Urmia University

In the Name of God

Faculty of Electrical Engineering, Computer Science and Advanced Technologies

Design and Implementation of an Intelligent Home Object Detection System Using Deep Neural Networks and Raspberry Pi 5 Processor

Submitted in Partial Fulfillment of the Requirements for the Degree of Bachelor of Science in Electrical Engineering – Electronics

Author: Mahya Kheirandish
Supervisor: Dr. Mashoufi

Academic Year: 2024–2025

Abstract

With the rapid advancement of artificial intelligence technologies, computer vision, and the Internet of Things (IoT), the integration of these technologies into the field of robotics has become a central focus of research and development. The ability of robots to perceive and analyze their surroundings is a fundamental requirement for performing complex tasks and interacting effectively with humans and objects.

This thesis presents the design and implementation of an intelligent indoor **instance segmentation** system, intended to serve as a visual subsystem in service and domestic robots. The proposed system is based on convolutional neural networks and utilizes the lightweight **YOLOv8n** model. It is capable of detecting various objects in real-time, accurately segmenting them from the background, and displaying the results simultaneously on both a graphical user interface (GUI) and an LCD screen for the operator or user.

The system is implemented using **Raspberry Pi 5** hardware and a USB camera, which, thanks to their compact size and low power consumption, enable real-time model execution in resource-constrained environments. Key objectives of this project include model optimization, memory usage reduction, and increased processing speed to ensure reliable performance in robotic scenarios such as autonomous navigation, obstacle avoidance, object search and recognition, item sorting, and collaboration with robotic arms.

Additionally, a user-friendly graphical interface has been developed to allow real-time monitoring and management of detection results, and the simultaneous display of data on an LCD facilitates interaction with the robot, even for non-expert users. Beyond applications in smart homes and service robots, the system is also suitable for use in educational robotics, research projects, lightweight industrial robots, and monitoring systems.

Ultimately, this project combines modern deep learning techniques, image processing, and user interface design to deliver a practical, cost-effective, and scalable framework for integrating computer vision into robotics—laying a foundation for the development of the next generation of intelligent robots.

Keywords: Object Detection, YOLOv8, Deep Learning, Raspberry Pi 5, Computer Vision, Robotics, Smart Home, Image Processing, Resource Optimization

Chapter 1: Introduction and Conceptual Foundations

Chapter 2: Fundamentals of YOLOv8, Neural Networks, I²C Protocol, and Graphical User Interface (GUI)

Chapter 3: Hardware and Software System Design

3.1 Introduction
3.2 Hardware Design
3.3 Software Design
3.4 Hardware-Software Integration
3.5 Summary

Chapter 4: Implementation and Full Coding

4.1 Introduction
4.2 Hardware and Software Environment
4.3 Raspberry Pi Setup and Code Deployment
4.4 Step-by-Step Explanation of Coding on Laptop and Raspberry Pi
4.5 Project Execution Stages on Raspberry Pi
4.6 System Workflow on Raspberry Pi

Chapter 5: Testing, Performance Evaluation, and Results Analysis

5.1 Introduction
5.2 Experimental Scenario
5.3 Models and Hardware Specifications
5.4 Model Execution Results on Laptop
5.5 Model Execution Results on Raspberry Pi
5.6 Thesis Conclusion

1.1 Introduction

With the advancement of technology and the growing development of smart home systems and robotics, the design of systems capable of recognizing and analyzing surrounding objects has become increasingly significant. In robotics, such capabilities are not only essential for perceiving the environment but also for making decisions and executing precise operations. These systems enable robots to effectively interact with their surroundings, plan movement paths based on detected obstacles and objects, and even autonomously perform tasks such as moving, sorting, or identifying specific items.

1.1.1 Image Processing

Image processing acts as the "eyes" of robotic systems. This field of computer science focuses on analyzing and manipulating digital images to extract useful information, which is then provided to the robot's decision-making modules. The image processing pipeline typically involves the following stages:

1. **Image Preprocessing**: Enhancing quality, reducing noise, and adjusting size or contrast of images—essential for the reliable performance of the robot's vision system.
2. **Feature Extraction**: Identifying edges, shapes, textures, and specific patterns that help the robot accurately detect and distinguish objects.
3. **Analysis and Decision-Making**: Using artificial intelligence algorithms and models—such as convolutional neural networks (CNNs)—to detect, classify, or track objects. These capabilities are crucial in robotics for obstacle avoidance and mission execution.

Applications of Image Processing in Robotics are diverse and include:

- **Home Service Robots:** Object detection for moving items, cleaning specific areas, or finding lost objects.
- **Autonomous Robotic Systems:** Obstacle detection and safe path planning for navigation.
- **Industrial Robots:** Sorting parts, quality control of products, and precise control of robotic arms.
- **Rescue and Medical Robotics:** Identifying objects or vital signs in critical environments or during delicate surgeries.

In this project, image processing combined with the lightweight YOLOv8 models running on low-power **Raspberry Pi 5** hardware plays a key role in providing essential capabilities for robotic systems. These capabilities can be easily integrated into a robotic platform to enable real-time perception and appropriate reactive behaviors.

1.1.2 Artificial Intelligence and Artificial Neural Networks

Especially **Convolutional Neural Networks (CNNs)**, which form the backbone of robotic vision. These technologies enable robots not only to see their environment but also to understand it and make decisions accordingly. In this project, the lightweight and fast YOLOv8n model with **Instance Segmentation** capability enables precise object detection and accurate separation of objects from the background.

In the context of robotics, these two capabilities play critical roles:

1. **Object Detection for Navigation and Obstacle Avoidance**
   When a robot equipped with a camera scans its surroundings, YOLOv8n can identify items, obstacles, or target objects and localize them with bounding boxes. This data helps the robot's navigation system adjust its path or avoid collisions.
2. **Instance Segmentation for Precise Interaction**
   Instance segmentation is a more advanced stage that separates the boundaries of each object on a pixel-by-pixel basis. This is important in robotics because:

- A robotic arm can detect the exact edges of an object and safely grasp it.
- Sorting or packaging robots can independently identify and manipulate multiple closely spaced objects.
- Industrial or home robots can record the exact size, shape, and position of objects to make better decisions.

3. Integration with Robotic Control Systems

The data generated by **YOLOv8n** and **YOLOv8n-seg** can be directly sent to the robot's motion control and decision-making modules. In this way, the robot not only "sees" but also "understands" what surrounds it and how to interact with the environment.

Examples of Robotic Applications:

- **Home Robots:** Finding and fetching items, organizing objects, or assisting the elderly.
- **Industrial Robots:** Detecting and separating parts on production lines.
- **Rescue Robots:** Identifying obstacles and objects in hazardous or debris-filled environments.

In fact, the technology implemented in this project for smart homes can serve as the core vision system of a robot and, with slight modifications, be applied across various service, industrial, and rescue domains.

1.2 Relationship Between Object Detection, Instance Segmentation, and Artificial Intelligence

Detecting and segmenting objects in images without artificial intelligence and deep neural networks is almost impossible or very limited. Neural networks enable the system to:

1. **Automatic Feature Learning:**
   Convolutional Neural Networks (CNNs) can extract patterns, colors, shapes, and textures of objects from training data. These features allow the model to recognize and differentiate various objects without manual rule-setting.
2. **Accurate Detection Resistant to Environmental Variations:**
   YOLOv8 and YOLOv8n-seg models, powered by deep learning, can detect objects even under varying lighting conditions, diverse angles, and complex backgrounds. Neural networks analyze data instead of simple pattern matching, thus providing higher generalization capability.

3. **Advanced Instance Segmentation Capability:**
   Precise separation of each object from the background and other objects requires complex pixel-level analysis, which is difficult and inefficient with traditional algorithms. Neural networks learn relationships between pixels and objects, producing accurate masks for each object and enabling intelligent environment processing.
4. **Capability for Continuous Improvement and Learning:**
   AI-based systems can improve their models by collecting new data, increasing detection and segmentation accuracy. This feature makes the system dynamic and reliable in real-world environments.

Therefore, the use of neural networks and AI forms the foundation for the performance of intelligent object detection and segmentation systems. Without them, achieving the required accuracy, speed, and flexibility in home environments would be nearly impossible.

1.3 Problem Statement

With advancements in smart home technology, the demand for systems capable of automatically detecting and analyzing objects in the environment has steadily increased. These systems have diverse applications, including:

- **Smart Home Monitoring:** Motion detection and identification of notable objects.
- **Inventory Management of Household Items:** Tracking quantity and location of objects.
- **Assistance for Elderly or Mobility-Impaired Individuals:** Reminding about essential items and informing about their location.

1.3.1 Challenges of Implementing the System on Raspberry Pi 5

1. **Hardware Limitations:**
   The ARM processor of Raspberry Pi 5 has limited computational power and lacks a powerful GPU, making it challenging to run large deep learning models.
   Limited memory necessitates lightweight and optimized models to enable real-time execution.

2. **Complexity of Deep Learning Models:**
   YOLOv8 models are optimized for object detection and offer high processing speed. However, adding the instance segmentation capability to precisely delineate the boundaries of each object requires additional computations, which increases processing time. For example, the YOLOv8n-seg version takes approximately 3 seconds to process a single image.

3. **Need for User Interaction and Result Display:**
   Simply detecting objects is not sufficient; users must be able to view the information clearly and effectively.
   Displaying object names on an LCD via the I^2C protocol and providing a graphical user interface (GUI) delivers information quickly and in a user-friendly manner.
   The choice between Tkinter and PyQt5 depends on factors such as simplicity, lightweight operation, and advanced features like segmentation mask overlays and threading support.

4. **Balance Between Accuracy and Speed:**
   In real-time applications, both processing speed and detection accuracy are critical.
   The selection of the model and algorithm optimization for Raspberry Pi 5 must strike a balance between segmentation quality and inference speed.

5. **Operational Challenges in Real-World Environments:**
   Environmental lighting, camera angle, and object distance can reduce detection accuracy.
   Integrating peripheral hardware such as USB webcams and LCDs, and ensuring their synchronization with the software, requires careful design.

1.3.4 Technical Choices and Design Decisions

- **Operating System:** Raspberry Pi OS, based on Linux, was chosen for its lightweight nature, full support for Python libraries, OpenCV, and TensorFlow, and its efficient resource management on the limited hardware of Raspberry Pi 5. This OS provides a stable environment for installing and running deep learning models and integrates well with LCD displays and the I²C protocol.
- **Model Selection:** YOLOv8n, a lightweight and fast version of YOLOv8, is suitable for object detection on the constrained hardware of Raspberry Pi 5.
- **Segmentation:** YOLOv8n-seg provides precise masks for each object but requires about 3 seconds per image for inference. For real-time applications, optimization and possibly reducing image resolution may be necessary.
- **GUI Framework:** Tkinter is appropriate for simple and lightweight interfaces, but PyQt5 is preferred for advanced features such as mask overlay, threading, and enhanced user interaction.
- **LCD and I²C:** Displaying detected object names on the LCD helps users quickly understand the environment without looking at a monitor. The connection is made via I²C with two wires (SDA and SCL), supporting multiple peripheral devices.

1.3.5 Main Project Problem Statement

How can an intelligent object detection and segmentation system be implemented in a home environment that:

1. Runs effectively on the limited hardware of Raspberry Pi 5,
2. Achieves acceptable accuracy and speed,
3. Displays detection and segmentation results on both a GUI and an LCD,
4. Provides a simple and practical user experience?

This problem is directly related to neural networks, image processing, and deep learning, with the project focusing on designing a practical, cost-effective, and intelligent system capable of performing reliably in real home environments.

1.4 Project Objectives

The main goal of this project is to design and implement an intelligent object detection and segmentation system based on Raspberry Pi 5, which can serve as the "eyes" and vision system for service, home, and research robots. Using deep neural networks, this system enables precise recognition of objects, obstacles, and the surrounding environment to facilitate intelligent decision-making by the robot.

