ME 598
Introduction to Robotics
Fall 2024

Lab 3: Image Processing for Vision-Based Tasks

R2
21 Nov 2024

Undergraduate students:

"I pledge my honor that I have abided by the Stevens Honor System."

Graduate students:

"I pledge that I have abided by the Graduate Student Code of Academic Integrity."

The report has been prepared by:

1. Mohammad Althaf Syed *althaf syed*
2. Bhagyath Badduri *Bhagyath*
3. Matthew Casey *Matthew Casey*
4. Prayash Das *Prayash Das*

Abstract

To enable a mobile robot to recognize and locate visual landmarks, this lab investigates the use of image processing algorithms for vision-based activities. The main goal is to create and apply algorithms that use real-world photos to detect orange traffic cones within the robot's field of vision. Configuring a camera in MATLAB, taking pictures, and using the MATLAB Color Thresholder App to separate color ranges in the HSV spectrum are important activities. The detection system is then trained using these calibrated HSV parameters to ensure reliable cone recognition. Following cone detection, essential features, including the cone centroids, are extracted using sophisticated image processing techniques like blob analysis. This entails identifying the area and centroid of every observed blob, labeling related components, and filtering out noise. By using the camera's horizontal field of view to compute angular offsets and closeness, the detected cones are mapped according to their orientation and separation from the robot.

To make sure these algorithms are resilient and flexible in the face of changing illumination and positioning situations, the lab moves on to testing them on more datasets. In order to assess system performance and improve the detection pipeline, students also take fresh test images. A study of algorithmic performance at the end of the lab highlights implementation issues, such as optimizing computational efficiency and adjusting settings. Students who successfully complete this lab will have practical expertise with image processing tools and robotics-related techniques such feature extraction, object localization, and color identification. These abilities serve as the foundation for incorporating computer vision into mobile robotic platforms, connecting abstract ideas with real-world robotics and automation applications.

Introduction

Mobile robotic systems must be able to sense and understand their surroundings. Robots can locate and recognize things in their environment thanks to vision-based sensing, which is crucial for activities involving interaction, manipulation, and navigation. With an emphasis on the identification and localization of visual landmarks within the range of vision of a mobile robot, this lab presents fundamental ideas and methods in image processing.

The lab has a strong emphasis on using MATLAB to construct image processing workflows, such as setting up a camera, taking pictures, and creating algorithms for color-based object detection. The assignment specifically entails utilizing the HSV color model, a well-liked method for reliable color segmentation in a range of lighting conditions, to detect orange traffic cones. To isolate target items in photos, color thresholds are adjusted and improved using the MATLAB Color Thresholder App. Building on the detection skills, features including the area, centroid, and position of observed cones are extracted via blob analysis. Cones' angular orientation and relative distance to the robot are then calculated using this data and the camera's field of vision.

To automate image processing activities across datasets, the lab also introduces students to tools such as the MATLAB Image Batch Processor App. This guarantees that new datasets and circumstances may be accommodated by the created algorithms. To prepare for increasingly complex tasks in mobile robotics and autonomous systems, students will have a hands-on grasp of how to incorporate image processing techniques into vision-based robotic applications at the end of the lab.

Theory & Experimental Procedure

Part 1

a) Enable webcam within MATLAB

We installed MATLAB Webcam Support by ensuring the system had the necessary drivers for the connected webcam.
Initialized the webcam in MATLAB using the webcam function

```
>>> mycam = webcam;
>>> mycam.Resolution = '640x480';
```

The webcam was successfully recognized, and the specified resolution was applied.

b) Grab image in MATLAB

The captured image was displayed successfully and saved as *Image1.jpg* in the working directory.

Discussion:

Efficiently managing external factors like lighting and orientation was critical to obtaining usable images. Using a moderate resolution ensured the system was prepared for real-time processing tasks without unnecessary computational overhead.

Conclusion

In this part, the webcam was successfully set up and integrated with MATLAB. An image was captured, displayed, and saved for further processing. Challenges such as lighting conditions and resolution selection were addressed effectively, laying the groundwork for vision-based tasks in subsequent parts of the lab.

