Mobile Robot Challenge

Robotics Mobile Robot Challenge

Pranav Srivastava[1],
Wisal Alaz [1],
Yousef Swed[1],
Kevin Bretz [1],
Laurens Grote [1]
[1]Leiden University

Email:

May 27, 2025

Abstract

This document describes a system implemented on a Raspberry Pi for detecting two distinct objects based on changes between two images and calculating the necessary camera adjustment to center the midpoint between these objects. The system utilizes a camera sensor as the sole input and performs all image processing tasks directly on the Raspberry Pi using OpenCV. The methodology relies on image differencing, thresholding, and contour analysis to locate the objects, without employing any pre-trained deep learning models for object detection.

1 Introduction

This project is part of the "Mobile Robot Challenge", a series of 2 challenges in which an intelligent, 4-wheeled robot is placed into a room and has 1 minute to scan its surroundings. After this minute, 2 new objects are placed into the room on the floor somewhere in front of the robot and it must identify them and drive in between them. For challenge 1, the items are chosen by and placed in their positions in the room by the participants of the challenge. For challenge 2, the items are chosen by and placed in their positions in the room by the judges.

The primary goal of this project is to develop a lightweight, autonomous system capable of identifying the appearance or significant movement of two distinct objects within a camera's field of view. Once these two objects are identified, the system calculates their geometric midpoint. Subsequently, it determines the pixel displacement required to reorient the camera so that this midpoint aligns with the center of the camera's image frame. To achieve this, it turns its head by adjusting the wheels to make the robot steer to the determined object middle point. This entire process, from image acquisition to computation of the adjustment vector, is designed to run efficiently on a Raspberry Pi, using its camera module.

The approach avoids computationally intensive object detection algorithms, instead focusing on detecting changes between a reference background image ("before") and a current scene image ("after").

2 System Methodology

The system operates through a sequence of image processing steps, all executed on the Raspberry Pi. The input consists of two images: a static background image and an image captured after a change has occurred (e.g., two new objects introduced).

2.1 Image Acquisition and Preprocessing

1. **Image Input**: The system requires two images: `before.png` (representing the scene before the introduction of the object) and `after.png` (representing the scene with the two objects). In a live deployment, these would be captured sequentially by the Raspberry Pi's camera.

2. **Grayscale Conversion**: Both input images are converted from BGR (Blue, Green, Red) color space to grayscale. This reduces computational complexity and focuses on luminance changes.

$$I_{gray} = \text{cv2.cvtColor}(I_{BGR}, \text{cv2.COLOR\_BGR2GRAY})$$

2.2 Change Detection

The core of object detection lies in identifying differences between grayscale "before" and "after" images.

1. **Absolute Difference**: The absolute pixel-wise difference between the two grayscale images is calculated. This highlights regions where changes have occurred.

$$I_{diff} = \text{cv2.absdiff}(I_{gray\_before}, I_{gray\_after})$$

2. **Thresholding**: The difference image I_{diff} is thresholded to create a binary image I_{thresh}. Pixels with a difference greater than a predefined threshold (e.g., 30) are set to 255 (white), and others are set to 0 (black). This isolates significant changes.

$$I_{thresh} = \text{cv2.threshold}(I_{diff}, 30, 255, \text{cv2.THRESH\_BINARY})[1]$$

3. **Dilation**: The thresholded image is then dilated. Dilation helps to close small gaps in the detected regions and make the object areas more solid. This is performed for a few iterations (e.g., 2).

$$I_{dilated} = \text{cv2.dilate}(I_{thresh}, \text{None, iterations=2})$$

2.3 Object Localization

After change detection, contours are found in the dilated binary image to locate the individual objects.

1. **Contour Detection**: The `cv2.findContours` function is used to find the boundaries of the white regions (potential objects) in $I_{dilated}$. It retrieves external contours using a simple chain approximation method.

2. **Centroid Calculation**: For each detected contour, the image moments are calculated. The centroid (cX, cY) of the contour, which represents the center of the object, is then computed using these moments:

$$M = \text{cv2.moments(contour)}$$

If $M[\text{"}m00\text{"}] \neq 0$:

$$cX = \frac{M[\text{"}m10\text{"}]}{M[\text{"}m00\text{"}]}$$

$$cY = \frac{M[\text{"}m01\text{"}]}{M[\text{"}m00\text{"}]}$$

The system specifically expects to detect exactly two such objects. If not, an error is indicated. The centers of these two objects are stored as (x_1, y_1) and (x_2, y_2).

2.4 Target Point Calculation

Once the centers of the two objects are identified, their collective midpoint is calculated.