Specific objectives and explanations:

1. **Implementation of Object Detection with YOLOv8n:**
   Enable fast and accurate identification of objects and obstacles along the robot's path. This allows the robot to avoid collisions and perform tasks with greater accuracy in dynamic and complex environments.
2. **Adding Instance Segmentation Capability:**
   Precisely delineate identified objects to determine their size, shape, and position relative to the robot. This feature is essential for tasks such as picking objects with a robotic arm or performing delicate operations like arranging or sorting items.
3. **Development of a Graphical User Interface (GUI):**
   Design a graphical environment to monitor and control the robot's vision system. Operators can observe robot performance, review detected objects, and even send control commands via the GUI.
4. **Displaying Results on LCD:**
   Provide real-time display of detection and segmentation information on an LCD connected to the robot, delivering rapid feedback to users or operators without requiring a secondary computer.
5. **Optimizing Processing Speed and Resource Consumption:**
   Increase image processing frame rate (FPS) and reduce latency to enable real-time operation, allowing the robot to react swiftly in dynamic environments.
6. **Data Management and Analysis:**
   Store and process image data to improve algorithms, train new models, and assist autonomous robot decision-making in similar future scenarios.
7. **Enhancing System Flexibility:**
   Allow integration of complementary sensors (e.g., distance sensors, GPS, temperature sensors) to extend the robot's capabilities in specific applications such as rescue operations or intelligent transport.
8. **Creating an Educational and Research Platform:**
   Provide an environment for training students and researchers in machine vision, artificial intelligence, and robotics, enabling practical experimentation and development of more advanced systems.

1.5 Necessity of the Project

Implementing an intelligent object detection and segmentation system in a home environment using Raspberry Pi 5 and deep neural networks is necessary for several reasons:

1. **Providing a Practical and Economical Solution for Smart Homes:**
   Offer a low-cost, easily installable system for object recognition and environmental management, reducing reliance on expensive equipment and powerful server-grade processors.

2. **Enhancing Interaction and User Experience:**
   Simultaneous display of recognition results on both LCD and GUI simplifies use and facilitates accurate control and reporting.

3. **Practical Training and Education in Modern Technologies:**
   Combining image processing, deep learning, and GUI design creates an excellent platform for developing practical skills and research opportunities in applied AI and IoT.

4. **Evaluating Feasibility of Lightweight Models on Limited Hardware:**
   Testing the performance of YOLOv8n and YOLOv8n-seg models on Raspberry Pi 5 as a low-power hardware solution, analyzing speed, accuracy, and resource usage to guide future smart system designs.

5. **Potential for Development and Application in Various Fields:**
   Smart monitoring systems for object detection, lost object identification, and environmental status checks.
   Inventory management for home items, counting, and tracking in kitchens or storage areas.
   Data analysis and AI training: gathered data can enhance models or be utilized in other research projects.

1.6 Encouraging the Development of Personalized Intelligent Systems

- Providing a framework that allows users to customize models, displays, and interfaces according to their specific needs.
- Establishing a development platform for future projects in the fields of smart homes and robotics.

1.7 Overview of Key Technologies

YOLOv8n

- Lightweight and fast, suitable for real-time execution on Raspberry Pi 5.
- The segmentation version (YOLOv8n-seg) enables precise boundary detection of objects.

GUI: Tkinter vs PyQt5

- Tkinter: Simple, lightweight, suitable for basic interfaces.
- PyQt5: Advanced, appropriate for displaying segmentation masks and complex user interactions.

I²C Protocol and LCD

- A bidirectional serial protocol that allows connection of multiple devices to the Raspberry Pi.
- LCD is used for displaying object names and providing quick information to the user.

1.7 Thesis Structure

- **Chapter 1:** Introduction, basic concepts, and technologies related to image processing and robotics.
- **Chapter 2:** Fundamentals of YOLOv8, neural networks, I²C, and graphical user interfaces (GUI) in robotic systems.
- **Chapter 3:** Hardware and software design with a focus on robotics and real-time processing.
- **Chapter 4:** Implementation and coding of the machine vision system.
- **Chapter 5:** Testing, evaluation, and analysis of results with an emphasis on applications in intelligent robots.

1.8 Summary

This chapter discussed the importance of image processing, artificial intelligence, and neural networks in creating intelligent systems. The technologies used, their limitations, advantages, and the role of neural networks in object detection and instance segmentation were introduced. These topics provide not only the theoretical and practical foundation necessary for the design and implementation of the project but also create the groundwork for developing robotic vision systems. Such systems enable robots to accurately perceive their environment, detect obstacles, safely interact with objects, and perform complex tasks across industrial, service, and domestic domains.

Chapter 2: Fundamentals of YOLOv8, Neural Networks, I²C, and Graphical User Interface (GUI)

2.1 Introduction

This chapter provides a comprehensive explanation of the theoretical and technical foundations of three key components critical to the system's operation. First, the YOLOv8 model, one of the most advanced deep learning algorithms for object detection and segmentation, is examined. This model combines high processing speed with notable detection accuracy, enabling simultaneous identification of multiple objects in dynamic environments, and forms the core of the machine vision processing in the system.

Next, the I²C communication protocol is introduced, along with its mechanism for data exchange between the Raspberry Pi central controller and the LCD display. This two-wire protocol with multi-device addressing offers an efficient and straightforward method for controlling and displaying outputs.

Finally, the design of the graphical user interface (GUI) using either Tkinter or PyQt5 libraries is discussed. This section focuses on user interaction with the system, displaying detection results, and providing necessary controls to manage system performance.

In summary, a thorough understanding of these three components and their interconnections establishes a solid foundation for the design, implementation, and optimization of the system.

2.2 Object Segmentation with Deep Learning

2.2.1 Convolutional Neural Networks (CNNs)

Convolutional Neural Networks (CNNs) are among the most important and widely used models in machine vision and deep learning. The primary goal of CNNs is to automatically identify patterns and features within images, eliminating the need for manual feature extraction.

2.2.2 Structure and Main Components of CNN

A typical CNN consists of several key layer types:

1. **Convolutional Layer:**
   This layer uses filters (kernels) to detect local features of the image such as edges, corners, and textures. Each filter extracts a specific feature, producing a feature map. By sliding over the image, filters identify similar regions and help the network learn more complex patterns.
2. **Activation Layer:**
   Usually employs a nonlinear function like ReLU (Rectified Linear Unit) to enable the network to learn complex relationships between features, thereby enhancing the model's ability to recognize nonlinear object characteristics.
3. **Pooling Layer:**
   Reduces the spatial dimensions of feature maps while preserving important information, decreasing computational complexity. Common pooling types include Max Pooling and Average Pooling.
4. **Fully Connected Layer:**
   Located at the end of the network, these layers combine the extracted features with trained weights to perform prediction tasks such as image classification or object detection, producing the final output.

2.2.3 Application of CNN in Object Detection

CNN plays a vital role in object detection. Advanced models like YOLO use CNNs to extract image features and predict the locations and categories of objects. In other words, CNN forms the backbone of YOLO, and without it, accurate and fast object detection would not be possible.

The main advantage of CNN over traditional methods is its ability to learn features automatically from raw image data. This capability enables models like YOLO to detect multiple objects simultaneously, even under challenging conditions such as varying lighting, viewing angles, or complex backgrounds.

2.2.4 How CNN Works in YOLO

- The input image is passed through convolutional layers to extract important features.
- Generated feature maps are processed by pooling and activation layers to reduce dimensions and highlight key features.
- Fully connected layers then predict bounding box locations and class labels.
- Unlike multi-step processing methods, YOLO examines the entire image in one step and produces outputs simultaneously.

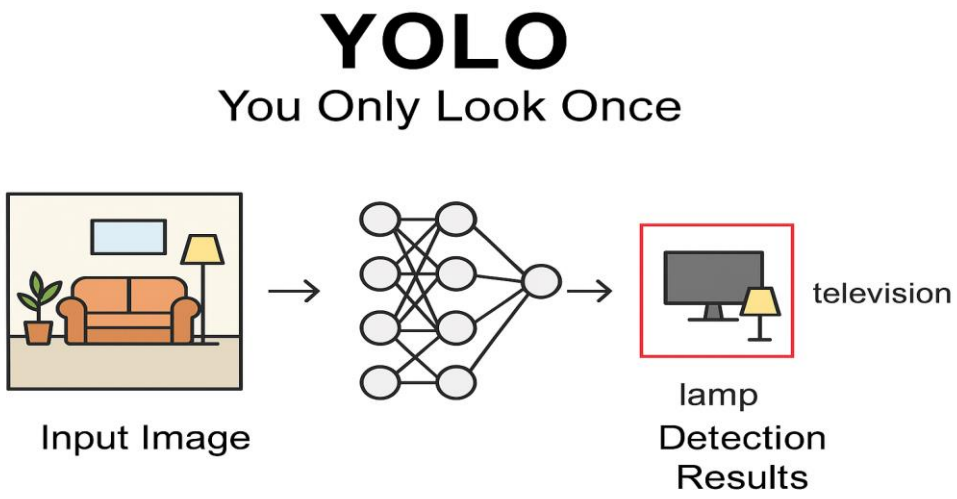
CNN is a powerful tool for feature extraction and learning complex image patterns. Combining CNN with models like YOLO enables precise real-time object detection and segmentation, even on low-power hardware like Raspberry Pi.

2.3 Background of the YOLO Algorithm

YOLO, an acronym for "You Only Look Once," is one of the most advanced and widely used object detection algorithms in computer vision and deep neural networks.

Unlike traditional methods that require multi-stage image processing to identify objects, YOLO processes the entire image in a single forward pass through the neural network and simultaneously predicts object locations (bounding boxes) and class labels.

This innovative approach dramatically increases speed, making YOLO suitable for real-time applications.



2.3.1 Key Features of YOLO

- **Single-stage processing:** Significantly faster than two-stage methods such as Faster R-CNN.
- **End-to-end learning:** The entire pipeline from input image to final output is trained within a single network.
- **Multi-object detection capability:** Simultaneous detection and labeling of multiple objects in one image.
- **Real-time performance:** Suitable for applications like robotics, video surveillance, and autonomous vehicles.

2.3.2 Common Applications of YOLO

- Intelligent surveillance and security systems
- Robotics and autonomous control
- Object detection in medical imaging
- Identification in satellite and drone imagery
- Smart homes and Internet of Things (IoT)

2.4 Introduction to YOLOv8 and the Nano Version

YOLOv8n (where "n" stands for Nano) is the smallest and lightest version of the YOLOv8 model series. Introduced by Ultralytics in 2023, its main goal is to deliver high speed and low resource consumption while maintaining acceptable accuracy.

YOLOv8n utilizes the advanced YOLOv8 architecture but with fewer parameters and layers. It is optimized to run efficiently on lightweight and low-power hardware such as Raspberry Pi, Jetson Nano, and IoT devices.

2.4.1 Key Features of YOLOv8n

1. **Lightweight and compact:** Nano architecture contains approximately 3 to 3.2 million parameters, enabling execution even on low-performance processors.
2. **High speed:** Capable of real-time processing at relatively high resolutions (e.g., 640×640) on low-power systems.
3. **Multi-task:** Supports three main tasks:
    - Object Detection
    - Instance Segmentation (pixel-level object delineation)
    - Image Classification

4. **Optimized for embedded hardware:** Suitable for projects with constraints on power consumption and memory.
5. **Acceptable accuracy relative to size:** Despite its small size, it performs close to larger versions in many everyday applications.

2.4.2 Advantages of Using YOLOv8n in Smart Home Projects

- Runs on Raspberry Pi 5 without requiring a powerful GPU.
- Real-time detection of household objects such as sofas, tables, TVs, vases, and other items.
- Low energy consumption, suitable for long-term operation systems.
- Easy training and updating using Ultralytics' Python library.