Part 2

Specify the minimum and maximum HSV values used by your code.

The following HSV ranges were determined to effectively segment the target object:

Parameter	Minimum	Maximum
Hue (H)	0.024	0.135
Saturation (S)	0.378	1.000
Value (V)	0.741	1.000

- Hue Range (0.024 - 0.135): Captures the specific color of the object based on its hue.
- Saturation Range (0.378 - 1.000): Ensures that only sufficiently vibrant colors are included, filtering out dull or grayish tones.
- Value Range (0.741 - 1.000): Focuses on brighter areas, eliminating shadows or darker regions that do not belong to the target object.

Image 1

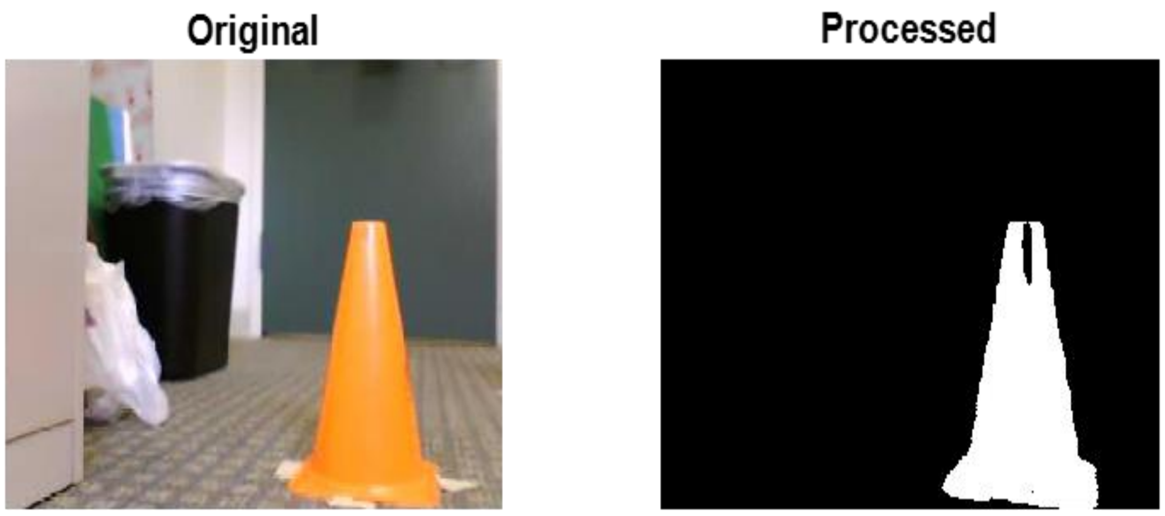


Image 2

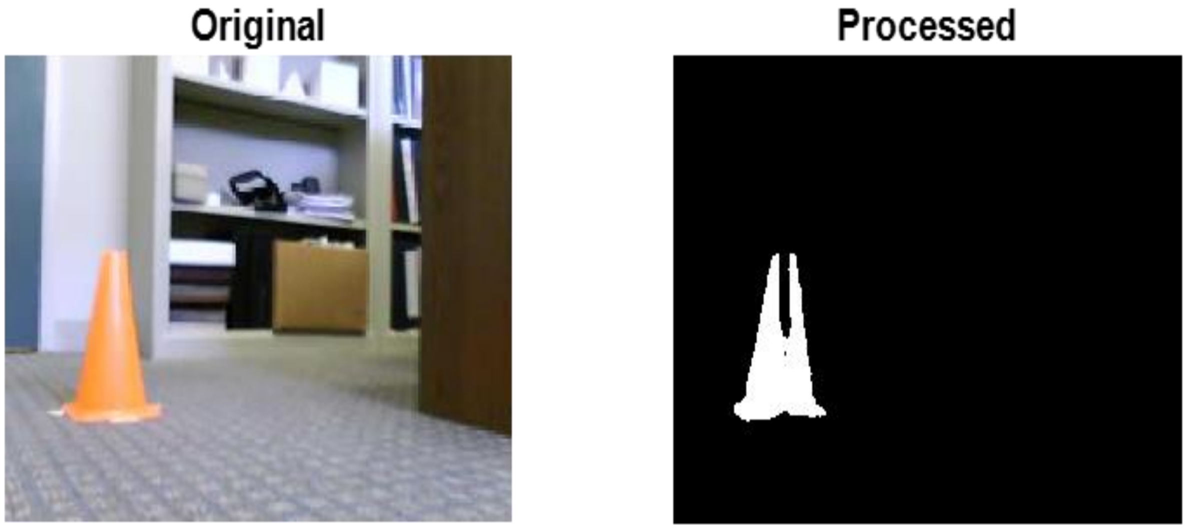


Image 3

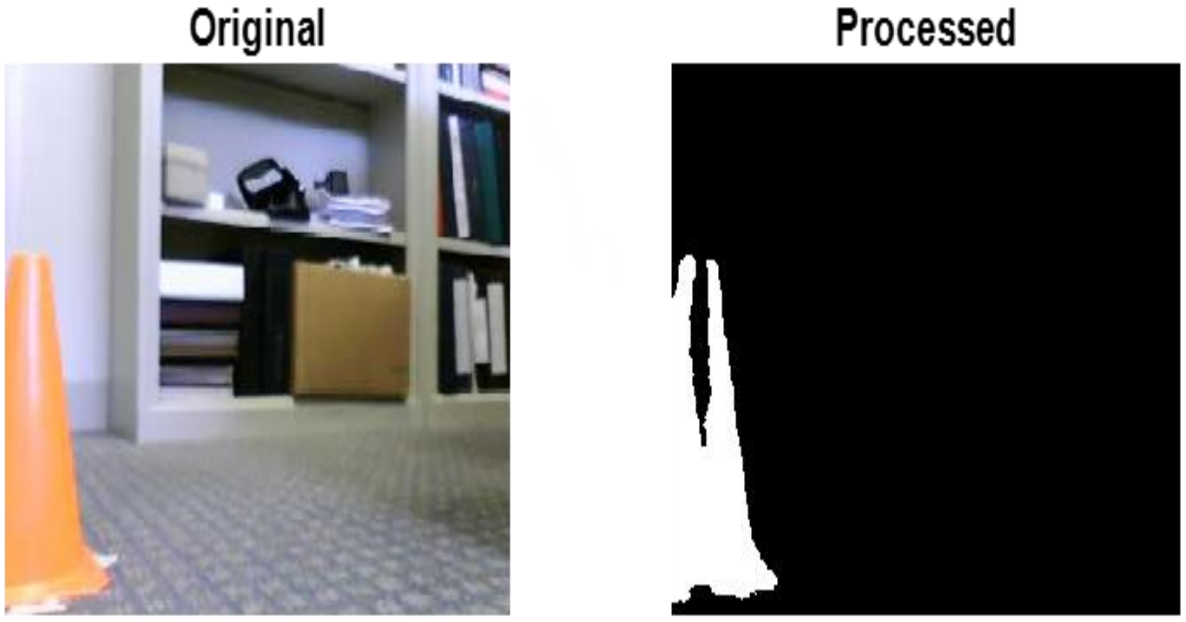


Image 4

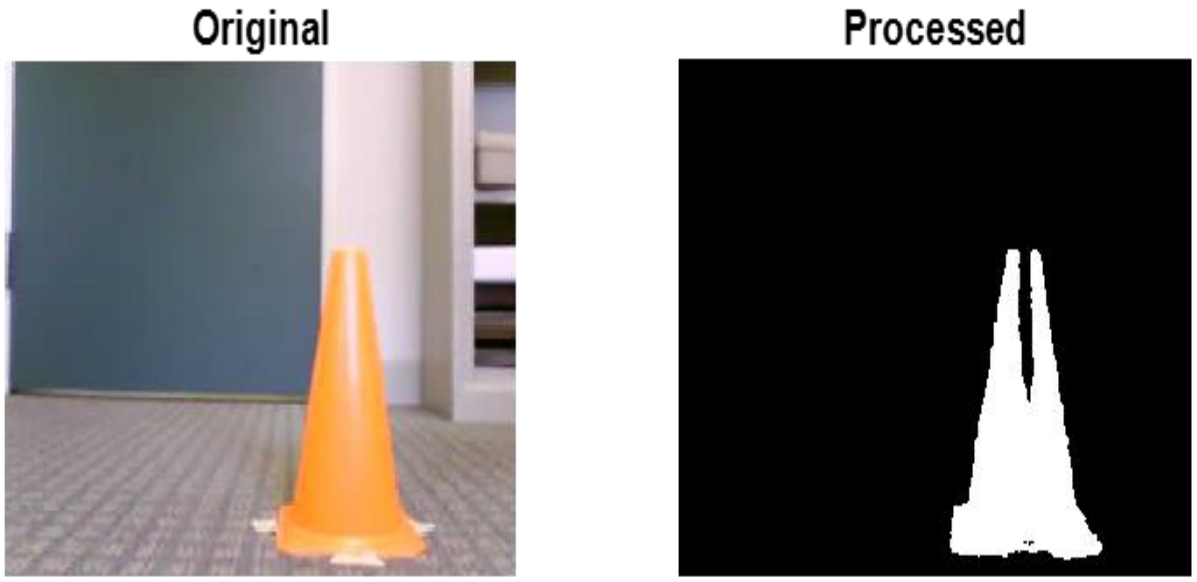


