جامعة نيويورك أبوظبي

NYU ABU DHABI

Embedded Systems
ENGR – UH 3530

Final Project Report

Fall, 2024

This report is entirely our own work, and we have kept a copy for our own records. We are aware of the University's policies on cheating, plagiarism, and the resulting consequences of their breach.

Submitted by:

Name	**Net ID**	**Signature**
Nasheed Ur Rehman	nur208	
Omar Rayyan	olr7742	
Firas Darwish	fbd2014	
Mahmoud Hafez	mah9935	

Abstract

This project focuses on the design and development of an autonomous line-following robot in compliance with the RoboCupJunior Rescue Line rules. Using the NVIDIA JetBot platform and OpenCV, we developed a vision-based control system to navigate modular tiles featuring diverse challenges such as ramps, intersections, and speed bumps. The robot's task was to follow a black line on a white floor while dynamically adapting to obstacles like gaps, debris, and changes in elevation.

Our implementation relies on classical computer vision techniques for robust line detection and navigation. OpenCV was used to preprocess camera inputs, identify the line's centroid, and compute steering corrections. Additional algorithms detected intersections using green markers and triggered predefined actions like turning or U-turns. We also implemented obstacle detection and recovery mechanisms to handle unexpected disruptions, ensuring continuous operation.

This project emphasizes real-world adaptability by adhering to competition constraints, such as unpredictable field layouts and environmental variations. By relying on efficient algorithms and OpenCV-based processing, we demonstrated that classical vision techniques could effectively solve autonomous navigation challenges in a competitive setting.

Contents

1 Introduction

Line-following robots are a fundamental application of robotics, where the robot autonomously navigates a marked path, typically a black line on a white background. These robots rely on sensors or cameras to detect the line and make real-time adjustments to stay on course. While techniques like proportional–integral–derivative (PID) control are often used for smooth and precise steering, simpler control methods, such as bang-bang control, offer a straightforward alternative for line-following tasks.

Computer vision enhances the robot's perception capabilities by interpreting visual data for decision-making. OpenCV, a widely-used open-source computer vision library, provides essential tools for processing images, including thresholding, contour detection, and color segmentation. These tools enable the robot to detect the line, identify navigation markers, and react to environmental features.

This project focuses on applying classical computer vision techniques and bang-bang control to solve the line-following problem. By leveraging OpenCV, we designed a system capable of detecting the line, identifying green intersection markers, and dynamically adjusting the robot's steering. While bang-bang control sacrifices the smoothness of advanced control techniques, it simplifies implementation and demonstrates the effectiveness of basic strategies in handling real-world challenges, including obstacles, gaps, and abrupt course changes.

This introduction outlines the foundation for a robust, vision-based, autonomous navigation system using OpenCV and highlights the simplicity and practicality of bang-bang control in addressing the requirements of the RoboCupJunior Rescue Line challenge.

2 Assumed Rules for the Line-Following Robot

2.1 Navigation Rules

- The robot must autonomously follow a continuous black line laid out on a white floor. The path may include varying terrains such as ramps, areas with uneven elevation, and sections with steps or gaps between tiles. These variations test the robot's ability to adapt to changing surface conditions.

- Discontinuities in the black line must be handled effectively. The robot should infer the line's continuation even when sections of it are missing or obscured. Additionally, the robot should smoothly navigate tight curves and intersections, relying on green visual markers to determine the appropriate direction to follow.

- The robot must stay entirely on the tiles during navigation. It should actively avoid veering off the edge of the field while simultaneously detecting and maneuvering around obstacles such as debris, speed bumps, and stationary objects, ensuring it does not lose track of the black line.

2.2 Environmental Rules

- The competition field is composed of modular tiles that may feature a variety of textures, including smooth, rough, or uneven surfaces. The robot must maintain consistent performance despite these variations.

- Lighting conditions across the field may change due to ambient light sources, shadows, or reflections. The robot should adapt dynamically to these variations to maintain accurate detection of the black line and markers.

- Small gaps or steps between tiles may create mechanical challenges for the robot, while magnetic interference or other environmental noise may impact sensor readings. The robot must be robust against these disturbances.

- The course may include various types of obstacles, ranging from small debris to larger fixed objects. The robot must be able to detect, classify, and either navigate around or adjust its behavior to safely interact with these obstacles.

2.3 Robot Control Rules

- The robot must function autonomously, with no human intervention allowed once a run begins. All decisions regarding navigation, obstacle avoidance, and course adaptation must be made by the robot in real time.

- The field configuration is randomized before each run, requiring the robot to dynamically adapt to new layouts. Pre-programmed solutions specific to known layouts are not allowed, emphasizing the robot's capacity for general problem-solving.

- A safety mechanism, such as an easily accessible kill switch, must be implemented. This feature ensures the robot can be stopped immediately if it encounters an issue or behaves in an unsafe manner.

3 Challenges and Implementation Goals

3.1 Potential Challenges with OpenCV

- **Line Detection under Varying Lighting:** The robot must reliably detect the black line even under changing lighting conditions, such as shadows, glare, or uneven illumination. These variations can reduce the contrast between the line and the white background, making detection more difficult.

- **Gaps and Discontinuities:** Discontinuities in the black line, such as gaps or interruptions, require the robot to predict the line's continuation. OpenCV-based methods may struggle to bridge these gaps, especially if they are long or poorly lit.

- **Handling Curves and Sharp Turns:** Navigating sharp curves and abrupt directional changes requires precise detection and control. Traditional computer vision techniques may have difficulty accurately estimating the path in such scenarios, leading to delays or errors in navigation.

- **Obstacle Recognition:** Detecting and avoiding obstacles while maintaining focus on the black line is a significant challenge. While OpenCV provides tools for detecting contours and shapes, integrating robust obstacle avoidance into a line-following framework requires careful design and optimization.

3.2 Implementation Goal

The primary goal of this project is to develop a robust and efficient line-following robot that leverages OpenCV for computer vision and bang-bang control for navigation. The system should operate in near real-time, providing accurate line detection and navigation even under challenging conditions.

The implementation focuses on addressing the following objectives:

- Ensuring reliable line detection and tracking despite variations in lighting, surface textures, and environmental noise.
- Developing a robust approach to handling discontinuities, sharp curves, and intersections using green markers as directional cues.
- Incorporating efficient algorithms for detecting and navigating around obstacles without deviating from the intended path.
- Achieving a balance between simplicity and performance, with OpenCV's lightweight processing enabling real-time operation on the NVIDIA JetBot platform.

The final implementation will demonstrate the robot's ability to adapt dynamically to randomized field layouts and complete the course autonomously while meeting the competition's requirements.

4 Our Approach: OpenCV with Bang-Bang Control

4.1 Overview

Our line-following solution leverages the NVIDIA JetBot platform, a single camera for vision input, and OpenCV-based processing for robust navigation on a modular field. We employ a bang-bang control scheme (i.e., simple on-off steering decisions) to keep the robot on course. The following sections detail each major component of the approach.

4.2 Regions of Interest and Thresholding

To reduce computational overhead and focus only on the critical portion of the image, we define regions of interest (ROI) for line detection and marker identification:

- **Line ROI:** The lower half of the frame (approximately 50%–100% vertically) is thresholded to isolate the black line against a white background.
- **Turn ROI:** A slightly higher region (about 60%–100% vertically) is separately processed to detect lower, extended parts of the line that help anticipate upcoming turns.
- **Green Marker ROI:** The bottom 20% of the frame is processed in HSV space to detect intersection markers (green squares) placed near the line.

After converting each ROI to grayscale and applying Gaussian blur, a binary inverse threshold is used, turning the black line into white contours for easier contour detection:

$$\text{thresh} = \begin{cases} 255 & \text{if blurredPixelValue} < \text{threshold} \\ 0 & \text{otherwise} \end{cases}$$

4.3 Weighted Contour Detection and Decaying Exponential

Within the ROI containing the line, we identify the largest contour and then apply a weighted centroid calculation using a *decaying exponential matrix*. This weighting biases contour pixels closer to the bottom center of the image, reflecting their importance for immediate navigation.

