Qatar University

College of Engineering

Department of Computer Science and Engineering

CMPE 483 Introduction to Robotics

Course Project Report

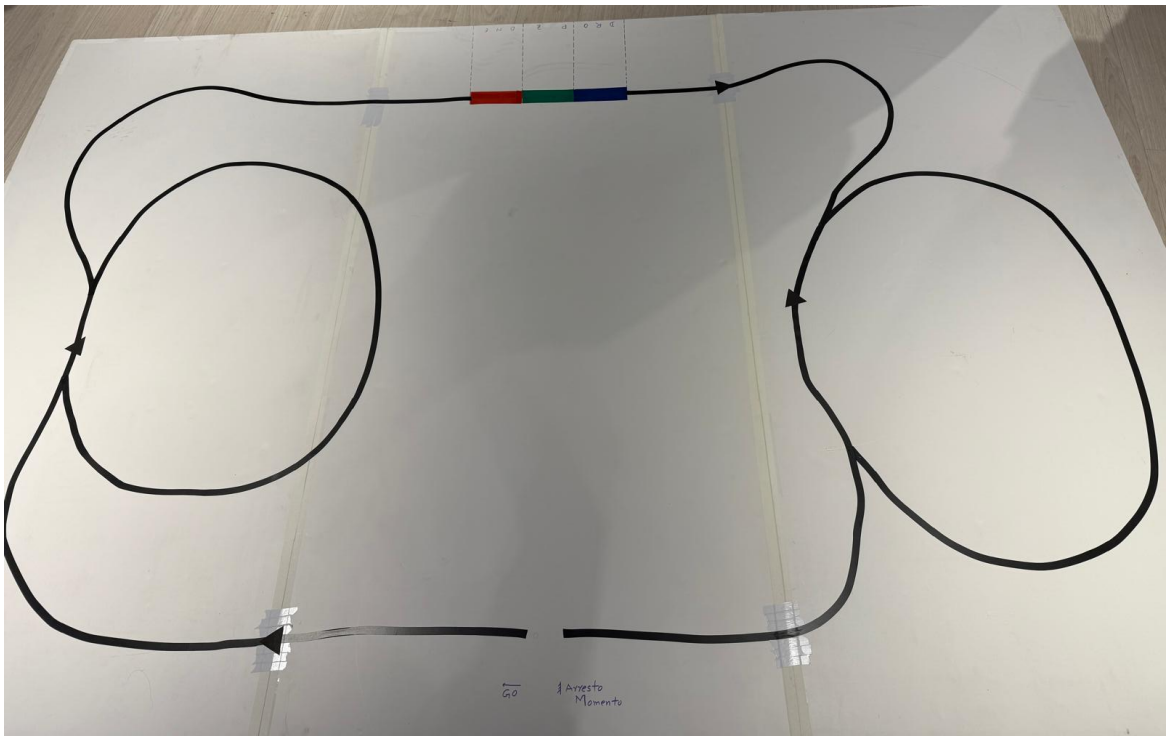
Line-following robot design using Master Pi

Project Group Member:

1. Munif al-Housani 201605496
2. Zabin Al-Dosari 202009034
3. Abdelaziz Shehata 202107370
4. Sultan al-harami 202103147

Problem Statement

In this project, you are required to design a simple autonomous mobile robot. The main track (size 2x3 m) is shown below which was used for an internal robotics contest on May 10th 2025:



Your robot is mainly a line follower which follows a 1-inch-wide black line on a white background. In addition, the black triangle symbols on the track which indicate the decision making needed in that level of the contest as follows:

1. Straight Pointing triangle: go straight and don't get distracted by the upcoming merging lines.
2. Right/Left pointing triangle: At the next bifurcation, stay on the right/left side.

The robot is shown a colored cube (Red, Green or Blue) at the start. It must detect the color and indicate it as beeps (1, 2, or 3 respectively), but our implementation displays the detected color on a debug screen in real-time for improved user feedback and testing clarity. When the same-colored section is spotted on the track, the robot must drop the cube on the left of the colored areas.

Table of Contents

Table of Figures

List of Tables

1. High Level Architecture

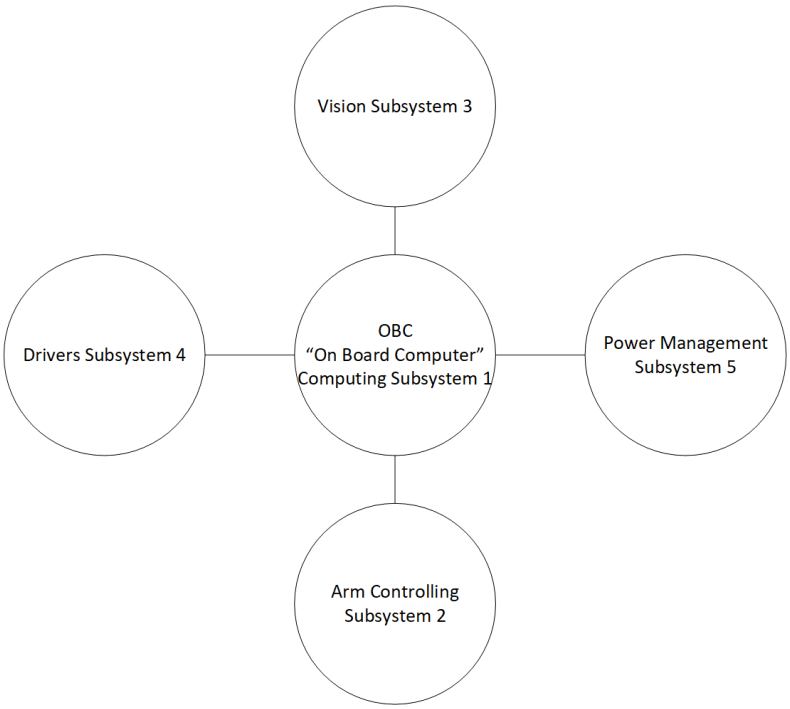


Figure 1 High Level Architecture

Figure 1 illustrates the system-level structure of the robot, with the Raspberry Pi 5 serving as the On-Board Computer (OBC) and coordinating all other hardware subsystems. The robot is designed in a modular fashion, with each subsystem handling a specific function and communicating with the Raspberry Pi through dedicated interfaces such as CSI (camera), GPIO (motor control), I2C (servo control), and USB serial (arm command signaling).

