKNOWLEDGEMENT

*First of all, I would like to thank the **Lord Almighty** who helped me to finish this project on time.*

I express my sincere thanks to , The Principal, Model Engineering College, Thrikkakara, for providing opportunity and the environment to do the project in my college.

I sincerely thank , Head of the Department, Dept. of Electronics, for his encouragement and constant support in making project successful.

*I would like to thank my class coordinator **Mrs. Sheeba P.S.** , Asst. Professor, Dept. of Electronics, for giving me timely instruction,for the completion the work.*

*I would like to thank my project coordinator **Mr. Manojkumar P.** , Asst. Professor, Dept. of Electronics, for giving me technical advice, without which I could never been able to complete the work in time.*

*I also wish to thank my project guide **Dr. Sumitha Mathew** , Associate. Professor, Dept. of Electronics, for providing valuable guidance.*

An excellent group of teaching and non-teaching staff helped me for this project. I owe much the assistance they gave me while doing the project.

Last, but not least I would like to thank my parents and friends for all the moral support and that they have given me.

Aswin Ajith (MDL22EC036)

Benjamin Joshy (MDL22EC039)

Eza Boby (MDL22EC047)

Noel Jaimon (MDL22EC092)

ABSTRACT

Human-elephant conflict is a major issue in regions where elephant intrusions threaten human settlements, leading to loss of life, crop damage, and economic hardships. To mitigate this problem, we propose an Elephant Detection and Warning System using Raspberry Pi and the YOLOv8 deep learning model for real-time object detection. The system consists of a camera module that continuously captures video feeds from forest fringes or agricultural lands. The Raspberry Pi processes these feeds using the YOLOv8 model to accurately detect the presence of elephants. Upon detection, the system automatically triggers an alert through a Telegram bot, instantly notifying local villagers and authorities. This enables timely preventive measures, reducing potential conflicts and ensuring both human and wildlife safety. The system is designed to be cost-effective, scalable, and easy to deploy, making it a practical solution for regions affected by human-elephant conflict.

Contents

List of Figures

List of Tables

Chapter 1

INTRODUCTION

Human-elephant conflict is a growing concern in regions where elephants encroach upon human settlements, leading to property damage, crop loss, and fatalities. To address this, we propose an Elephant Detection and Warning System using Raspberry Pi and YOLOv8 for real-time detection. The system processes live video feeds to identify elephants and instantly notifies local villagers via a Telegram bot, enabling timely action. This cost-effective, automated solution enhances safety and minimizes conflicts between humans and wildlife.[**?**]

1.1 Background of the Project

Human-elephant conflict (HEC) has become a critical issue in regions where elephant habitats overlap with human settlements. As human populations expand and forests shrink due to deforestation and agricultural activities, elephants often stray into villages in search of food, leading to property damage, crop destruction, and even loss of human and animal lives. Traditional methods such as manual monitoring, fences, and firecrackers have proven ineffective or unsustainable, necessitating technological solutions for early warning and conflict prevention. Advancements in computer vision and artificial intelligence provide an opportunity to develop automated wildlife monitoring systems. Object detection models like YOLOv8 offer high accuracy and real-time processing capabilities, making them suitable for identifying elephants in live video feeds. Single-board computers like Raspberry Pi enable cost-effective deployment in remote areas, reducing dependence on expensive infrastructure. By integrating these technologies, an automated detection and warning system can enhance human preparedness against elephant intrusions. Our proposed Elephant Detection and Warning System leverages Raspberry Pi and YOLOv8 to detect elephants in real-time and alert local communities via a Telegram bot. This system provides instant notifications, allowing villagers to take preven-

tive measures such as securing crops, avoiding elephant paths, and notifying authorities. The low-cost, scalable, and efficient nature of this system makes it a viable solution for minimizing human-elephant conflicts while promoting coexistence between humans and wildlife. [**?**]

1.2 Motivation

Human-elephant conflict (HEC) is a pressing issue in many regions, leading to loss of lives, destruction of property, and negative impacts on both human and elephant populations. With increasing habitat destruction due to deforestation and urban expansion, elephants are forced to venture into villages and farmlands in search of food. Traditional methods such as manual monitoring, electric fences, and loud noise deterrents have proven to be either ineffective, expensive, or harmful to the environment. There is a growing need for an automated, cost-effective, and reliable solution to detect and prevent elephant intrusions before they cause damage. Recent advancements in artificial intelligence and embedded systems provide a promising way to address this issue. Object detection models like YOLOv8 offer real-time processing with high accuracy, making them suitable for detecting elephants in video feeds. Additionally, Raspberry Pi, being affordable and low-power, enables remote deployment in affected areas. By integrating these technologies, we can create a real-time warning system that alerts villagers and authorities about elephant intrusions, allowing them to take timely action. This project is motivated by the goal of enhancing human and wildlife safety through technology. By developing an automated Elephant Detection and Warning System, we aim to reduce human-elephant conflicts, protect farmlands and villages, and ensure a peaceful coexistence between humans and elephants. The system's affordability, scalability, and real-time capabilities make it a practical and impactful solution for regions struggling with HEC.

1.3 Importance of the problem

Human-elephant conflict (HEC) is a critical issue that affects millions of people, especially in rural and forest-adjacent areas. Elephants, in search of food and water, often enter villages and farmlands, causing extensive damage to crops, property, and infrastructure. In

some cases, these encounters lead to human fatalities and injuries, creating tension between local communities and wildlife conservation efforts. Reducing HEC is essential not only for the safety of humans but also for the well-being and protection of elephants, as retaliatory actions against them are often deadly. The economic impact of HEC is substantial, particularly in agricultural communities where crop damage can destroy livelihoods. Farmers, who depend on their harvests for survival, face immense losses, leading to increased poverty and frustration. As the human population grows and habitats shrink, these conflicts are likely to become more frequent, worsening the already fragile relationship between communities and wildlife. Traditional methods of conflict mitigation, such as fences, firecrackers, and manual monitoring, have proven costly and ineffective over time. These approaches are often resource-intensive and lack real-time capabilities, leaving communities vulnerable to unexpected elephant intrusions. The need for a more sustainable, efficient, and affordable solution has never been greater.

1.4 Objective and Scope

The primary objective of this project is to develop an Elephant Detection and Warning System that uses a Raspberry Pi combined with the YOLOv8 object detection model to detect elephants in real-time. The key objectives are:

1. Detect elephants using real-time video feeds captured by cameras placed in vulnerable areas.
2. Process video feeds through the YOLOv8 model for accurate and efficient elephant identification.
3. Send real-time alerts to local villagers and authorities through a Telegram bot to inform them of elephant presence.
4. Enable timely action by providing early warnings to prevent property damage, injuries, and fatalities.
5. Reduce human-elephant conflict by offering a sustainable, automated solution for monitoring and alerting.

This system aims to empower communities with early detection and quick response mechanisms, promoting safety and coexistence between humans and elephants. The scope of this project includes the design and implementation of a cost-effective, automated detection system that can be deployed in regions where human-elephant conflict is prevalent. The system will focus on detecting elephants in specific areas, such as near agricultural fields or villages, using real-time video feeds. It will send alerts via a Telegram bot to local communities and authorities when an elephant is detected, allowing for quick responses. The system is scalable and can be expanded to cover larger areas or adapted for other wildlife monitoring purposes. Additionally, the system is designed to be low-power, making it suitable for remote deployment.

Chapter 2

LITERATURE REVIEW

Human-elephant conflict (HEC) has been a significant issue in countries where elephant habitats overlap with human settlements, particularly in regions such as South Asia and Africa. Studies have shown that the expansion of agricultural land, coupled with human encroachment into forested areas, forces elephants into close contact with human populations, leading to frequent conflicts. The World Wildlife Fund (WWF) reports that these conflicts result in significant economic losses, both in terms of property damage and agricultural crop destruction, as well as human and elephant casualties. Various traditional methods of managing HEC, such as electric fences and watchtowers, have been employed with limited success due to their high costs, inefficiency, and potential harm to the environment. There is, therefore, a growing need for alternative, more effective, and sustainable solutions.

In recent years, technology has been applied to monitor wildlife and reduce human-wildlife conflict. The use of computer vision and artificial intelligence has gained momentum in detecting and tracking animals in real-time. Object detection models like YOLO (You Only Look Once) have proven effective in various wildlife monitoring applications, including detecting poaching activities and identifying endangered species. YOLOv8, a real-time, high-accuracy version of this model, has demonstrated exceptional performance in detecting objects in real-world scenarios with minimal computational requirements, making it suitable for low-cost devices such as Raspberry Pi. Previous work has utilized YOLO models for identifying large animals like elephants and rhinoceroses, showcasing their capability to provide real-time alerts and data to relevant stakeholders. Raspberry Pi, a low-cost, energy-efficient, and versatile single-board computer, has been widely used in environmental monitoring systems due to its affordability and ease of integration with various sensors and devices. Several research projects have successfully utilized Raspberry Pi in conjunction with camera modules for wildlife monitoring, including the detection of elephants. The small size and low power consumption of

Raspberry Pi make it an ideal choice for remote deployment in areas where power supply and infrastructure are limited. This affordability and flexibility make it a suitable tool for rural and developing regions that face human-elephant conflicts.

Further integration of these technologies with communication platforms like Telegram allows for quick dissemination of alerts. Previous studies on wildlife monitoring have demonstrated the effectiveness of using messaging services for real-time alerts. For instance, systems that alert local authorities via SMS or WhatsApp have been used to manage elephant movements. By integrating the Telegram bot with the detection system, the proposed solution can efficiently notify villagers about elephant presence, allowing them to take preventive actions.

In conclusion, the integration of YOLOv8, Raspberry Pi, and Telegram bot notifications presents a promising approach to reducing human-elephant conflict. By leveraging advancements in computer vision and affordable embedded systems, this project aims to provide an automated, cost-effective, and scalable solution for early detection and warning systems, improving safety and reducing the economic impact of HEC in vulnerable regions.

Chapter 3

PROBLEM STATEMENT AND PROPOSED SOLUTION

Human-elephant conflict (HEC) is a growing issue, especially in areas where elephants intrude on human settlements, causing crop damage and posing risks to human lives. Traditional methods like electric fences and manual monitoring are ineffective and costly. To address this, we propose an Elephant Detection and Warning System using Raspberry Pi and YOLOv8 object detection. The system will process real-time video feeds to detect elephants and send instant alerts via a Telegram bot. This cost-effective, automated solution aims to reduce HEC, ensuring the safety of both humans and elephants by providing timely warnings.

3.1 Problem Statement

Human-elephant conflict (HEC) is a growing issue in regions where elephants encroach on human settlements, causing crop damage, property destruction, and even fatalities. As elephant habitats shrink due to deforestation and urban expansion, elephants increasingly enter villages and farmlands in search of food and water. Traditional methods, such as electric fences, watchtowers, and manual monitoring, are costly, ineffective, and not scalable, often failing to provide timely responses. These systems lack real-time detection and proactive alert mechanisms, leaving communities vulnerable to unexpected elephant intrusions. The absence of early warning systems further exacerbates the issue. There is a pressing need for an affordable, automated solution that can detect elephants and alert local communities in real time, enabling them to take preventive actions and mitigate the risks of damage and harm.

1. Human-elephant conflict leads to property damage, crop destruction, and fatalities.
2. Traditional methods like electric fences and manual monitoring are inefficient and expensive.

3. Lack of real-time detection and early warning systems leaves communities vulnerable.
4. Growing elephant population and shrinking habitats increase the frequency of conflicts.
5. There is a need for an affordable, automated solution to reduce the risks and provide timely alerts to local communities.

3.2 Proposed Solution

To address the growing issue of human-elephant conflict (HEC), we propose an Elephant Detection and Warning System using Raspberry Pi and the YOLOv8 object detection model. This system will process real-time video feeds from cameras placed in elephant-prone areas, accurately detecting elephants and triggering automated alerts. The alerts will be sent to local villagers and authorities through a Telegram bot, enabling timely action to mitigate potential damage or harm. This solution is cost-effective, scalable, and easy to deploy, providing an automated, proactive approach to prevent HEC.

3.2.1 How Elephant Detection solves the problem

1. **Real-time detection:** Utilizes YOLOv8 for accurate and immediate detection of elephants in video feeds, providing timely alerts.
2. **Affordable:** Uses Raspberry Pi, a low-cost, energy-efficient platform, making it accessible for remote, resource-limited areas.
3. **Automated alerts:** Sends instant notifications via a Telegram bot, enabling quick responses from local communities and authorities.
4. **Scalable solution:** Can be easily deployed in multiple locations, making it suitable for large areas prone to elephant intrusions.
5. **Proactive approach:** Reduces reliance on manual monitoring by providing early warnings, helping to prevent crop damage, property destruction, and human-elephant fatalities.