Let (y, x) be pixel coordinates within the ROI, where x ranges from 0 to $W_{\text{roi}} - 1$ and y ranges from 0 to $H_{\text{roi}} - 1$. If x_c is the ROI's horizontal midpoint, the final weighting array W_{xy} is defined as:

$$W_{xy} = \exp(-a\, y) \cdot \exp\big(-b\,|x - x_c|\big),$$

where a and b are positive constants controlling vertical and horizontal decay rates, respectively. Once we obtain a binary mask of the largest contour, we compute the weighted moments:

$$M_{00} = \sum_{x,y} W_{xy}{\cdot}\text{mask}(x, y), \quad M_{10} = \sum_{x,y} x{\cdot}W_{xy}{\cdot}\text{mask}(x, y), \quad M_{01} = \sum_{x,y} y{\cdot}W_{xy}{\cdot}\text{mask}(x, y).$$

From these weighted moments, the centroid $(\tilde{c}_x, \tilde{c}_y)$ is

$$\tilde{c}_x = \frac{M_{10}}{M_{00}}, \quad \tilde{c}_y = \frac{M_{01}}{M_{00}}.$$

We then translate $\tilde{c}_x, \tilde{c}_y$ back to the original image frame. This approach better anchors the contour detection, especially in challenging conditions like partial gaps or intersections.

4.4 Bang-Bang Control Logic

Bang-bang control, a simple and efficient approach for on-off control, forms the foundation of our robot's navigation system. Using the centroid of the detected line, we calculate the *error* relative to the center of the camera frame, and this error determines the steering direction and motor actions.

Error Calculation

The error is calculated as the horizontal displacement of the line's centroid (c_x) from the center of the frame. For a frame of width W, the error is given by:

$$\text{error} = c_x - \frac{W}{2}.$$

This value is positive if the centroid is to the right of the center and negative if to the left.

Control Logic

A threshold parameter THRESHOLD is used to classify the magnitude of the error into three discrete regions, which correspond to the robot's movement commands:

$$\begin{cases} \text{Go Straight:} & \text{if } |\,\text{error}\,| < \text{THRESHOLD} \\ \text{Turn Left:} & \text{if error} < -\text{THRESHOLD} \\ \text{Turn Right:} & \text{if error} > \text{THRESHOLD}. \end{cases}$$

Motor Speed Assignment

The robot's motors are controlled based on the error magnitude and sign:

- **Go Straight:** When the line is approximately centered ($|\,\text{error}\,| < \text{THRESHOLD}$), both motors are set to the base speed SPEED. This ensures forward motion along the detected line:

$$\text{Left\_motor\_speed} = \text{Right\_motor\_speed} = \text{SPEED}.$$

- **Turn Left:** If the error is negative and exceeds the threshold ($\text{error} < -\text{THRESHOLD}$), the line is to the left of the robot. To correct this, the left motor is stopped or slowed, while the right motor continues at the base speed:

$$\text{Left\_motor\_speed} = 0, \quad \text{Right\_motor\_speed} = \text{SPEED}.$$

- **Turn Right:** If the error is positive and exceeds the threshold ($\text{error} > \text{THRESHOLD}$), the line is to the right of the robot. To correct this, the right motor is stopped or slowed, while the left motor continues at the base speed:

$$\text{Left\_motor\_speed} = \text{SPEED}, \quad \text{Right\_motor\_speed} = 0.$$

Incorporating Smooth Transitions with Deceleration

To prevent abrupt halts, a brief forward movement precedes each turn in the functions `turn_left()` and `turn_right()`. This reduces the likelihood of overshooting during sharp directional changes. The sequence is as follows:

1. A small forward motion to stabilize the robot:

$$\text{robot.forward(speed)}.$$

2. Gradual steering by reducing the speed of one motor:

$$\text{robot.left(speed)} \quad \text{or} \quad \text{robot.right(speed)}.$$

3. A short stop to ensure accurate re-centering of the line:

$$\text{robot.stop()}.$$

Error Dynamics and Response Time

The robot's response time depends on the magnitude of the error. Larger errors lead to faster adjustments, as the robot aggressively corrects misalignments by stopping one motor. While this ensures quick course correction, it may also introduce oscillations (zigzag motion) in scenarios where the line is nearly centered or under slight deviations. These oscillations are inherent to bang-bang control but are mitigated by carefully tuning THRESHOLD and SPEED.

Challenges and Observations

While bang-bang control is computationally efficient and easy to implement, it introduces some trade-offs:

- **Oscillatory Behavior:** The robot tends to oscillate between left and right corrections when the error fluctuates near zero. This behavior is most pronounced during slight misalignments.
- **Abrupt Turns:** Sharp transitions between states can result in jerky movements, especially on tight curves or uneven surfaces.
- **Energy Inefficiency:** Frequent motor state changes increase energy consumption, particularly during prolonged runs.

Despite these limitations, our implementation successfully balances simplicity and reliability, demonstrating robust performance under real-time constraints.

4.5 Green Marker Detection for Turns and U-Turns

Green markers play a critical role in directing the robot at intersections and dead ends. Our implementation relies on detecting these markers using HSV-based color segmentation and geometric contour analysis. Based on the spatial configuration of the markers, the robot decides whether to perform a left turn, right turn, or U-turn.

HSV-Based Green Marker Detection

To detect green markers, the robot processes a specific region of interest (ROI) in the bottom portion of the frame, corresponding to the expected location of markers near the black line. The detection pipeline is as follows:

1. Convert the ROI from RGB to HSV (Hue, Saturation, Value) color space, as HSV provides better discrimination for color-based segmentation under varying lighting conditions.
2. Apply predefined HSV thresholds for green:

   $$\text{lower\_green} = [62, 55, 60], \quad \text{upper\_green} = [107, 255, 255].$$

   Pixels within this range are retained in a binary mask:

   $$\text{mask}(x, y) = \begin{cases} 255 & \text{if pixel } (x, y) \text{ is within green range,} \\ 0 & \text{otherwise.} \end{cases}$$

3. Identify contours in the binary mask, representing green regions. Only contours exceeding a minimum area (`GREEN_AREA_THRESHOLD`) are considered valid to filter out noise.

Marker Classification and Action Decisions

Based on the spatial arrangement of detected green markers, the robot classifies the intersection type and selects the appropriate maneuver.

1. Markers on Both Sides: U-Turn Detection When green markers are detected on both the left and right side of the line, the robot interprets this configuration as a dead end, requiring a U-turn. To ensure accurate detection, the robot evaluates the vertical position of the markers and the proximity to the black line. If the markers are sufficiently close (below the `APPROACH_Y_THRESHOLD`), the robot executes a U-turn:

if (left_marker_detected $\wedge$ right_marker_detected) and $y_{\text{marker}} <$ `APPROACH_Y_THRESHOLD` :

execute `u_turn()`.

The U-turn logic involves the following sequence:

1. Slow down the robot to enhance precision during the turn.
2. Pivot the robot by running one motor in reverse while the other moves forward.
3. Stop briefly to allow re-centering on the black line after completing the turn.

2. Marker on One Side: Left or Right Turn A single marker on either side of the line indicates a left or right turn. The robot decides the direction based on the marker's horizontal position relative to the frame center:

if left_marker_detected: execute `turn_left()`.

if right_marker_detected: execute `turn_right()`.

Similar to the U-turn logic, the robot slows down before initiating the turn and verifies the marker's vertical position to avoid premature actions.