The Raspberry Pi 5 functions as the On-Board Computer (OBC) and is the central controller of the robotic system. It is responsible for processing camera input, executing decision-making algorithms, and managing communication between all subsystems. It receives video input from the PiCamera v2.1 and performs real-time image processing to handle tasks such as line following, triangle detection, object color identification, and drop zone recognition. Based on this visual input, the Raspberry Pi determines navigation paths and behavioral transitions. Motor control is achieved by sending PWM signals from the Raspberry Pi to the L298N motor driver, allowing for dynamic speed and direction adjustments based on PID control logic. It directly controls both DC motors used for movement. The Raspberry Pi also communicates with the Arduino Uno over USB serial. When a valid drop zone is identified, the Pi sends a signal to trigger a predefined servo movement sequence for the robotic arm. This centralized architecture enables the Raspberry Pi 5 to coordinate sensing, movement, and task execution in a autonomous manner.

The Arm Controlling Subsystem consists of an Arduino Uno connected to an Adafruit PCA9685 servo driver, which controls eight servo motors responsible for the robotic arm's movement. This subsystem operates independently of the main robot logic and responds to commands received from the Raspberry Pi 5. Communication between the Raspberry Pi and the Arduino is established via USB serial. When the robot detects the correct drop zone color, the Raspberry Pi sends a single-character signal ('0', '1', or '2') to the Arduino. Each value corresponds to a predefined pose: idle, grab, or release.

Upon receiving the signal, the Arduino executes the associated servo movements by sending PWM signals through the PCA9685 driver. The use of the PCA9685 allows for precise control of all eight servos simultaneously using I2C communication. This subsystem handles all mechanical interactions with the object, including gripping it at the start and releasing it at the correct drop location, enabling the robot to complete the delivery task without manual intervention.

The Vision Subsystem is built around the Raspberry Pi Camera v2.1, which connects to the Raspberry Pi 5 through the CSI interface. It captures continuous video frames that are processed in real time to support the robot's core perception tasks. This subsystem enables the detection of four critical visual cues: the initial object color (red, green, or blue), the black line used for navigation, directional triangle signs indicating turns, and wide color zones used to trigger the object drop. All image processing is performed on the Raspberry Pi using the OpenCV library. The camera feed is used to extract features such as contours, shapes, and color segments, which are then passed to the control logic for decision-making. The Vision Subsystem plays a key role in ensuring the robot navigates the track correctly, interprets the environment accurately, and performs the object drop action at the appropriate location.

The Motor Driver Subsystem is responsible for controlling the movement of the robot using two DC motors connected through an L298N motor driver. It receives direction and speed commands from the Raspberry Pi 5 via GPIO pins. The Raspberry Pi generates PWM signals to control the motor speeds and sets the motor direction based on real-time feedback from the Vision Subsystem. A PID control algorithm adjusts the left and right motor speeds independently to correct for any deviation from the black line. This subsystem allows the robot to move forward, turn, or stop based on visual inputs and decision logic processed by the On-Board Computer. Its reliable response to control commands ensures smooth and accurate navigation along the predefined track.

The Power Management Subsystem supplies electrical power to all components of the robot through a combination of sources. A 12V battery powers the L298N motor driver and DC motors directly, while a buck converter steps down the voltage to 5V to supply the servo control board. The Raspberry Pi 5 is powered independently via a USB power bank, ensuring stable operation during processing and camera usage. The Arduino Uno receives power through the USB connection from the Raspberry Pi, allowing for synchronized startup and communication. A digital voltage display is included to monitor the battery level in real time. This subsystem ensures that each part of the robot receives the appropriate voltage and current levels for safe and reliable operation throughout the mission.

This modular architecture allows each subsystem to focus on a specific role while being coordinated centrally through the Raspberry Pi, resulting in a responsive and autonomous robotic system.

2. Design Details

To implement a functional, efficient, and reliable robotic system, our design approach was divided into distinct but interdependent hardware and software components. Each subsystem was carefully selected to meet the specific operational requirements and constraints of the project, including real-time performance, power management, physical integration, and modular control. The following subsections provide an in-depth breakdown of both the hardware and software components, supported by diagrams and flowcharts where necessary. Each part is justified based on its role in ensuring the system meets its intended functionality.

2.1. Hardware

The following subsection outlines the hardware components and their roles in fulfilling the robot's functional and performance requirements. It includes a breakdown of how each device was selected, connected, and integrated to ensure efficient operation within the project's design constraints.

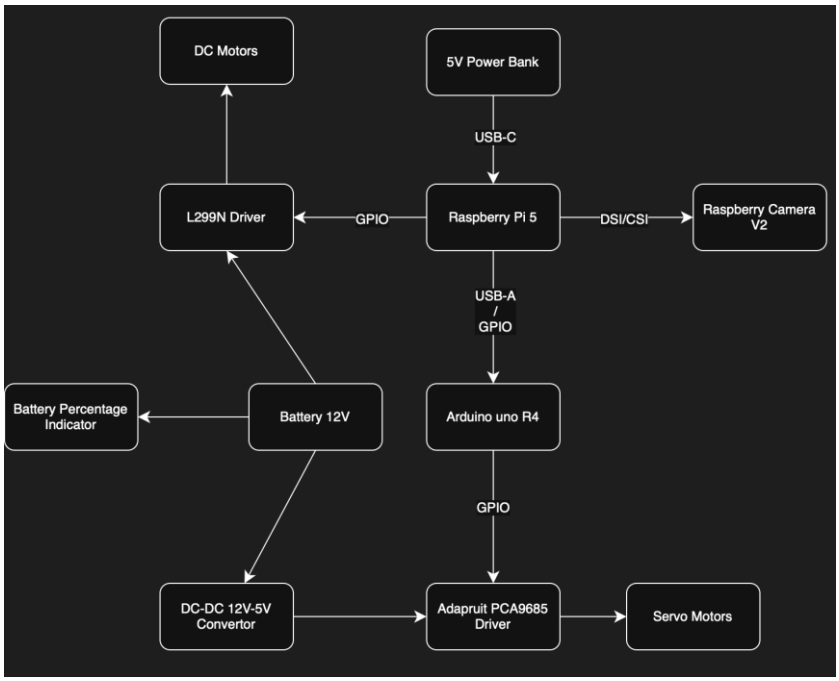


Figure 2 Connectivity Diagram

Table 1 Tools and Frameworks Used

Tool/Framework	Purpose in Project
Raspberry Pi 5	Central controller, processes camera input.
Arduino Uno R4	Through the servo driver, controls the servo motors on the robotic arm
Adafruit PCA9685 Servo Driver	Controls 8 servo motors. Connected to Arduino for arm movement.
L298N Motor Driver	Drives two DC motors (robot movement) using GPIO signals from Raspberry Pi.
Pi Camera v2.1	Captures real-time visual input
12V Battery Pack	Powers DC motors and Servo Driver through a buck converter.
Buck Converter (12V to 5V)	Converts 12V battery output to 5V for powering the servo driver.
Power Bank	Powers Raspberry Pi via USB-C.