3.3 Figure

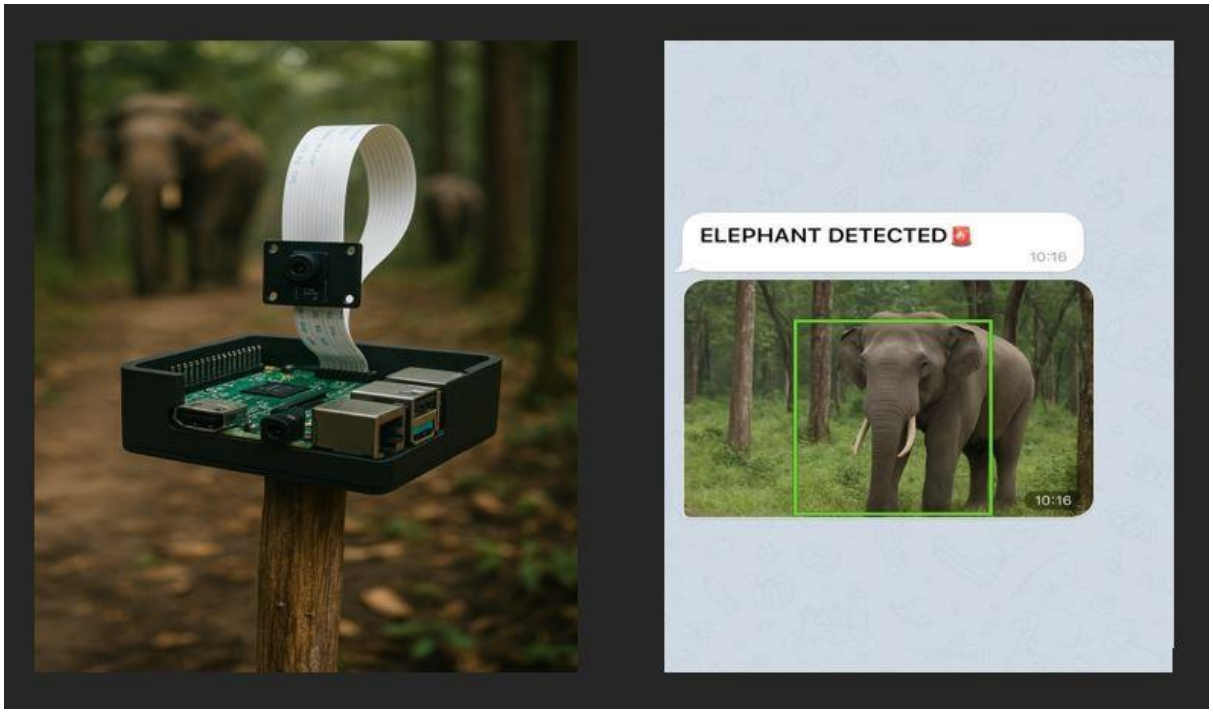


Figure 3.1: Proposed Solution

Chapter 4

BLOCK DIAGRAM AND EXPLANATION

4.1 Block Diagram

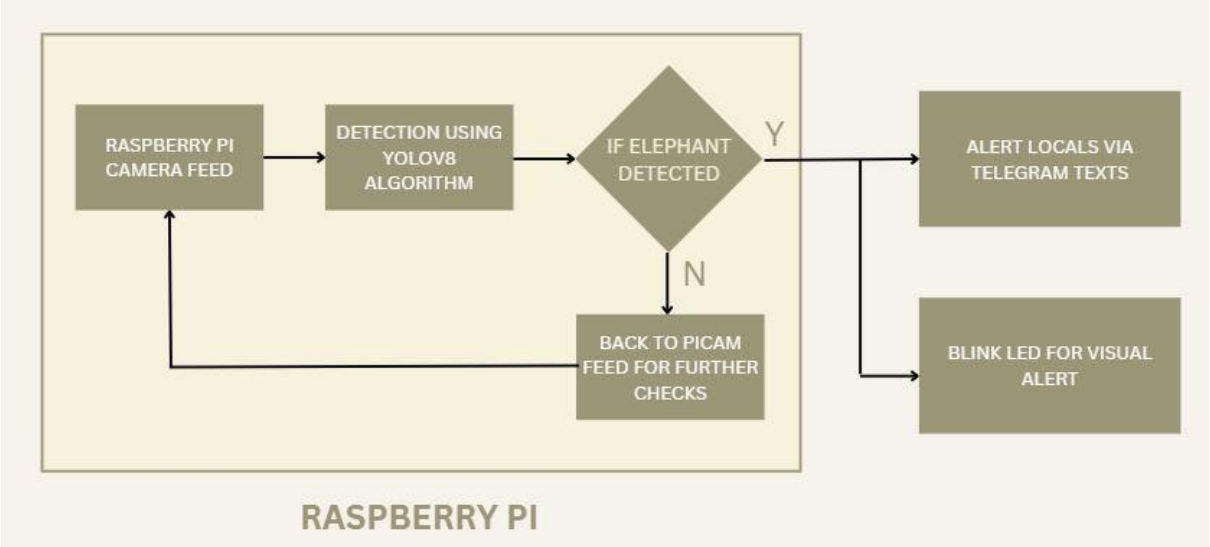


Figure 4.1: Block Diagram

4.2 System Overview

The Elephant Detection and Warning System is designed to detect elephants in real-time and alert local communities to prevent human-elephant conflicts. The system uses a camera module to capture video feeds from high-risk areas, which are processed by a Raspberry Pi running the YOLOv8 object detection model. If an elephant is detected, the system immediately triggers an automated alert via a Telegram bot, notifying villagers and authorities to take preventive action. The system is designed to be cost-effective, scalable, and power-efficient, making it suitable for deployment in remote locations where infrastructure is limited.

4.3 Component Breakdown

4.3.1 Hardware Components:

1. **Raspberry Pi 5:** Acts as the central processing unit for running the YOLOv8 model, handling image processing, and executing scripts for real-time detection and alerts. Its powerful performance and GPIO support make it ideal for IoT applications.

2. **Pi Camera Module:** Captures real-time video feed for object detection, providing a direct input to the detection algorithm. Its compact size and compatibility with the Raspberry Pi make it perfect for field use.

3. **Power Supply:** Ensures a consistent and reliable power source for the Raspberry Pi and camera module, enabling continuous operation in remote locations.

4. **Protective Enclosure/Mounting Setup:** Protects sensitive components from environmental elements like dust, moisture, and wildlife, ensuring durability and stability in harsh conditions.

5. **SD Card (64 GB or Higher):** Stores the operating system, YOLOv8 model, Python scripts, and captured images or videos. A high storage capacity ensures efficient data handling.

6. **Cooling Fan/Heat Sink:** Prevents overheating during prolonged operation by managing heat dissipation, ensuring optimal performance and stability.

4.3.2 Software Components:

1. **Raspberry Pi OS (64-bit):** Provides a stable and optimized environment for running Python scripts, handling device drivers, and managing hardware interfaces. It's well-suited for IoT and machine learning applications.

2. **YOLOv8 Object Detection Model (.onnx):** The trained model identifies elephants and

other objects with high accuracy. Using the ONNX format allows efficient model deployment on edge devices like the Raspberry Pi.

3. **OpenCV and Ultralytics Libraries:** OpenCV handles image processing and manipulation, while Ultralytics simplifies YOLOv8 implementation, making real-time object detection efficient and straightforward.

4. **Python Scripts:** Automate the detection process, capture camera input, run the YOLOv8 model, and trigger alert mechanisms based on detection results.

5. **Telegram Bot API:** Facilitates real-time alerting by sending messages directly to users, enabling quick responses to threats. The API is reliable, scalable, and user-friendly.

4.4 Algorithm

1. **System Initialization:** Start the Raspberry Pi, load the YOLOv8 model, initialize the Pi Camera, and set up the Telegram bot.

2. Capture Video Feed: Continuously capture and preprocess frames from the Pi Camera.

3. **Object Detection:** Pass each frame to the YOLOv8 model and filter detections based on confidence scores.

4. **Elephant Detection Check:** If an elephant is detected, annotate the image and save it locally.

5. **Send Alert via Telegram:** Send ”ELEPHANT DETECTED!” with the annotated image using the Telegram bot.

6. **Post-Alert Process:** Repeat the detection and alert process in a loop.

7. **System Shutdown:** Safely stop video capture and power down the Raspberry Pi when needed.

Chapter 5

CIRCUIT DIAGRAM AND EXPLANATION

5.1 Circuit Diagram

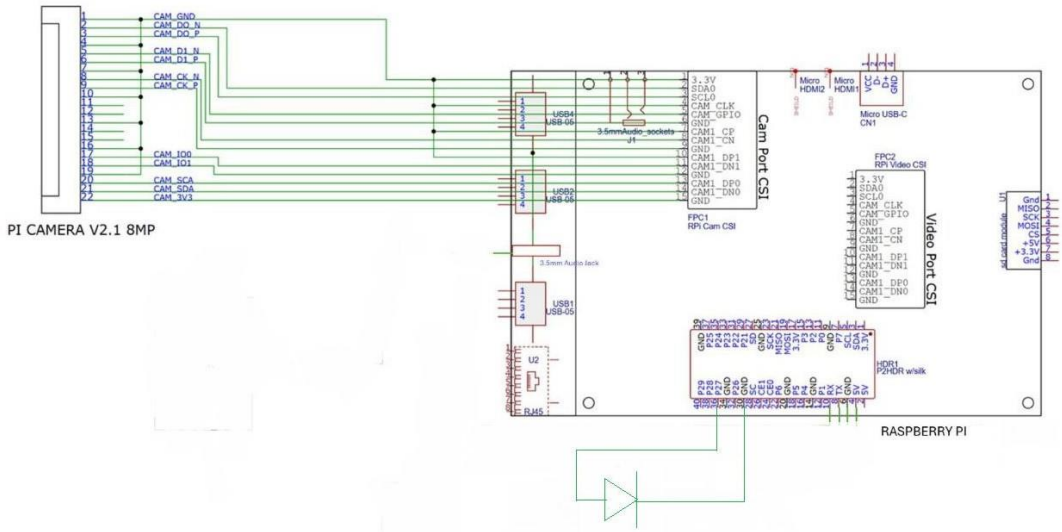


Figure 5.1: Circuit Diagram

5.2 Explanation of Connections

This configuration enables the system to efficiently process camera input, perform object detection, and send alerts while providing a local warning signal using the LED. Step by step explanation is given below.

1. Pi Camera Module Connection: Connect the Pi Camera module to the CSI port on the Raspberry Pi 5, ensuring the ribbon cable is properly aligned with the contacts facing the right direction.

2. Connect the Raspberry Pi 5 to a 5V/3A power supply using the USB-C port to ensure stable power for continuous operation.

3. LED Indicator Connection: Connect the positive leg (anode) of the LED to GPIO Pin 17 (BCM GPIO 17) on the Raspberry Pi. Connect the negative leg (cathode) of the LED to Ground (GND) on the Raspberry Pi.

4. Internet Connectivity: Connect the Raspberry Pi to the internet using WiFi or Ethernet to enable Telegram alerts.

5.3 Bill of Materials

No	Particular	Quantity	Unit Price	Amount
1	Raspberry pi 5	1	8209	8209
2	Picam version 1.3	1	400	400
3	SD Card 64 GB	1	500	500
3	LED	1	10	10
3	Jumper Wires	2	10	10
Total				**9029**

Table 5.1: List of Devices

Chapter 6

COMPONENTS USED

6.1 Raspberry Pi 5 (8 GB RAM):

1. Specifications: Quad-core ARM Cortex-A76 processor, 8 GB RAM, USB 3.0, GPIO pins, CSI camera interface, and WiFi/Ethernet connectivity.

2. Reason for Use: Provides ample processing power for real-time object detection using the YOLOv8 model, while built-in connectivity options enable remote alerts.

Figure 6.1: Raspberry Pi 5

6.2 Pi Camera Module (V1.3):

1. Specifications: 5 MP sensor, 1080p video recording at 30 fps, CSI interface for high-speed data transfer.
2. Reason for Use: The V1.3 camera module offers reliable image quality and seamless integration with the Raspberry Pi, making it suitable for real-time detection tasks.

Figure 6.2: Picam v 1.3

6.3 Power Supply (5V/3A USB-C Adapter):

1. Specifications: Provides 5V DC at 3A with over-voltage and over-current protection.
2. Reason for Use: Ensures stable power delivery for continuous operation of the Raspberry Pi and peripherals.

Figure 6.3: Power Supply

6.4 LED and Jumper Wires (Indicator Setup):

1. Specifications: Standard 5mm LED with a 220ohm resistor for current limiting; Jumper wires for reliable connections.
2. Reason for Use: The LED serves as a visual alert for elephant detection, connected to GPIO Pin 17 for easy control via scripts.

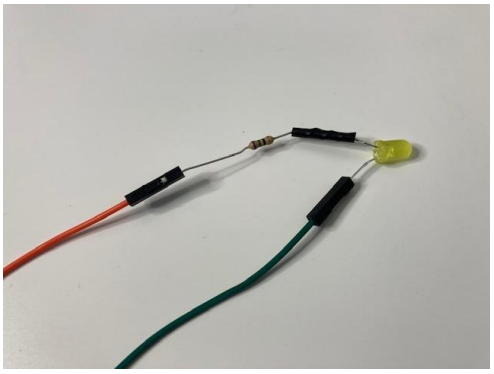


Figure 6.4: LED connected using jumper wires

6.5 Micro SD Card (64 GB or Higher):

1. Specifications: Class 10 or UHS-1 speed rating, minimum 64 GB storage capacity.
2. Reason for Use: Provides ample storage for the operating system, object detection model, and project scripts.

Figure 6.5: SD Card

6.6 Protective Enclosure:

1. Specifications: Weatherproof and durable with access for power, camera, and indicator LED.
2. Reason for Use: Ensures the safety of components in harsh outdoor environments, protecting against dust, moisture, and wildlife.

Figure 6.6: Case for Raspberry Pi

Chapter 7

IMPLEMENTATION AND DESIGN

This project focuses on developing a real-time elephant detection and alert system using the Raspberry Pi 5 and YOLOv8 object detection model. The hardware setup includes the Raspberry Pi 5, Pi Camera Module V1.3, and an LED indicator for local alerts. The software efficiently captures video input, detects elephants, and sends alerts through a Telegram bot, providing a reliable solution for reducing human-elephant conflicts.

7.1 Initial Setup without casing:

Design and Implementation Steps for Initial Setup (Without Casing and LED Indicators):

Figure 7.1: Initial Setup

1. **Hardware Setup:**

   - Connect the Pi Camera Module V1.3 to the CSI port on the Raspberry Pi 5, ensuring proper alignment.

   - Insert a 64 GB microSD card with the Raspberry Pi OS installed into the Pi.

   - Power the Raspberry Pi 5 using a 5V/3A USB-C power adapter.

2. **Software Configuration:**

   - Boot the Raspberry Pi and connect it to the internet via WiFi.

   - Enable the Pi Camera using the Raspberry Pi configuration settings.

   - Install required dependencies, including **Python, OpenCV, PyTorch, and YOLOv8 model files.

3. **Object Detection Model Setup:**

   - Copy the trained YOLOv8 .onnx model to the Raspberry Pi.

   - Create a Python script to load the model and preprocess input frames from the camera.

4. **Testing the Detection System:**

   - Run the script to capture real-time video input and perform object detection.

   - Verify that the model accurately identifies elephants and other classes in the camera feed.

5. **Telegram Bot Integration:**

   - Set up the Telegram bot and obtain the API token.