Complete List of Classes in the Default YOLOv8n Version:

شماره	کلاس (انگلیسی)	شماره	کلاس (انگلیسی)	شماره	کلاس (انگلیسی)	شماره	کلاس (انگلیسی)
1	person	21	Elephant	41	wine glass	61	dining table
2	bicycle	22	Bear	42	Cup	62	toilet
3	car	23	Zebra	43	Fork	63	TV
4	motorcycle	24	Giraffe	44	Knife	64	laptop
5	airplane	25	Backpack	45	Spoon	65	mouse
6	bus	26	Umbrella	46	Bowl	66	remote
7	train	27	Handbag	47	Banana	67	keyboard
8	truck	28	Tie	48	Apple	68	cell phone
9	boat	29	Suitcase	49	Sandwich	69	microwave
10	traffic light	30	Frisbee	50	Orange	70	oven
11	fire hydrant	31	Skis	51	Broccoli	71	toaster
12	stop sign	32	Snowboard	52	Carrot	72	sink
13	parking meter	33	sports ball	53	hot dog	73	refrigerator
14	Bench	34	Kite	54	Pizza	74	book
15	Bird	35	baseball bat	55	donut	75	clock
16	Cat	36	baseball glove	56	cake	76	vase
17	dog	37	Skateboard	57	chair	77	scissors
18	horse	38	Surfboard	58	couch	78	teddy bear
19	sheep	39	tennis racket	59	potted plant	79	hair drier
20	cow	40	Bottle	60	bed	80	toothbrush

YOLOv8n is an ideal choice for projects that require speed, lightweight design, and local execution on small hardware. Therefore, in this project, YOLOv8n has been used so that the system can detect objects in the home environment on a Raspberry Pi 5 with minimal latency.

2.5. Segmentation with YOLOv8n-seg

YOLOv8n-seg is a version of YOLOv8n that, in addition to Object Detection, provides Instance Segmentation capabilities. In this mode, the model not only detects the type and general position (Bounding Box) of an object but also identifies all the pixels belonging to that object.

Feature	Object Detection	Instance Segmentation
Output	Bounding Box (rectangle)	Precise pixel-level mask
Shape accuracy	Low (rectangle only)	High (true object boundary)
Applications	Counting and locating objects	Detailed processing, shape separation, robotics

In other words, the output of YOLOv8n-seg provides each detected object with an accurate mask that shows the true area of the object within the image. This capability is highly valuable for projects requiring precise object shapes and boundaries.

2.5.1 Advantages of YOLOv8n-seg

1. **Higher accuracy in object boundary detection** – Instead of a simple bounding box, the actual shape of objects is precisely identified.
2. **Optimized for complex processing** – For example, in home robotics, accurately determining object boundaries is essential for grasping or moving items.
3. **Maintains high speed** – Despite the heavier task of segmentation, the Nano version remains suitable for real-time processing on lightweight hardware like Raspberry Pi 5.
4. **Direct applications in smart homes** – Such as detecting the exact area of a glass on a table for a robotic arm or precisely verifying the presence of specific objects in the scene.

In the home object detection project, using YOLOv8n-seg allows not only detecting and labeling household items (e.g., sofa, table, or TV) but also extracting the precise boundaries of each item. This feature is particularly useful for scene analysis, robot interaction with objects, and more accurate image processing.

2.6 I²C Protocol and LCD Control with Raspberry Pi

The **I²C protocol** (Inter-Integrated Circuit) is a bidirectional serial communication interface that allows communication between a single master controller and multiple slave devices using only two lines. These two lines are:

- **SDA (Serial Data Line):** Data line for sending and receiving information between devices
- **SCL (Serial Clock Line):** Clock line for synchronization and timing of data transfer

On the **Raspberry Pi 5**, the I²C pins are defined as follows:

Physical Pin	GPIO Number	Function
3	GPIO 2	SDA
5	GPIO 3	SCL

In this project, the Raspberry Pi 5 acts as the Master, and the LCD display with an I²C module acts as the Slave. The hardware connection is as follows:

1. Connect the LCD's data line (SDA) to GPIO 2 on the Raspberry Pi
2. Connect the LCD's clock line (SCL) to GPIO 3 on the Raspberry Pi
3. Connect the LCD's power (VCC) to 3.3V or 5V on the Raspberry Pi
4. Connect the LCD's ground (GND) to GND on the Raspberry Pi

Due to the shared bus nature of I²C, multiple devices can be connected to the same SDA and SCL lines, as long as each device has a unique address to avoid data collisions.

Advantages of the I²C Protocol:

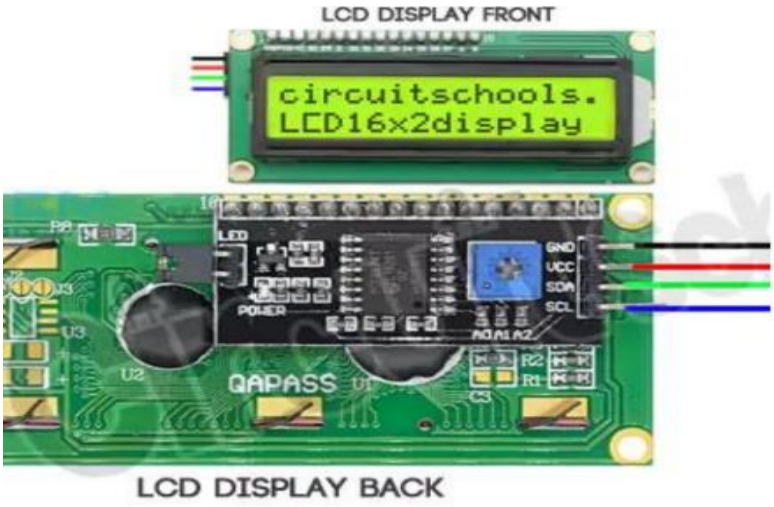
- Saves wiring: only two lines are needed to communicate with multiple devices.
- Multi-addressing: allows connecting multiple devices to one Master without interference.
- High flexibility: suitable for embedded systems and microcontrollers.
- Bidirectional data transfer: the Master can send or receive data.

Application in the Project:
After processing images with the YOLOv8 model, the detection results are sent to the Raspberry Pi. The Raspberry Pi then transfers this information to the LCD display via the I²C protocol so that the user can view the detection results in real-time.

2.7 Introduction and Connection of LCD

Introduction to LCD1602
The LCD1602 is one of the most commonly used character LCD displays in electronics and embedded systems projects. Its name is derived from the following features:

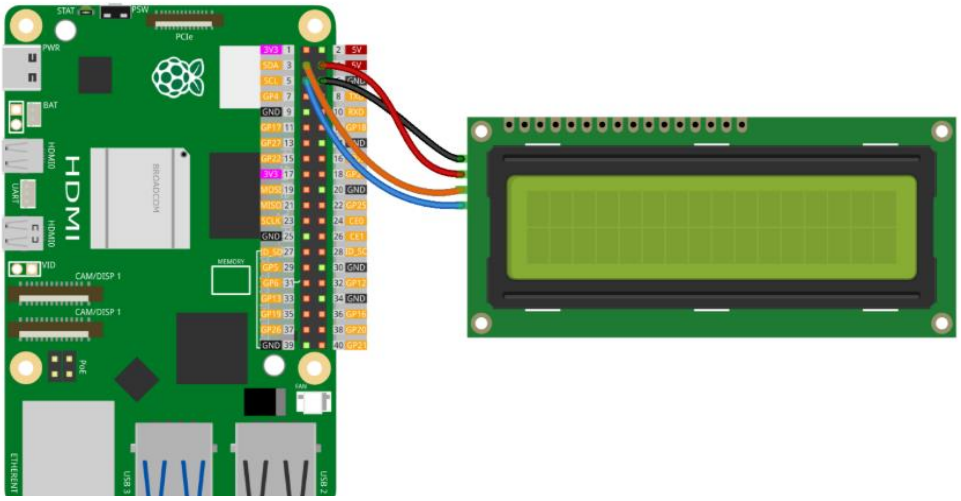
- **16 columns:** (Displays 16 characters per line)
- **2 rows:** (Two display lines)



This display is capable of showing standard ASCII characters as well as some special symbols. Each character is formed in a 5×8 pixel matrix. The LCD1602 typically uses the HD44780 controller, which is supported by most microcontrollers and development boards.

By default, the LCD1602 operates in parallel mode with 8 or 4 data lines plus several control lines. However, to reduce the number of required wires, an I²C converter module is attached to its back. This module transmits data using only two lines, SDA and SCL, and is fully compatible with the I²C protocol.

Pin	Name	Description
GND	Ground	Connected to Raspberry Pi ground (GND)
VCC	Power	Connected to 5V or 3.3V power supply on Raspberry Pi (usually 5V)
SDA	Serial Data	Data line for sending and receiving information
SCL	Serial Clock	Clock line for synchronizing data transfer



2.7.2 Method of Connecting LCD1602 to Raspberry Pi 5 via I²C

Hardware Connections:

- Connect the VCC pin to the Raspberry Pi 5's 5V pin (physical pin 2 or 4)
- Connect the GND pin to the Raspberry Pi ground (any GND pin such as physical pin 6)
- Connect the SDA pin to GPIO 2 (physical pin 3)
- Connect the SCL pin to GPIO 3 (physical pin 5)

Enabling I²C on Raspberry Pi:

- Run the command: `sudo raspi-config`
- Enable the I2C Interface option under the Interface Options menu
- Save the changes and reboot the Raspberry Pi

Identifying the I²C Address of the Display:
(Usually done via the `i2cdetect` tool after enabling I²C)

Programming and Controlling the LCD

2.7.3 Application of LCD1602 in the Project

In this project, the LCD1602 is used to display the results from the YOLOv8 model. After objects are detected by the Raspberry Pi 5, the results — such as the object names or the number of detected objects — are sent as text to the LCD via the I²C protocol, allowing the user to view the output live.

2.8 Graphical User Interface (GUI) Design and Tools Used

A graphical user interface (GUI) is a part of the system that enables interaction between humans and devices or software. Instead of using complex text commands, the GUI simplifies and speeds up interaction by displaying windows, buttons, images, and text.

In smart object detection projects, the GUI allows the user to:

- View the input images
- See the objects detected by the YOLOv8 model
- Receive live system results and information
- Control system operations (start/stop processing, save data, etc.)

2.8.1 Tools Used

For implementing the GUI in Python, there are two main libraries:

1. **Tkinter**
    - The standard Python library for graphical interfaces
    - Simple and lightweight
    - Suitable for small to medium projects

2. **PyQt5**
    - A powerful and professional library for advanced GUI design
    - Supports creation of complex windows, tables, animations, and precise control over elements
    - Suitable for large and industrial projects

In this project, depending on the needs and complexity of the interface, one of these two can be used.

2.8.2 Main GUI Elements in the Project

The system's user interface includes the following components:

- **Main Window:** The primary space for displaying all elements
- **Image Panel:** Displays the input image from the camera and the detected objects
- **Labels:** Show textual information such as the names or counts of detected objects
- **Buttons:** Start or stop processing, save results
- **List or Table (List/Tree View):** Display details of objects in a categorized manner

2.8.3 GUI Design Steps

1. Create the main window: set size, title, and background color
2. Add elements: place buttons, labels, and image panel using frames
3. Link elements to system functions: buttons control start/stop, image and data update in real-time
4. Real-time GUI updates: using loops or callbacks, the image and information refresh continuously without interrupting the program

2.8.4 GUI Application in the Project

The GUI enables the user to:

- View the input image and detected objects simultaneously
- See the names and counts of objects live
- Control system operations without using command line
- Receive important information and system status

2.9 Chapter Summary

In this chapter, we became familiar with the three main pillars of the project. The YOLOv8 model and its Nano version are the core tools for object detection and segmentation in our system. The I²C protocol allows displaying detection information on the LCD. Finally, the GUI, implemented using Tkinter or PyQt5, facilitates user interaction with the system. This foundational knowledge is crucial for the practical implementation in Chapter 3.