2.1.1 Justification For Approach

As part of the project constraints, we were required to use the Raspberry Pi 5 as the main controller and predefined DC motors. Beyond these fixed components, several design decisions were made to optimize performance. First, we selected a 12V 10A battery to power the system, as the robot's weight demanded a strong and stable power supply to drive the motors effectively. Second, although we initially received two connected robotic arms, we determined that only one was necessary to complete the object-handling tasks. Therefore, we removed the extra arm to reduce weight and simplify control. For controlling the 8 servo motors of the remaining arm, we used an Arduino Uno connected to a PCA9685 servo driver. This was essential because the Raspberry Pi's Raspbian OS lacks native support for the Adafruit servo libraries.

2.1.2 Interface Approach Used

Table 2 Interface Approach Used

Interface Type	Components Connected	Purpose
GPIO	Raspberry Pi → L298N Motor Driver	Controls DC motors' direction and speed via PWM
USB-A	Raspberry Pi ↔ Arduino Uno	Sends predefined signals (0,1,2) to control arm pose
I²C	Arduino Uno → Adafruit PCA9685 Servo Driver	Controls up to 8 servo motors on the robotic arm
Power Wiring	Battery Pack → Buck Converter & Driver	Supplies required voltage
USB-C	Power Bank → Raspberry pi	Supplies Power for the Ras

2.1.3 Novel Aspects of Interface Design

- Predefined servo positions are stored in Arduino, offloading servo logic from Raspberry Pi.
- Object color detection is handled only once before switching to navigation, reducing Raspberry Pi processing load.
- A hybrid control system is used: Raspberry Pi handles vision and movement, while Arduino controls the arm servos. This simplifies design and avoids servo library issues on Raspberry Pi.

2.2. Software

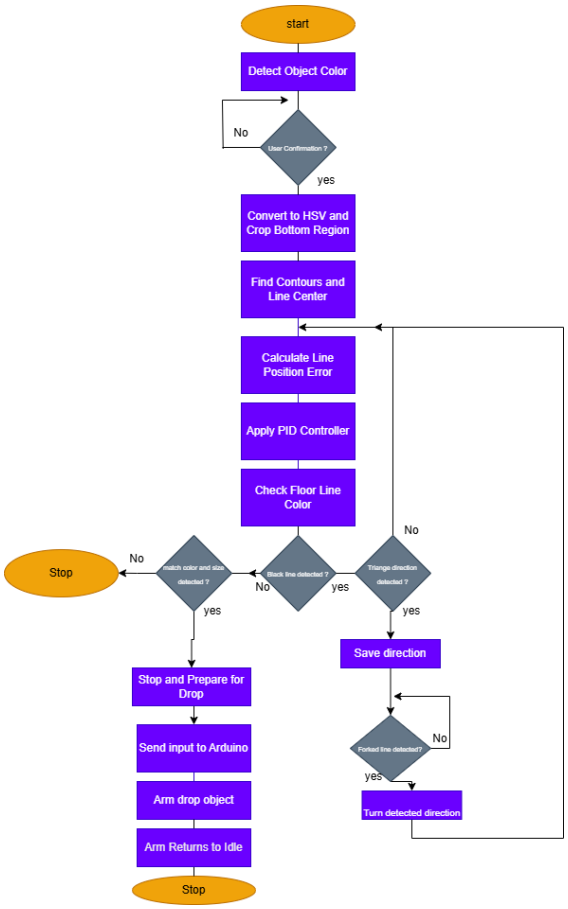


Figure 3 Data Flowchart

Table 3 Implemented Software

Software	Purpose
Visual Studio Code	Used for managing Python code in the Raspberry Pi for the robot's logic.
Arduino IDE	Used to write C code for the Arduino that controls the robot arm via PCA9685 by choosing 0,1,2 to move the arm.
Raspberry Pi OS	An operating system that is running on the Raspberry Pi supports code execution and hardware interfacing.
Thonny IDE	A lightweight Python editor that is used to test and run Python code directly on the Raspberry Pi.

Because of their ease of use and support for GPIO, OpenCV, and real-time camera interfacing, we decided to use Visual Studio Code and Thonny as our Python programming environments for Raspberry Pi OS. VS Code offers better access to files. Thonny is lightweight and perfect for rapid testing. We used the Arduino IDE to write and upload code to control the robotic arm. It supports communication and provides a simple way to program Arduino using C/C++. Raspberry Pi OS was chosen as the operating system because it is fully compatible with Python, GPIO libraries, and camera modules. It also provides a stable environment for running the robot's main control logic.

The robot uses a camera to detect the black line on the floor. First, it converts the image to the HSV color format to better detect the line. It focuses only on the lower part of the image, where the line is expected. Using color thresholding, it creates a black-and-white image where the black line appears as white. Then, it filters the image to remove noise and finds the center of the line. The robot compares this center with the middle of the image to calculate how far it has drifted. This difference is called the error. A PID controller uses that error to adjust the motor speeds, helping the robot stay on track. If the robot doesn't detect any line, it simply stops and displays "No line detected".

3. Implementation

3.1. System Integration and Assembly

The robot's physical structure is based on a tracked platform with two DC motors for movement and a multi-jointed robotic arm mounted on top. The central computing unit, Raspberry Pi 5, is mounted on the chassis and serves as the On-Board Computer (OBC). It communicates with and coordinates all other subsystems. A PiCamera v2.1 is mounted on the end-effector (gripper) of the robotic arm, giving the robot a dynamic and frontal view of its environment. This camera handles all visual tasks, including line following, triangle detection, and drop zone identification. It connects directly to the Raspberry Pi via the CSI interface. Two DC motors drive the tracks and are connected to an L298N motor driver (Figure 12), which receives PWM and direction signals from the Raspberry Pi through GPIO pins. The robotic arm is powered by eight servo motors connected to an Adafruit PCA9685 PWM driver board (Figures 4 and 6). This driver is controlled by an Arduino Uno R4 WiFi (Figure 5), which receives movement commands from the Raspberry Pi over a USB serial link. The PCA9685 board receives power from a 5V buck converter (Figure 8), which steps down the 12V input from the Li-ion battery pack (Figure 7). The Raspberry Pi is powered separately using a 20,000 mAh USB power bank (Figure 9), ensuring stable and isolated power delivery. The entire assembly is mounted securely to avoid loose connections or vibration during movement. Servo wires, motor cables, and data connections are routed and taped to maintain mechanical reliability. The completed integration of all hardware components is shown in Figures 10 and 11.

The tracked chassis features a hollow base structure that serves as the internal housing for core electronics. Inside this compartment, the Raspberry Pi 5 and the L298N motor driver were securely mounted on the floor using double-sided adhesive foam for stability and vibration dampening. A rectangular slot inside the base was used to insert the 12V battery pack snugly, keeping it locked in place during motion. Above the battery compartment, a strong double-sided adhesive was used to attach the 20,000 mAh power bank, ensuring the Raspberry Pi remained powered independently and without accidental disconnection, a small digital voltage display was mounted near the battery to provide real-time monitoring of the power level during testing and operation. This helped ensure the robot did not enter undervoltage states that could affect servo or motor performance.