   - Modify the detection script to send alerts with images using the Telegram API when an elephant is detected.

This setup ensures a functional detection system, ready for field deployment with casing and indicators in the next phase.

7.2 Final Setup

Design and Implementation Steps for Initial Setup (Without Casing and LED Indicators):

Figure 7.2: Final Setup

1. **Connecting LED Indicator and Updating Script:**

   - Connect the LED's anode to GPIO Pin 17 and cathode to Ground (GND) and update the detection script to blink the LED when an elephant is detected.

2. **Enclosure Assembly and Final Testing:**

   - Secure all components inside a weatherproof enclosure, ensure proper cable routing, and perform final testing to verify LED, camera, and Telegram alerts before field deployment.

7.3 Details of Software used

The software aspect of the project involves configuring the Raspberry Pi, setting up object detection, and enabling remote alerts. The following components are essential for successful implementation:

1. **Operating System**:

   Raspberry Pi OS (64-bit): Provides a stable environment with necessary drivers for Pi Camera integration and GPIO control.

2. **Programming Language**:

   Python 3.11: Chosen for its extensive libraries, ease of use, and support for AI/ML frameworks.

3. **Object Detection Model**:

   YOLOv8 (.onnx format): Pre-trained for detecting elephants and other objects with high accuracy.

4. **Libraries and Dependencies**:

   OpenCV (cv2): For capturing video input from the Pi Camera and preprocessing images.

   Ultralytics (YOLOv8 library): For loading and running the YOLOv8 model efficiently.

   NumPy: For numerical computations and handling image arrays.

   PIL (Pillow): For image processing and handling detected object frames.

   Requests: For sending alert messages via the Telegram API.

   RPi.GPIO or GPIO Zero: For controlling the LED indicator using GPIO pins.

   Telegram Bot API: For communicating with the Telegram bot to send alerts.

5. **Software Setup Steps**:

Install dependencies using pip and configure the Raspberry Pi. Load the YOLOv8 model, preprocess input, and run real-time detection. Implement Telegram API integration for alert messaging. Control the LED indicator using GPIO output.

These tools and libraries ensure efficient detection, alerting, and control capabilities, making the solution effective and reliable.

Chapter 8

EXPLANATION OF CODE

8.1 Overview

The objective of this program is to detect elephants using YOLOv8 algorithm running on a Raspberry Pi 5. The program obtains video feed from the raspberry pi's camera and sends an alert through telegram when an elephant is detected. In addition, an led is also lit to alert local residents.

8.2 Included Libraries

The following libraries are imported at the beginning of the program

- **cv2(Open CV)** - It is an open-source computer vision library used for image processing, video analysis, and machine learning tasks.
- **picamera2** - It is a Python library for controlling Raspberry Pi cameras, providing an easy-to-use interface for capturing images and videos.
- **requests** - This library is used for getting chat id for sending message using telegram bot.
- **ultralytics** - YOLO is imported from ultralytics library, an object detection framework with high speed and good accuracy.
- **time** - Time library is used to set delay.
- **gpiozero** - Used for interfacing with GPIO pins of the Raspberry Pi.

8.3 Initializing Camera

```
picam2 = Picamera2()
```

```
    picam2.preview_configuration.main.size = (1280, 1280)
    picam2.preview_configuration.main.format = "RGB888"
    picam2.preview_configuration.align()
    picam2.start()
```

- **Picamera2** - This is used to set picam2 variable.
- **main.size** - main.size is used to set the input resolution of the camera to 1280x1280.
- **main.format** - It is used to set 24bit color format for the picamera.
- **align** - This sets the alignment of the camera input.
- **start** - start command starts the picamera with the set parameters.

8.4 Functions of bot

```
    BOT_TOKEN = "bot_token"


def getchatid():
    url = f"https://api.telegram.org/bot{BOT_TOKEN}/getUpdates"
    response = requests.get(url).json()

    if "result" in response and len(response["result"]) > 0:
        chat_id = response["result"][-1]["message"]["chat"]["id"]
        return chat_id
    else:
        print("No chat ID found. Send a message to the bot first!")
        return None

def sendtelegrammessage(message):
    chat_id = get_chat_id()
```

```
    if not chat_id:
        return

    url = f"https://api.telegram.org/bot{BOT_TOKEN}/sendMessage"
    payload = {"chat_id": chat_id, "text": message}
    requests.post(url, json=payload)

def sendtelegramphoto(photo_path):
    chat_id = getchatid()
    if not chat_id:
        return

    url = f"https://api.telegram.org/bot{BOT_TOKEN}/sendPhoto"
    with open(photo_path, "rb") as photo:
        files = {"photo": photo}
        requests.post(url, data={"chat_id": chat_id}, files=files)
```

- **getchatid** - Gets the chat id using the bot token and prints an error if failed.
- **sendtelegrammessage** - Sends telegram message using requests library.
- **sendtelegramphoto** - Sends the photo with bounding box and labels plotted.

8.5 Initializing the Model

```
    model = YOLO("best.onnx")
    elephant_detected = False
    led = LED(17)
```

The trained model weights are loaded into the program using the YOLO library and sets the elephant_detected variable to False. LED(17) is then called to initialize the led.

8.6 Running The Model

```
while True:
    frame = picam2.capture_array()
    # Run YOLOv8 detection
    results = model(frame)

    for result in results:
        for box in result.boxes:
            class_id = int(box.cls[0])
            class_name = model.names[class_id]

            if class_name.lower() == "elephant":
                x1, y1, x2, y2 = map(int, box.xyxy[0])
                cv2.rectangle(frame, (x1, y1), (x2, y2), (0, 255, 0),
                cv2.putText(frame, class_name, (x1, y1 - 10), cv2.FONT

                if not elephant_detected:
                    send_telegram_message(" Elephant detected!")

                    # Save the detected frame
                    photo_path = "elephant_detected.jpg"
                    cv2.imwrite(photo_path, frame)

                    # Send the saved image to Telegram
                    send_telegram_photo(photo_path)

                    elephant_detected = True  # Prevent spam
                    led.on()
```

```
                time.sleep(300)
                led.off()
                elephant_detected = False
                continue
```

The output from the camera is captured using the .capture_array() command. The detection algorithm is run on the captured frame and stored in the results array. The results are then traversed and checked for the class elephant. If an elephant is found, the telegram bot is triggered to send message along with a photo containing a labeled bounding box around the detection. Additionally an led is also lit when elephant is detected.

Chapter 9

TESTING AND RESULTS

Testing involves validating the accuracy of the YOLOv8 model in detecting elephants, assessing the responsiveness of the Telegram alert system, and verifying the functionality of the LED indicator. The tests were conducted in controlled environments to ensure consistent and accurate results, demonstrating the system's effectiveness in reducing human-elephant conflicts.

9.1 Testing Procedure

1. Model Accuracy Testing:

   Conduct real-time detection tests using the Pi Camera Module V1.3 to validate the YOLOv8 model's accuracy in identifying elephants and distinguishing them from other objects.

   Record detection results, including true positives, false positives, and false negatives, to calculate precision and recall.

2. Telegram Alert Verification:

   Trigger elephant detection scenarios to confirm that the Telegram bot sends alerts promptly.

   Validate that the alert message contains the correct text, image, and bounding boxes around detected objects.

3. LED Indicator Testing:

   Simulate elephant detection and observe the LED connected to GPIO Pin 17 to ensure it blinks correctly when detection occurs.

   Test multiple scenarios to verify consistent LED behavior.

4. End-to-End System Validation:

Perform a complete run-through, from initial detection to alert notification, and confirm seamless integration of all components.

Verify real-time detection and alert delivery efficiency.

9.2 Observations and Output

9.2.1 Testing Phase 1: Trained Model using Colab

During the first phase of our project, we trained the YoloV8 model using Colab, and used a usb camera with our laptop as the processing unit to run the code, which resulted in detection of elephants with an MPA (Mean Precision Average) value of almost 0.5.

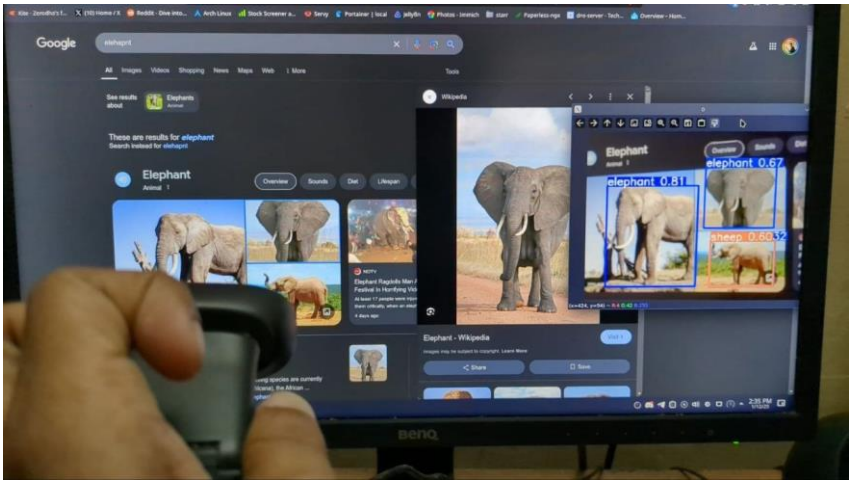


Figure 9.1: Testing Phase 1

9.2.2 Testing Phase 2: Trained model using college GPU (Quadro RTX 400)

During the second training phase of our project, we trained the YoloV8 model using an NVIDIA Quadro GPU via ssh from the college lab, resulting in exceptionally better MPA (Mean Precision Average) values, ranging from 0.9 to 1.0.

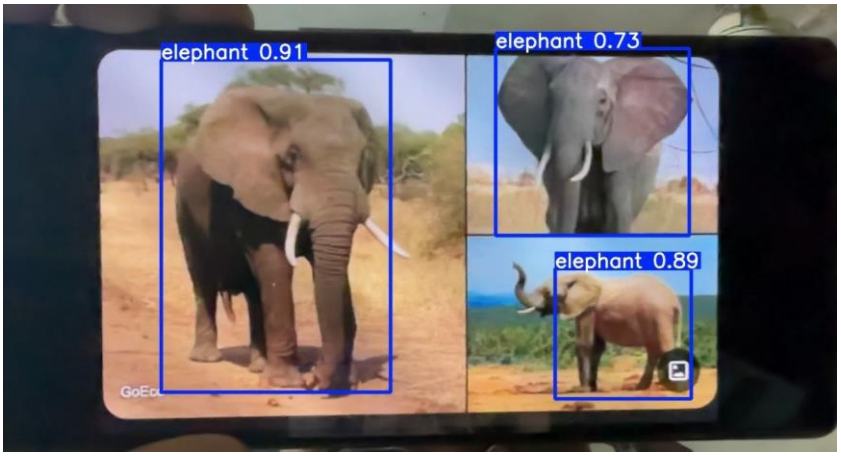


Figure 9.2: Testing Phase 2

9.2.3 Testing Phase 3: Working of Telegram Bot

During the third training phase of our project, we tested the working of our Python script, which triggers the Telegram api to send alerts via bot.

Figure 9.3: Testing Phase 3

9.3 Performance Analysis

The performance of the YOLOv8 model was evaluated using multiple graphs that provide insights into the model's accuracy, precision, and overall effectiveness in detecting elephants. These graphs include:

9.3.1 Precision Confidence Curve

This curve helps determine the optimal confidence threshold for deployment. Setting a threshold too low may result in false alarms, while too high a threshold might miss some detections.

The ideal curve shows high precision even at lower confidence levels, indicating the model is consistently accurate.

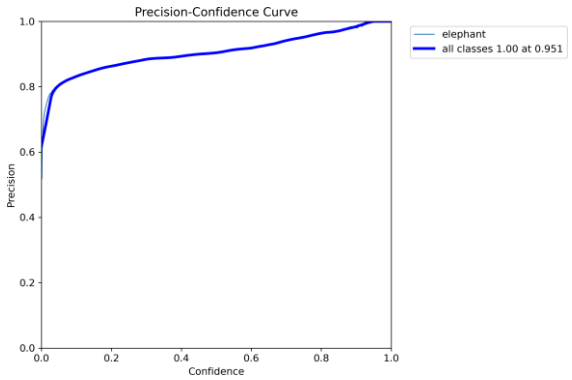


Figure 9.4: Precision Confidence Curve

9.3.2 Recall Confidence Curve

This curve helps determine the optimal confidence threshold that balances recall with acceptable precision.

A flat, high recall curve means the model reliably detects elephants, even at higher confidence levels.

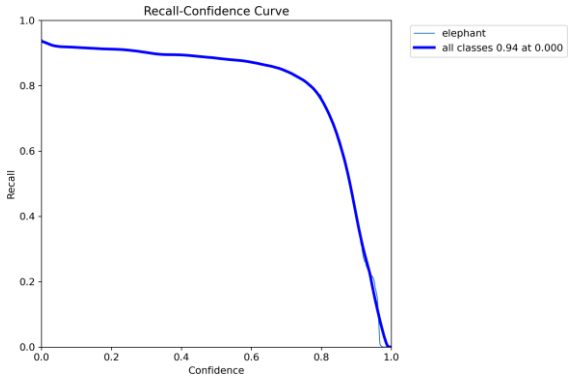


Figure 9.5: Recall Confidence Curve

9.3.3 Precision Recall Curve

This curve helps find the optimal confidence threshold that balances detection accuracy (recall) and reliability (precision).

A well-performing model for this project would show a curve maintaining high precision even as recall increases.

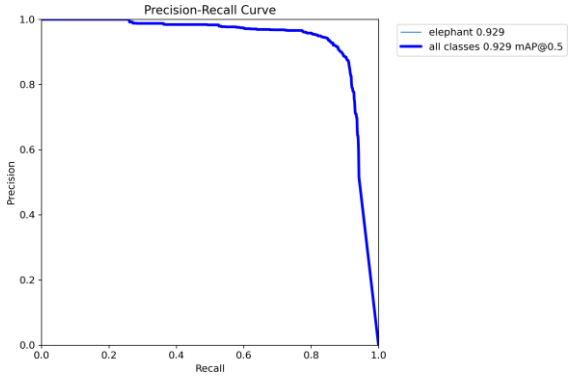


Figure 9.6: Precision Recall Curve

9.3.4 Confusion Matrix

For elephant detection, a good confusion matrix would show a high value in the ”Elephant-Elephant” cell (correct detections) and low or zero values in cells indicating misclassification.