False Positive Reduction Using Spatial Constraints

To improve robustness and reduce false positives:

- **Vertical Position Validation:** Detected markers are considered valid only if their vertical position (y_{marker}) lies below the `APPROACH_Y_THRESHOLD`. This ensures that distant markers do not trigger actions prematurely.
- **Cross-Referencing with Line Position:** The algorithm checks whether the green marker is adjacent to or overlaps with the detected black line. If the black line remains visible beneath the marker, the robot avoids turning unnecessarily and continues forward.
- **Area Thresholding:** Only markers with an area exceeding `GREEN_AREA_THRESHOLD` are considered valid. This step filters out noise or reflections misclassified as markers.

Motion Control During Marker Detection

When approaching green markers, the robot reduces its speed (`SLOW_SPEED`) to enhance detection stability and ensure precise execution of maneuvers:

Left_motor_speed = Right_motor_speed = `SLOW_SPEED`.

This deceleration improves the robot's ability to detect markers accurately and reduces overshooting during turns.

Summary of Marker Detection and Actions

The robot's green marker detection logic ensures reliable navigation through intersections and dead ends. By combining HSV-based segmentation with geometric constraints, the system effectively differentiates between turning scenarios:

- **Left Turn:** Triggered by a green marker on the left side of the line.
- **Right Turn:** Triggered by a green marker on the right side of the line.
- **U-Turn:** Triggered by green markers on both sides of the line.

These actions allow the robot to autonomously navigate complex field layouts, including multi-branch intersections and abrupt dead ends.

4.6 Red Line Detection for Permanent Stop

The robot uses red lines as a signal to permanently stop movement, indicating the end of the course or a safety boundary. Detection relies on HSV-based color segmentation, contour analysis, and orientation calculations to determine if a detected line is sufficiently horizontal. Once a valid red line is identified, the robot sets a `permanent_stop` flag, halting all actions.

HSV-Based Red Line Segmentation

To detect red lines, the robot processes the entire frame using predefined HSV ranges. This approach provides robustness to varying lighting conditions:

$$\text{lower\_red1} = [0, 120, 70], \quad \text{upper\_red1} = [10, 255, 255],$$

$$\text{lower\_red2} = [170, 120, 70], \quad \text{upper\_red2} = [179, 255, 255].$$

Two separate masks are generated for these ranges, capturing the dual hues of red (near 0° and 180° in the HSV color wheel):

$$\text{mask\_red} = \text{mask\_red1} \vee \text{mask\_red2}.$$

Contours are extracted from the binary mask to identify potential red regions in the frame.

Contour Filtering and Area Thresholding

To reduce noise, only contours with an area larger than `RED_AREA_THRESHOLD` are considered. This ensures that small red artifacts, such as noise or reflections, are ignored:

$$\text{if contour\_area} > \texttt{RED\_AREA\_THRESHOLD}.$$

Determining Horizontal Orientation

For each valid red contour, the robot computes its orientation using the minimum area bounding rectangle. This rectangle encloses the contour with the smallest possible area, providing its width w, height h, center (c_x, c_y), and rotation angle θ. The orientation is analyzed as follows:

1. **Calculate the aspect ratio:** The aspect ratio (AR) is the ratio of the longer side to the shorter side of the bounding rectangle:
$$AR = \frac{\max(w, h)}{\min(w, h)}.$$
A high aspect ratio (e.g., $AR > 5$) indicates that the contour is elongated, resembling a line.

2. **Check the deviation angle:** The angle θ (in degrees) of the bounding rectangle's major axis relative to the horizontal is obtained from OpenCV's `cv2.minAreaRect()` function. A horizontal line is identified if:
$$|\theta| < \theta_{\max} \quad \text{or} \quad |\theta - 90^\circ| < \theta_{\max},$$
where $\theta_{\max}$ is the allowable deviation angle (e.g., 10°). This accounts for minor variations in the alignment of the red line.

3. **Final validation:** A contour is considered a valid horizontal red line if both the aspect ratio and deviation angle criteria are satisfied:
$$\text{if } AR > 5 \text{ and } |\theta| < \theta_{\max} \quad \text{or} \quad |\theta - 90^\circ| < \theta_{\max}.$$

Permanent Stop Logic

If a valid horizontal red line is detected, the robot sets the `permanent_stop` flag. This immediately halts all motor actions:

$$\text{if valid\_red\_line: } \texttt{permanent\_stop = True}.$$

Once this flag is set, the robot stops moving and disregards further navigation commands.

Summary of Red Line Detection

The red line detection process ensures precise identification of horizontal stop signals using a combination of color segmentation, contour analysis, and orientation checks. Key steps include:

- **Color-Based Segmentation:** Dual HSV thresholds capture the full range of red hues.
- **Contour Filtering:** Area and aspect ratio constraints eliminate irrelevant contours.
- **Orientation Validation:** Deviation angles and aspect ratios ensure only horizontal lines are considered valid.

This approach minimizes false positives and ensures that the robot reliably stops at designated endpoints or boundaries, enhancing safety and compliance with competition rules.

4.7 Obstacle Detection

Obstacle detection ensures that the robot can identify and respond to unexpected blockages in its path. This feature leverages a dedicated *obstacle region of interest (ROI)* positioned above the line-following ROI to focus on potential obstacles rather than floor features.

Detection Logic

The detection process involves the following steps:

1. **Region of Interest:** The ROI for obstacle detection spans the upper portion of the thresholded image, above the ROI used for line following. This placement ensures that the robot identifies obstacles before encountering them.

2. **Binary Masking:** The same binary thresholding applied during line detection is reused to simplify the processing pipeline. Dark objects within the ROI appear as white regions in the binary mask.

3. **Contour Extraction:** Contours are extracted from the binary mask. Each contour represents a potential obstacle.

4. **Size Filtering:** To eliminate noise, only contours with an area exceeding the `OBSTACLE_AREA_THRESHOLD` are considered:

   $$\text{if contour\_area} > \texttt{OBSTACLE\_AREA\_THRESHOLD}.$$

   Larger contours are assumed to correspond to physical obstacles, such as debris or large objects.

Response to Detected Obstacles

When a valid obstacle is detected:

- The robot immediately halts by setting both motor speeds to zero:

  $$\text{Left\_motor\_speed} = \text{Right\_motor\_speed} = 0.$$

- The robot remains stationary until the obstacle is removed, which is determined by the absence of large contours in the ROI.

Summary of Obstacle Detection

The obstacle detection module ensures that the robot can dynamically adapt to unpredictable conditions on the field. Key features include:

- **Dedicated ROI:** Focuses on the region where obstacles are most likely to appear.
- **Contour Filtering:** Size-based filtering minimizes false positives.
- **Real-Time Response:** Halts the robot promptly upon detecting a blockage.

This functionality enhances the robot's autonomy and robustness, ensuring safe navigation even in cluttered or dynamic environments.

4.8 Stall Detection

Stall detection addresses situations where the robot fails to move despite receiving motor commands, such as when it is caught on debris or encounters mechanical resistance. By monitoring frame differences over time, the system identifies potential stalls and triggers recovery actions.

Detection Logic

The stall detection process involves the following steps:

1. **Frame Differencing:** Consecutive grayscale frames are compared using pixel-wise absolute differences:

$$\text{frame\_diff} = |\,\text{frame}(t) - \text{frame}(t-1)\,|.$$

2. **Cumulative Difference:** The sum of all pixel differences in the frame provides a scalar measure of motion:

$$\text{diff\_sum} = \sum_{x,y} \text{frame\_diff}(x, y).$$

3. **Threshold Check:** If the cumulative difference falls below `FRAME_DIFF_THRESHOLD` for `STALL_THRESHOLD` consecutive frames, the robot is assumed to be stalled.

Recovery Actions

When a stall is detected:

- The robot performs a brief forward burst by setting both motors to `STALL_SPEED` for a short duration (`STALL_DURATION`):

$$\text{Left\_motor\_speed} = \text{Right\_motor\_speed} = \texttt{STALL\_SPEED}.$$

- After the burst, the robot stops momentarily to reassess its environment before resuming normal operation.