The top of the chassis provides structural support for the robotic arm. The servo wires were routed neatly along the arm structure using electrical tape and zip ties, then passed through the rotating base into the lower housing where they connect to the PCA9685 board. All connections were organized to avoid interference with moving parts, and stress-relieved to prevent detachment during movement. The PiCamera ribbon cable was also fed carefully through the arm's segments and fixed using black tape to maintain flexibility during arm motion.

3.2. Servo Pose Calibration GUI

To ensure precise arm movement during the drop-off operation, a custom Python-based graphical interface was developed to calibrate and record the servo positions of the 8-degree-of-freedom robotic arm. The GUI was created using the Tkinter library and interfaced with the Arduino Uno R4 over USB serial communication. The interface featured eight horizontal sliders, each mapped to a specific servo motor. Adjusting a slider sent a real-time PWM value (ranging from 100 to 500) to the corresponding servo via the Arduino, enabling fine-tuned manual control of each joint. This allowed the user to visually position the arm into any desired configuration and save it under named reference poses.

Three key positions were recorded using the GUI:

- Ref1_Idle: A default resting position used when the robot is not actively grabbing or releasing.
- Ref2_Grab: The position in which the gripper is closed around an object.
- Ref3_Release: The position where the gripper opens to release the object in the drop zone.

Each saved pose was stored in a Python dictionary and printed to the console in list format. These values were then hard coded into the Arduino sketch, allowing the arm to immediately respond to serial commands sent by the Raspberry Pi during execution. This manual calibration approach simplified the tuning process and ensured that the robotic arm performed consistently in physical trials. The GUI also reduced the risk of hardware damage by allowing careful pose adjustment without repeated code uploads or guesswork. This tool was essential for ensuring accurate and repeatable mechanical behavior during drop-off tasks.

3.3. Real-World Testing and Validation

Testing was conducted iteratively in multiple phases, starting with basic motor validation and gradually building toward autonomous delivery. Each phase introduced new functionality motors, line following, triangle detection, drop zone logic, and arm actuation and was tested independently before integration. Early testing began by verifying the connection and directional control of the DC motors using the L298N driver. Once motor control was confirmed, a basic black line-following prototype was developed and tested using black electrical tape on a white floor surface. These initial trials were conducted in the team's own workspace and later repeated successfully at the instructor's office.

As development progressed, additional logic for color-based object detection and directional triangle signs was added. Triangle detection using custom-printed markers proved to be highly reliable, with the system correctly identifying and responding to left, right, and straight symbols. Color detection also worked consistently under normal lighting, with no recorded misclassifications between red, green, or blue drop zones.

The robot was tested extensively on a custom-built track created by the team using white floor panels, black, blue, red, and green electrical tape for the navigation path, and printed triangle markers for turn instructions. The layout was designed to simulate the conditions of the final contest, including colored starting zones, intersections, and drop areas. In over 10 test cycles, the robot consistently completed approximately 80% of the required steps per run. Two full delivery cycles including detection, navigation, triangle handling, drop execution, and resume were completed successfully. Each run provided valuable feedback that was used to refine the codebase and stabilize the behavior of all subsystems.

Across more than ten test cycles, the robot consistently completed around 80% of the required behaviors in each run. In two full delivery cycles, the robot successfully detected the object color, followed the line, interpreted triangle signs, matched the drop zone, and executed the drop action. These successful runs demonstrated complete end-to-end system functionality and confirmed the robot’s ability to perform autonomously under real conditions.

During these trials, the robot was able to recover from minor issues such as triangle misalignment or brief loss of tracking. In cases where the black tape reflected ambient light (particularly under strong overhead lighting), the line-following algorithm occasionally failed to detect the path, causing the robot to stop. Once the surface glare was minimized, the robot performed at its best, with smooth navigation and accurate visual detection.

During the final competition demo, the robot met expectations and performed its tasks as designed. The system successfully detected the colored object, followed the line, responded to directional signs, matched the drop zone color, and executed the arm movement using the pre-calibrated servo poses. The demo validated the integration between mechanical components, visual intelligence, and actuation logic.

Figure 6 Adafruit PCA9685 Servo Driver (Rear View)

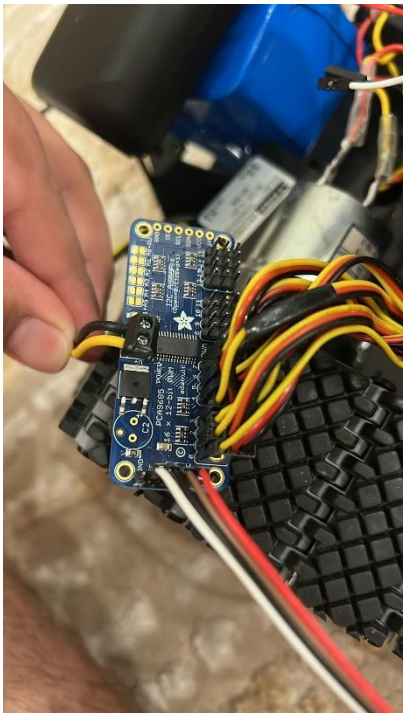


Figure 4 Adafruit PCA9685 Servo Driver (Front Wiring View)

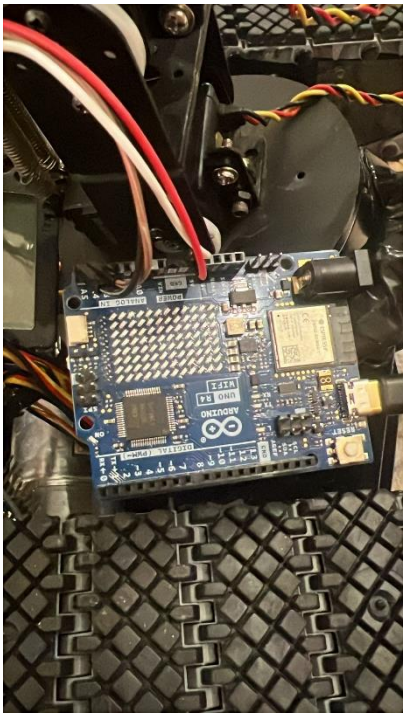


Figure 5 Arduino Uno R4 WiFi Controller

Figure 9 20,000 mAh USB Power Bank

Figure 7 12V Li-ion Battery Pack

Figure 8 Buck Converter (12V to 5V)