It helps identify specific weaknesses, such as false positives (e.g., misclassifying other animals as elephants) or false negatives (missing actual elephants).

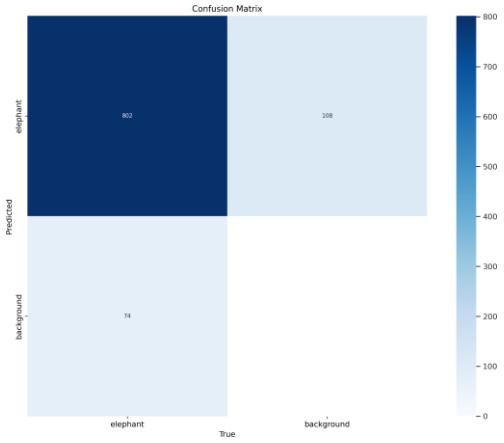


Figure 9.7: Confusion Matrix

9.3.5 Model Testing Results

A sample image is attached to verify the accuracy of detection of elephant images after the training process of over 30,000 images.

Figure 9.8: Final Prediction

Chapter 10

APPLICATIONS, LIMITATIONS AND FUTURE SCOPE

This section explores the potential real-world applications, identifies the current limitations, and discusses the future scope of the elephant detection project using the YOLOv8 object detection algorithm. The project's adaptability makes it suitable for diverse scenarios, while understanding its limitations helps refine the solution. Future enhancements can make the system more robust, reliable, and effective, especially in remote areas with connectivity challenges.

10.1 Applications

The elephant detection system using the YOLOv8 object detection algorithm offers significant benefits in mitigating human-elephant conflicts and supporting wildlife conservation efforts. Its versatile design enables deployment in various environments and scenarios, making it an effective solution for real-time wildlife monitoring and protection.

- Wildlife Conservation and Protection: Detects elephants in real-time to prevent human-wildlife conflicts and support conservation initiatives.
- Early Warning Systems for Villages and Farmlands: Alerts locals and authorities of elephant intrusions to safeguard lives and crops.
- Forest Department Monitoring: Assists forest officials in tracking elephant movements and planning mitigation strategies.
- Scalable Solution for Remote Monitoring: Can be adapted for use in remote areas using alternative communication protocols for better coverage.

10.2 Limitations

While the elephant detection system using the YOLOv8 algorithm effectively identifies and alerts about elephant intrusions, certain challenges limit its performance and deployment in real-world scenarios. Understanding these limitations helps in refining the system for better reliability and scalability.

- Dependency on Stable Internet Connection: The system requires a reliable internet connection for real-time alerts, limiting its use in remote areas.
- Limited Detection Range: The Pi Camera 1.3 has a fixed focal length, restricting the detection range.
- Power Supply Constraints: Continuous operation requires a stable power source, which can be challenging in forested regions.
- Hardware Limitations: The Raspberry Pi 5's processing power can limit performance for real-time object detection at high resolutions.

10.3 Future Scope

The elephant detection system using the YOLOv8 algorithm offers significant potential for future enhancements. By addressing current limitations and incorporating advanced technologies, the project can become a more robust and scalable solution for wildlife monitoring and human-wildlife conflict mitigation.

- Integration of Alternative Communication Protocols: Use of LoRa, Zigbee, or satellite communication for reliable alerts in remote areas with limited internet connectivity.
- Improved Detection Capabilities: Upgrading to advanced cameras and sensors for better accuracy in varying weather and lighting conditions.
- Multi-Species Detection: Training the model to recognize additional wildlife species for broader conservation applications.

Chapter 11

CONCLUSION

The elephant detection system developed using the YOLOv8 object detection algorithm provides an effective solution for addressing human-elephant conflicts and promoting wildlife conservation. By leveraging the processing capabilities of the Raspberry Pi 5, the precision of the Pi Camera 1.3, and the real-time alerting features of a Telegram bot, the project successfully detects elephants and sends instant alerts to concerned authorities and local communities. This proactive approach helps mitigate threats to both human lives and wildlife by enabling timely interventions.

Despite its success, the current setup faces certain limitations, including dependency on a stable internet connection, environmental challenges, and power constraints in remote locations. These challenges can be overcome through future upgrades, such as integrating alternative communication protocols like LoRa, Zigbee, or satellite networks for reliable operation in low-connectivity areas. Additionally, improvements in camera specifications, the use of solar power, and the inclusion of multi-species detection capabilities would further enhance the system's efficiency and versatility.

This project demonstrates how affordable, scalable, and intelligent solutions can effectively contribute to wildlife protection and peaceful coexistence. It lays a strong foundation for future research and development, paving the way for advanced, AI-driven wildlife monitoring systems that can be deployed in diverse environments.

. . .

Bibliography

[1] G. Van Horn et al., *"The iNaturalist Species Classification and Detection Dataset,"* , arXiv:1707.06642 [cs], Apr. 2018, Accessed: Dec. 01, 2021. [Online]. Available: https://arxiv.org/abs/1707.06642.

[2] IMAGENET 1000 class list, *"IMAGENET 1000 Class List - Weka Deeplearning4j. (n.d.). Retrieved December 8, 2021,"* , Available: https://deeplearning.cms.waikato.ac.nz/user-guide/class-maps/IMAGENET/.

[3] M. Körschens and J. Denzler., *"ELPephants: A Fine-Grained Dataset for Elephant Re-Identification,"* , IEEE/CVF International Conference on Computer Vision Workshop (ICCVW), 2019, pp. 263-270, doi: 10.1109/ICCVW.2019.00035.

[4] ULTRALYTICS YOLO Docs., Available: https://docs.ultralytics.com/models/yolov8/

[5] K. S. P. Premarathna, R. M. K. T. Rathnayaka and J. Charles,.*"An Elephant Detection System to Prevent Human-Elephant Conflict and Tracking of Elephant Using Deep Learning,"* , 2020 5th International Conference on Information Technology Research (ICITR), Moratuwa, Sri Lanka, 2020, pp. 1-6, doi: 10.1109/ICITR51448.2020.9310798.

[6] M. Fazil and M. Firdhous,.*"IoT-Enabled Smart Elephant Detection System for Combating Human Elephant Conflict,"* ,2018 3rd International Conference on Information Technology Research (ICITR), Moratuwa, Sri Lanka, 2018, pp. 1-6, doi: 10.1109/ICITR.2018.8736149.

Appendices

Appendix A

Coding

```
import cv2
from picamera2 import Picamera2, Preview
import requests
from ultralytics import YOLO
import time
from gpiozero import LED

picam2 = Picamera2()
picam2.preview_configuration.main.size = (1280, 1280)
picam2.preview_configuration.main.format = "RGB888"
picam2.preview_configuration.align()
picam2.start()

# Telegram bot token
BOT_TOKEN = "7301257374:AAHhcYhBFN8s1EgSVKQzC51D4w8p4pyP-MM"

def get_chat_id():
    url = f"https://api.telegram.org/bot{BOT_TOKEN}/getUpdates"
    response = requests.get(url).json()

    if "result" in response and len(response["result"]) > 0:
        chat_id = response["result"][-1]["message"]["chat"]["id"]
        return chat_id
```

```
    else:
        print("No chat ID found. Send a message to the bot first!")
        return None

def send_telegram_message(message):
    chat_id = get_chat_id()
    if not chat_id:
        return

    url = f"https://api.telegram.org/bot{BOT_TOKEN}/sendMessage"
    payload = {"chat_id": chat_id, "text": message}
    requests.post(url, json=payload)

def send_telegram_photo(photo_path):
    chat_id = get_chat_id()
    if not chat_id:
        return

    url = f"https://api.telegram.org/bot{BOT_TOKEN}/sendPhoto"
    with open(photo_path, "rb") as photo:
        files = {"photo": photo}
        requests.post(url, data={"chat_id": chat_id}, files=files)

# Load YOLOv8 model
model = YOLO("best.onnx")
elephant_detected = False
led = LED(17)

while True:
```

```
frame = picam2.capture_array()
# Run YOLOv8 detection
results = model(frame)

for result in results:
    for box in result.boxes:
        class_id = int(box.cls[0])
        class_name = model.names[class_id]

        if class_name.lower() == "elephant":
            x1, y1, x2, y2 = map(int, box.xyxy[0])
            cv2.rectangle(frame, (x1, y1), (x2, y2), (0, 255, 0),
            cv2.putText(frame, class_name, (x1, y1 - 10), cv2.FONT

            if not elephant_detected:
                send_telegram_message(" Elephant detected!")

                # Save the detected frame
                photo_path = "elephant_detected.jpg"
                cv2.imwrite(photo_path, frame)

                # Send the saved image to Telegram
                send_telegram_photo(photo_path)
                led.on()
                elephant_detected = True  # Prevent spam

                time.sleep(20)

                led.off()
```

```
                    elephant_detected = False

                    continue
#       cv2.imshow("YOLOv8 Elephant Detection", frame)

#       if cv2.waitKey(1) & 0xFF == ord("q"):
#           break

cap.release()
cv2.destroyAllWindows()
```

Appendix B

Project Estimate

B.1 Bill of Materials

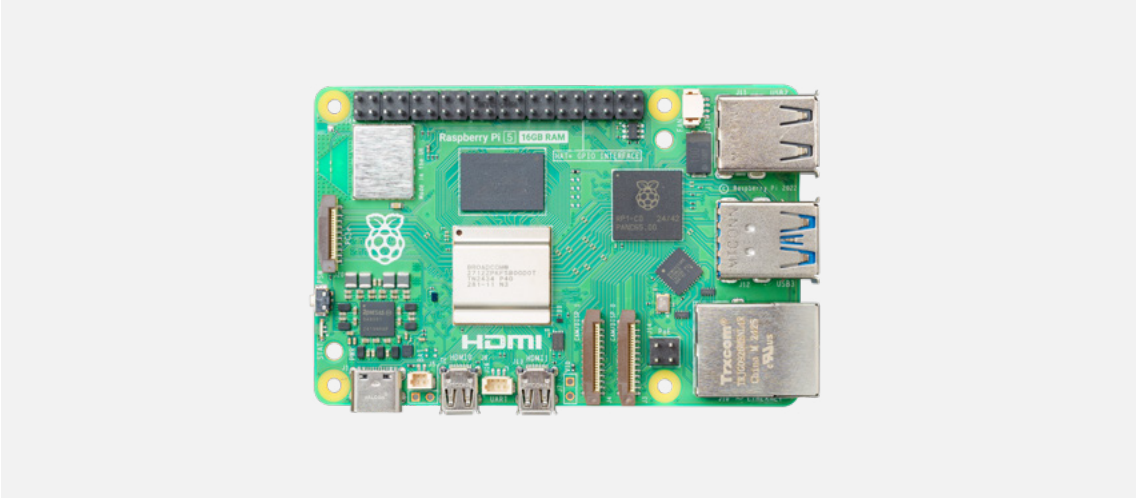
No	Particular	Quantity	Unit Price	Amount
1	Raspberry pi 5	1	8209	8209
2	Picam version 1.3	1	400	400
3	SD Card 64 GB	1	500	500
3	LED	1	10	10
3	Jumper Wires	2	10	10
Total				**9029**

Table B.1: List of Devices

Appendix C

Datasheets

Overview



Welcome to the latest generation of Raspberry Pi: the everything computer.

Featuring a 64-bit quad-core Arm Cortex-A76 processor running at 2.4GHz, Raspberry Pi 5 delivers a 2–3× increase in CPU performance relative to Raspberry Pi 4. Alongside a substantial uplift in graphics performance from an 800MHz VideoCore VII GPU; dual 4Kp60 display output over HDMI; and state-of-the-art camera support from a rearchitected Raspberry Pi Image Signal Processor, it provides a smooth desktop experience for consumers, and opens the door to new applications for industrial customers.

For the first time, this is a full-size Raspberry Pi computer using silicon built in-house at Raspberry Pi. The RP1 “southbridge” provides the bulk of the I/O capabilities for Raspberry Pi 5, and delivers a step change in peripheral performance and functionality. Aggregate USB bandwidth is more than doubled, yielding faster transfer speeds to external UAS drives and other high-speed peripherals; the dedicated two-lane 1Gbps MIPI camera and display interfaces present on earlier models have been replaced by a pair of four-lane 1.5Gbps MIPI transceivers, tripling total bandwidth, and supporting any combination of up to two cameras or displays; peak SD card performance is doubled, through support for the SDR104 high-speed mode; and for the first time the platform exposes a single-lane PCI Express 2.0 interface, providing support for high-bandwidth peripherals.