Summary of Stall Detection

The stall detection module ensures continuous operation by enabling the robot to recover from temporary immobilization. Key features include:

- **Frame-Based Analysis:** Detects stalls by monitoring changes in consecutive frames.
- **Dynamic Recovery:** Executes a short forward burst to overcome mechanical resistance.
- **Efficiency:** Operates with minimal computational overhead, leveraging existing image processing data.

By integrating stall detection, the robot maintains its autonomy and resilience, ensuring reliable performance under diverse field conditions.

4.9 Summary

Our implementation successfully integrates traditional computer vision techniques with a bang-bang control strategy to achieve effective navigation on a modular field. The key features of the system include:

- **Robust Line Detection:** A decaying exponential weighting scheme enhances the reliability of centroid calculations, allowing the robot to handle gaps and intersections in the line.
- **Marker-Based Navigation:** Distinct HSV thresholds for green and red markers enable precise decisions for intersection turns and permanent stop signals.
- **Simplified Steering Control:** Bang-bang control provides a straightforward and efficient approach to real-time steering, with trade-offs in oscillatory behavior.
- **Obstacle and Stall Recovery:** Advanced techniques like frame differencing and contour analysis ensure the robot maintains autonomy in dynamic and unpredictable environments.

This cohesive design meets the RoboCupJunior-inspired requirements, enabling the robot to navigate autonomously and respond effectively to challenges in near real-time.

5 Control Logic Flow (Pseudo-Code)

```
Initialize Robot()
Initialize Camera(width=224, height=224)
Display camera widgets (side by side)

Initialize constants:
    MOVE_ROBOT = True
    SPEED = 0.11
    THRESHOLD = 11
    GREEN_AREA_THRESHOLD = 20
    SLOW_SPEED = 0.08
    APPROACH_Y_THRESHOLD = 80
    RED_AREA_THRESHOLD = 4000
    STALL_THRESHOLD = 50
    STALL_SPEED = 0.2
    STALL_DURATION = 0.12
    FRAME_DIFF_THRESHOLD = 100000
    OBSTACLE_AREA_THRESHOLD = 5000
    PAUSE_ON_RED = True

Define turn_left(), turn_right(), u_turn() to handle maneuvering.

Set permanent_stop = False
Set green_marker_detected = False
Set green_marker_side = None
Set obstacle_detected = False
Initialize stall_counter = 0
```

```
Initialize prev_frame = None

WHILE True:

    IF permanent_stop == True:
        - Capture camera frame for display only
        - Sleep briefly
        - CONTINUE (skip control logic)

    # Capture Frame
    current_frame = camera.value
    frame_rgb = Convert current_frame to RGB
    gray_frame = Convert frame_rgb to grayscale
    Show frame_rgb on image_widget

    # Stall Detection
    IF prev_frame is not None:
        frame_diff = absdiff(gray_frame, prev_frame)
        diff_sum = sum of all values in frame_diff
        IF diff_sum < FRAME_DIFF_THRESHOLD:
            stall_counter += 1
        ELSE:
            stall_counter = 0

        IF stall_counter >= STALL_THRESHOLD AND MOVE_ROBOT ==
           True AND obstacle_detected == False:
            Print("Stall detected. Executing burst of speed.")
            robot.forward(STALL_SPEED)
            Sleep(STALL_DURATION)
            robot.stop()
            stall_counter = 0
    ELSE:
        stall_counter = 0

    prev_frame = copy(gray_frame)

    # Line Detection (Weighted Centroid)
    blur = GaussianBlur(gray_frame)
    thresh = BinaryInverseThreshold(blur, threshold_value=60)
    height, width = shape of thresh
    roi_x, roi_width, roi_y, etc. computed based on constants

    Extract roi = thresh[roi_y : height, roi_x : (roi_x +
       roi_width)]
    Extract roi_turn = thresh[roi_y_turn : height, roi_x : (roi_x
        + roi_width)]

    Find contours in roi
    Find contours in roi_turn

    IF contours in roi_turn:
```

```
75          Compute centroid cx_turn, cy_turn using largest contour
76
77      IF contours in roi:
78          largest_contour = max contour by area
79          Compute weighted centroid using decaying exponentials:
80              weighted_mask = drawContour(largest_contour, 1) *
                    weight_array
81              (weighted_cx, weighted_cy) = moments(weighted_mask)
82          IF weighted_cx, weighted_cy valid:
83              cx = weighted_cx + roi_x
84              cy = weighted_cy + roi_y
85              error = cx - (width // 2)
86              IF not green_marker_detected AND not
                    obstacle_detected:
87                  IF abs(error) < THRESHOLD:
88                      left_motor_speed = SPEED
89                      right_motor_speed = SPEED
90                  ELSE IF error < 0:
91                      left_motor_speed = 0
92                      right_motor_speed = SPEED
93                  ELSE:
94                      left_motor_speed = SPEED
95                      right_motor_speed = 0
96          ELSE:
97              IF MOVE_ROBOT:
98                  robot.stop()
99              Update processed_widget
100             CONTINUE
101     ELSE:
102         IF MOVE_ROBOT:
103             robot.stop()
104         Update processed_widget
105         CONTINUE
106
107     # Green Marker Detection
108     green_roi = bottom portion of frame_rgb
109     hsv_green_roi = Convert(green_roi to HSV)
110     mask_green = inRange(hsv_green_roi, lower_green, upper_green)
111     Find contours_green in mask_green
112
113     green_left = False
114     green_right = False
115     marker_center_y = None
116
117     FOR each contour_g in contours_green:
118         IF contour_area > GREEN_AREA_THRESHOLD:
119             Identify bounding rect (x_g, y_g, w_g, h_g)
120             Compute center_x_g, center_y_g
121             # Additional checks to confirm approach
122
123             IF valid marker:
```

```
                Determine green_left or green_right
                Slow robot if obstacle_detected == False:
                    robot.left_motor.value = SLOW_SPEED
                    robot.right_motor.value = SLOW_SPEED

    # Handle intersection logic
    IF obstacle_detected == False:
        IF green_left AND green_right:
            # U-turn logic
            ...
        ELSE IF (green_left OR green_right):
            # Single side turn logic
            ...
        ELSE:
            # Normal line-follow if no marker
            ...
    # Update line follow speeds if no obstacle and no marker
       triggered

    # Red Line Detection
    hsv_frame = Convert frame_rgb to HSV
    mask_red = inRange(hsv_frame, red ranges)
    Find contours_red in mask_red

    red_line_detected = False
    FOR cnt_r in contours_red:
        IF area_r > RED_AREA_THRESHOLD:
            rect = minAreaRect(cnt_r)
            Evaluate angle, aspect_ratio
            IF horizontal_condition AND aspect_ratio > 5:
                red_line_detected = True
                BREAK

    IF PAUSE_ON_RED AND red_line_detected:
        Print("Red line detected. Pausing.")
        robot.stop()
        permanent_stop = True

    # Obstacle Detection
    obstacle_roi = thresh[0 : roi_y, 0 : width]
    contours_obstacle in obstacle_roi
    current_obstacle_detected = False
    FOR cnt_o in contours_obstacle:
        IF area_o > OBSTACLE_AREA_THRESHOLD:
            current_obstacle_detected = True
            BREAK

    IF current_obstacle_detected AND not permanent_stop:
        obstacle_detected = True
        IF MOVE_ROBOT:
            robot.stop()
```

```
    ELSE:
        IF obstacle_detected AND not permanent_stop:
            obstacle_detected = False

    Show processed_frame on processed_widget

END WHILE

# Cleanup
camera.stop()
IF MOVE_ROBOT:
    robot.stop()
```

Listing 1: Pseudo-code for our JetBot Control Logic

Explanation

This pseudo-code highlights the overall structure of our control loop:

- **Frame Acquisition and Visualization:** Continuously captures frames from the camera and displays both raw and processed images.
- **Stall Detection:** Compares consecutive frames to detect motion stalls. A short forward burst helps free the robot.
- **Line Detection:** Uses Gaussian blur, binary thresholding, and weighted centroid calculations for robust line tracking.
- **Green Marker Detection:** Identifies intersection markers and triggers turns or U-turns based on the marker's position.
- **Red Line Detection:** Searches for a sufficiently large, horizontally oriented red contour and permanently stops if found.
- **Obstacle Detection:** Halts the robot upon detecting large contours in a specified ROI, resuming movement only after the obstacle is removed.