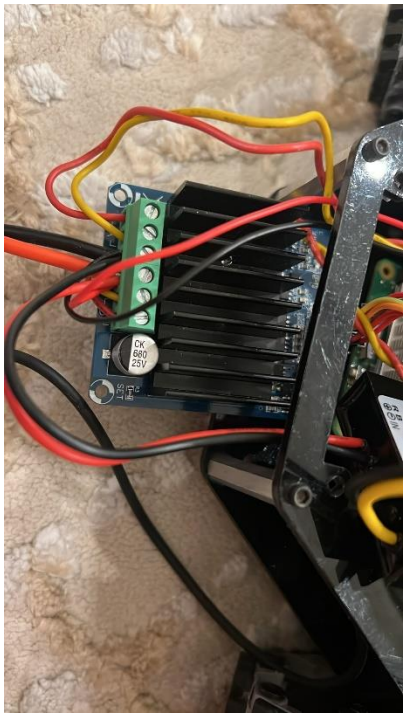


Figure 12 L298N Motor Driver Module

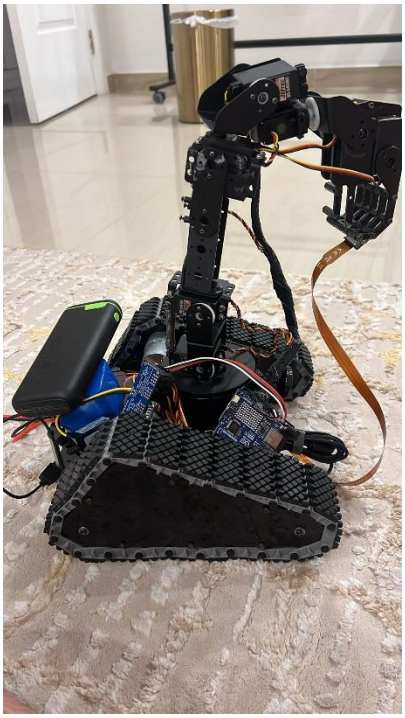


Figure 11 Robot Side View – Fully Integrated System

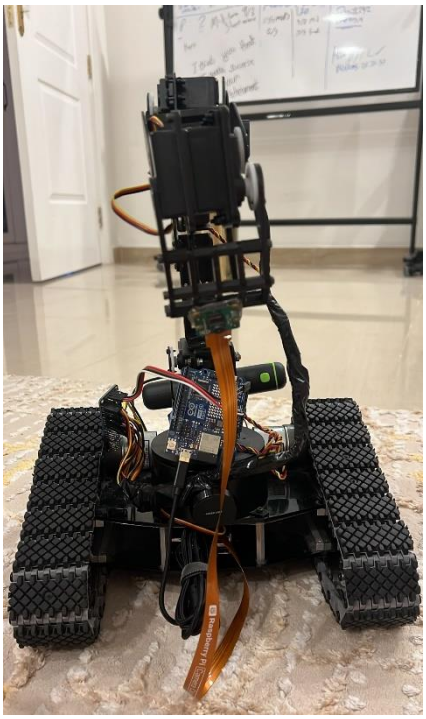


Figure 10 Front View of Final Robot

Appendix

```
#!/usr/bin/env python3
import time
import cv2
import numpy as np
from gpiozero import OutputDevice, PWMOutputDevice
from picamera2 import Picamera2
import os
import json

# ========== Motor Setup ==========
# SWAPPED: Left motor is now what was previously right motor
# Right motor is now what was previously left motor

# RIGHT motor (formerly left motor)
in1 = OutputDevice(23)   # Pin 16
in2 = OutputDevice(24)   # Pin 18
ena = PWMOutputDevice(12)  # Pin 32 (PWM)

# LEFT motor (formerly right motor)
in3 = OutputDevice(17)  # Pin 11
in4 = OutputDevice(27)  # Pin 13
enb = PWMOutputDevice(13)  # Pin 33 (PWM)

# ========== PID Parameters ==========
Kp = 0.25
Ki = 0.0005
Kd = 0.05
last_error = 0
integral = 0

# Speed settings - same as your working sample
base_speed = 0.35
max_speed = 0.55
search_speed = 0.3

# ========== Global Variables ==========
target_color = None
normal_line_width = 0  # Will be dynamically set
next_turn = None
detected_triangles = []
at_intersection = False
drop_handled = False
processing_triangle = False
color_detection_complete = False
waiting_for_object = True
stop_motors_flag = False
```

```
debug_mode = True  # Always enable debug mode

# Create data directory in the current working directory
data_dir = "./robot_data"
os.makedirs(data_dir, exist_ok=True)

# ========== Logging ==========
def save_detection_data():
    data = {
        "target_color": target_color,
        "detected_triangles": detected_triangles
    }
    with open(f"{data_dir}/detection_data.json", "w") as f:
        json.dump(data, f)

# ========== Motor Functions ==========
def set_motor(left_dir, right_dir, left_speed, right_speed):
    # SWAPPED: We're swapping which motors get which commands
    # What was previously sent to left now goes to right
    # What was previously sent to right now goes to left

    # Ensure speeds are proper
    left_speed = min(max(abs(left_speed), 0.0), 1.0)
    right_speed = min(max(abs(right_speed), 0.0), 1.0)

    # LEFT motor (formerly right motor) - uses in3, in4, enb
    if left_dir == "forward":
        in3.off()
        in4.on()
    else:
        in3.on()
        in4.off()
    enb.value = left_speed

    # RIGHT motor (formerly left motor) - uses in1, in2, ena
    if right_dir == "forward":
        in1.off()
        in2.on()
    else:
        in1.on()
        in2.off()
    ena.value = right_speed

    # Debug output for motor values
    print(f"Motors: L={left_dir}({left_speed:.2f}),
R={right_dir}({right_speed:.2f})")

def stop_motors():
```

```
    ena.off()
    enb.off()
    in1.off()
    in2.off()
    in3.off()
    in4.off()
    print("Motors stopped")

# ========== Turning Functions ==========
def turn_left_90_degrees():
    print("\n🔄 Turning left 90 degrees...")
    stop_motors()
    time.sleep(0.5)
    set_motor("backward", "forward", 0.4, 0.4)
    time.sleep(0.8)
    stop_motors()
    time.sleep(0.5)

def turn_right_90_degrees():
    print("\n🔄 Turning right 90 degrees...")
    stop_motors()
    time.sleep(0.5)
    set_motor("forward", "backward", 0.4, 0.4)
    time.sleep(0.8)
    stop_motors()
    time.sleep(0.5)

# ========== Drop Area Behavior ==========
def handle_drop_zone():
    global drop_handled
    print(f"\n🩹 {target_color.upper()} Drop point detected! Executing drop
routine...")
    stop_motors()
    time.sleep(0.5)

    # Turn left 90 degrees
    turn_left_90_degrees()

    # Wait at drop point
    time.sleep(5)

    # Turn right 90 degrees to return to line
    turn_right_90_degrees()

    drop_handled = True
    print(f"\n☑️ Drop handling complete! Now ignoring other drop areas.")

# ========== Color Detection Using Single Camera ==========
```

```
def detect_object_color(frame):
    global target_color, waiting_for_object, color_detection_complete

    # Define a central region of interest
    h, w = frame.shape[:2]
    crop = frame[int(h*0.3):int(h*0.7), int(w*0.3):int(w*0.7)]

    # Draw a rectangle around the detection area
    cv2.rectangle(frame, (int(w*0.3), int(h*0.3)), (int(w*0.7), int(h*0.7)),
(0, 255, 255), 2)

    hsv = cv2.cvtColor(crop, cv2.COLOR_BGR2HSV)

    # Improved color ranges
    red1 = cv2.inRange(hsv, (0, 100, 100), (10, 255, 255))
    red2 = cv2.inRange(hsv, (160, 100, 100), (180, 255, 255))
    green = cv2.inRange(hsv, (40, 50, 50), (85, 255, 255))
    blue = cv2.inRange(hsv, (100, 50, 50), (130, 255, 255))
    red = cv2.bitwise_or(red1, red2)

    r = np.sum(red) / 255
    g = np.sum(green) / 255
    b = np.sum(blue) / 255
    total = crop.shape[0] * crop.shape[1]

    detected = None
    # Higher thresholds for more confident detection
    if r / total > 0.3: detected = "red"
    elif g / total > 0.25: detected = "green"
    elif b / total > 0.25: detected = "blue"

    # Display color percentages for debugging
    r_percent = round(r / total * 100, 1)
    g_percent = round(g / total * 100, 1)
    b_percent = round(b / total * 100, 1)

    cv2.putText(frame, f"Detecting color: {detected if detected else 'none'}",
(10, 30),
                cv2.FONT_HERSHEY_SIMPLEX, 0.7, (0, 255, 255), 2)
    cv2.putText(frame, f"R: {r_percent}% G: {g_percent}% B: {b_percent}%",
(10, 60),
                cv2.FONT_HERSHEY_SIMPLEX, 0.6, (0, 255, 255), 2)

    # Check if we have a confident detection
    if detected and (r / total > 0.3 or g / total > 0.25 or b / total > 0.25):
        if waiting_for_object:
            target_color = detected
            print(f"\n🎯 Detected object color: {target_color.upper()}")
```

```
            waiting_for_object = False
            color_detection_complete = True
            save_detection_data()
            cv2.putText(frame, "COLOR DETECTED! Press any key to start", (50,
240),
                        cv2.FONT_HERSHEY_SIMPLEX, 0.8, (0, 255, 0), 2)
            cv2.imshow("Robot Vision", frame)
            cv2.waitKey(0)  # Wait for key press before starting

    return frame

# ========== Triangle Detection ==========
def detect_triangle(frame):