Specification

Processor

Broadcom BCM2712 2.4GHz quad-core 64-bit Arm Cortex-A76 CPU, with Cryptographic Extension, 512KB per-core L2 caches, and a 2MB shared L3 cache

Features:

- VideoCore VII GPU, supporting OpenGL ES 3.1, Vulkan 1.2
- Dual 4Kp60 HDMI® display output with HDR support
- 4Kp60 HEVC decoder
- LPDDR4X-4267 SDRAM (options for 2GB, 4GB, 8GB and 16GB)
- Dual-band 802.11ac Wi-Fi®
- Bluetooth 5.0 / Bluetooth Low Energy (BLE)
- microSD card slot, with support for high-speed SDR104 mode
- 2 × USB 3.0 ports, supporting simultaneous 5Gbps operation
- 2 × USB 2.0 ports
- Gigabit Ethernet, with PoE+ support (requires separate PoE+ HAT)
- 2 × 4-lane MIPI camera/display transceivers
- PCIe 2.0 x1 interface for fast peripherals (requires separate M.2 HAT or other adapter)
- 5V/5A DC power via USB-C, with Power Delivery support
- Raspberry Pi standard 40-pin header
- Real-time clock (RTC), powered from external battery
- Power button

Production lifetime:

Raspberry Pi 5 will remain in production until at least January 2036

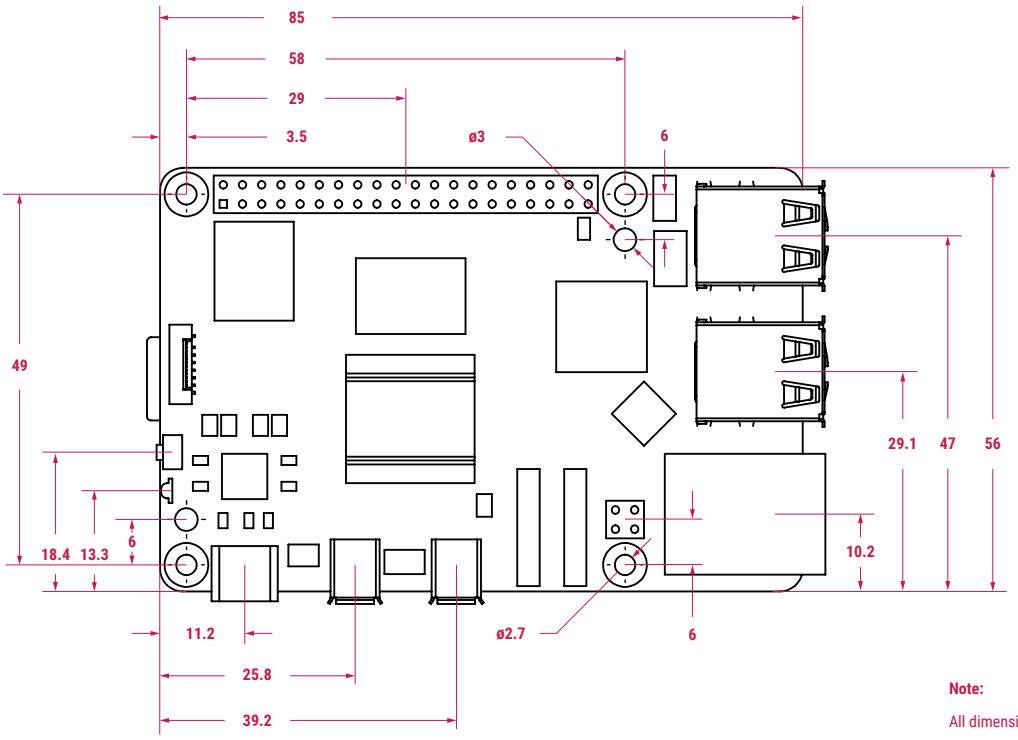
Compliance:

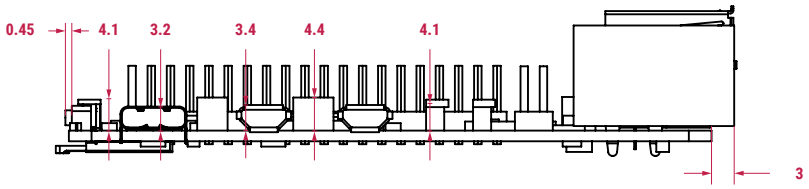
For a full list of local and regional product approvals, please visit pip.raspberrypi.com

List price:

2GB	**$50**
4GB	**$60**
8GB	**$80**
16GB	**$120**

Physical specification





Note:

All dimensions in mm

All dimensions are approximate and for reference purposes only. The dimensions shown should not be used for producing production data

The dimensions are subject to part and manufacturing tolerances

Not all of the board components are shown. Please reference a physical board for representation of componentry

Dimensions may be subject to change

WARNINGS

- This product should be operated in a well ventilated environment, and if used inside a case, the case should not be covered.
- While in use, this product should be firmly secured or should be placed on a stable, flat, non-conductive surface, and should not be contacted by conductive items.
- The connection of incompatible devices to Raspberry Pi 5 may affect compliance, result in damage to the unit, and invalidate the warranty.
- All peripherals used with this product should comply with relevant standards for the country of use and be marked accordingly to ensure that safety and performance requirements are met.

SAFETY INSTRUCTIONS

To avoid malfunction or damage to this product, please observe the following:

- Do not expose to water or moisture, or place on a conductive surface while in operation.
- Do not expose to heat from any source; Raspberry Pi 5 is designed for reliable operation at normal ambient temperatures.
- Store in a cool, dry location.
- Take care while handling to avoid mechanical or electrical damage to the printed circuit board and connectors.
- While it is powered, avoid handling the printed circuit board, or handle it only by the edges, to minimise the risk of electrostatic discharge damage.

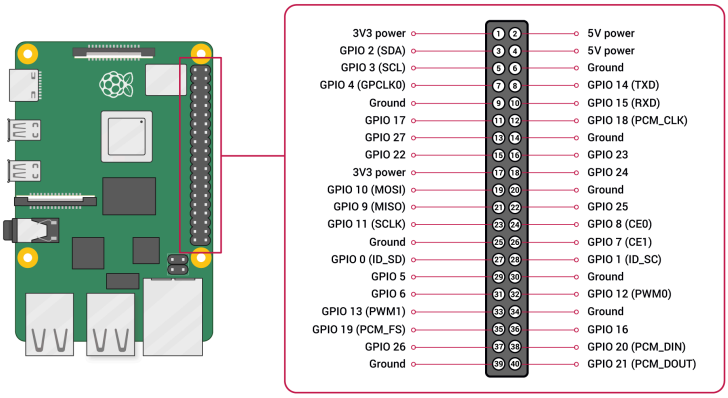
§ GPIO and the 40-pin header

Edit this on GitHub

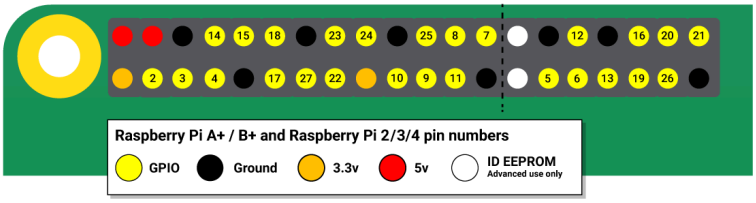
You can find a 40-pin GPIO (general-purpose input/output) header on all current Raspberry Pi boards. The GPIO headers on all boards have a 0.1in (2.54mm) pin pitch.

NOTE

The header is unpopulated (has no headers) on Zero and Pico devices that lack the "H" suffix.



General Purpose I/O (GPIO) pins can be configured as either general-purpose input, general-purpose output, or as one of up to six special alternate settings, the functions of which are pin-dependent.



NOTE

The GPIO pin numbering scheme is not in numerical order. GPIO pins 0 and 1 are present on the board (physical pins 27 and 28), but are reserved for advanced use.

Outputs

A GPIO pin designated as an output pin can be set to high (3.3V) or low (0V).

Inputs

A GPIO pin designated as an input pin can be read as high (3.3V) or low (0V). This is made easier with the use of internal pull-up or pull-down resistors. Pins GPIO2 and GPIO3 have fixed pull-up resistors, but for other pins this can be configured in software.

View a GPIO pinout for your Raspberry Pi

A GPIO reference can be accessed on your Raspberry Pi by opening a terminal window and running the command `pinout`. This tool is provided by the GPIO Zero Python library, which is installed by default in Raspberry Pi OS.

WARNING

While connecting simple components to GPIO pins is safe, be careful how you wire things up. LEDs should have resistors to limit the current passing through them. Do not use 5V for 3.3V components. Do not connect motors directly to the GPIO pins, instead use an H-bridge circuit or a motor controller board.

Permissions

In order to use the GPIO ports, your user must be a member of the `gpio` group. The default user account is a member by default, but you must add other users manually using the following command:

```
$ sudo usermod -a -G gpio <username>
```

GPIO pads

The GPIO connections on the BCM2835 package are sometimes referred to in the peripherals data sheet as "pads" — a semiconductor design term meaning "chip connection to outside world".

The pads are configurable CMOS push-pull output drivers/input buffers. Register-based control settings are available for:

- Internal pull-up / pull-down enable/disable
- Output drive strength
- Input Schmitt-trigger filtering

Power-on states

All GPIO pins revert to general-purpose inputs on power-on reset. The default pull states are also applied, which are detailed in the alternate function table in the Arm peripherals datasheet. Most GPIOs have a default pull applied.

Interrupts

Each GPIO pin, when configured as a general-purpose input, can be configured as an interrupt source to the Arm. Several interrupt generation sources are configurable:

- Level-sensitive (high/low)
- Rising/falling edge
- Asynchronous rising/falling edge

Level interrupts maintain the interrupt status until the level has been cleared by system software (e.g. by servicing the attached peripheral generating the interrupt).

The normal rising/falling edge detection has a small amount of synchronisation built into the detection. At the system clock frequency, the pin is sampled with the criteria for generation of an interrupt being a stable transition within a three-cycle window, i.e. a record of 1 0 0 or 0 1 1. Asynchronous detection bypasses this synchronisation to enable the detection of very narrow events.

Alternative functions

Almost all of the GPIO pins have alternative functions. Peripheral blocks internal to the SoC can be selected to appear on one or more of a set of GPIO pins, for example the I2C buses can be configured to at least three separate locations. Pad control, such as drive strength or Schmitt filtering, still applies when the pin is configured as an alternate function.

Some functions are available on all pins, others on specific pins:

- PWM (pulse-width modulation)
  - Software PWM available on all pins
  - Hardware PWM available on GPIO12, GPIO13, GPIO18, GPIO19
- SPI
  - SPI0: MOSI (GPIO10); MISO (GPIO9); SCLK (GPIO11); CE0 (GPIO8), CE1 (GPIO7)

  - SPI1: MOSI (GPIO20); MISO (GPIO19); SCLK (GPIO21); CE0 (GPIO18); CE1 (GPIO17); CE2 (GPIO16)
- I2C
  - Data: (GPIO2); Clock (GPIO3)
  - EEPROM Data: (GPIO0); EEPROM Clock (GPIO1)
- Serial
  - TX (GPIO14); RX (GPIO15)

Voltage specifications

Two 5V pins and two 3.3V pins are present on the board, as well as a number of ground pins (GND), which can not be reconfigured. The remaining pins are all general-purpose 3.3V pins, meaning outputs are set to 3.3V and inputs are 3.3V-tolerant.

The table below gives the various voltage specifications for the GPIO pins for BCM2835, BCM2836, BCM2837 and RP3A0-based products (e.g. Raspberry Pi Zero or Raspberry Pi 3+). For information about Compute Modules you should see the relevant datasheets.

Symbol	Parameter	Conditions	Min	Typical	Max	Unit
V_{IL}	Input Low Voltage	-	-	-	0.9	V
V_{IH}	Input high voltage[a]	-	1.6	-	-	V
I_{IL}	Input leakage current	TA = +85○C	-	-	5	µA
C_{IN}	Input capacitance	-	-	5	-	pF
V_{OL}	Output low voltage[b]	IOL = -2mA	-	-	0.14	V
V_{OH}	Output high voltage[b]	IOH = 2mA	3.0	-	-	V
I_{OL}	Output low current[c]	VO = 0.4V	18	-	-	mA
I_{OH}	Output high current[c]	VO = 2.3V	17	-	-	mA
R_{PU}	Pullup resistor	-	50	-	65	kΩ
R_{PD}	Pulldown resistor	-	50	-	65	kΩ

[a] Hysteresis enabled

[b] Default drive strength (8mA)

[c] Maximum drive strength (16mA)

The table below gives the voltage specifications for the GPIO pins on BCM2711-based products (4-series devices). For information about Compute Modules you should see the relevant datasheets.

Symbol	Parameter	Conditions	Min	Typical	Max	Unit
V_{IL}	Input Low Voltage	-	-	-	0.8	V
V_{IH}	Input high voltage[a]	-	2.0	-	-	V

Symbol	Parameter	Conditions	Min	Typical	Max	Unit
I_{IL}	Input leakage current	TA = +85○C	-	-	10	µA
V_{OL}	Output low voltage[b]	IOL = -4mA	-	-	0.4	V
V_{OH}	Output high voltage[b]	IOH = 4mA	2.6	-	-	V
I_{OL}	Output low current[c]	VO = 0.4V	7	-	-	mA
I_{OH}	Output high current[c]	VO = 2.6V	7	-	-	mA
R_{PU}	Pullup resistor	-	33	-	73	kΩ
R_{PD}	Pulldown resistor	-	33	-	73	kΩ

[a] Hysteresis enabled

[b] Default drive strength (4mA)

[c] Maximum drive strength (8mA)

GPIO pads control

Edit this on GitHub

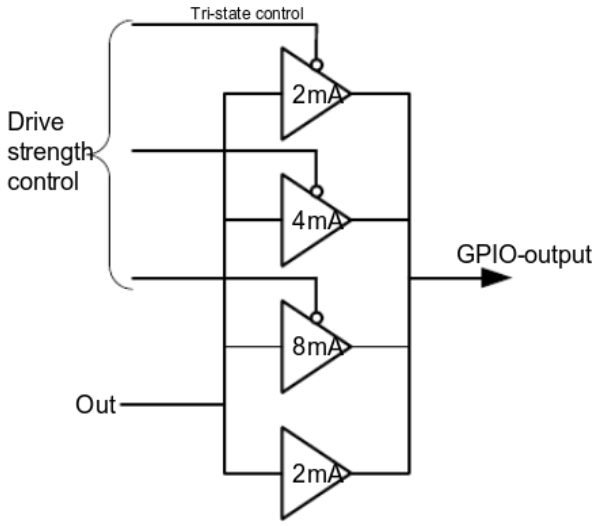
GPIO drive strengths do not indicate a maximum current, but a maximum current under which the pad will still meet the specification. You should set the GPIO drive strengths to match the device being attached in order for the device to work correctly.

Control drive strength

Inside the pad are a number of drivers in parallel. If the drive strength is set low (`0b000`), most of these are tri-stated so they do not add anything to the output current. If the drive strength is increased, more and more drivers are put in parallel. The diagram shows that behaviour.

WARNING

On 4-series devices, the current level is half the value shown in the diagram.



Current value

The current value specifies the maximum current under which the pad will still meet the specification.

Current value is *not* the current that the pad will deliver, and is *not* a current limit.

The pad output is a voltage source:

- If set high, the pad will try to drive the output to the rail voltage (3.3V)
- If set low, the pad will try to drive the output to ground (0V)

The pad will try to drive the output high or low. Success will depend on the requirements of what is connected. If the pad is shorted to ground, it will not be able to drive high. It will try to deliver as much current as it can, and the current is only limited by the internal resistance.