Chapter 3: Hardware and Software System Design

3.1 Introduction

In this chapter, the overall design of the system—including both the hardware and software components—is discussed. The main goal is to develop an intelligent system based on the Raspberry Pi 5 for object detection, classification, and counting using the advanced YOLOv8 model. The system is designed to have sufficient processing power to run deep learning models while maintaining low power consumption and cost-effectiveness, ensuring reliable performance.

The proposed system consists of three main modules:

1. Processing Module

- The Raspberry Pi 5 serves as the central core of the system.
- Its main responsibilities include executing the YOLOv8 model, processing input images, managing data flow, and coordinating between input and output components.
- The Raspberry Pi 5 was selected due to its appropriate processing power, low energy consumption, compatibility with webcams and displays, and support for Python and OpenCV libraries.

2. Input Module

- This module is responsible for capturing live images from the environment so the system can detect objects in real-time.
- A high-quality webcam with a decent frame rate is used to provide clear, delay-free images to the YOLOv8 model.
- The captured input data is sent to the processing module for object detection and counting.

3. Output Module

- This includes an LCD display and a graphical user interface (GUI), which are responsible for showing the detection results and enabling user interaction with the system.
- The LCD is used to quickly display textual information such as detected object names, their counts, and system status.
- The GUI allows users to view the live input feed, detected objects with bounding boxes, statistical information, and control system functions.

3.1.1 Data Flow and Component Interaction

- Input images are captured by the webcam and sent to the processing module.
- The YOLOv8 model detects and classifies the objects in the image.
- Processed data, including object names and counts, are transmitted to the LCD and GUI for real-time display and user interaction.
- This data flow operates in real-time with minimal latency, providing a smooth and efficient user experience.

3.1.2 Advantages of the System Design

- **High Performance:** With the YOLOv8 model and Raspberry Pi 5, object detection is performed quickly and accurately.
- **Low Cost:** Using a Raspberry Pi and webcam is cost-effective and eliminates the need for expensive hardware.
- **Scalability:** The system can be easily expanded by adding more cameras or upgrading the model.
- **User-Friendly Interaction:** The GUI and LCD provide a simple and intuitive experience for the user.

3.2 Hardware Design

3.2.1 Introduction and Configuration of Raspberry Pi

The Raspberry Pi is a single-board mini-computer designed and developed by the Raspberry Pi Foundation in the UK. Initially aimed at promoting computer science education in schools and developing countries, it quickly gained popularity among electronics enthusiasts, robotics developers, IoT practitioners, and AI researchers due to its low cost, compact size, and versatile features.

This board integrates all essential computer components on a small PCB, including:

- **CPU (Processor)**
- **GPU (Graphics Processor)**
- **RAM (Memory)**
- **I/O Ports:** USB, HDMI, Ethernet, etc.
- **MicroSD Card Slot** for OS and data storage

3.2.2 Key Features of the Raspberry Pi

1. **Compact Size:** Roughly the size of a credit card (85 × 56 mm)
2. **Low Power Consumption:** Suitable for 24/7 operation
3. **Low Cost:** Much cheaper than laptops or industrial mini-PCs
4. **High Compatibility:** Connects to cameras, displays, sensors, and various modules
5. **Extensive Software Support:** Compatible with Linux-based OSs like Raspberry Pi OS and Ubuntu, and supports AI libraries such as OpenCV, PyTorch, etc.

Raspberry Pi 5 – The Next Generation

The Raspberry Pi 5 is the latest and most powerful version in the family, released in 2023. Compared to its predecessor (Raspberry Pi 4), it offers significant improvements:

- Quad-core ARM Cortex-A76 processor @ 2.4 GHz
- 4 GB or 8 GB LPDDR4X RAM
- USB 3.0 ports for fast peripheral connectivity
- 4K@60fps video output support
- Optimized power usage with significantly higher performance

3.2.3 Applications of Raspberry Pi in AI Projects

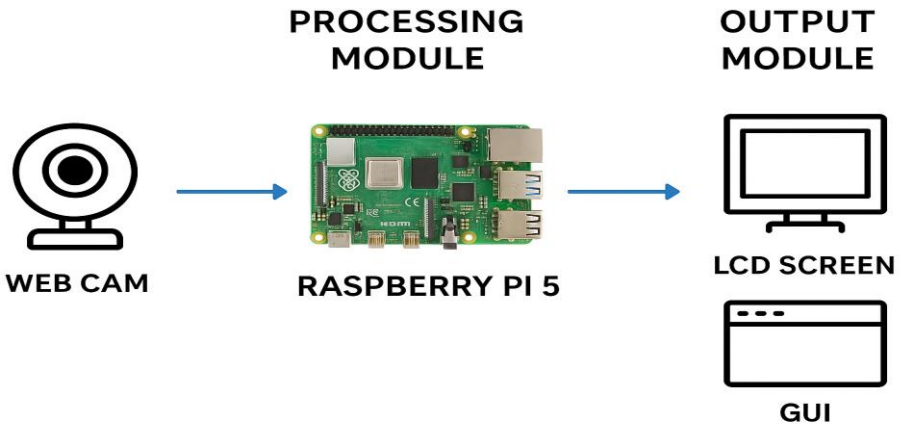
Due to its small size, affordability, low power consumption, and ability to run lightweight deep learning models, the Raspberry Pi is ideal for projects like:

- **Object Detection and Recognition**
- **Smart Surveillance Systems**
- **Robotics and Automated Control**
- **Smart Home Systems**
- **Computer Vision and Image Processing**

The Raspberry Pi combines the capabilities of a complete computer in a compact and affordable form factor, making it one of the most popular hardware choices for AI and IoT applications. The latest Raspberry Pi 5, in particular, offers the processing power needed to run deep learning models like YOLOv8n in real-time.

3.2.4 Selection and Connection of the USB Camera

Camera Type:
This project utilizes a USB webcam. These cameras are easy to connect, affordable, and fully compatible with the Raspberry Pi and OpenCV. A 720p resolution webcam was chosen to maintain a good balance between image quality and processing speed.



3.2.5 Webcam Function in the System

- Capturing real-time live images from the environment
- Providing input to the YOLOv8 model for object detection and classification
- Allowing simultaneous display of results on the GUI or LCD screen

3.2.6 Challenges of Using a Webcam

- Environmental lighting changes can reduce detection accuracy
- Frame delays may occur due to high resolution or limited processing resources
- Some webcam models may not be automatically recognized due to incompatibility with the V4L2 (Video4Linux2) driver

3.2.7 Connecting an I²C-Based LCD Display

The LCD is one of the output modules in the system that displays textual information and object detection results to the user. Using the I²C (Inter-Integrated Circuit) protocol simplifies the connection of the LCD to the Raspberry Pi 5 and reduces wiring complexity. In this project, an **LCD1602 with I²C module** is used, capable of displaying 2 lines with 16 characters each.

3.2.8 Features of LCD1602 with I²C

- **Character Display**: Capable of showing letters, numbers, and symbols across two lines with 16 characters per line
- **I²C Protocol**: Utilizes only two wires (SDA and SCL) for data and command transmission
- **5V Compatibility**: Can be connected directly to the Raspberry Pi
- **Adjustable I²C Address**: Typically set to 0x27 or 0x3F for communication with the processor
-

Physical Wiring to Raspberry Pi 5

LCD Pin	Raspberry Pi Pin
VCC	3.3V
GND	GND
SDA	GPIO 2 (SDA)
SCL	GPIO 3 (SCL)

Important Notes:

- Use short, high-quality wires to reduce electrical noise
- Ensure proper voltage levels to avoid damage to the LCD or Raspberry Pi

3.2.9 Software Configuration

1. Enable the I²C interface in Raspberry Pi OS
2. Install necessary Python libraries
3. Detect the I²C address of the LCD

LCD Usage in the System

- Displaying object names and counts as detected by the YOLOv8 model
- Showing system status such as start, stop, or errors
- Providing simple messages and notifications to the user without requiring a monitor

3.3 Software Design

The software acts as the brain of the system, coordinating hardware control, data processing, and user interaction. It is designed to ensure the system performs real-time, accurate, and stable object detection and result presentation.

The software architecture includes three main components:

1. Image processing and object detection using YOLOv8
2. Output module management including the LCD and GUI
3. Data flow control and inter-module coordination

3.3.1 Image Processing and Object Detection

1. Live Image Acquisition

- Captured via USB webcam and sent to the Raspberry Pi
- Images are preprocessed (e.g., resized and formatted) for compatibility with the YOLOv8 model

2. YOLOv8 Execution

- YOLOv8 detects, classifies, and localizes objects in each frame
- Outputs include bounding boxes, object classes, and confidence scores

3. Processing Optimization

- Reducing image resolution to increase detection speed
- Using parallel processing (threading) to enable simultaneous image acquisition, detection, and output
- Implementing caching to reduce computational load in consecutive frames

3.3.2 LCD Display Module

- Communicates via the I²C protocol using only SDA and SCL lines
- Displays short text messages such as object counts, system messages, and errors
- Supports queuing of multiple messages for sequential display

3.3.3 Graphical User Interface (GUI)

- Designed using **Tkinter** or **PyQt5**
- Shows live camera feed with **bounding boxes** and object labels
- Displays object types and counts
- Provides user controls (Start, Stop, Reset, Camera Settings)
- Capable of displaying system messages, alerts, and status notifications

3.3.4 Software Architecture and Modules

1. Capture Module

- Captures live images from the webcam and performs basic preprocessing
- Resizes images and converts them to YOLO-compatible formats

2. Detection Module

- Executes YOLOv8 for object detection and classification
- Outputs bounding boxes, object classes, and counts

3. Display Module

- Manages output to LCD and GUI simultaneously
- Updates the user interface in real time
- Displays alerts, messages, and additional data

4. Control Module

- Manages user inputs from the GUI
- Coordinates between Capture, Detection, and Display modules
- Handles exceptions and system errors

3.3.6 Optimization and Expandability

- **Image Processing Optimization**: Reducing resolution, implementing threading and caching
- **Concurrent Execution**: Enables real-time detection and display without lag
- **Scalability**: Allows for additional cameras, advanced YOLO models, or alternative displays
- **Hardware-Software Integration**: Easily integrates new sensors and modules with minimal software changes

3.3.7 Advantages of the Software Design

- Real-time and accurate image processing
- Intuitive and user-friendly graphical interface
- Low-cost and scalable solution
- Seamless integration between hardware and software components

3.3.8 Proposed Segmentation Module

In the extended version of the system, in addition to object detection, a **segmentation module** is proposed for more precise object isolation. The system utilizes **YOLOv8n-seg**, a lightweight segmentation-capable version of YOLOv8.

Benefits of This Approach

- **Higher detection accuracy**, even in cluttered scenes or when objects are close together
- **Background separation**: Allows individual object processing or removal of the background
- **Statistical analysis**: Accurately measures size, count, and position of objects

Integration with AI and Computer Vision

- A CNN learns to identify visual features of objects
- Image processing is used for frame preparation and detection optimization
- Combined, these technologies enable automatic object detection and pixel-level segmentation

3.4 Hardware and Software Integration

In this project, the hardware and software components are designed to operate concurrently and in sync. The flow of information between modules is as follows:

1. **Webcam**: Captures live frames from the environment and sends them to the software
2. **YOLOv8 Model**: Processes frames and detects and localizes objects
3. **GUI**: Displays the live feed with bounding boxes, allowing the user to monitor results
4. **LCD Display**: Shows up to two detected objects at a time for quick and simple monitoring

Design Features:

- Real-time frame processing minimizes delays and increases responsiveness
- Integration enables synchronized display on both GUI and LCD
- Modular design allows easy expansion by adding sensors or additional displays

3.5 Summary

In this chapter, the complete **hardware and software design** of the project was presented. The intelligent selection of components such as **Raspberry Pi 5**, **USB webcam**, and **LCD display** made the system portable, cost-effective, and high-performing. On the software side, the integration of **YOLOv8**, **GUI**, and support for **segmentation** provides the user with a comprehensive real-time object detection experience.