    global detected_triangles, processing_triangle, next_turn

    # Skip if we're already processing a triangle
    if processing_triangle:
        return None

    h, w = frame.shape[:2]
    # Look in the upper portion of the frame for triangles
    t, b = int(h * 0.2), int(h * 0.6)
    roi = frame[t:b, :]

    # Convert to grayscale and threshold - using your approach from working
sample
    gray = cv2.cvtColor(roi, cv2.COLOR_BGR2GRAY)
    blurred = cv2.GaussianBlur(gray, (5, 5), 0)
    _, thresh = cv2.threshold(blurred, 70, 255, cv2.THRESH_BINARY_INV)

    # Apply morphological operations
    kernel = np.ones((5, 5), np.uint8)
    processed = cv2.erode(thresh, kernel, iterations=1)
    processed = cv2.dilate(processed, kernel, iterations=1)

    # Display the mask for debugging
    mask_display = cv2.cvtColor(processed, cv2.COLOR_GRAY2BGR)
    mask_scaled = cv2.resize(mask_display, (320, 120))
    frame[10:130, w-330:w-10] = mask_scaled

    # Find contours in the mask
    contours, _ = cv2.findContours(processed, cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)

    valid_triangles = []

    for contour in contours:
        # Increase area threshold to avoid false detections
```

```
        if cv2.contourArea(contour) < 200:  # Increased from 100
            continue

        # Use higher epsilon for better shape approximation
        epsilon = 0.04 * cv2.arcLength(contour, True)
        approx = cv2.approxPolyDP(contour, epsilon, True)

        # Only accept shapes with exactly 3 vertices
        if len(approx) == 3:
            # Check if it's a valid triangle (has reasonable area)
            area = cv2.contourArea(approx)
            if area < 200 or area > 5000:  # Filter out very small or very
large triangles
                continue

            # Draw the triangle contour
            cv2.drawContours(frame[t:b, :], [approx], 0, (0, 255, 0), 2)

            # Calculate centroid of the triangle
            M = cv2.moments(approx)
            if M["m00"] != 0:
                cx = int(M["m10"] / M["m00"])
                cy = int(M["m01"] / M["m00"]) + t  # Adjust back to original
frame coordinates

                # Use a better method to determine triangle orientation
                # Find the top vertex (minimum y value)
                sorted_by_y = sorted(approx[:, 0, :], key=lambda p: p[1])
                top_point = sorted_by_y[0]
                bottom_points = sorted_by_y[1:]

                # Sort bottom points by x
                bottom_points = sorted(bottom_points, key=lambda p: p[0])
                left_point, right_point = bottom_points

                # Draw points on the triangle vertices for debugging
                cv2.circle(frame[t:b, :], (top_point[0], top_point[1]), 5,
(255, 0, 0), -1)  # Blue for top
                cv2.circle(frame[t:b, :], (left_point[0], left_point[1]), 5,
(0, 255, 0), -1)  # Green for left
                cv2.circle(frame[t:b, :], (right_point[0], right_point[1]), 5,
(0, 0, 255), -1)  # Red for right

                # Calculate midpoint of the bottom side
                mid_x = (left_point[0] + right_point[0]) // 2
                mid_y = (left_point[1] + right_point[1]) // 2

                # Draw line from midpoint to top
```

```
                cv2.line(frame[t:b, :], (mid_x, mid_y), (top_point[0],
top_point[1]), (255, 0, 255), 2)

                # Determine direction based on top vertex position relative to
midpoint
                if abs(top_point[0] - mid_x) < 20:
                    # Top point is centered above the bottom edge -> pointing
up
                    direction = "up"
                elif top_point[0] < mid_x:
                    # Top point is to the left of the midpoint -> pointing
left
                    direction = "left"
                else:
                    # Top point is to the right of the midpoint -> pointing
right
                    direction = "right"

                # Visualize the detected direction
                direction_text = f"Direction: {direction}"
                cv2.putText(frame, direction_text, (10, 180),
                            cv2.FONT_HERSHEY_SIMPLEX, 0.7, (255, 0, 255), 2)

                valid_triangles.append({"x": cx, "y": cy, "direction":
direction, "area": area})

    # If we found valid triangles, pick the largest one
    if valid_triangles:
        best_triangle = max(valid_triangles, key=lambda x: x["area"])

        # Check if it's a new triangle we haven't seen before
        is_new = True
        for t in detected_triangles:
            if abs(t["x"] - best_triangle["x"]) < 40 and abs(t["y"] -
best_triangle["y"]) < 40:
                is_new = False
                break

        if is_new:
            detected_triangles.append(best_triangle)
            print(f"\n △  Triangle detected pointing
{best_triangle['direction']}! Area: {best_triangle['area']}")
            next_turn = best_triangle["direction"]
            processing_triangle = True
            return best_triangle["direction"]

    return None
```

```
# ========== Process Triangle Decision ==========
def handle_triangle_direction(direction):
    global processing_triangle

    if direction == "left":
        print("\n⬅️ Following triangle direction: LEFT")
        turn_left_90_degrees()
    elif direction == "right":
        print("\n➡️ Following triangle direction: RIGHT")
        turn_right_90_degrees()
    elif direction == "up":
        print("\n⬆️ Following triangle direction: STRAIGHT")
        # Continue straight - no need to turn
        pass

    processing_triangle = False

# ========== Image Processing ==========
def process_image(image):
    global normal_line_width, drop_handled

    # Get image dimensions
    height, width = image.shape[:2]

    # EXACTLY match your working sample's ROI location:
    # Wider vertical ROI: 60% to 80% of frame height
    crop_top = int(height * 0.6)
    crop_bottom = int(height * 0.8)
    roi = image[crop_top:crop_bottom, 0:width]

    # Process image EXACTLY like your working sample:
    gray = cv2.cvtColor(roi, cv2.COLOR_BGR2GRAY)
    blurred = cv2.GaussianBlur(gray, (5, 5), 0)
    _, thresh = cv2.threshold(blurred, 70, 255, cv2.THRESH_BINARY_INV)

    # Apply morphological operations