If the pad is driven high and it is shorted to ground, in due time it will fail. The same holds true if you connect it to 3.3V and drive it low.

Meeting the specification is determined by the guaranteed voltage levels. Because the pads are digital, there are two voltage levels, high and low. The I/O ports have two parameters which deal with the output level:

- V_{OL}, the maximum low-level voltage (0.14V at 3.3V VDD IO)
- V_{OH}, the minimum high-level voltage (3.0V at 3.3V VDD IO)

V_{OL}=0.14V means that if the output is Low, it will be <= 0.14V. V_{OH}=3.0V means that if the output is High, it will be >= 3.0V.

As an example, a drive strength of 16mA means that if you set the pad high, you can draw up to 16mA, and the output voltage is guaranteed to be >=V_{OH}. This also means that if you set a drive strength of 2mA and you draw 16mA, the voltage will **not** be V_{OH} but lower. In fact, it may not be high enough to be seen as high by an external device.

There is more information on the physical characteristics of the GPIO pins.

NOTE

On the Compute Module devices, it is possible to change the VDD IO from the standard 3.3V. In this case, V_{OL} and V_{OH} will change according to the table in the GPIO section.

The Raspberry Pi 3.3V supply was designed with a maximum current of ~3mA per GPIO pin. If you load each pin with 16mA, the total current is 272mA. The 3.3V supply will collapse under that level of load. Big current spikes will happen, especially if you have a capacitive load. Spikes will bounce around all the other pins near them. This is likely to cause interference with the SD card, or even the SDRAM behaviour.

Safe current

All the electronics of the pads are designed for 16mA. This is a safe value under which you will not damage the device. Even if you set the drive strength to 2mA and then load it so 16mA comes out, this will not damage the device. Other than that, there is no guaranteed maximum safe current.

GPIO addresses

- 0x 7e10 002c PADS (GPIO 0-27)
- 0x 7e10 0030 PADS (GPIO 28-45)
- 0x 7e10 0034 PADS (GPIO 46-53)

Bits	Field name	Description	Type	Reset
31:24	PASSWRD	Must be 0x5A when writing; accidental write protect password	W	0
23:5		**Reserved** - Write as 0, read as don't care		

Bits	Field name	Description	Type	Reset
4	SLEW	Slew rate; 0 = slew rate limited; 1 = slew rate not limited	RW	0x1
3	HYST	Enable input hysteresis; 0 = disabled; 1 = enabled	RW	0x1
2:0	DRIVE	Drive strength, see breakdown list below	RW	0x3

Beware of Simultaneous Switching Outputs (SSO) limitations which are device-dependent as well as dependent on the quality and layout of the PCB, the amount and quality of the decoupling capacitors, the type of load on the pads (resistance, capacitance), and other factors beyond the control of Raspberry Pi.

Drive strength list

- 0 = 2mA
- 1 = 4mA
- 2 = 6mA
- 3 = 8mA
- 4 = 10mA
- 5 = 12mA
- 6 = 14mA
- 7 = 16mA

Industrial use of the Raspberry Pi

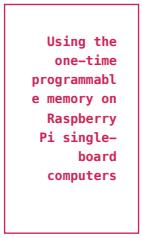
Edit this on GitHub

Raspberry Pi is often used as part of another product. This documentation describes some extra facilities available to use other capabilities of your Raspberry Pi.

One-time programmable settings

WHITE PAPER

Using the one-time programmable memory on Raspberry Pi single-board computers



All Raspberry Pi single-board computers (SBCs) have an inbuilt area of one-time programmable (OTP) memory, which is actually part of the main system on a chip (SoC). As its name implies, OTP memory can be written to (i.e. a binary 0 can be changed to a 1) only once. Once a bit has been changed to 1, it can never be returned to 0. One way of looking at the OTP is to consider each bit as a fuse. Programming it involves deliberately blowing the fuse — an irreversible process, as you cannot get inside the chip to replace it!

This whitepaper assumes that the Raspberry Pi is running the Raspberry Pi operating system (OS), and is fully up-to-date with the latest firmware and kernels.

There are a number of OTP values that can be used. To see a list of all the OTP values, run the following command:

```
$ vcgencmd otp_dump
```

Some interesting lines from this dump are:

- 28 - Serial number

2 Overview

We have already seen the camera board connected through the ribbon cable to (one of) the CSI port(s) on the Raspberry Pi itself.

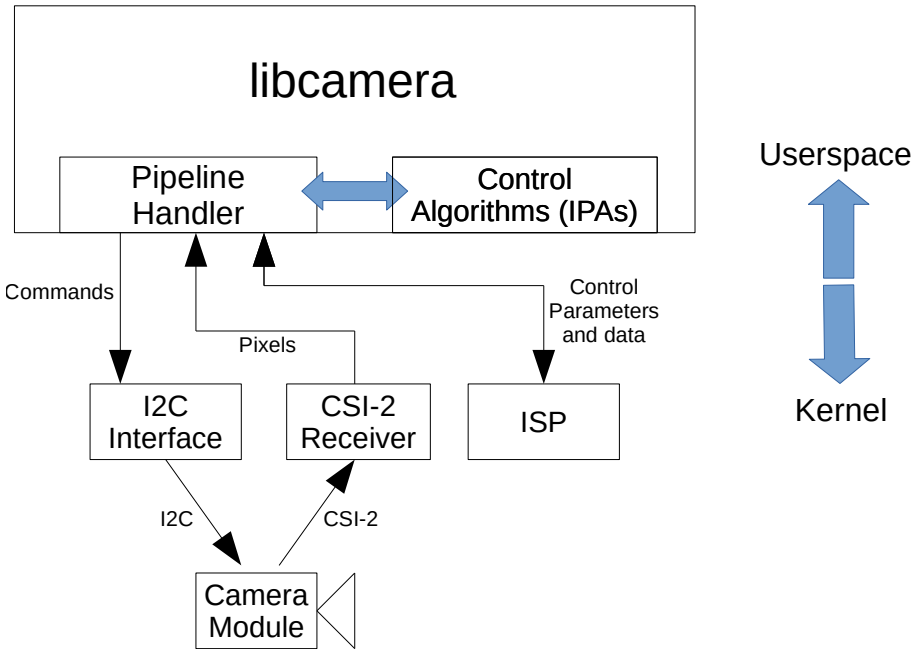


Figure 2: Overview of the libcamera system, explained below.

2.1 The Camera Module

Raspberry Pi already produces compatible camera boards, as listed in the introduction.

These are Bayer sensors, that is, they output so-called "raw" Bayer images which have not yet been processed into anything a user could recognise. These raw pixels are transmitted back to the Raspberry Pi in discrete image *frames*, using the MIPI CSI-2 protocol. This is the 2nd version of the CSI (Camera Serial Interface) defined by the MIPI (originally "Mobile Industry Processor Interface") Alliance, an organisation dedicated to standardising interfaces between different components of mobile devices.

2.2 CSI Connector

Besides supplying power and clock signals, the ribbon cable performs two principal functions. It allows commands (typically register updates) to be sent to the image sensor using the I^2C interface. These values define the operating mode of the sensor, including the output image size and framerate, as well as commands to start and stop streaming images.

Secondly, the output images from the camera are transmitted through the cable along the CSI-2 interface back to the Raspberry Pi.

There are a number of different types of CSI-2 connectors. Raspberry Pi 4s and other older full-sized Raspberry Pis use the larger 15-pin connector. Compute Module 4s and Raspberry Pi 5s use smaller connectors accepting a 22-pin cable.

A more detailed specification of the modules and connector, including a link to the schematics, can be found at https://www.raspberrypi.org/documentation/hardware/camera/README.md.

2.3 On-Chip Hardware

The Raspberry Pi has the following hardware.

- An I^2C interface, for sending commands to the image sensor.
- A CSI-2 Receiver, for receiving image frames, in the form of pixel data, back from the sensor. The CSI-2 Receiver is on the main processor on Raspberry Pi 4 or older devices, where it is known as *Unicam*. On Raspberry Pi 5s, the CSI-2 receiver, along with some early "front end" parts of the ISP (Image Signal Processor) are on the RP1 I/O chip. Collectively, these entities are referred to as the *Camera Front End* (often abbreviated to CFE).
- And an ISP (Image Signal Processor) which converts the raw Bayer images from the sensor into something a user might recognise. On Pi 4 and earlier devices, the ISP also produces statistics from the pixel data, from which a number of software control algorithms will deduce appropriate parameters to be fed to the ISP in order to produce pleasing output images. On Pi 5, the statistics are produced by the CFE (Camera Front End), though the bulk of the processing (the *ISP Back End*) still happens on the main processor.

Interactions between these on-chip hardware devices is mediated though interrupts, responded to by the camera software stack.

2.4 Software and Control Algorithms

libcamera is responsible for handling and fulfilling application requests using Raspberry Pi's *pipeline handler*. In turn, the pipeline handler must ensure that the camera and ISP control algorithms are invoked at the correct moment, and must fetch up to date parameter values for each image frame.

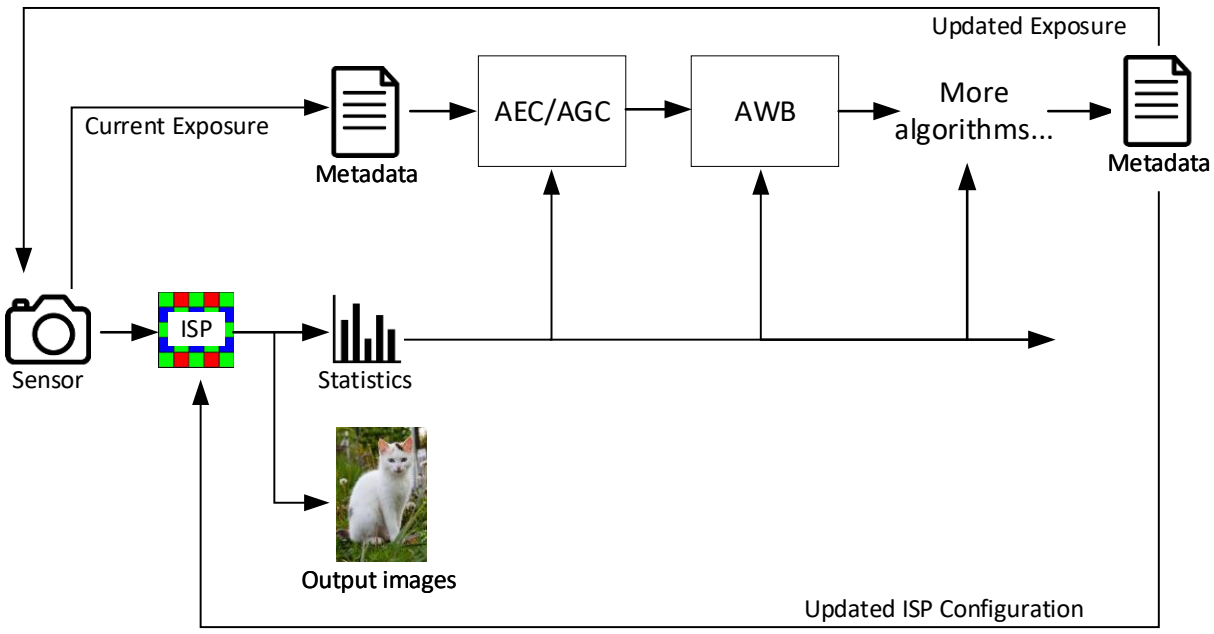


Figure 3: Control algorithms use image metadata and statistics to update their parameters.

The Raspberry Pi control algorithm framework associates a buffer of *metadata* with each image, containing exposure and other information about it. The algorithms each receive this metadata and statistics about the image which were calculated by the hardware ISP (or CFE on Pi 5). They run in turn to compute new and updated values which are deposited back into the metadata.

Finally, the pipeline handler reads values out of the metadata and uses them to update the camera's exposure settings and the ISP parameters for the next frame. There will be more details on this in chapter 4 (Control Algorithm Overview).

3 Driver Framework

There are a number of different types of driver involved in the camera system.

Firstly there are drivers for the various Raspberry Pi hardware devices, including Unicam (Pi 4 and earlier devies), the CFE (Camera Front End) on Pi 5, the ISP and so forth. These are all published in the `raspberrypi/linux` Github repository (https://github.com/raspberrypi/linux). Users and third parties should not need to change these drivers, but if you feel that you do and need help, please reach out on the Raspberry Pi support forums.

3.1 Camera Drivers

Those wishing to add support for new cameras will need to provide a standard Linux kernel driver for the new image sensor. Before embarking on this, we would recommend gaining some experience of Linux kernel drivers, and also of the Linux V4L2 framework.

Much the easiest route to creating a new camera driver is to start from an existing one, and to replace the calculations and register settings with ones that you have obtained from the sensor manufacturer. Both the imx477 (https://github.com/raspberrypi/linux/tree/rpi-6.1.y/drivers/media/i2c/imx477.c) and ov5647 (https://github.com/raspberrypi/linux/tree/rpi-6.1.y/drivers/media/i2c/ov5647.c) would be suitable starting points.

You should endeavour to duplicate all the ioctls and V4L2 controls that you find in those drivers, as most of them will be mandatory for *libcamera* to work correctly.

Note that some sensors are able to return "embedded data" that lists the registers (including the exposure and gain registers) used for each frame. If your sensor is able to do this, then the imx477 driver is probably a better starting point. Otherwise we would recommend looking at the ov5647 driver first.

3.2 Device Tree

The camera driver needs to be listed as part of the Linux device tree so that it is correctly loaded. The details are largely beyond the scope of this document, however, we draw the reader's attention to the following resources.

1. In the `raspberrypi/linux` Github repository (https://github.com/raspberrypi/linux), please consult the `Documentation/devicetree/bindings/media/i2c` directory for example bindings, such as `imx219.txt` or `ov5647.txt`.
2. The existing overlays in the linux source tree at `arch/arm64/boot/dts/overlays` (use `arm` in place of `arm64` for 32-bit platforms). For example, look for `imx219-overlay.dts` or `ov5647-overlay.dts`.