Chapter 4: Full Implementation and Coding

4.1 Introduction

This chapter presents the complete implementation of the object detection system using YOLOv8 and the Python programming language. The system runs on a Raspberry Pi and supports displaying results on a GUI built with Tkinter as well as on an LCD screen. Additionally, users can select the objects they want to detect through the user interface.

4.2 Hardware and Software Environment

Hardware Used:

- Raspberry Pi 5
- USB camera or Pi Camera
- I2C LCD Display
- Appropriate cables and power supply

Software and Libraries:

- Python 3.x
- OpenCV for image processing
- PIL (Pillow) for converting images into displayable format for Tkinter
- Tkinter for graphical user interface
- ultralytics.YOLO for YOLOv8 model
- RPLCD and RPi.GPIO for controlling the LCD

4.3 Preparing Raspberry Pi and Transferring the Code

Writing the code on the main computer:

- The complete program with all components was written and saved in a specified file.
- The code includes the following parts:
    - Loading the YOLOv8 model
    - Camera configuration and resolution settings
    - Creating the Tkinter user interface
    - Controlling the LCD display
    - Processing frames and detecting objects
    - Managing user selections

File Transfer to Raspberry Pi:
• The file was transferred to the Raspberry Pi using USB, SCP, or an SD card.
• File path on the Pi:

```
/home/pi/mahya_test_yolo_final.py
```

Installing the required libraries on the Pi:

```
pip3 install opencv-python pillow ultralytics RPLCD RPi.GPIO
```

Running the code on the Raspberry Pi:

```
python3 mahya_test_yolo_final.py
```

The system begins by initializing the camera, loading the YOLO model, and preparing the GUI and LCD.

4.4.1 Section-by-Section Code Explanation on Raspberry Pi 5

Importing Libraries:

```
1 import sys
2 import tkinter as tk
3 from tkinter import ttk, messagebox
4 from PIL import Image, ImageTk
5 import cv2
6 from ultralytics import YOLO
7 import time
8 import threading
```

- **sys**: Used to handle command-line arguments and to exit the program.
- **tkinter**, **ttk**, **messagebox**: Used to build the graphical user interface (GUI) of the application.
- **PIL**: Used to convert and display images within the GUI.
- **cv2**: The OpenCV library, used for capturing images from the camera and performing image processing.
- **ultralytics**: The YOLOv8 library used for object detection.
- **time**: Used for time-related operations (delays, measuring intervals).
- **threading**: Enables concurrent execution (e.g., running image processing and the GUI simultaneously).

Adding support for Raspberry Pi hardware LCD.

```
10 # Add LCD support
11 try:
12     import RPi.GPIO as GPIO
13     from RPLCD.i2c import CharLCD
14     LCD_AVAILABLE = True
15 except ImportError:
16     print("RPi.GPIO or RPLCD not available. Running without LCD.")
17     LCD_AVAILABLE = False
```

It attempts to import the specific modules required to control the LCD via the I2C protocol on the Raspberry Pi.
If these libraries are not available, an error message is shown, and the program continues to run without LCD support.
The variable **`LCD_AVAILABLE`** determines whether the LCD is present or not.

Loading the YOLOv8 Model

```
19 # Your existing model load and LCD initialization
20 model_detection = YOLO('yolov8n.pt')
```

The **yolov8n.pt** model, which is the small version of YOLOv8, is loaded to perform object detection on the input images.

Camera Settings

```
22 cap = cv2.VideoCapture(0, cv2.CAP_V4L2)
23 cap.set(cv2.CAP_PROP_FRAME_WIDTH, 320)
24 cap.set(cv2.CAP_PROP_FRAME_HEIGHT, 240)
25 cap.set(cv2.CAP_PROP_FPS, 15)
26 cap.set(cv2.CAP_PROP_BUFFERSIZE, 1)
27
```

The default camera (webcam or camera connected to the Raspberry Pi) is opened.
The image resolution is set to 320x240 (reducing size for better speed).
The frame rate is set to 15 frames per second.
The camera buffer is set to 1 to read new frames more quickly.

Initializing the LCD if it is enabled

```
28 # Initialize LCD (adjust address and port as needed)
29 if LCD_AVAILABLE:
30     try:
31         lcd = CharLCD('PCF8574', 0x27, port=1, cols=16, rows=2, dotsize=8)
32         lcd.clear()
33         lcd.write_string("Starting..")
34         time.sleep(2)
35         lcd.clear()
36     except Exception as e:
37         print(f"LCD initialization failed: {e}")
38         LCD_AVAILABLE = False
```

The LCD is created with I2C address 0x27 and port 1, which are the most common settings for I2C LCDs.
The LCD screen is cleared, and the message "Starting.." is displayed for 2 seconds.
If an error occurs, the error message is printed and the LCD is disabled.

Function to display objects on the LCD

```
40 def display_on_lcd(objects, object_counts):
41     """Display detected objects on LCD"""
42     if not LCD_AVAILABLE:
43         return
44 
45     try:
46         lcd.clear()
47         if not objects:
48             lcd.write_string("No objects\ndetected")
49         else:
50             # Display up to 2 objects (one per line) with counts
51             if len(objects) == 1:
52                 obj_name = objects[0]
53                 count = object_counts[obj_name]
54                 line1 = f"{obj_name}({count})"[:16]
55                 line2 = ""
56             else:
57                 obj1 = objects[0]
58                 count1 = object_counts[obj1]
59                 line1 = f"{obj1}({count1})"[:16]
60 
61                 if len(objects) > 1:
62                     obj2 = objects[1]
63                     count2 = object_counts[obj2]
64                     line2 = f"{obj2}({count2})"[:16]
65                 else:
66                     total_count = sum(object_counts.values())
67                     line2 = f"Total: {total_count}"[:16]
68 
```

If the LCD is not enabled, the function does nothing.
Otherwise, it clears the screen.
If no objects are detected, it displays the message "No objects detected."
Otherwise, it shows up to two detected objects along with their counts.

Global variables for target objects.

```
73 # Initialize target object filter
74 target_objects = ["ALL"]  # Default to detect all objects
75 pending_objects = ["ALL"]  # Objects selected but not yet confirmed
```

target_objects: Objects that are intended to be recognized after user confirmation.
pending_objects: Objects that are currently being selected but have not yet been confirmed by pressing the confirmation button.

Defining the main program class (interface and processing)
Setting up the main window and frames

```
77 class DetectionApp:
78     def __init__(self, root):
79         self.root = root
80         self.root.title("YOLOv8 Object Detection - Mahya Kheirandish")
81         self.root.geometry("1200x800")
82 
83         # Create main frame
84         main_frame = tk.Frame(root)
85         main_frame.pack(fill=tk.BOTH, expand=True, padx=10, pady=10)
```

The main class in which the entire program runs.
It includes the definition and configuration of the graphical user interface, as well as control of the camera and model.
The window title is set.
The window size is set to 1200x800 pixels.
A main frame is created to organize the components.

Video display section (on the left side).

```
87         # Left side - Video display
88         video_frame = tk.Frame(main_frame)
89         video_frame.pack(side=tk.LEFT, fill=tk.BOTH, expand=True, padx=(0, 10))
90 
91         self.image_label = tk.Label(video_frame, bg='black')
92         self.image_label.pack(fill=tk.BOTH, expand=True)
```

A dedicated frame for video display is created.
A Label widget is made to show the video image, with a black background color.

Object selection section (on the right side)

```
94          # Right side - Object selection
95          selection_frame = tk.Frame(main_frame, width=250)
96          selection_frame.pack(side=tk.RIGHT, fill=tk.Y)
97          selection_frame.pack_propagate(False)
```

A right-side frame with a fixed width of 250 pixels is created for object selection.

It includes a title and a list for selecting objects.

```
 99         tk.Label(selection_frame, text="Select Objects to Detect:", font=('Arial', 12, 'bold')).pack(pady=(0, 10))
100
101         # Object list widget with scrollbar
102         list_frame = tk.Frame(selection_frame)
103         list_frame.pack(fill=tk.BOTH, expand=True, pady=(0, 10))
104
105         scrollbar = tk.Scrollbar(list_frame)
106         scrollbar.pack(side=tk.RIGHT, fill=tk.Y)
107
108         self.object_listbox = tk.Listbox(list_frame, selectmode=tk.EXTENDED, yscrollcommand=scrollbar.set)
109         self.object_listbox.pack(side=tk.LEFT, fill=tk.BOTH, expand=True)
110         scrollbar.config(command=self.object_listbox.yview)
```

A label above the list displays the title "Select Objects to Detect."
A multi-select listbox with a vertical scrollbar is created for the object list.

The list is populated with the YOLO classes.

```
112         # Populate with YOLO class names
113         self.populate_object_list()
```

A separate function that adds the YOLO model classes to the list.

Binding the event for selection changes in the list.

```
115         # Bind selection change
116         self.object_listbox.bind('<<ListboxSelect>>', self.update_pending_objects)
```

When the user selects or deselects an item, the `update_pending_objects` function is executed.

Buttons for managing selections

```
118         # Buttons
119         self.confirm_btn = tk.Button(selection_frame, text="Confirm Selection",
120                                      command=self.confirm_selection, bg='#4CAF50', fg='white',
121                                      font=('Arial', 10, 'bold'))
122         self.confirm_btn.pack(fill=tk.X, pady=2)
123 
124         self.select_all_btn = tk.Button(selection_frame, text="Select All",
125                                         command=self.select_all_objects)
126         self.select_all_btn.pack(fill=tk.X, pady=2)
127 
128         self.clear_all_btn = tk.Button(selection_frame, text="Clear All",
129                                        command=self.clear_all_objects)
130         self.clear_all_btn.pack(fill=tk.X, pady=2)
```

- Confirm selections button
- Select all objects button
- Clear selections button
- Display current selection status

```
132         # Status label
133         self.status_label = tk.Label(selection_frame, text="Current: ALL objects",
134                                      fg='blue', font=('Arial', 10, 'bold'), wraplength=230)
135         self.status_label.pack(pady=10)
```

A label to display which objects are currently selected.

Helper variables and starting video processing in a separate thread

```
137         self.frame_count = 0
138         self.last_display_time = 0
139         self.display_interval = 2  # seconds
140         self.running = True
141 
142         # Start video processing in separate thread
143         self.video_thread = threading.Thread(target=self.video_loop, daemon=True)
144         self.video_thread.start()
145 
146         # Handle window close
147         self.root.protocol("WM_DELETE_WINDOW", self.on_closing)
```

To reduce processing load, frame counting is performed.
Results are displayed on the console and LCD every 2 seconds.
Video processing and object detection run in a separate thread to prevent the interface from slowing down.
A function is registered to execute when the window is closed.

A function to populate the object list.

```
149     def populate_object_list(self):
150         """Populate the object list with YOLO class names"""
151         # Add "ALL" option first
152         self.object_listbox.insert(0, "ALL objects")
153         self.object_listbox.select_set(0)  # Select by default
154 
155         # Add individual object classes in alphabetical order
156         class_names = model_detection.names
157         sorted_classes = sorted(class_names.items(), key=lambda x: x[1])
158 
159         for class_id, class_name in sorted_classes:
160             self.object_listbox.insert(tk.END, class_name.title())
```

First, the option "ALL objects" is added.
Then, all YOLO model classes are added in order.