    kernel = np.ones((3, 3), np.uint8)
    processed = cv2.erode(thresh, kernel, iterations=1)
    processed = cv2.dilate(processed, kernel, iterations=1)

    # Create debug image the same way as your working sample
    debug_image = np.copy(image)
    debug_roi = cv2.cvtColor(processed, cv2.COLOR_GRAY2BGR)
    debug_image[crop_top:crop_bottom, 0:width] = debug_roi
    cv2.rectangle(debug_image, (0, crop_top), (width, crop_bottom), (0, 255,
255), 2)

    # For drop zone detection
```

```
    hsv_roi = cv2.cvtColor(roi, cv2.COLOR_BGR2HSV)
    mask_red = cv2.inRange(hsv_roi, (0, 100, 100), (10, 255, 255)) |
cv2.inRange(hsv_roi, (160, 100, 100), (180, 255, 255))
    mask_green = cv2.inRange(hsv_roi, (40, 50, 50), (85, 255, 255))
    mask_blue = cv2.inRange(hsv_roi, (100, 50, 50), (130, 255, 255))

    # Find contours EXACTLY like your working sample
    contours, _ = cv2.findContours(processed, cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)

    # Now process contours and line detection the same way as your working
sample
    if contours:
        largest = max(contours, key=cv2.contourArea)
        if cv2.contourArea(largest) > 50:  # Same threshold as your working
code
            M = cv2.moments(largest)
            if M["m00"] != 0:
                cx = int(M["m10"] / M["m00"])
                cy = int(M["m01"] / M["m00"])

                # Get bounding box for line width measurement
                x, y, w_box, h_box = cv2.boundingRect(largest)
                line_width = w_box

                # If normal_line_width is not set yet, calibrate it
                if normal_line_width == 0 and line_width > 0:
                    normal_line_width = line_width
                    print(f"\n📏 Normal line width calibrated:
{normal_line_width} pixels")

                # Draw visualization elements EXACTLY like your working sample
                cv2.drawContours(debug_image[crop_top:crop_bottom], [largest],
-1, (0, 255, 0), 2)
                cv2.circle(debug_image[crop_top:crop_bottom], (cx, cy), 5, (0,
0, 255), -1)
                cv2.line(debug_image, (width // 2, 0), (width // 2, height),
(255, 0, 0), 1)
                cv2.line(debug_image[crop_top:crop_bottom], (roi.shape[1] //
2, cy), (cx, cy), (255, 0, 0), 2)

                # Display error value the same way as your working sample
                error = cx - roi.shape[1] // 2
                cv2.putText(debug_image, f"Error: {error}", (10, 30),
                            cv2.FONT_HERSHEY_SIMPLEX, 0.7, (0, 255, 0), 2)

                # Use line width for drop zone detection
```

```
                # Check if we're at a drop zone (line width significantly
larger)
                is_wide_area = normal_line_width > 0 and line_width >= 2.0 *
normal_line_width

                # Add visualization for debugging
                cv2.putText(debug_image, f"Line width: {line_width}px", (10,
60),
                            cv2.FONT_HERSHEY_SIMPLEX, 0.6, (255, 255, 0), 2)
                cv2.putText(debug_image, f"Normal width:
{normal_line_width}px", (10, 90),
                            cv2.FONT_HERSHEY_SIMPLEX, 0.6, (255, 255, 0), 2)

                # Mark wide area if detected
                if is_wide_area:
                    cv2.putText(debug_image, "WIDE AREA DETECTED", (10, 120),
                                cv2.FONT_HERSHEY_SIMPLEX, 0.6, (0, 0, 255), 2)

                    # Only check for target color in drop zones if we haven't
handled a drop yet
                    if not drop_handled:
                        color_found = False

                        for color, mask in zip(["red", "green", "blue"],
[mask_red, mask_green, mask_blue]):
                            color_contours, _ = cv2.findContours(mask,
cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)

                            for c in color_contours:
                                if cv2.contourArea(c) > 150:  # Must be
significant color area
                                    # Draw the color contour
                                    cv2.drawContours(debug_image[crop_top:crop
_bottom], [c], 0, (0, 255, 255), 2)

                                    if color == target_color:
                                        cv2.putText(debug_image, f"TARGET
COLOR MATCH: {color.upper()}", (10, 150),
                                                    cv2.FONT_HERSHEY_SIMPLEX,
0.6, (0, 255, 0), 2)
                                        print(f"\n🎯 Found {color} area
(Target color match!)")
                                        handle_drop_zone()
                                        color_found = True
                                        break
                                    else:
                                        cv2.putText(debug_image, f"Found
{color} area (not target)", (10, 150),
```

```
                                                    cv2.FONT_HERSHEY_SIMPLEX,
0.6, (0, 0, 255), 2)

                            if color_found:
                                break

                # Add target color indicator
                if target_color:
                    cv2.putText(debug_image, f"Target:
{target_color.upper()}", (width - 200, 30),
                                cv2.FONT_HERSHEY_SIMPLEX, 0.7, (0, 255, 255),
2)

                # Add drop status indicator
                if drop_handled:
                    cv2.putText(debug_image, "DROP COMPLETED", (width - 200,
60),
                                cv2.FONT_HERSHEY_SIMPLEX, 0.7, (0, 255, 0), 2)

                return error, debug_image, True

    # No line found - use your working sample's approach
    cv2.putText(debug_image, "NO LINE DETECTED", (width // 2 - 100, height //
2),
                cv2.FONT_HERSHEY_SIMPLEX, 0.7, (0, 0, 255), 2)
    return 100, debug_image, False  # Return value of 100 just like your
working sample

# ========== PID Control ==========
def pid_control(error, found_line):
    global last_error, integral, stop_motors_flag

    if stop_motors_flag:
        stop_motors()
        return

    if not found_line:
        # Search pattern if line is lost - EXACTLY like your working sample
        print("SEARCH PATTERN - Line Lost")  # Additional debug info