1. **Midpoint Computation**: The coordinates of the target center point (C_x, C_y) between the two detected object centers are calculated as:

$$C_x = \frac{x_1 + x_2}{2}$$

$$C_y = \frac{y_1 + y_2}{2}$$

This (C_x, C_y) becomes the point of interest that the camera should ideally be centered on.

2.5 Camera Adjustment Vector

Finally, the system calculates the necessary adjustment for the camera to center its view on the calculated target midpoint.

1. **Image Center**: The center of the camera's image frame is determined by its width (W) and height (H). For an image of 640×480 pixels:

$$I_{center\_x} = \frac{W}{2} = \frac{640}{2} = 320$$

$$I_{center\_y} = \frac{H}{2} = \frac{480}{2} = 240$$

2. **Delta Calculation**: The difference (delta) between the target midpoint (C_x, C_y) and the image center $(I_{center\_x}, I_{center\_y})$ is computed. This gives the pixel offset required for centering.

$$\Delta x = C_x - I_{center\_x}$$

$$\Delta y = C_y - I_{center\_y}$$

These Δx and Δy values represent the horizontal and vertical pixel adjustments the camera needs to make. These values can then be used by a motor control system (like the imported `MotorControl`) to physically pan and tilt the camera.

2.6 Steering

Due to the inconsistent nature of our hardware, specifically the low friction and general design of the robot's wheels, which we did not have a replacement for, it is extremely difficult to get reproducible results using the basic motor control functions that the robot provides. Because of this high inaccuracy when trying to steer the robot along a pre-determined route, it is necessary to implement a safety function that corrects the trajectory of the robot. With our setup, ideally we would be able to adjust the robots motor control signals based on visual feedback in regular, frequent intervals. However, due to our object detection and trajectory calculation being based

on a before and after picture taken from a single camera perspective and the robot not having any further information about it's surroundings, as soon as what the robot's camera sees changes the object detection no longer works, which means the robot can no longer visually determine where it needs to go and has no sense of position or direction. This means that the robot has to know exactly what it is going to do before it moves.

We were not able to come up with a more reliable way to steer and correct the robots trajectory, so our solution for this sub-task is simply to rotate the robot a fixed value to the left or to the right, depending on where the center of both objects is determined as being in relation to the center of the robots camera view, and then to drive forward for a fixed amount of time. If the center of both objects is within a certain threshold from the center of the image, the robot will move straight ahead. The amount of left and right rotation, as well as driving forward, were chosen based on values that performed well during testing.

In a future implementation, a robust trajectory correction logic should be added to the process to greatly increase the robot's chance of arriving at it's goal location. Using wheels with a higher friction coefficient will also aid in this task.

3 Implementation Notes

The system is implemented in Python using the OpenCV library (`cv2`) for all image processing tasks. The computations are performed entirely on the Raspberry Pi. The script `object_diff.py` encapsulates this logic, including functions such as `find_object_locations`, `calculate_center_point`, and `rotate_camera` (which calculates the deltas). While `Picamera2` and `MotorControl` are imported, the core detection logic focuses on processing pre-loaded images and calculating the adjustment vector.

Demo link

YouTube Link for the mobile robot challenge

4 Conclusion

This document outlined a method for detecting two objects based on image differences and calculating the necessary camera adjustments to center their midpoint, designed for a Raspberry Pi platform. By relying on fundamental image processing techniques available in OpenCV, such as grayscale conversion, image differencing, thresholding, and contour analysis, the system provides a computationally efficient solution that uses only the camera sensor. This approach is suitable for applications where specific object recognition is not required, but rather the detection of change and the localization of new elements in the scene. All processing is self-contained on the Raspberry Pi.

A Appendix 1

Detailed instructions for installing all required dependencies and executing the program can also be found in the README.md.

A.1 Dependencies

Following libraries should be installed:

```bash
sudo apt install python3-click
sudo apt install python3-flask-cors
sudo apt install python3-flask-socketio
sudo apt install python3-flask
sudo apt install python3-gi-cairo
sudo apt install python3-gi
sudo apt install python3-git
sudo apt install python3-gpiozero
sudo apt install python3-gunicorn
sudo apt install python3-kms++
sudo apt install python3-libcamera
sudo apt install python3-libgpiod
sudo apt install python3-matplotlib
sudo apt install python3-numpy
sudo apt install python3-opencv
sudo apt install python3-paho-mqtt
sudo apt install python3-pgzero
sudo apt install python3-picamera2
sudo apt install python3-pigpio
sudo apt install python3-pyqt5
sudo apt install python3-rpi-lgpio
sudo apt install python3-serial
sudo apt install python3-smbus2
sudo apt install python3-spidev
sudo apt install python3-tk
sudo apt install python3-torch
```

A.2 Running the code

To run the code that was used during the presentation, just run ‘python main.py‘, instructions on what to do will be prompted.