Function to update selected but unconfirmed objects

```
162     def update_pending_objects(self, event=None):
163         """Update pending objects based on user selection (not yet confirmed)"""
164         global pending_objects
165         selected_indices = self.object_listbox.curselection()
166 
167         if not selected_indices:
168             pending_objects = ["ALL"]
169             return
170         pending_objects = []
171         all_selected = False
172         specific_objects = []
173 
174         for index in selected_indices:
175             item_text = self.object_listbox.get(index)
176             if item_text == "ALL objects":
177                 all_selected = True
178             else:
179                 specific_objects.append(item_text.lower())
180         # If "ALL" is selected along with specific objects, prioritize specific objects
181         if all_selected and specific_objects:
182             # Deselect "ALL objects" item
183             self.object_listbox.selection_clear(0)
184             pending_objects = specific_objects
185         elif all_selected:
186             # Clear all other selections when ALL is selected
187             self.object_listbox.selection_clear(0, tk.END)
188             self.object_listbox.select_set(0)
189             pending_objects = ["ALL"]
190         elif specific_objects:
191             pending_objects = specific_objects
192         else:
193             pending_objects = ["ALL"]
```

When the selection in the list changes, the selection status is updated.
If "ALL objects" and specific objects are selected together, only the specific objects are kept.
If no objects are selected or only "ALL objects" is selected, all objects are considered active.

Confirm selections function

```
195     def confirm_selection(self):
196         """Confirm the selected objects for detection"""
197         global target_objects
198         target_objects = pending_objects.copy()
199 
200         if target_objects[0] == "ALL":
201             status_text = "Current: ALL objects"
202             print("Detection confirmed: ALL objects")
203         else:
204             status_text = f"Current: {', '.join(target_objects[:3])}"
205             if len(target_objects) > 3:
206                 status_text += f" (+{len(target_objects)-3} more)"
207             print(f
208             if len(target_objects) > 3:
209                 status_text += f" (+{len(target_objects)-3} more)"
210             print(f"Detection confirmed: {', '.join(target_objects)}")
211 
```

The final selections are confirmed and saved in **target_objects**.
The status text in the interface is updated.

Select All and **Clear All** functions:

- **Select All:** Selects all items in the list.
- **Clear All:** Clears all selections from the list.

```
214     def select_all_objects(self):
215         """Select all specific objects (not including ALL objects option)"""
216         # First, deselect "ALL objects"
217         all_item = self.object_list.item(0)
218         all_item.setSelected(False)
219 
220         # Then select all specific object classes
221         for i in range(1, self.object_list.count()):
222             self.object_list.item(i).setSelected(True)
223 
224     def clear_all_objects(self):
225         """Clear all selections and select ALL by default"""
```

`select_all_objects`: All objects except "ALL objects" are selected.
`clear_all_objects`: All selections are cleared, and only "ALL objects" is selected.

Processing each frame, updating the image, and displaying the result

```
    def update_frame(self):
        ret, frame = cap.read()
        if not ret:
            return
        self.frame_count += 1
        if self.frame_count % 3 != 0:
            # Skip frames to reduce load
            return
        # Resize frame for processing
        small_frame = cv2.resize(frame, (320, 240))
        results = model_detection.predict(source=small_frame, conf=0.4, imgsz=320, verbose=False)
        detected_objects = []
        for result in results:
            if result.boxes is not None:
                for box in result.boxes:
                    cls_id = int(box.cls)
                    label = model_detection.names[cls_id]
                    # Filter based on confirmed target objects
                    if target_objects[0] == "ALL" or label.lower() in target_objects:
                        detected_objects.append(label)
                        # Draw bounding box on original frame scaled appropriately
                        x1, y1, x2, y2 = map(int, box.xyxy[0])
                        # Scale back to original frame size (if different)
                        scale_x = frame.shape[1] / 320
                        scale_y = frame.shape[0] / 240
                        x1 = int(x1 * scale_x)
                        x2 = int(x2 * scale_x)
                        y1 = int(y1 * scale_y)
                        y2 = int(y2 * scale_y)
                        cv2.rectangle(frame, (x1, y1), (x2, y2), (0,255,0), 2)
                        cv2.putText(frame, label, (x1, y1-10), cv2.FONT_HERSHEY_SIMPLEX,
                                    0.5, (0,255,0), 1)
```

```
        current_time = time.time()
        unique_objects = list(set(detected_objects))
        # Count occurrences of each object
        object_counts = {}
        for obj in detected_objects:
            object_counts[obj] = object_counts.get(obj, 0) + 1
        if current_time - self.last_display_time >= self.display_interval:
            if unique_objects:
                # Display with counts for console
                console_output = []
                for obj in unique_objects[:5]:
                    count = object_counts[obj]
                    if count > 1:
                        console_output.append(f"{obj}({count})")
                    else:
                        console_output.append(obj)
                print(f"Detected: {', '.join(console_output)}")
            else:
                print("No objects detected")
            # Update LCD display with counts
            display_on_lcd(unique_objects, object_counts)
            self.last_display_time = current_time
        # Convert BGR OpenCV image to RGB for Qt
        rgb_image = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
        h, w, ch = rgb_image.shape
        bytes_per_line = ch * w
        qt_image = QImage(rgb_image.data, w, h, bytes_per_line, QImage.Format_RGB888)
        # Scale image to fit the full label size
        pixmap = QPixmap.fromImage(qt_image)
        scaled_pixmap = pixmap.scaled(self.image_label.size(), Qt.KeepAspectRatio, Qt.SmoothTransformation)
        self.image_label.setPixmap(scaled_pixmap)
```

A frame is read.
Processing is performed every 3 frames to reduce resource usage.
The frame is resized, and the YOLO model is run on it.
Objects that are in **target_objects** are saved, and green bounding boxes are drawn.
The boxes are drawn on the original-sized frame.
The count of each object is tracked and displayed on the console and LCD every 2 seconds.
The image is converted to a format compatible with Tkinter and updated in the interface.

Window close management

```
295     def closeEvent(self, event):
296         cap.release()
297         if LCD_AVAILABLE:
298             try:
299                 lcd.clear()
300                 lcd.write_string("Detection\nStopped")
301                 time.sleep(1)
302                 lcd.clear()
303             except:
304                 pass
305         super().closeEvent(event)
```

When closing the program, the `running` variable is set to False to stop the thread.
The camera is released.
The LCD displays the message "Detection Stopped" and then is cleared.
The program window is closed.

Running the program

```
307 if __name__ == "__main__":
308     app = QApplication(sys.argv)
309     window = DetectionApp()
310     window.show()
311     sys.exit(app.exec())
```

Project Execution Steps on Raspberry Pi

Installing Libraries and Preparing the Environment

Before running the code, the Python environment and the required libraries must be installed on the Raspberry Pi.

Installing the basic libraries:

```
sudo apt update
sudo apt install python3-pip python3-tk python3-pil python3-opencv -y
```

Installing YOLOv8 and ultralytics:

```
pip install ultralytics
```

Installing LCD libraries:

```
pip install RPLCD
sudo apt install i2c-tools python3-smbus
```

- `python3-tk` for the GUI
- `python3-pil` for image conversion
- `python3-opencv` for image processing and capturing camera frames
- `RPLCD` for controlling the LCD

4.4.24 Section-by-Section Code Explanation on Laptop
Importing Required Libraries

```
from ultralytics import YOLO
import cv2
import numpy as np
from PyQt5.QtWidgets import QApplication, QWidget, QPushButton, QVBoxLayout, QLabel, QLineEdit, QHBoxLayout, \
    QListWidget, QScrollArea
from PyQt5.QtCore import QThread, pyqtSignal
from PyQt5.QtGui import QImage, QPixmap
```

In this section, the necessary libraries are imported for:

- Image processing (OpenCV)
- Using YOLOv8 models (ultralytics)
- Creating a graphical user interface (PyQt5)
- Working with arrays (NumPy)

Loading YOLOv8 models

```
# Load YOLOv8 pretrained model (coco dataset has chairs, tables, etc.)
model_detection = YOLO('yolov8n.pt')  # YOLOv8 model for detection
model_segmentation = YOLO('yolov8n-seg.pt')  # YOLOv8 model for segmentation
```

In this part, two models are loaded:

- The YOLOv8 nano model for object detection.
- The YOLOv8 nano model with mask output for object segmentation.

Defining global variables for the program

```
12 # Initialize variables
13 cap = cv2.VideoCapture(1)
14 detection_enabled = False
15 segmentation_enabled = False
16 target_object = ""  # Empty means detect all objects
```

Connecting to the camera using `cv2.VideoCapture`.
Defining flags to control detection and segmentation modes.
Setting the target object name to filter the model's output.

DetectionThread class for background processing
Defining the class and its signals.

```
17 class DetectionThread(QThread):
18     frame_updated = pyqtSignal(np.ndarray)
```

This class is used for processing live video in a separate thread to prevent the user interface from slowing down. The `frame_updated` signal is defined to send processed frames to the UI.

The run function executes the frame processing loop.

```
20     def run(self):
21         global detection_enabled, segmentation_enabled, target_object
22         while True:
23             ret, frame = cap.read()
24             if not ret:
25                 break
```

In this function, live frames are captured from the camera, and depending on the active mode, either detection or segmentation is performed on them.

Processing in detection mode

```
30            if detection_enabled and not segmentation_enabled:
31                # Perform object detection only
32                results = model_detection.predict(source=frame, conf=0.5, stream=True)
33
34                detected_objects = []
35                annotated_frame = frame.copy()
36
37                for result in results:
38                    if target_object:
39                        # Filter results for target object only
40                        if result.boxes is not None and len(result.boxes) > 0:
41                            for box in result.boxes:
42                                object_name = model_detection.names[int(box.cls)]
43                                if target_object.lower() in object_name.lower():
44                                    detected_objects.append(object_name)
45                                    # Draw filtered detection manually
46                                    x1, y1, x2, y2 = map(int, box.xyxy[0])
47                                    cv2.rectangle(annotated_frame, (x1, y1), (x2, y2), (0, 255, 0), 2)
48                                    cv2.putText(annotated_frame, object_name, (x1, y1 - 10),
49                                                cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 255, 0), 2)
50                    else:
51                        # Show all detected objects
52                        annotated_frame = result.plot()
53                        if result.boxes is not None:
54                            detected_objects.extend([model_detection.names[int(box.cls)] for box in result.boxes])
55
56                # Overlay detected object names on the OpenCV frame
57                y_offset = 20
58                for obj in set(detected_objects):  # Use `set` to ensure unique names
59                    cv2.putText(annotated_frame, obj, (10, y_offset), cv2.FONT_HERSHEY_SIMPLEX, 0.6, (0, 255, 0), 2)
60                    y_offset += 20
```


If detection mode is active:

- The `yolov8n.pt` model is run on the frame.
- The detected objects are drawn.
- If the user has specified a particular object, only that object is filtered and displayed.

Processing in segmentation mode

```
62            elif segmentation_enabled and not detection_enabled:
63                # Perform semantic segmentation only
64                results = model_segmentation.predict(source=frame, conf=0.5, stream=True)
65
66                annotated_frame = frame.copy()
67                for result in results:
68                    if hasattr(result, 'masks') and result.masks is not None:
69                        if target_object:
70                            # Filter segmentation results for target object only
71                            if result.boxes is not None and len(result.boxes) > 0:
72                                for i, box in enumerate(result.boxes):
73                                    object_name = model_segmentation.names[int(box.cls)]
74                                    if target_object.lower() in object_name.lower():
75                                        # Draw filtered segmentation manually
76                                        if i < len(result.masks.data):
77                                            mask = result.masks.data[i].cpu().numpy()
78                                            colored_mask = np.zeros_like(annotated_frame)
79                                            colored_mask[mask > 0.5] = [0, 255, 0]
80                                            annotated_frame = cv2.addWeighted(annotated_frame, 1, colored_mask, 0.3, 0)
81
82                                        # Draw bounding box and label
83                                        x1, y1, x2, y2 = map(int, box.xyxy[0])
84                                        cv2.rectangle(annotated_frame, (x1, y1), (x2, y2), (0, 255, 0), 2)
85                                        cv2.putText(annotated_frame, object_name, (x1, y1 - 10),
86                                                    cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 255, 0), 2)
87                        else:
88                            annotated_frame = result.plot()
89                    else:
90                        annotated_frame = frame.copy()
91            else:
92                annotated_frame = frame.copy()
93
```


If segmentation mode is active:

- The `yolov8n-seg.pt` model is executed.
- Colored masks are applied to the detected objects.
- If a target object is specified, only that object is displayed.