This structured control flow emphasizes real-time responsiveness and autonomy in navigating a modular field under RoboCupJunior-inspired rules.

6 Control Logic Flow Diagram

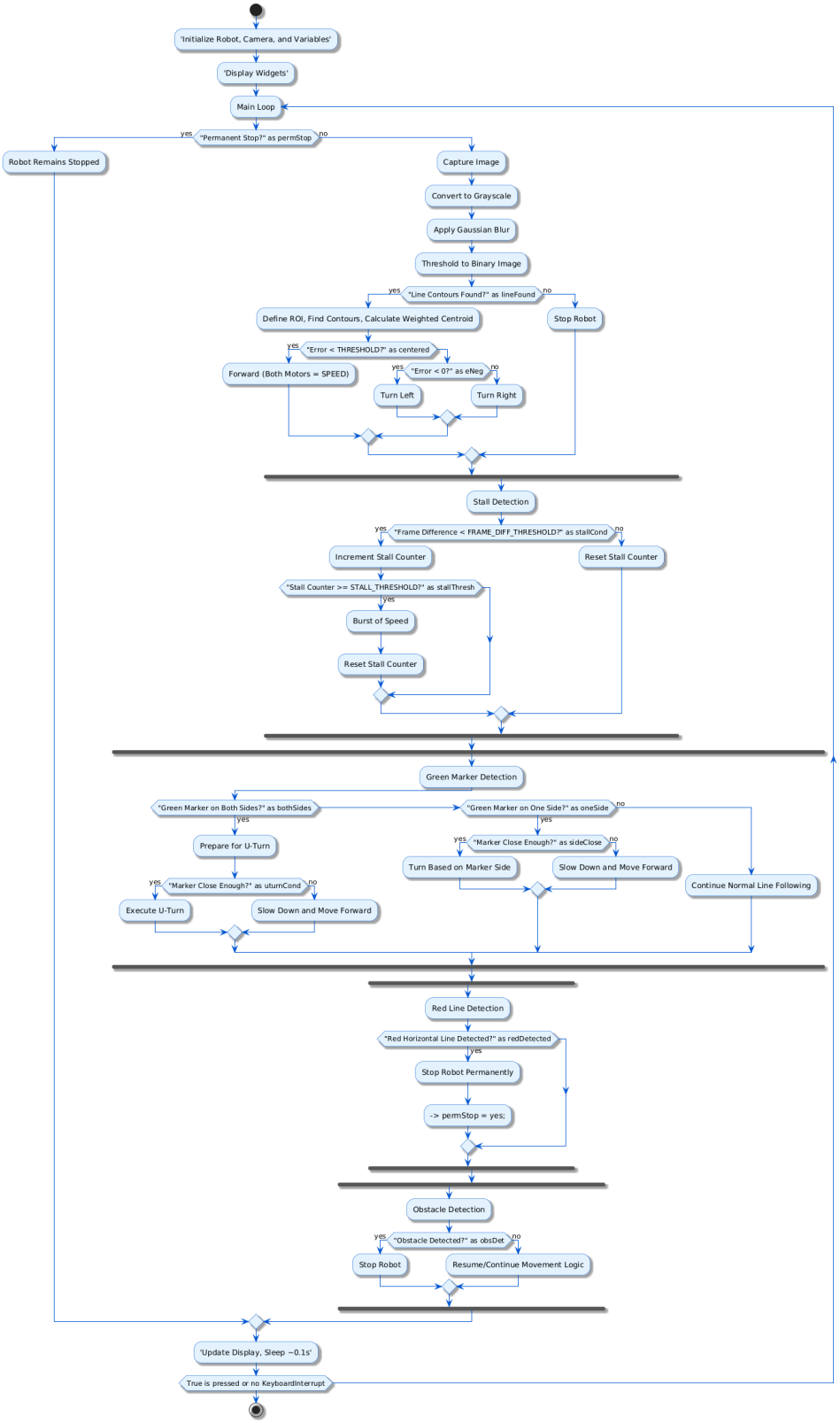

Figure 1: Control Logic Flow Diagram for the Line-Following Robot

7 Results & Discussion

The performance of the autonomous line-following robot was systematically evaluated on a modular test field designed to replicate the challenges of the RoboCupJunior Rescue Line competition. The results highlight the robot's ability to navigate straight paths, handle intersections and sharp turns, avoid obstacles, and detect course boundaries. This section provides an in-depth analysis of the robot's behavior and performance in these key scenarios. Figures 2, 3, and 4 provide visual demonstrations of these scenarios.

7.1 Line Following Performance

The robot demonstrated robust line-following capabilities under diverse field conditions, including varying lighting intensities, textured surfaces, and minor discontinuities in the line. The decaying exponential weighting scheme for centroid computation played a critical role in maintaining stability, ensuring that the robot could track the line even when parts of it were missing or obscured by shadows or reflections.

Notable observations include:

- **Handling Gaps in the Line:** The robot was able to infer the trajectory of the line through partial gaps, thanks to the weighting scheme that emphasized pixels closer to the bottom center of the image.
- **Adapting to Surface Variations:** The robot maintained reliable line detection and tracking despite changes in floor textures, such as transitions from smooth to rough tiles.
- **Performance in Dynamic Lighting:** The system exhibited resilience to moderate changes in lighting, such as shadows cast by moving objects or reflections on glossy surfaces.

Figure 2 shows the robot maintaining a steady trajectory along a straight black line, demonstrating the effectiveness of the vision-based approach.

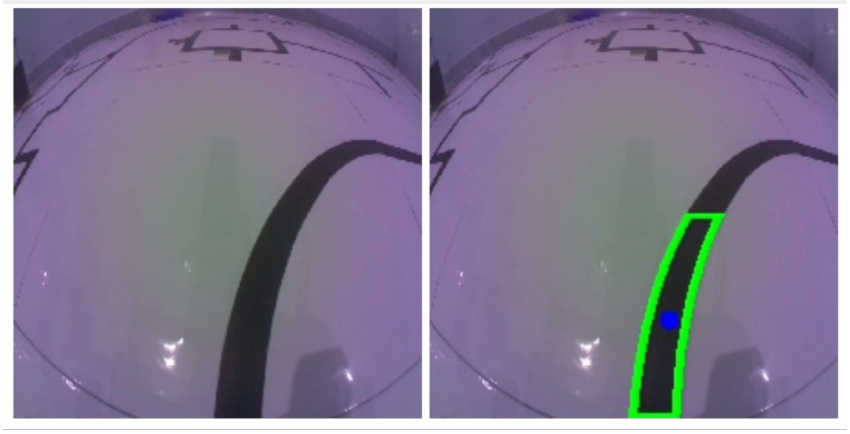

Figure 2: Robot following a straight black line on the test field.

7.2 Turn Handling

The robot's ability to navigate intersections and execute sharp turns was tested extensively using green markers placed strategically along the field. The bang-bang control

mechanism enabled the robot to make quick adjustments, ensuring it could realign with the line after completing each turn.

Detailed results include:

- **Left and Right Turns:** The robot accurately detected green markers on either side of the line and performed the corresponding turn. It reduced speed as it approached the marker, improving precision during directional changes.
- **U-Turns at Dead Ends:** When green markers were detected on both sides of the line, the robot correctly interpreted this as a dead end and executed a U-turn. The turning motion was smooth, with minimal overshooting or misalignment upon rejoining the line.
- **Error Recovery:** In cases where the robot initially misaligned during a turn, it quickly corrected its trajectory using the error-based steering logic.