3.3 The `CamHelper` Class

Unfortunately there is a certain amount of information that we need to know about cameras and camera modes but which the kernel driver framework (V4L2) does not supply. For this reason we provide the `CamHelper` class, along with derived class implementations for all of Raspberry Pi's supported sensors. These are all device-specific functions, but implemented in userspace rather than by extending the existing kernel framework. The principal functions of this class are:

1. To convert exposure times (in microseconds) to and from the device-specific representation (usually a number of lines of pixels) used by V4L2.
2. To convert analogue gain values to and from the device-specific "gain codes" that must be supplied to V4L2.
3. To allow parsing of embedded data buffers returned by a sensor.
4. To indicate how many frames of delay there are when updating the sensor's exposure time or analogue gain values.
5. To indicate how many frames are invalid and may need to be dropped on startup, or after changing the sensor mode and re-starting streaming.

Generally speaking it should be sufficient to copy one of our existing implementations and work from there. In particular note that

- Sensors that do not supply embedded data buffers should copy the OV5647 implementation (`cam_helper_ov5647.cpp`).
- Sensors that do supply embedded data buffers should copy the imx477 version (`cam_helper_imx477.cpp`). If the sensor in question follows the general SMIA pattern then this implementation, and the metadata parser, should in fact be quite close to what is required.

The `CamHelper` class methods are well documented in `cam_helper.hpp`, however, we do further draw the reader's attention to the `CameraMode` class, which contains the important parameters related to each different camera mode. The fields of the `CameraMode` are populated by the IPA, so they should automatically contain the correct values once the kernel driver is functioning properly. These fields documented in `controller/camera_mode.h`.

4 Control Algorithm Overview

4.1 Framework

Whilst Raspberry Pi provides a complete implementation of all the necessary ISP control algorithms, these actually form part of a larger framework. The principal components of this framework are described as follows.

- The `Controller` class is a relatively thin layer which manages all the control algorithms running under its *aegis*. It is responsible for loading the control algorithms at startup and invoking their methods when instructed to through *libcamera*.
- The `Algorithm` class is specialised to implement each individual control algorithm. The `Controller` invokes the methods of each algorithm as instructed.

The principal methods of the `Algorithm` class are

1. The `Prepare` method which is invoked just before the ISP is started. It is given information about the captured image (including shutter speed and amount of analogue gain applied by the sensor) and must decide the actual values to be programmed into the ISP.
2. The `Process` method which is invoked once statistics for the image frame are available. This method is given these statistics and is expected to initiate new computations which will determine the values written by the `Prepare` method when it runs again. The `Process` method runs just after the ISP has produced an output frame on Pi 4 and earlier devices, because it is the act of running the ISP that produces that statistics. On a Pi 5, however, statistics are collected by the Camera Front End, so `Process` runs just before `Prepare`, which programs the Back End ISP.

Other methods include the `Read` method, which loads the algorithm and may also initialise default values, and the `SwitchMode` method, which is invoked whenever a new `CameraMode` is selected for use.

In the following discussion, pathnames of specific files are given relative to the location of the Raspberry Pi specific code in the *libcamera* source tree, namely `libcamera/src/ipa/rpi`, unless we explicitly state otherwise.

4.2 Defining and Loading Algorithms

At startup, the `Controller` loads and initialises those algorithms that are listed in a JSON file. The idea is that a separate JSON file (the *camera tuning*) is loaded for each different kind of camera. Every algorithm is registered with the system using a static `RegisterAlgorithm` initialiser, making it easy to add more algorithms to the system.

The JSON file lists camera algorithms by name, conventionally given in a *<vendor>.<moniker>* format. The *moniker* can be thought of as the *type* of the algorithm, such as AWB (Automatic White Balance) or AGC/AEC (Automatic Gain/Exposure Control). For instance, Raspberry Pi's implementation of AWB is named `rpi.awb`. The implementation of each algorithm defines what parameters it has and in its `Read` method how it parses them from the file or sets default values.

If, for example, the mythical *Foo Corporation* were to produce their own implementation of AWB, it might be named `foo.awb`, and it could be listed directly in the JSON camera tuning file. Any algorithms no longer wanted could be deleted from the JSON file, or commented out by renaming them. For instance, changing `rpi.awb` to `x.rpi.awb` will cause the Raspberry Pi version to be silently ignored. We encourage developers to use a separate folder for any algorithms they implement, so while Raspberry Pi's algorithms all live in `controller/rpi`, Foo Corporation's algorithms might live in `controller/foo`.

Finally, we note that the order in which algorithms are listed in the JSON file is the order in which the controller will load and initialise them, and is also the order in which the `Prepare` and `Process` methods of the algorithms will run.

4.3 Standard Algorithms

The framework defines a number of abstract classes derived from `Algorithm` that provide standard interfaces to particular kinds of algorithms. For example, the `class AwbAlgorithm` (`controller/awb_algorithm.hpp`) requires the implementation of a `setMode` method which allows applications to set the AWB mode to, for example, `"tungsten"`, `"daylight"`, and so forth. Developers are encouraged to use these standard algorithm types when this is possible. Where it may be generally helpful, Raspberry Pi will also consider pull requests to add new features to the standard algorithms.

Furthermore, implementations of these standard algorithms are expected, by convention, to have a name which ends with the appropriate moniker for that type of algorithm. So an AWB algorithm would have a name always ending in `awb`, and AGC/AEC (Automatic Gain/Exposure Control) would always end in `agc`, facilitating the type of code shown below.

```
Algorithm *algorithm = controller->GetAlgorithm("awb");
AwbAlgorithm *awb_algorithm = dynamic_cast<AwbAlgorithm *>(algorithm);
if (awb_algorithm) {
        awb_algorithm->setMode("sunny");
}
```

The full set of standard monikers for our different types of algorithm is set out in the table below, though there is no reason in principle why developers should not invent new ones if they are appropriate for their own use cases. The algorithms themselves are explained in detail in the following chapter.

Moniker	Description
`alsc`	Automatic Lens Shading Correction algorithm.
`agc`	Automatic Gain/Exposure Control algorithm (AGC/AEC).
`awb`	Automatic White Balance algorithm (AWB).
`black_level`	Algorithm to set the correct black level for the sensor.
`cac`	Algorithm to mitigate colour fringing caused by lateral chromatic aberration. Pi 5 only.
`ccm`	Algorithm to calculate the correct Colour Correction Matrix (CCM).
`contrast`	Adaptive global contrast and gamma control algorithm.
`denoise`	Algorithm to jointly control spatial denoise, temporal denoise and colour denoise. Pi 5 only.
`dpc`	Algorithm to set the appropriate level of Defective Pixel Correction.
`geq`	Algorithm to set an appropriate level of Green Equalisation.
`lux`	Algorithm that estimates an approximate lux level for the scene.
`noise`	Algorithm to calculate the noise profile of the image given the current conditions.
`sdn`	Algorithm to set an appropriate strength for the Spatial Denoise block. Pi 4 and earlier devices only.
`sharpen`	Algorithm to set appropriate sharpness parameters.
`af`	Algorithm to control a lens driver for Auto Focus.

Table 2: Algorithm type monikers.

4.4 Algorithm Communication

Algorithms communicate with the outside world, and with each other, through *image metadata*. Image metadata is an instance of the `Metadata` class (`controller/metadata.hpp`) which is created for each new image from the camera, and travels alongside the image through the entire processing pipeline; it is available to both the `Prepare` and `Process` methods. Code external to the algorithms themselves will look in the image metadata to find updated values to program into the ISP, and some algorithms may also look in the metadata to find the results calculated by other algorithms that they wish to use.

This information written into the image metadata is normally referred to as the algorithm's *status*, and is implemented by a class whose name ends in `Status`, normally found in the files `controller/*_status.h`. For example, the CCM algorithm (which calculates the colour correction matrix for the ISP) relies on the estimated colour temperature which can be found in the `AwbStatus` object (`controller/awb_status.h`) in the image metadata. In particular, items are stored in the image metadata indexed by a string key, and by convention this key is made up of two parts joined by a period (.), the first being the algorithm moniker (`awb`) and the second the word `status`.

Furthermore, we normally arrange that the status metadata contains any user-programmable settings that affect the result of the algorithm. This is so that an application can request that a setting be changed, and then sit back and monitor the image metadata to discover when it has actually happened. For example, we have seen the `setMode` method of the AWB algorithm. When the `AwbStatus` metadata shows the new mode name in its `mode` field, then we know the change has taken effect.

When replacing existing algorithms, care must be taken still to generate the correct status information in the image's metadata, as other (un-replaced) algorithms may still expect it. So any replacement AWB algorithm would still have to write out an `AwbStatus` object for the CCM algorithm (under the key `awb.status`), otherwise the CCM algorithm would need amending or replacing also. Custom algorithms are free to communicate using the image metadata too, only in this case the expectation is that any metadata keys would begin with the organisation name. To wit, our friends at Foo Corporation might file their secret AWB parameters under `foo.awb.secret_parameters`.

Recall that the order in which `Prepare` and `Process` methods are run matches that of the JSON file, meaning that when one algorithm has a dependency on the status information of another then that determines their relative ordering within the file. So in the case of our example, we will have to list `rpi.ccm` **after** `rpi.awb`.

4.5 Performance Considerations

When writing new control algorithms, care must be taken not to delay the operation of the imaging pipeline any longer than necessary. This means that both the `Prepare` and `Process` methods for any algorithm must take considerably less than a millisecond, and that any significant amount of computation must be backgrounded using an asynchronous thread. Amongst the Raspberry Pi algorithms, examples of this can be found in the AWB (`controller/rpi/awb.cpp`) and ALSC (Automatic Lens Shading Correction - `controller/rpi/alsc.cpp`) algorithms.

Although the details are of course up to developers, the general pattern is for the `Process` method to re-start the computation (in the asynchronous thread) if the previous computation has finished. Often it makes sense to wait a few frames before re-starting it as the results are not usually so critical that we need them immediately, and we can thereby save on CPU load too.

The `Prepare` method normally has a notion of *target values*, and it filters the values that it outputs slowly towards these target values with every frame (for example, using an IIR filter). It watches for any in-progress asynchronous computation to finish though it does not block waiting for it. If unfinished, it will simply leave the target values unchanged and try again on the next frame; if finished, this merely causes an update to its target values, which it will immediately start filtering towards.

and more user-friendly than YOLOv5. It is an advanced model that improves upon the success of YOLOv5 by incorporating modifications that enhance its power and user-friendliness in various computer vision tasks. These enhancements include a modified backbone network, an anchor-free detection head, and a new loss function. Furthermore, it provides built-in support for image classification tasks. YOLOv8 is distinctive in that it delivers unmatched speed and accuracy performance while maintaining a streamlined design that makes it suitable for different applications and easy to adapt to various hardware platforms.

2 Architecture of YOLOv8

As of the current writing, there is no published paper yet on YOLOv8, so detailed insights into the research techniques and ablation studies conducted during its development are unavailable. However, an analysis of the YOLOv8 repository [24] and its documentation [23] over its predecessor YOLOv5 [20], reveals several key features and architectural improvements.

2.1 Architecture components

The YOLOv8 architecture is composed of two major parts, namely the backbone and the head, both of which use a fully convolutional neural network.

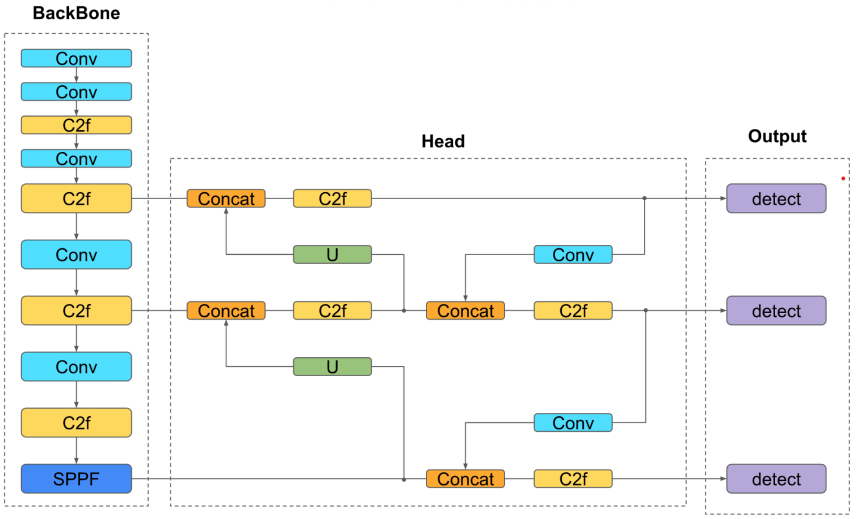


Fig. 2. YOLOv8 Architecture visualization, Arrows represent data flow between layers

Backbone. YOLOv8 features a new backbone network which is a modified version of the CSPDarknet53 architecture [26] which consists of 53 convolutional layers and employs a technique called cross-stage partial connections to enhance

the transmission of information across the various levels of the network.

This Backbone of YOLOv8 consists of multiple convolutional layers organized in a sequential manner that extract relevant features from the input image. The new C2f module integrates high-level features with contextual information to enhance detection accuracy. The SPPF [11] (spatial pyramid pooling faster) module, and the other following convolution layers, process features at various scales. [25]

Head. The head then takes the feature maps produced by the Backbone and further processes them to provide the model's final output in the form of bounding boxes and object classes. In YOLOv8, the head is created to be detachable, which implies that it manages objectness scores, classification, and regression duties in an independent manner. This approach allows each branch to focus on its own task while improving the model's overall accuracy. The U layers (Upsample layers) in Fig. 2. increase the resolution of the feature maps. [25] The head uses a sequence of convolutional layers to analyse the feature maps, followed by a linear layer for predicting the bounding boxes and class probabilities. The head's design is optimised for speed and accuracy, with special consideration given to the number of channels and kernel sizes of each layer to maximise performance.

Finally, the Detection module uses a set of convolution and linear layers to map the high-dimensional features to the output bounding boxes and object classes. The entire structure is designed to be quick and effective, yet it maintains high precision in object detection.