Sending the frame to the user interface

```
94              self.frame_updated.emit(annotated_frame)
95
96          cap.release()
97          cv2.destroyAllWindows()
```

The final processed frame is sent to the QLabel for display in the UI using a signal.

MainWindow class – The program's graphical user interface

Creating the main widgets

```
100 class MainWindow(QWidget):
101     def __init__(self):
102         super().__init__()
103         self.setWindowTitle("Object Detection and Segmentation")
104         self.resize(1200, 700)  # Make window larger
105
106         # Create buttons
107         self.toggle_detection_button = QPushButton("Enable Detection")
108         self.toggle_detection_button.clicked.connect(self.toggle_detection)
109
110         self.toggle_segmentation_button = QPushButton("Enable Segmentation")
111         self.toggle_segmentation_button.clicked.connect(self.toggle_segmentation)
112
113         # Create text input and confirm button for object filtering
114         self.object_input = QLineEdit()
115         self.object_input.setPlaceholderText("Enter object name (e.g., person, car, chair)")
116         self.confirm_button = QPushButton("Confirm")
117         self.confirm_button.clicked.connect(self.confirm_target_object)
118
119         # Create labels for displaying video
120         self.video_label = QLabel()
121         self.video_label.setFixedSize(640, 480)
```

Creating the buttons, input fields, and labels needed to control the different modes of the program.

Displaying the logo and creator information

```
123         # Create logo label
124         self.logo_label = QLabel()
125         self.logo_label.setFixedSize(100, 80)
126         try:
127             logo_pixmap = QPixmap("urmia_logo.png")
128             scaled_logo = logo_pixmap.scaled(80, 80, aspectRatioMode=1)  # Keep aspect ratio
129             self.logo_label.setPixmap(scaled_logo)
130         except:
131             self.logo_label.setText("Logo")
132             self.logo_label.setStyleSheet("border: 1px solid gray; text-align: center;")
133 
134         # Create creator label
135         self.creator_label = QLabel("Creator: Mahya Kheirandish")
136         self.creator_label.setStyleSheet("font-size: 12px; color: black; text-align: left;")
137         self.creator_label.setFixedSize(200, 20)
138 
139         # Create side panel with available objects list
140         self.objects_list = QListWidget()
141         self.objects_list.setMaximumWidth(200)
142         self.objects_list.setMinimumWidth(200)
```

Displaying the university or project logo and the software designer’s name at the top of the window.

List of classes detectable by the model

```
145         available_objects = list(model_detection.names.values())
146         for obj in sorted(available_objects):
147             self.objects_list.addItem(obj)
148 
149         # Connect list item click to input field
150         self.objects_list.itemClicked.connect(self.on_object_selected)
151 
152         # Layout for main content
153         input_layout = QHBoxLayout()
154         input_layout.addWidget(QLabel("Target Object:"))
155         input_layout.addWidget(self.object_input)
156         input_layout.addWidget(self.confirm_button)
157 
158         button_layout = QVBoxLayout()
159         button_layout.addWidget(self.toggle_detection_button)
160         button_layout.addWidget(self.toggle_segmentation_button)
161 
162         # Create top header layout with logo and creator at very top
163         header_layout = QHBoxLayout()
164         logo_info_layout = QVBoxLayout()
165         logo_info_layout.addWidget(self.logo_label)
166         logo_info_layout.addWidget(self.creator_label)
167         header_layout.addLayout(logo_info_layout)
168         header_layout.addStretch()  # Push everything else to the right
169 
170         # Video layout centered with proper spacing
171         video_container_layout = QHBoxLayout()
172         video_container_layout.addStretch()
173         video_container_layout.addWidget(self.video_label)
174         video_container_layout.addStretch()
```

```
176         main_content_layout = QVBoxLayout()
177         main_content_layout.addLayout(header_layout)  # Logo at top
178         main_content_layout.addSpacing(20)  # Add space between logo and video
179         main_content_layout.addLayout(video_container_layout)  # Video centered
180         main_content_layout.addLayout(input_layout)
181         main_content_layout.addLayout(button_layout)
182 
183         # Side panel layout
184         side_panel_layout = QVBoxLayout()
185         side_panel_layout.addWidget(QLabel("Available Objects:"))
186         side_panel_layout.addWidget(self.objects_list)
```

Setting layouts and arranging components

```
188         # Main horizontal layout combining content and side panel
189         main_layout = QHBoxLayout()
190         main_layout.addLayout(main_content_layout)
191         main_layout.addLayout(side_panel_layout)
192         self.setLayout(main_layout)
193 
194         # Detection thread
195         self.detection_thread = DetectionThread()
196         self.detection_thread.frame_updated.connect(self.update_frame)
197         self.detection_thread.start()
198 
199     def on_object_selected(self, item):
200         """Fill the input field when an object is selected from the list"""
201         self.object_input.setText(item.text())
```

Visual layout of the user interface organized logically and professionally using `QVBoxLayout` and `QHBoxLayout`.

Control functions for UI operation

Selecting an object from the list

```
199     def on_object_selected(self, item):
200         """Fill the input field when an object is selected from the list"""
201         self.object_input.setText(item.text())
```

When clicking on an item in the object list, its name appears in the input field.

Confirming the selected object

```
203     def confirm_target_object(self):
204         global target_object
205         target_object = self.object_input.text().strip()
206         if target_object:
207             print(f"Target object set to: {target_object}")
208         else:
209             print("Target object cleared - showing all objects")
```

Enable or disable detection or segmentation

```
211     def toggle_detection(self):
212         global detection_enabled, segmentation_enabled
213         if not detection_enabled:  # Enable detection
214             detection_enabled = True
215             segmentation_enabled = False  # Ensure segmentation is disabled
216             self.toggle_detection_button.setText("Disable Detection")
217             self.toggle_segmentation_button.setText("Enable Segmentation")
218         else:  # Disable detection
219             detection_enabled = False
220             self.toggle_detection_button.setText("Enable Detection")
221 
222     def toggle_segmentation(self):
223         global detection_enabled, segmentation_enabled
224         if not segmentation_enabled:  # Enable segmentation
225             segmentation_enabled = True
226             detection_enabled = False  # Ensure detection is disabled
227             self.toggle_segmentation_button.setText("Disable Segmentation")
228             self.toggle_detection_button.setText("Enable Detection")
229         else:  # Disable segmentation
230             segmentation_enabled = False
231             self.toggle_segmentation_button.setText("Enable Segmentation")
```

These functions are used to enable or disable detection and segmentation modes. Only one of them can be active at any given time.

Image update in the graphical interface

```
233     def update_frame(self, frame):
234         # Convert OpenCV frame to QImage
235         rgb_frame = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
236         h, w, ch = rgb_frame.shape
237         bytes_per_line = ch * w
238         q_image = QImage(rgb_frame.data, w, h, bytes_per_line, QImage.Format_RGB888)
239 
240         # Display QImage in QLabel
241         self.video_label.setPixmap(QPixmap.fromImage(q_image))
```

Converting the received frame into a displayable format in PyQt and displaying it in a `QLabel` as the live output of the camera.

Managing the proper termination of the application.

```
243     def closeEvent(self, event):
244         cap.release()
245         cv2.destroyAllWindows()
246         self.detection_thread.terminate()
247         event.accept()
```

Releasing resources (such as the camera) and terminating threads upon closing the main window.

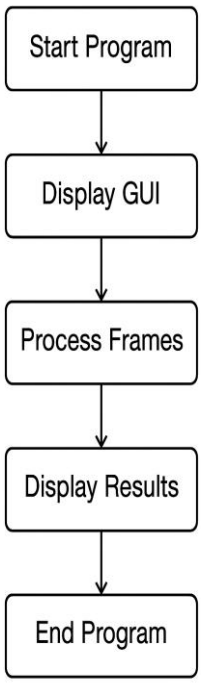
Running the application

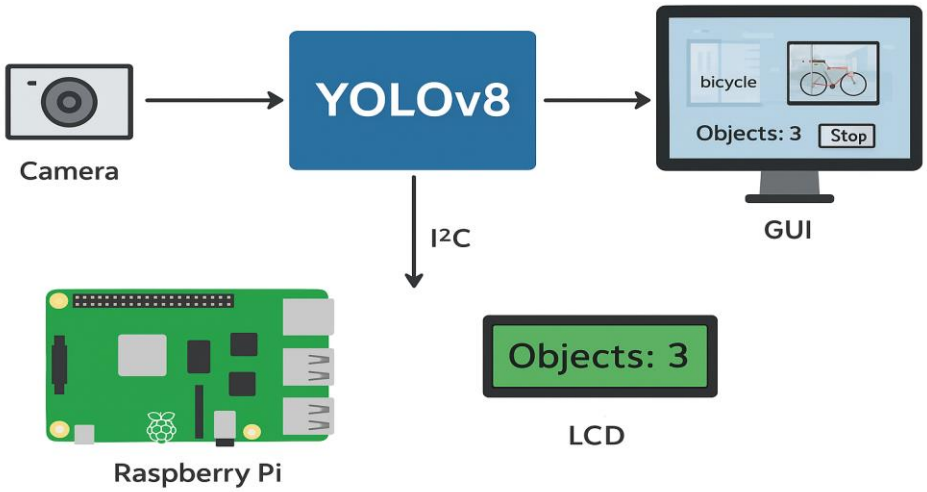
```
250 # Run the application
251 app = QApplication([])
252 window = MainWindow()
253 window.show()
254 app.exec_()
```

Creating an instance of the PyQt application, displaying the main window, and starting the application's main event loop**.**

4.6 System Workflow on Raspberry Pi

1. **Program Start:** The Pi runs the program, initializing the camera and LCD.
2. **GUI Display:** The Tkinter window opens, allowing the user to select the desired objects.
3. **Frame Processing:** Each camera frame is analyzed, objects are detected, and the results are displayed on both the image and the LCD.
4. **Results Display:** The number and types of detected objects are shown to the user.
5. **Program Termination:** Upon closing the GUI, resources are released and the program safely terminates.





Chapter 5: Testing, Performance Evaluation, and Results Analysis

5.1 Introduction

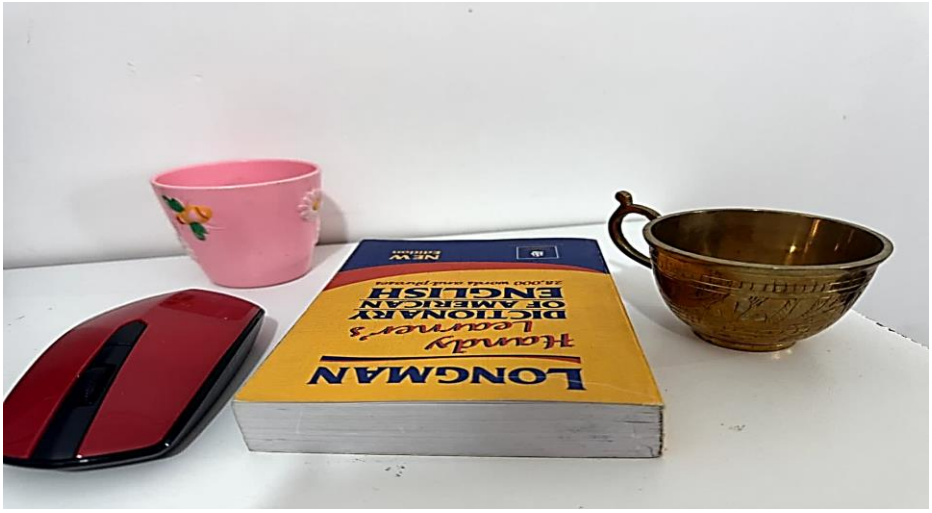
This chapter presents the results obtained from the practical implementation of the YOLOv8-based object detection system on Raspberry Pi hardware and a laptop. Additionally, the performance of two main approaches in the field of computer vision, namely Object Detection and Instance Segmentation, are compared, with their strengths and weaknesses evaluated under real project conditions. The data and images collected from practical experiments form the basis of the analyses presented in this chapter.