Figures 3 and 4 illustrate the robot executing a left turn and a right turn, respectively, in response to detected green markers.

Figure 3: Robot detecting a green marker to the left of centroid to indicate left turn.

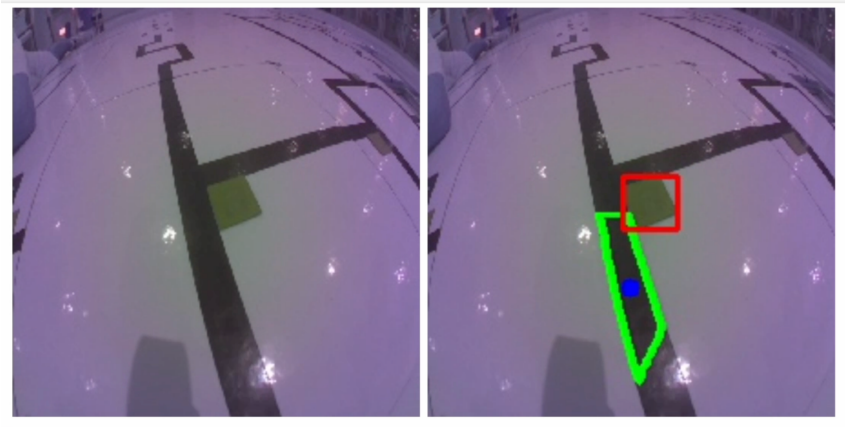

Figure 4: Robot detecting a green marker to the right of centroid to indicate right turn.

7.3 Obstacle Avoidance and Stall Recovery

The robot's obstacle detection and stall recovery mechanisms were tested by introducing various physical challenges on the course, such as large debris and mechanical resistance.

Key observations include:

- **Obstacle Detection:** The robot successfully identified large objects in its path using contour analysis within the designated obstacle ROI. Upon detection, it halted promptly, preventing collisions.
- **Resuming After Obstacle Removal:** The robot resumed movement as soon as the obstacle was removed from the field of view, demonstrating dynamic adaptability.
- **Stall Recovery:** The stall recovery mechanism was effective in scenarios where the robot encountered mechanical resistance, such as getting caught on uneven surfaces or debris. A short forward burst allowed the robot to free itself and continue operation.

These capabilities ensured that the robot could navigate cluttered environments without manual intervention, enhancing its autonomy and reliability.

7.4 Red Line Detection for Stop Signals

Red lines were used to signal the robot to halt permanently, marking the end of the course or a safety boundary. The HSV-based segmentation and contour orientation analysis ensured accurate and reliable detection of horizontal red lines.

Performance highlights include:

- **Precise Line Detection:** The robot consistently identified horizontal red lines, even under varying lighting conditions and surface textures.
- **Permanent Stop Mechanism:** Once a valid red line was detected, the robot halted all motor actions and ignored further navigation commands, ensuring compliance with the stop condition.
- **False Positive Avoidance:** By applying aspect ratio and angle constraints, the robot avoided misclassifying small or non-horizontal red objects as stop signals.

This functionality enhances the robot's ability to safely navigate and terminate operations as required.

7.5 Discussion

The results demonstrate the effectiveness of the robot's design in addressing the challenges of autonomous navigation on a modular field. The following points summarize the key findings:

- **Line Detection Robustness:** The decaying exponential weighting scheme significantly improved the reliability of line tracking, allowing the robot to handle partial gaps and challenging lighting conditions.

- **Marker-Based Navigation:** The combination of HSV-based segmentation and spatial validation enabled precise detection of green and red markers, ensuring accurate turn execution and course termination.
- **Obstacle and Stall Handling:** The robot's ability to dynamically detect and respond to obstacles, as well as recover from stalls, ensured uninterrupted operation under diverse conditions.
- **Control Simplicity and Trade-Offs:** The bang-bang control scheme provided a straightforward and computationally efficient solution but introduced oscillatory behavior during straight-line navigation. This trade-off is acceptable for scenarios requiring quick responses but could be optimized in future iterations.

Despite its limitations, the robot performed reliably across various scenarios, meeting the requirements of the RoboCupJunior-inspired challenge.

Future work could focus on the following improvements:

- Integrating a proportional–integral–derivative (PID) controller to reduce oscillations and improve smoothness during navigation.
- Enhancing the vision system with machine learning-based object detection for more sophisticated obstacle recognition and marker classification.
- Improving hardware robustness to handle extreme environmental conditions, such as highly reflective surfaces or severe surface irregularities.
- Extending the algorithm to handle more complex field layouts, such as multi-branch intersections or dynamic obstacles.

These advancements would further enhance the robot's adaptability and performance, enabling it to tackle more challenging autonomous navigation tasks in competitive and real-world applications.

8 Conclusion

The development and implementation of the autonomous line-following robot for the RoboCupJunior Rescue Line challenge demonstrate the effective integration of classical computer vision techniques with the NVIDIA JetBot platform. By leveraging OpenCV for line detection, marker identification, and obstacle avoidance, the project showcases how relatively simple algorithms can address complex navigation challenges in dynamic environments.

Key achievements include the successful handling of modular course layouts, dynamic adaptation to unpredictable obstacles, and robust responses to environmental variations such as changes in lighting and surface texture. The integration of bang-bang control simplified the system design, ensuring real-time responsiveness while maintaining alignment with competition requirements. Despite its limitations in terms of oscillatory behavior and abrupt turns, this control approach proved adequate for the task.

The project also highlights the practicality of using HSV-based segmentation for detecting green markers and red stop signals, enhancing the robot's ability to autonomously navigate intersections and recognize course boundaries. Additionally, the implementation

of stall detection and recovery mechanisms, combined with obstacle detection, significantly improved the robot's resilience and reliability under real-world conditions.

Future work could focus on further optimizations in contour weighting and marker detection to enhance the precision of navigation decisions. Expanding the vision system to include machine learning-based object detection could address some limitations of the current rule-based approach, allowing for more sophisticated handling of complex environments.

Overall, this project demonstrates a strong foundation in robotics, computer vision, and embedded systems, contributing valuable insights into the design of autonomous navigation systems for competitive and practical applications. The results validate the effectiveness of classical computer vision techniques in achieving real-time performance and adaptability within constrained computational environments.