2.2 Architectural Advancements

Anchor-Free Detection. Similar to YOLOv6 and YOLOv7, YOLOv8 is a model that does not rely on anchors. This means that it predicts the centre of an object directly rather than the offset from a known anchor box. Anchor boxes were a well-known challenging aspect of early YOLO models (YOLOv5 and earlier) since these could represent the target benchmark's box distribution but not the distribution of the custom dataset. The use of anchor-free detection minimises the number of box predictions, which speeds up Non-Maximum Suppression (NMS), a complex post-processing phase that sifts through candidate detections following inference. [27]

New Convolution layer. The convolutional (conv) layers in YOLO architecture are responsible for detecting features in input images using learnable filters. These layers detect features at different scales and resolutions, allowing the network to detect objects of varying sizes and shapes. The output of these layers is then passed through other layers to generate bounding boxes and class predictions for each object detected in the image.

Unlike YOLOv5, YOLOv8 uses a different convolution layer called C2f. This

new layer replaces the C3 layer of YOLOv5. The C2f layer in YOLOv8 concatenates the outputs of all the Bottleneck layers, while in the C3 layer of YOLOv5, only the output of the last Bottleneck layer is utilized.

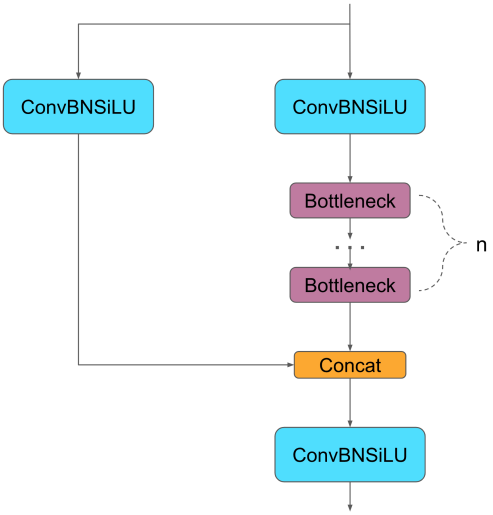


Fig. 3. C3 Module of YOLOv5, the number of bottleneck layers are n. ConvBNSiLU is a block composed of a Conv, a BatchNorm and a SiLU layer.

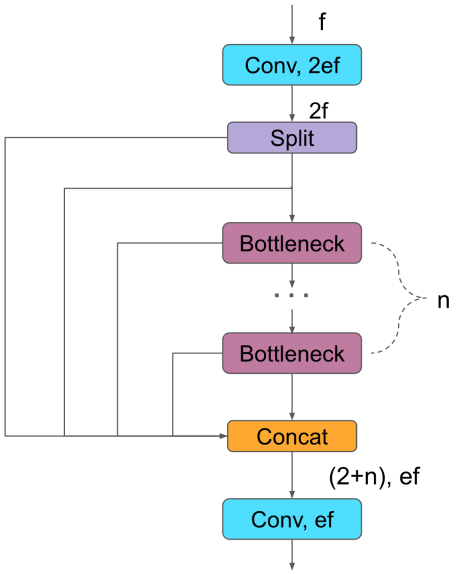


Fig. 4. C2f Module of YOLOv8, Conv is a block composed of a Conv2d, a BatchNorm and a SiLU layer.

The bottleneck in this system is similar to that of YOLOv5 but with a change in the first convolution layer. The kernel size has been increased from 1x1 to 3x3, which is similar to the ResNet block described in 2015. In the neck of the system, the features are concatenated without requiring them to have the same channel dimensions, which reduces the number of parameters and the total size of the tensors.

3 Training and Inference

3.1 Downloadable Python Package via pip

YOLOv8 can now be installed through a PIP package, making it easy for users to install and manage YOLOv5 for training and inference. This simplifies the installation process and allows for easy updates and compatibility with other Python libraries. Users can simply use the pip package manager to install YOLOv8 and start using it for their computer vision tasks, further increasing the accessibility and usability of the model.

YOLOv8 can also be installed from the source on GitHub.

3.2 Command Line Interface (CLI)

One of the most useful features of YOLOv8 is its ultralytics package distributed with a CLI. It supports easy single-line commands without requiring a Python environment. CLI does not require any customisation or Python code. The CLI provides options for specifying dataset paths, model architecture, training parameters, and output directories. This allows users to easily customize the training process according to their specific requirements. Compared to previous versions of YOLO, the CLI in YOLOv8 offers additional options for fine-tuning, model evaluation, and distributed training, providing enhanced flexibility and control over the training process.

3.3 YOLOv8 Python SDK

The YOLOv8 model also comes with a Pythonic Model and Trainer interface, making it easier to integrate the YOLO model into custom Python scripts with just a few lines of code. This enables users to leverage the power of YOLOv8 for object detection, image classification, and instance segmentation tasks with minimal effort. This streamlined integration process is a significant advancement, as it eliminates the need for complex configurations or lengthy setup procedures, making YOLOv8 more accessible and convenient for developers to use in their Python-based projects.

3.4 YOLOv8 Tasks and Modes

The YOLOv8 framework can be used to perform computer vision tasks such as detection, segmentation, classification, and pose estimation. It comes with pre-trained models for each task. The pretrained models for detection, Segmentation and Pose are pretrained on the COCO dataset [25-26], while Classification models are pretrained on the ImageNet dataset.

YOLOv8 introduces scaled versions such as YOLOv8n (nano), YOLOv8s (small), YOLOv8m (medium), YOLOv8l (large), and YOLOv8x (extra big). These several versions provide variable model sizes and capabilities, catering to various requirements and use scenarios.

For Segmentation, Classification and Pose estimation; these various scaled versions use suffixes such as -seg, -cls and -pose respectively. These tasks don't require additional commands and scripts for making masks, contours or for classifying the images.

With a well-labelled and sufficient dataset, the accuracy can be high. Also using a GPU over a CPU is recommended for the training process to further enhance the performance by decreasing computation time.

YOLOv8 offers multiple modes that can be used either through a command line interface (CLI) or through Python scripting, allowing users to perform different tasks based on their specific needs and requirements. These modes are

Train. This mode is used to train a custom model on a dataset with specified hyperparameters. During the training process, YOLOv8 employs adaptive techniques to optimize the learning rate and balance the loss function. This leads to enhanced model performance.

Val. This mode is used to evaluate a trained model on a validation set to measure its accuracy and generalization performance. This mode can help in tuning the hyperparameters of the model for improved performance.

Predict. This mode is used to make predictions using a trained model on new images or videos. The model is loaded from a checkpoint file, and users can input images or videos for inference. The model predicts object classes and locations in the input file.

Export. This mode is used to convert a trained model to a format suitable for deployment in other software applications or hardware devices. This mode is useful for deploying the model in production environments. Commonly used YOLOv8 export formats are PyTorch, TorchScript, TensorRT, CoreML, and PaddlePaddle.

Track. This mode is used to perform real-time object tracking in live video streams. The model is loaded from a checkpoint file and can be used for applications like surveillance systems or self-driving cars.

Benchmark. This mode is used to profile the performance of different export formats in terms of speed and accuracy. It provides information on the size of the exported format, mAP50-95 metrics for object detection, segmentation, and pose, or accuracy_top5 metrics for classification, as well as inference time per image. This enables users to select the most suitable export format for their particular use case, considering their requirements for speed and accuracy.

3.5 User Experience (UX) Enhancements

YOLOv8 improves on prior versions by introducing new modules such as spatial attention, feature fusion, and context aggregation.

Each YOLO task has its own Trainers, Validators and Predictors which can be customized to support user custom tasks or research and development ideas. Callbacks are utilized as points of entry at strategic stages during the train, val, export, and predict modes in YOLOv8. These callbacks accept an object of either a trainer, validator, or predictor, depending on the type of operation being performed.

The model's performance, speed, and accuracy are heavily influenced by YOLO settings, hyperparameters, and augmentation which can also influence model behaviour at various stages of model development, such as training, validation, and prediction. These can be configured by using either CLI or Python scripts.

4 Performance Evaluation

4.1 Object Detection Metrics

mAPval. mAPval stands for Mean Average Precision on the validation set. It is a popular metric used to evaluate the accuracy of an object detection model. Average precision (AP) is calculated for each class in the validation set, and then the mean is taken across all classes to obtain the mAPval score. A higher mAPval score indicates better accuracy of the object detection model in detecting objects of different classes in the validation set.

Speed CPU ONNX. This relates to the object identification model's speed while running on a CPU (Central Processing Unit) using the ONNX (Open Neural Network Exchange) runtime. ONNX is a prominent deep learning model representation format, and model speed can be quantified in terms of inference time or frames per second (FPS). Higher values for Speed CPU ONNX indicate faster inference times on a CPU, which can be important for real-time or near-real-time applications.

Speed A100 TensorRT. This refers to the speed of the object detection model when running on an A100 GPU (Graphics Processing Unit) using TensorRT, which is an optimization library developed by NVIDIA for deep learning inference. Similar to Speed CPU ONNX, the speed can be measured in terms of inference time or frames per second (FPS). Higher values for Speed A100 TensorRT indicate faster inference times on a powerful GPU, which can be beneficial for applications that require high throughput or real-time processing.

Latency A100 TensorRT FP16 (ms/img). This refers to the latency or inference time of the object detection model when running on an NVIDIA A100 GPU with TensorRT optimization, using the FP16 (half-precision floating point) data type. It indicates how much time the model takes to process a single image, typically measured in milliseconds per image (ms/img). Lower values indicate faster inference times, which are desirable for real-time or low-latency applications.

Params (M). Params (M) refers to the number of model parameters in millions. It represents the size of the model, and generally larger models tend to have more capacity for learning complex patterns but may also require more computational resources for training and inference.

FLOPs (B). FLOPs (B) stands for Floating point operations per second in billions. It is a measure of the computational complexity of the model, indicating the number of floating-point operations the model performs per second during inference. Lower FLOPs (B) values indicate less computational complexity and can be desirable for resource-constrained environments, while higher values indicate more computational complexity and may require more powerful hardware for efficient inference.

4.2 Benchmark Datasets and Computational Efficiency

Performance of YOLOv8 on COCO. The COCO val2017 dataset [28-29] is a commonly used benchmark dataset for evaluating object detection models. It consists of a large collection of more than 5000 diverse images with 80 object categories, and it provides annotations for object instances, object categories, and other relevant information. The dataset is an Industry-Standard benchmark for object detection performance and for comparing the accuracy and speed of different object detection models.

Table 1. Performance of YOLOv8 Pretrained Models for Detection. Sourced from the Ultralytics GitHub Repository (https://github.com/ultralytics/ultralytics)

Model	size (pixels)	mAPval 50-95	Speed CPU ONNX (ms)	Speed A100 TensorRT (ms)	params (M)	FLOPs (B)
YOLOv8n	640	37.3	80.4	0.99	3.2	8.7
YOLOv8s	640	44.9	128.4	1.20	11.2	28.6
YOLOv8m	640	50.2	234.7	1.83	25.9	78.9
YOLOv8l	640	52.9	375.2	2.39	43.7	165.2
YOLOv8x	640	53.9	479.1	3.53	68.2	257.8

The training and results were sourced from Ultralytics GitHub repository [24][1]. All scaled versions of YOLOv8 along with previous versions of YOLO i.e. YOLOv5, YOLOv7 were trained on COCO. Here mAPval values are for single-model single-scale on the COCO val2017 dataset and Speed is averaged over COCO val images using an Amazon EC2 P4d instance.

[1] Ulatralytics GitHub repository https://github.com/ultralytics/ultralytics

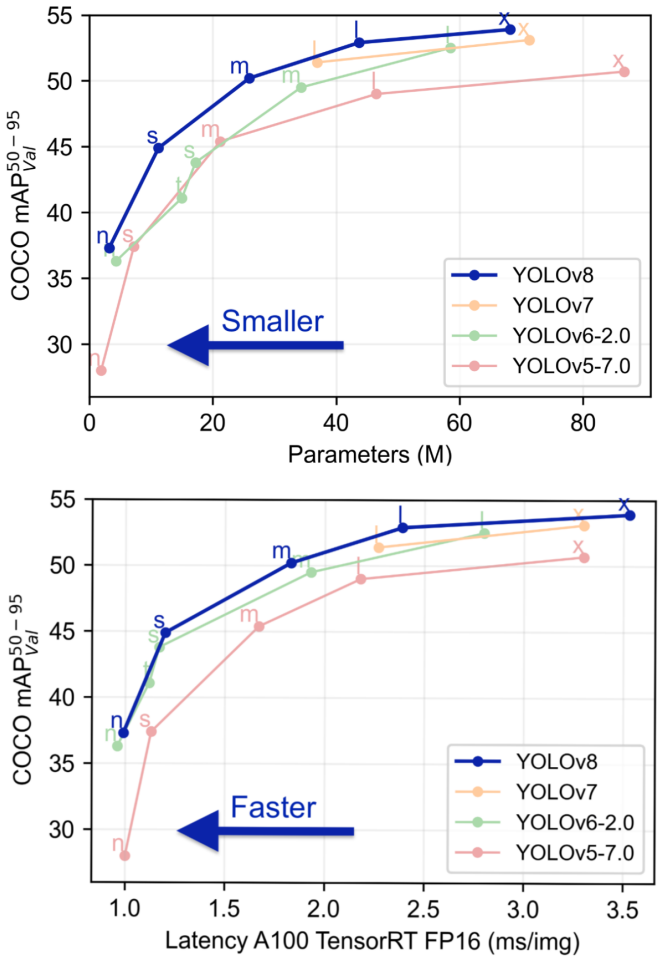


Fig. 5. Performance comparison of all scaled versions of YOLOv8, YOLOv7, YOLOv6 and YOLOv5 (Source: GitHub repository "ultralytics/ultralytics" by Ultralytics, `https://github.com/ultralytics/ultralytics`)

Performance of YOLOv8 on RF100. The Roboflow 100 (RF100) dataset [30-31] is a diverse, multi-domain benchmark comprising 100 datasets created by using over 90,000 public datasets and 60 million public images from the Roboflow Universe, a web application for computer vision practitioners. The dataset aims to provide a more comprehensive evaluation of object detection models by offering a wide range of real-life domains, including satellite, microscopic, and gaming images. With RF100, researchers can test their models' generalizability on semantically diverse data.