Considering the application of this system in robotics, the experiments were designed to assess the model's usability in real-world scenarios involving service and domestic robots. This includes evaluating the system's ability to identify obstacles in the path, accurately segment objects for mechanical tasks such as picking or moving, and respond swiftly to environmental changes. The results not only demonstrate the model's accuracy and speed but also analyze its compatibility with the processing and energy constraints typical of robotic hardware.

5.2 Experimental Scenario

To evaluate the system, a controlled scenario was designed. Four different objects were placed on a table:

- A book
- A mouse
- Two cups



The USB camera connected to the Raspberry Pi 5 captured live images and sent them to the YOLOv8n model for processing. In the initial stage, the system simultaneously detected all objects present, identified their locations, and displayed them with text labels. Then, using the class filter feature in the user interface, only the class "cup" was selected in the first phase and the class "mouse" in the second phase. This capability demonstrated that the system can selectively focus on specific objects, which is highly useful in robotic scenarios.

In service robots, such functionality enables the robot to purposefully recognize the target object and perform subsequent actions such as picking, moving, or avoiding collisions. For example, a robot can monitor the environment under normal conditions and focus only on specific objects like "cup" or "tool" when needed, thereby improving efficiency and response speed.

5.3 Models and Hardware Specifications

Models:

- YOLOv8n (Object Detection)
- YOLOv8n-Seg (Instance Segmentation, pre-trained on the COCO dataset)

Hardware:

- Laptop
- Raspberry Pi

5.4 Results of Model Execution on the Laptop

Processing Speed:

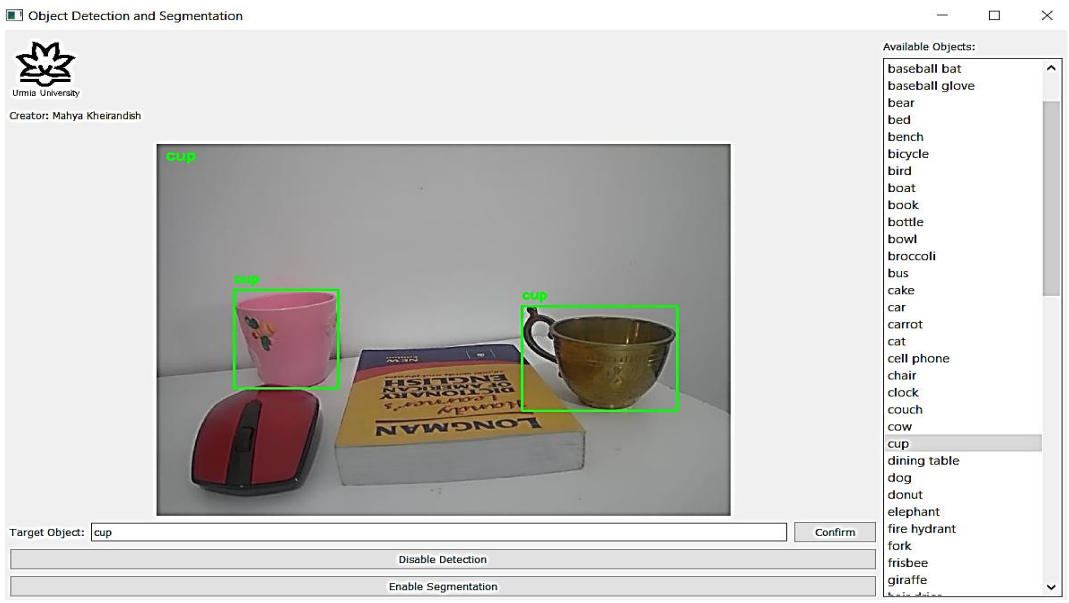
- Detection ≈ 45 FPS
- Segmentation ≈ 25 FPS

Accuracy:

Both models were able to accurately detect the objects present in the experimental scenario.

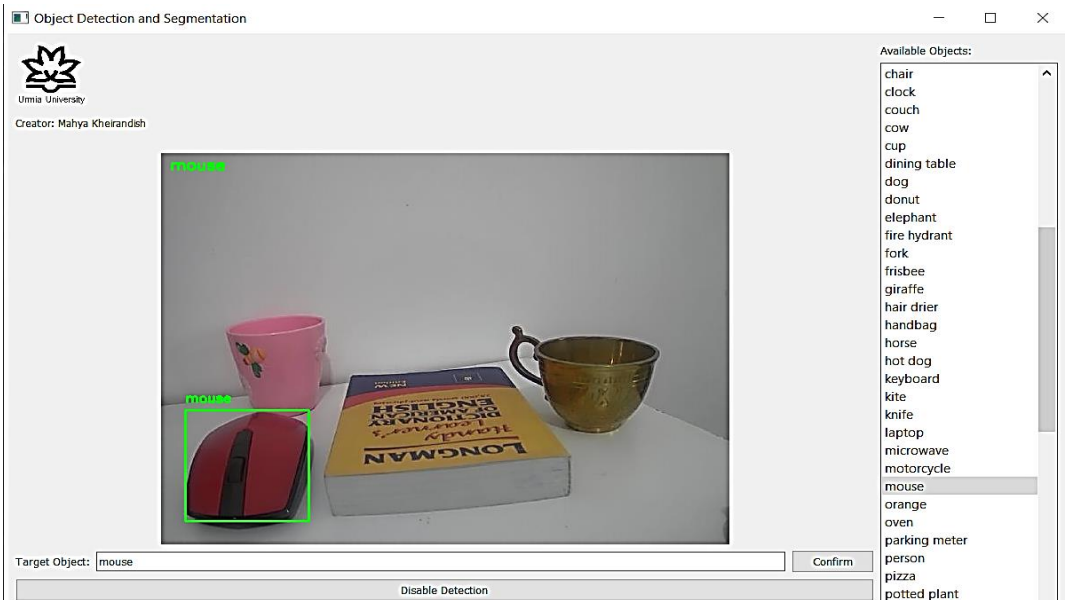
Output:

- **Detection:** Objects were displayed with bounding boxes and class labels.
- **Segmentation:** In addition to detection, precise pixel-wise masks were applied to each object.

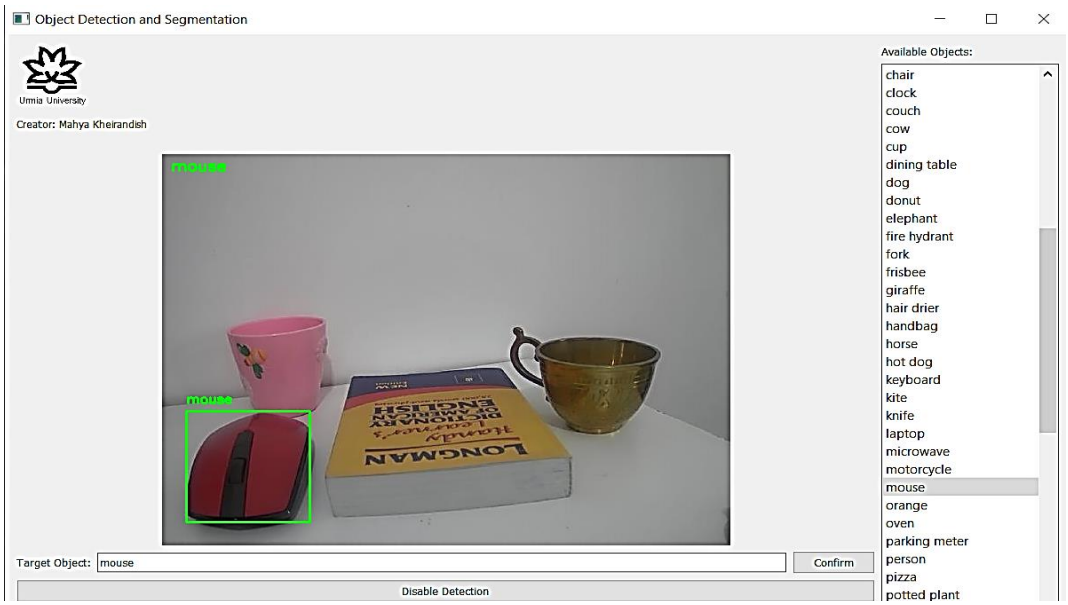


Detection

Segmentaion



Detection



Segmentaion

5.5 Results of Model Execution on Raspberry Pi

- YOLOv8n (Detection), by reducing the input resolution to 640×480 and utilizing threading, was able to detect objects at approximately 6 FPS and display the results on both the LCD and GUI.
- YOLOv8n-Seg (Segmentation), due to its high computational demands and the hardware limitations of the Raspberry Pi, was practically unusable in real-time and therefore was not executed.

5.6 Thesis Summary

In this project, the YOLOv8n and YOLOv8n-Seg models were evaluated on different hardware platforms, including a laptop and the Raspberry Pi 5. The main goal was to design an intelligent object detection and segmentation system for a home environment, capable of providing precise information for automated analysis and decision-making, and to serve in the future as the visual brain for robots. The key results are as follows:

Model Execution on Laptop

- Successful execution of both YOLOv8n and YOLOv8n-Seg on the laptop enabled fast and accurate object processing.
- The segmentation model was able to delineate precise boundaries for each object, facilitating advanced environmental analysis.
- This level of accuracy provides a foundation for robots to identify, classify, and interact with objects, such as picking up a cup or organizing items.

Model Execution on Raspberry Pi 5

- The Raspberry Pi was able to run the lightweight YOLOv8n model at an acceptable speed, but running YOLOv8n-Seg was not feasible due to hardware resource limitations.
- This indicates that small robots or IoT devices can perform real-time processing and basic decision-making with lightweight models, whereas more precise segmentation and advanced interaction require more powerful hardware.

Comparison of Object Detection and Instance Segmentation

- **Object Detection:** Fast and resource-efficient, suitable for robots needing quick object identification.
- **Instance Segmentation:** More precise, enabling advanced analysis such as measurement, counting, and object management; applicable for industrial or service robots requiring detailed environmental interaction.
- The choice between these approaches in robotic projects should be based on hardware capabilities and accuracy requirements.

Practical Applications in Robotics and Smart Homes

- **Smart Homes:** The system can identify, segment, and display various objects, interacting effectively with users.
- **Future Robotics:** Extracted data enables robots to navigate, detect obstacles, pick up and organize objects, or interact safely with their surroundings. This system can become the visual brain of robots, serving as a foundation for intelligent service and domestic robots.

Recommendations and Key Points

- Optimization of models and hardware for real-time performance on small robots is essential.
- More powerful hardware is required for advanced robotic applications and precise object segmentation.
- Data collection and result analysis can serve as a basis for future research and improvement of intelligent robotic systems.

Final Conclusion

The integration of artificial intelligence, image processing, and low-power hardware has enabled the development of an intelligent and practical system usable even on small robots. This project demonstrated that robots can, in the future, identify, classify, and interact with environmental objects using such systems, functioning as smart domestic or service robots. Therefore, this system lays a strong foundation for the development of future robots with advanced vision capabilities and autonomous decision-making.