We also implemented a top down view mechanism that converts a picture into a top down view, trying to estimate the distances from the robot. To run this, minimum following libaries need to be installed:

```
pip install numpy matplotlib
```

After that inside the top_down_view folder a ‘python bird_view.py‘ starts the program. The program asks you to draw two rectangles (by putting four points).

B Appendix 2

B.1 Top-Down View Images

We were regrettably unable to obtain any top-down sketches that were of a satisfactory degree of accuracy using the tools at our disposal. Due to our object detection technique being focused on being lightweight, we do not obtain any absolute spatial information which can be used to

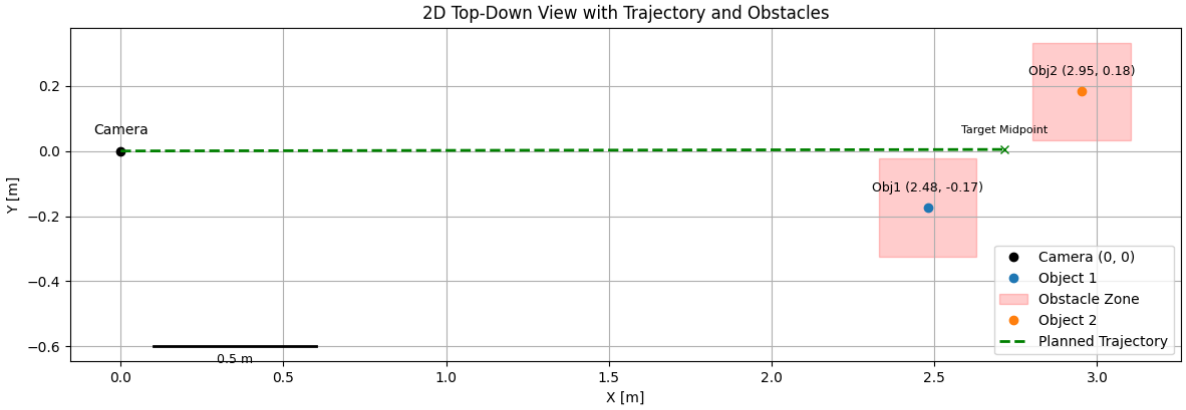

Figure 1: 2D Top-Down View with Trajectory and Obstacles

infer a map of the room. We experienced steep performance drops from the RaspberryPi when we attempted to run any computer vision models on it that we were considering using, and we were also unable to correctly deduce the distance to a point on the ground using perspective projection mapping, so we instead opted for a fast and computationally lightweight solution focused on the main task of detecting and passing between two objects.

We do however have a working python script that enables interactive localization of objects in an image and visualizes their positions from a top-down perspective. The main goal is to estimate the real-world positions of two objects relative to a camera and to plot a planned trajectory for navigation or obstacle avoidance.

B.2 Functionality

- **Interactive Object Selection:**
  The script loads an image and prompts the user to click on the top-left and bottom-right corners of two objects (four clicks in total). Each click is visually marked and labeled for clarity.

- **Distance and Offset Calculation:**
  Using camera calibration parameters (such as focal length and real object width), the script calculates the distance from the camera to each object and their lateral offsets from the camera's centerline. These calculations are based on the pinhole camera model and assume the real width of the object is known.

- **Top-Down Visualization:**
  The script generates a 2D plot showing the camera at the origin and the two objects at their computed positions. Each object is marked and surrounded by a shaded "obstacle

zone." If two objects are selected, a planned trajectory from the camera to the midpoint between the objects is also plotted. The visualization includes a scale bar and axis labels, and is saved as an image file.

B.3 How to Run the Script

1. **Dependencies:**
   The script requires Python 3.x, OpenCV, NumPy, and Matplotlib. These can be installed with:

   ```
   pip install opencv-python numpy matplotlib
   ```

2. **Image Preparation:**
   Place the target image (e.g., `testImage.jpg`) in the same directory as the script, or update the `image_path` variable in the script to point to the correct file.

3. **Execution:**
   Run the script from the terminal:

   ```
   python bird_view.py
   ```

   The image will open in a window. Click the top-left and bottom-right corners of each object (four clicks total). After the fourth click, press any key to continue.

4. **Results:**
   The script prints the calculated distances and offsets for each object. A top-down view is displayed and saved as `challenge1_view.png`.

B.4 Customization

Camera parameters such as focal length and real object width can be adjusted at the top of the script to match the specific camera and objects used. The size of the obstacle zones can be changed in the plotting function.

B.5 Applications

This approach is useful in robotics for tasks such as object localization, obstacle avoidance, and trajectory planning using camera images.