9 Appendix

9.1 Python Code for Line-Following Robot

```
import cv2
import numpy as np
from jetbot import Robot, Camera, bgr8_to_jpeg
import ipywidgets as widgets
from IPython.display import display
import time

# - - - - - - - - - - - - - - - - - - - - - - -
# Configuration Flags
# - - - - - - - - - - - - - - - - - - - - - - -
MOVE_ROBOT = True   # Set to False if you want the robot to remain stationary

# - - - - - - - - - - - - - - - - - - - - - - -
# Initialize Hardware
# - - - - - - - - - - - - - - - - - - - - - - -
robot = Robot()
camera = Camera.instance(width=224, height=224)

# Display widgets: shows original camera image and processed image side-by-side
image_widget = widgets.Image(format='jpeg', width=224, height=224)
processed_widget = widgets.Image(format='jpeg', width=224, height=224)
display(widgets.HBox([image_widget, processed_widget]))

# - - - - - - - - - - - - - - - - - - - - - - -
# Line Following Parameters
# - - - - - - - - - - - - - - - - - - - - - - -
SPEED = 0.11          # Base forward speed
THRESHOLD = 11         # Threshold for line offset decisions

# - - - - - - - - - - - - - - - - - - - - - - -
# Green Marker Detection Parameters
# - - - - - - - - - - - - - - - - - - - - - - -
GREEN_AREA_THRESHOLD = 20    # Minimum area for valid green marker
SLOW_SPEED = 0.08            # Reduced speed near green markers
APPROACH_Y_THRESHOLD = 80    # Vertical threshold for action near green markers

# - - - - - - - - - - - - - - - - - - - - - - -
# Red Line Detection Parameters
# - - - - - - - - - - - - - - - - - - - - - - -
RED_AREA_THRESHOLD = 4000     # Area threshold to identify a red horizontal line
PAUSE_ON_RED = True           # If True, robot stops permanently when red line is
    detected

# HSV range for red
lower_red1 = np.array([0, 120, 70])
upper_red1 = np.array([10, 255, 255])
```

```
lower_red2 = np.array([170, 120, 70])
upper_red2 = np.array([179, 255, 255])

# - - - - - - - - - - - - - - - - - - - - - - -
# Stall Detection Parameters
# - - - - - - - - - - - - - - - - - - - - - - -
STALL_THRESHOLD = 50
STALL_SPEED = 0.2
STALL_DURATION = 0.12
FRAME_DIFF_THRESHOLD = 100000

# - - - - - - - - - - - - - - - - - - - - - - -
# Obstacle Detection Parameters
# - - - - - - - - - - - - - - - - - - - - - - -
# Obstacles can be any color or shape. We'll consider a large dark object appearing in front of the robot.
# We'll use the thresholded image to detect large unexpected contours in the region above the line detection area.
# If a large contour is detected there, we treat it as an obstacle.
OBSTACLE_AREA_THRESHOLD = 5000  # Adjust as needed to filter out noise. Larger means more certain obstacle.

# - - - - - - - - - - - - - - - - - - - - - - -
# Turn and Movement Functions
# - - - - - - - - - - - - - - - - - - - - - - -
def turn_left():
    robot.forward(0.1)
    time.sleep(0.8)
    robot.left(0.13)
    time.sleep(0.41)
    robot.forward(0.08)
    time.sleep(0.05)
    robot.stop()

def turn_right():
    robot.forward(0.1)
    time.sleep(0.8)
    robot.right(0.13)
    time.sleep(0.5)
    robot.forward(0.08)
    time.sleep(0.05)
    robot.stop()

def u_turn():
    robot.left(0.2)
    time.sleep(1.0)
    robot.stop()

# Variable to stop permanently once red line is detected
permanent_stop = False

try:
    green_marker_detected = False
    green_marker_side = None

    prev_frame = None
    stall_counter = 0

    # Variable to track obstacle presence
    obstacle_detected = False

    while True:
        # If permanently stopped due to red line, just show frames and do nothing else
        if permanent_stop:
            image = camera.value
            frame = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
            image_widget.value = bgr8_to_jpeg(frame)
            processed_widget.value = bgr8_to_jpeg(frame)
            time.sleep(0.1)
            continue

        # - - - - - - - - - - - - - - - - - - - - - - -
        # Capture Frame
```

```
        # - - - - - - - - - - - - - - - - - - - - - - - - -
        image = camera.value
        frame = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
        gray_frame = cv2.cvtColor(frame, cv2.COLOR_RGB2GRAY)
        image_widget.value = bgr8_to_jpeg(frame)

        # - - - - - - - - - - - - - - - - - - - - - - - - -
        # Stall Detection
        # - - - - - - - - - - - - - - - - - - - - - - - - -
        if prev_frame is not None:
            frame_diff = cv2.absdiff(gray_frame, prev_frame)
            diff_sum = np.sum(frame_diff)
            if diff_sum < FRAME_DIFF_THRESHOLD:
                stall_counter += 1
            else:
                stall_counter = 0
            if stall_counter >= STALL_THRESHOLD and MOVE_ROBOT and not
                obstacle_detected:
                print("Stall detected. Executing burst of speed.")
                robot.forward(STALL_SPEED)
                time.sleep(STALL_DURATION)
                robot.stop()
                stall_counter = 0
        else:
            stall_counter = 0

        prev_frame = gray_frame.copy()

        # - - - - - - - - - - - - - - - - - - - - - - - - -
        # Line Detection
        # - - - - - - - - - - - - - - - - - - - - - - - - -
        blur = cv2.GaussianBlur(gray_frame, (5, 5), 0)
        _, thresh = cv2.threshold(blur, 60, 255, cv2.THRESH_BINARY_INV)

        height, width = thresh.shape
        roi_y = int(height * 0.5)
        roi_y_turn = int(height * 0.6)
        ROI_WIDTH_PERCENTAGE = 0.9
        roi_width = int(width * ROI_WIDTH_PERCENTAGE)
        roi_x = (width - roi_width) // 2

        roi = thresh[roi_y:height, roi_x:roi_x+roi_width]
        roi_turn = thresh[roi_y_turn:height, roi_x:roi_x+roi_width]

        contours, _ = cv2.findContours(roi, cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)
        contours_turn, _ = cv2.findContours(roi_turn, cv2.RETR_TREE,
            cv2.CHAIN_APPROX_SIMPLE)

        cx = cy = None
        cx_turn = cy_turn = None
        processed_frame = frame.copy()

        # Check turn reference line in lower ROI
        if contours_turn:
            largest_contour_turn = max(contours_turn, key=cv2.contourArea)
            M_turn = cv2.moments(largest_contour_turn)
            if M_turn['m00'] != 0:
                cx_turn = int(M_turn['m10'] / M_turn['m00']) + roi_x
                cy_turn = int(M_turn['m01'] / M_turn['m00']) + roi_y

        # Find weighted centroid of the largest line contour in the main ROI
        if contours:
            largest_contour = max(contours, key=cv2.contourArea)
            roi_height, roi_width_actual = roi.shape
            center_x = roi_width_actual // 2

            a = 0.020
            b = 0.02
            y = np.arange(roi_height).reshape(-1, 1)
            x_coords = np.arange(roi_width_actual).reshape(1, -1)
            weight_y = np.exp(-a * y)
            weight_x = np.exp(-b * np.abs(x_coords - center_x))
            weight_array = weight_y * weight_x
```

```
187
188             mask = np.zeros_like(roi, dtype=np.float32)
189             cv2.drawContours(mask, [largest_contour], -1, 1, -1)
190             weighted_mask = mask * weight_array
191             M = cv2.moments(weighted_mask)
192             if M['m00'] != 0:
193                 weighted_cx = M['m10'] / M['m00']
194                 weighted_cy = M['m01'] / M['m00']
195
196                 cx = int(weighted_cx) + roi_x
197                 cy = int(weighted_cy) + roi_y
198
199                 cv2.drawContours(processed_frame[roi_y:height, roi_x:roi_x+roi_width],
                        [largest_contour], -1, (0, 255, 0), 3)
200                 cv2.circle(processed_frame, (cx, cy), 5, (255, 0, 0), -1)
201
202                 error = cx - (width // 2)
203                 # Only adjust speeds if no green marker being approached and no
                        obstacle
204                 if not green_marker_detected and not obstacle_detected:
205                     if abs(error) < THRESHOLD:
206                         left_motor_speed = SPEED
207                         right_motor_speed = SPEED
208                     elif error < 0:
209                         left_motor_speed = 0
210                         right_motor_speed = SPEED
211                     else:
212                         left_motor_speed = SPEED
213                         right_motor_speed = 0
214             else:
215                 if MOVE_ROBOT:
216                     robot.stop()
217                 processed_widget.value = bgr8_to_jpeg(processed_frame)
218                 continue
219         else:
220             if MOVE_ROBOT:
221                 robot.stop()
222             processed_widget.value = bgr8_to_jpeg(processed_frame)
223             continue
224
225         # - - - - - - - - - - - - - - - - - - - - - - - -
226         # Green Marker Detection
227         # - - - - - - - - - - - - - - - - - - - - - - - -
228         green_roi_y_start = int(height * 0.8)
229         green_roi = frame[green_roi_y_start:height, 0:width]
230
231         hsv_green_roi = cv2.cvtColor(green_roi, cv2.COLOR_RGB2HSV)
232         lower_green = np.array([62, 55, 60])
233         upper_green = np.array([107, 255, 255])