        set_motor("forward", "backward", search_speed, search_speed)
        print("Searching for line...")
        return

    # PID calculation - EXACTLY like your working sample
    proportional = error
    integral += error
    derivative = error - last_error
```

```
    # Limit integral windup
    integral = max(min(integral, 500), -500)
    last_error = error

    # Calculate adjustment
    adjustment = Kp * proportional + Ki * integral + Kd * derivative

    # SWAPPED: Left and right speeds are now swapped
    # Since motors are physically swapped, we need to apply adjustments
correctly
    left_speed = base_speed + adjustment / 100  # Now adding adjustment
    right_speed = base_speed - adjustment / 100  # Now subtracting adjustment

    # Limit speeds - EXACTLY like your working sample
    left_speed = max(min(left_speed, max_speed), -max_speed)
    right_speed = max(min(right_speed, max_speed), -max_speed)

    # Determine directions
    left_dir = "forward" if left_speed >= 0 else "backward"
    right_dir = "forward" if right_speed >= 0 else "backward"

    # Print speeds for debugging
    print(f"MOTOR CONTROL: Left={left_dir}({left_speed:.2f}),
Right={right_dir}({right_speed:.2f}), Error={error}")

    # Set motors
    set_motor(left_dir, right_dir, abs(left_speed), abs(right_speed))

# ========== Simple Test Motor Function ==========
def test_motors():
    print("Testing motors...")
    # Test both motors forward
    print("Both motors forward...")
    set_motor("forward", "forward", 0.3, 0.3)
    time.sleep(2)

    # Stop
    stop_motors()
    time.sleep(1)

    # Test left turn
    print("Testing left turn...")
    set_motor("backward", "forward", 0.3, 0.3)
    time.sleep(1)

    # Stop
    stop_motors()
    time.sleep(1)
```

```
    # Test right turn
    print("Testing right turn...")
    set_motor("forward", "backward", 0.3, 0.3)
    time.sleep(1)

    # Stop
    stop_motors()
    print("Motor test complete")

# ========== MAIN ==========
if __name__ == "__main__":
    # Setup the camera with the same parameters as your working sample
    camera = Picamera2()
    camera_config = camera.create_preview_configuration(main={"size": (800,
480)})
    camera.configure(camera_config)
    camera.start()

    print("Initializing camera...")
    time.sleep(2)

    # Use the same window name and size as your working sample
    cv2.namedWindow("Robot Vision", cv2.WINDOW_NORMAL)
    cv2.resizeWindow("Robot Vision", 800, 600)

    print("\n🔍 Place the colored object in front of the camera...")
    print("Press 't' to test motors, 'q' to quit, 'p' to pause/resume, 'r' to
reset")

    try:
        while True:
            # Capture and process frame - FLIP the image just like your
working sample
            frame = cv2.flip(camera.capture_array(), 1)
            frame = cv2.cvtColor(frame, cv2.COLOR_RGBA2BGR)

            # First phase: Color detection
            if not color_detection_complete:
                frame = detect_object_color(frame)
                cv2.imshow("Robot Vision", frame)
                key = cv2.waitKey(1) & 0xFF

                # Handle key presses
                if key == ord("q"):
                    break
                elif key == ord("t"):
                    test_motors()
```

```
                elif key == ord("r"):  # Reset
                    target_color = None
                    waiting_for_object = True
                    color_detection_complete = False
                    print("\n🔄 Reset color detection. Place new object.")

                # Skip the rest of the loop until color detection is done
                continue

            # Second phase: Line following and task execution

            # Check for triangle direction if not already processing one
            if not processing_triangle:
                triangle_direction = detect_triangle(frame)
                if triangle_direction:
                    # Stop motors before handling triangle
                    stop_motors()
                    time.sleep(0.3)
                    handle_triangle_direction(triangle_direction)

            # Process image and get line position error
            error, debug_frame, found_line = process_image(frame)

            # Apply PID control if not processing a triangle
            if not processing_triangle:
                pid_control(error, found_line)

            # Add instructions like your working sample
            cv2.putText(debug_frame, "Press 'q' to quit, 'p' to pause/resume,
'r' to reset, 't' to test motors",
                        (10, 450), cv2.FONT_HERSHEY_SIMPLEX, 0.5, (255, 255,
255), 1)

            # Display the frame
            cv2.imshow("Robot Vision", debug_frame)

            # Check for key presses - just like your working sample
            key = cv2.waitKey(1) & 0xFF
            if key == ord("q"):
                break
            elif key == ord("t"):
                test_motors()
            elif key == ord("p"):  # Pause/Resume - from your working sample
                stop_motors_flag = not stop_motors_flag
                print("Motors paused." if stop_motors_flag else "Motors
resumed.")
            elif key == ord("r"):  # Reset
                print("\n🔄 Resetting robot...")
```

```
                stop_motors()
                target_color = None
                normal_line_width = 0
                next_turn = None
                detected_triangles = []
                at_intersection = False
                drop_handled = False
                processing_triangle = False
                color_detection_complete = False
                waiting_for_object = True
                last_error = 0
                integral = 0
                print("\n🔍 Place the colored object in front of the
camera...")

    except KeyboardInterrupt:
        print("\n🛑 Stopped by user.")
    except Exception as e:
        print(f"\n❌ Error occurred: {e}")
        import traceback
        traceback.print_exc()

    finally:
        # Clean up resources
        stop_motors()
        camera.stop()
        cv2.destroyAllWindows()
        print("\n📄 Program Ended.")
        print(f"Final data: Target color = {target_color}, Triangles detected
= {len(detected_triangles)}")
```