234         mask_green = cv2.inRange(hsv_green_roi, lower_green, upper_green)
235
236         contours_green, _ = cv2.findContours(mask_green, cv2.RETR_EXTERNAL,
                cv2.CHAIN_APPROX_SIMPLE)
237         green_left = False
238         green_right = False
239         marker_center_y = None
240         marker_top_y = None
241         marker_center_x = None
242
243         for contour_g in contours_green:
244             area_g = cv2.contourArea(contour_g)
245             if area_g > GREEN_AREA_THRESHOLD:
246                 x_g, y_g, w_g, h_g = cv2.boundingRect(contour_g)
247                 center_x_g = x_g + w_g // 2
248                 center_y_g = y_g + h_g // 2
249                 center_y_full = center_y_g + green_roi_y_start
250                 cv2.rectangle(processed_frame, (x_g, y_g + green_roi_y_start),
251                               (x_g + w_g, y_g + h_g + green_roi_y_start), (0, 0, 255),
                                    2)
252
253                 if marker_center_y is None or center_y_full > marker_center_y:
254                     marker_center_y = y_g + h_g + green_roi_y_start
255                     marker_top_y = y_g + green_roi_y_start
```

```
                    marker_center_x = center_x_g

                check_y = max(marker_top_y - 35, 0)
                check_x = int(marker_center_x)
                pixel_value = gray_frame[check_y, check_x]

                if pixel_value < 128:
                    if cx_turn is not None:
                        if center_x_g < cx_turn and x_g < cx_turn:
                            green_left = True
                        else:
                            green_right = True
                    else:
                        if center_x_g < (width // 2) and x_g < (width // 2):
                            green_left = True
                        else:
                            green_right = True

                    # Slow down near green markers (if no obstacle)
                    if MOVE_ROBOT and not obstacle_detected:
                        robot.left_motor.value = SLOW_SPEED
                        robot.right_motor.value = SLOW_SPEED

        # Handle actions based on detected green markers
        if not obstacle_detected:
            if green_left and green_right:
                if not green_marker_detected:
                    print("Green markers on both sides. Preparing for U-turn.")
                    green_marker_detected = True
                    green_marker_side = 'both'
                else:
                    if marker_center_y and marker_center_y > APPROACH_Y_THRESHOLD:
                        check_y = max(marker_top_y - 35, 0)
                        check_x = int(marker_center_x)
                        pixel_value = gray_frame[check_y, check_x]

                        if pixel_value < 128:
                            print("Performing U-turn.")
                            if MOVE_ROBOT:
                                u_turn()
                            green_marker_detected = False
                        else:
                            if MOVE_ROBOT:
                                robot.left_motor.value = SLOW_SPEED
                                robot.right_motor.value = SLOW_SPEED
                    else:
                        if MOVE_ROBOT:
                            robot.left_motor.value = SLOW_SPEED
                            robot.right_motor.value = SLOW_SPEED
            elif green_left or green_right:
                side = 'left' if green_left else 'right'
                if not green_marker_detected:
                    print(f"Green marker on {side}. Approaching.")
                    green_marker_detected = True
                    green_marker_side = side
                else:
                    if marker_center_y and marker_center_y > APPROACH_Y_THRESHOLD:
                        check_y = max(marker_top_y - 35, 0)
                        check_x = int(marker_center_x)
                        pixel_value = gray_frame[check_y, check_x]

                        if pixel_value < 128:
                            print(f"Turning to the {green_marker_side} at green
                                marker.")
                            if MOVE_ROBOT:
                                if green_marker_side == 'left':
                                    turn_left()
                                else:
                                    turn_right()
                            green_marker_detected = False
                        else:
                            if MOVE_ROBOT:
                                robot.left_motor.value = SLOW_SPEED
```

```
                                robot.right_motor.value = SLOW_SPEED
                    else:
                        if MOVE_ROBOT:
                            robot.left_motor.value = SLOW_SPEED
                            robot.right_motor.value = SLOW_SPEED
            else:
                # No green marker: normal line following if no obstacle
                green_marker_detected = False
                if MOVE_ROBOT and not obstacle_detected:
                    robot.left_motor.value = left_motor_speed
                    robot.right_motor.value = right_motor_speed

            # If not approaching a green marker and no obstacle, continue normal line
                follow
            if not green_marker_detected and MOVE_ROBOT and not obstacle_detected:
                robot.left_motor.value = left_motor_speed
                robot.right_motor.value = right_motor_speed

        # - - - - - - - - - - - - - - - - - - - - - - - -
        # Red Line Detection (Permanent Stop)
        # - - - - - - - - - - - - - - - - - - - - - - - -
        hsv_frame = cv2.cvtColor(frame, cv2.COLOR_RGB2HSV)
        mask_red1 = cv2.inRange(hsv_frame, lower_red1, upper_red1)
        mask_red2 = cv2.inRange(hsv_frame, lower_red2, upper_red2)
        mask_red = cv2.bitwise_or(mask_red1, mask_red2)

        contours_red, _ = cv2.findContours(mask_red, cv2.RETR_EXTERNAL,
            cv2.CHAIN_APPROX_SIMPLE)

        red_line_detected = False
        for cnt_r in contours_red:
            area_r = cv2.contourArea(cnt_r)
            if area_r > RED_AREA_THRESHOLD:
                rect = cv2.minAreaRect(cnt_r)
                (cx_r, cy_r), (w_r, h_r), angle_r = rect
                angle_r = abs(angle_r)
                horizontal_condition = (angle_r < 10 or abs(angle_r - 90) < 10)

                if h_r > 0:
                    aspect_ratio = w_r / h_r if w_r > h_r else h_r / w_r
                else:
                    aspect_ratio = 0

                if horizontal_condition and aspect_ratio > 5:
                    red_line_detected = True
                    box_r = cv2.boxPoints(rect)
                    box_r = np.int0(box_r)
                    cv2.drawContours(processed_frame, [box_r], -1, (255, 0, 0), 2)
                    break

        if PAUSE_ON_RED and red_line_detected:
            print("Red horizontal line detected. Pausing.")
            if MOVE_ROBOT:
                robot.stop()
            permanent_stop = True  # Robot will never move again once red line is
                detected

        # - - - - - - - - - - - - - - - - - - - - - - - -
        # Obstacle Detection
        # - - - - - - - - - - - - - - - - - - - - - - - -
        # Define an ROI in the frame above the line detection area to look for large
            dark objects.
        # If a large contour is found here, consider it an obstacle.
        # We'll use the same `thresh` image, which marks dark objects as white regions.
        obstacle_roi = thresh[0:roi_y, 0:width]
        contours_obstacle, _ = cv2.findContours(obstacle_roi, cv2.RETR_EXTERNAL,
            cv2.CHAIN_APPROX_SIMPLE)

        current_obstacle_detected = False
        for cnt_o in contours_obstacle:
            area_o = cv2.contourArea(cnt_o)
            if area_o > OBSTACLE_AREA_THRESHOLD:
```

```
                # Large contour in the upper region indicates something is blocking the view
                current_obstacle_detected = True
                break

        # Update obstacle state: If detected now, stop. If not detected now, resume normal logic.
        if current_obstacle_detected and not permanent_stop:
            obstacle_detected = True
            if MOVE_ROBOT:
                robot.stop()
            # When obstacle is present, the robot stays stopped until obstacle is removed.
        else:
            # If no obstacle currently and previously it was detected, we can resume movement
            if obstacle_detected and not permanent_stop:
                # Resume normal operation if not influenced by red line or green markers
                obstacle_detected = False
                # Speeds are set in line/green logic above. Just ensure no obstacle logic stops movement here.

        processed_widget.value = bgr8_to_jpeg(processed_frame)

except KeyboardInterrupt:
    print("Interrupted by user. Stopping robot.")
finally:
    camera.stop()
    if MOVE_ROBOT:
        robot.stop()
```

Listing 2: Python Code for Robot Control