Image 5

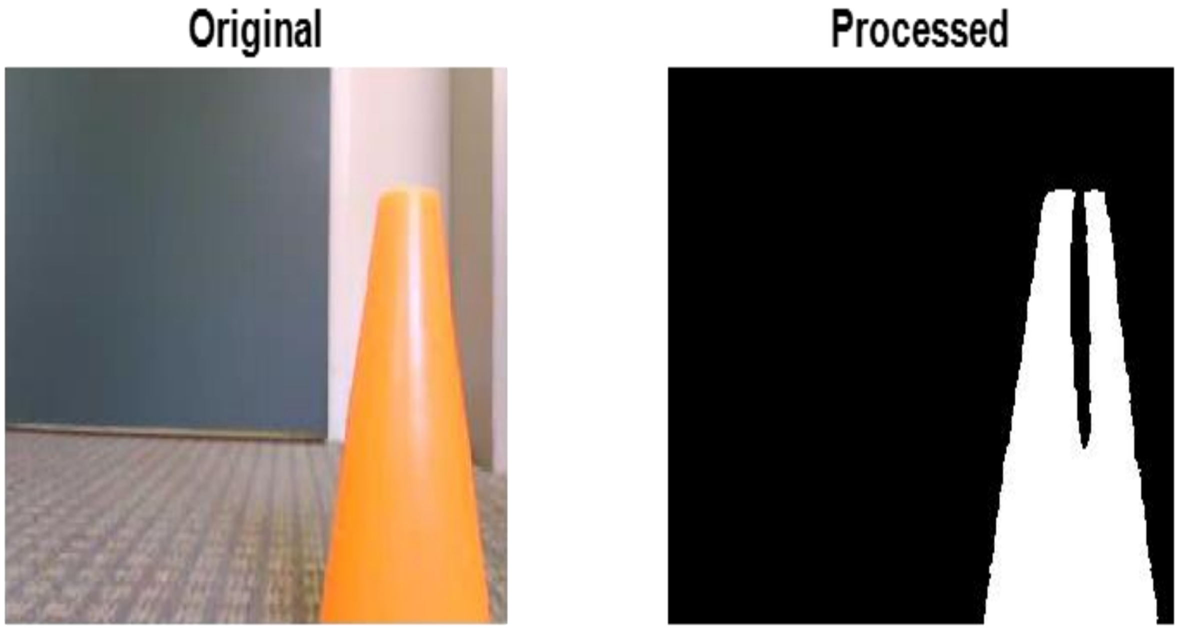


Image 6

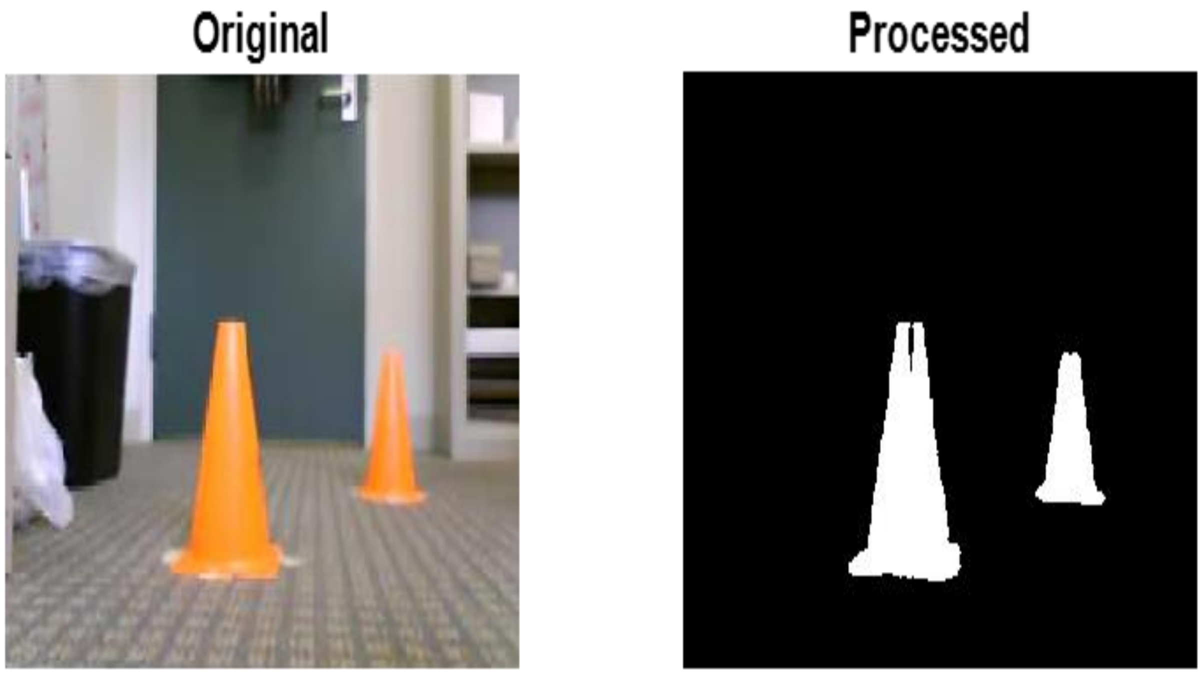


Image 7

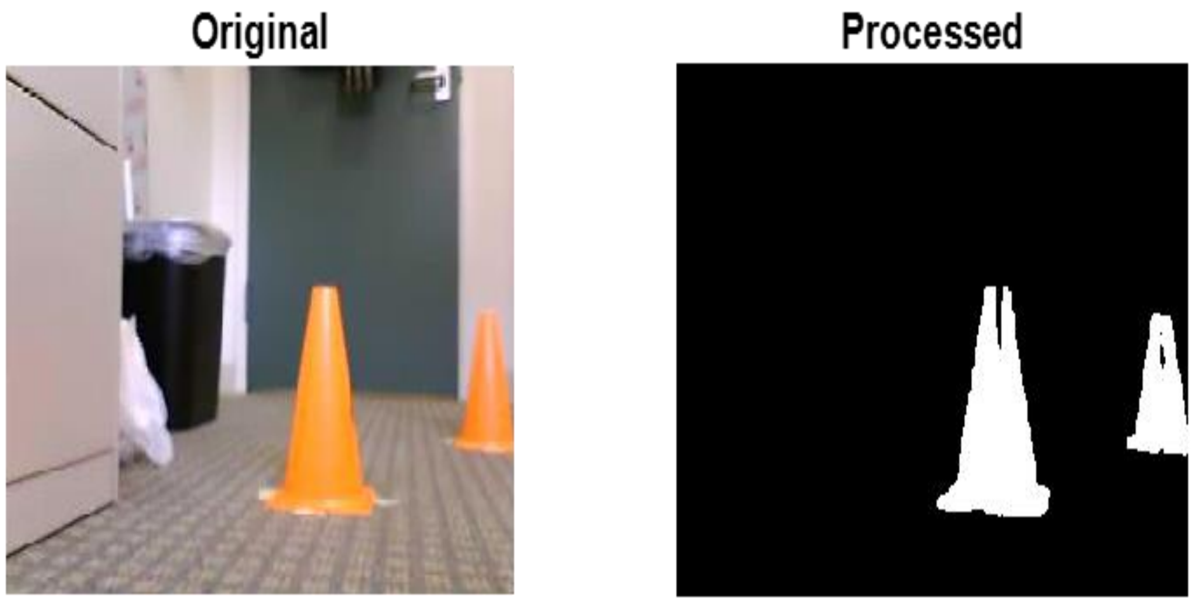


Image 8

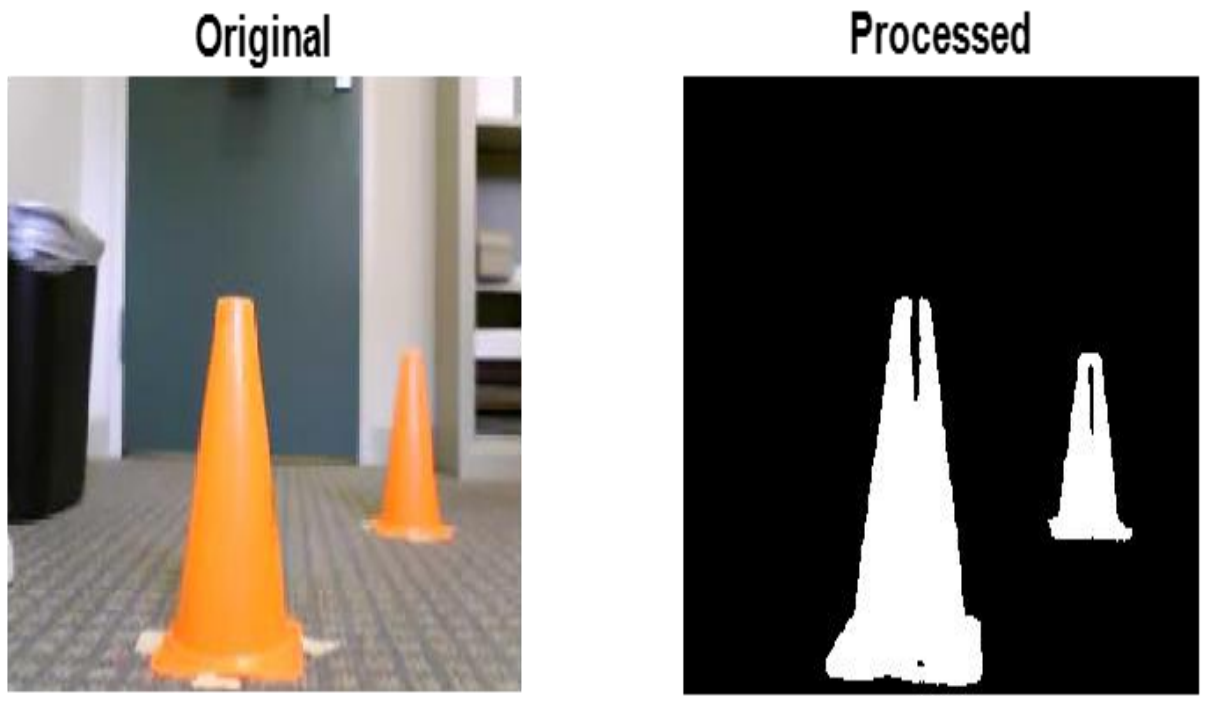
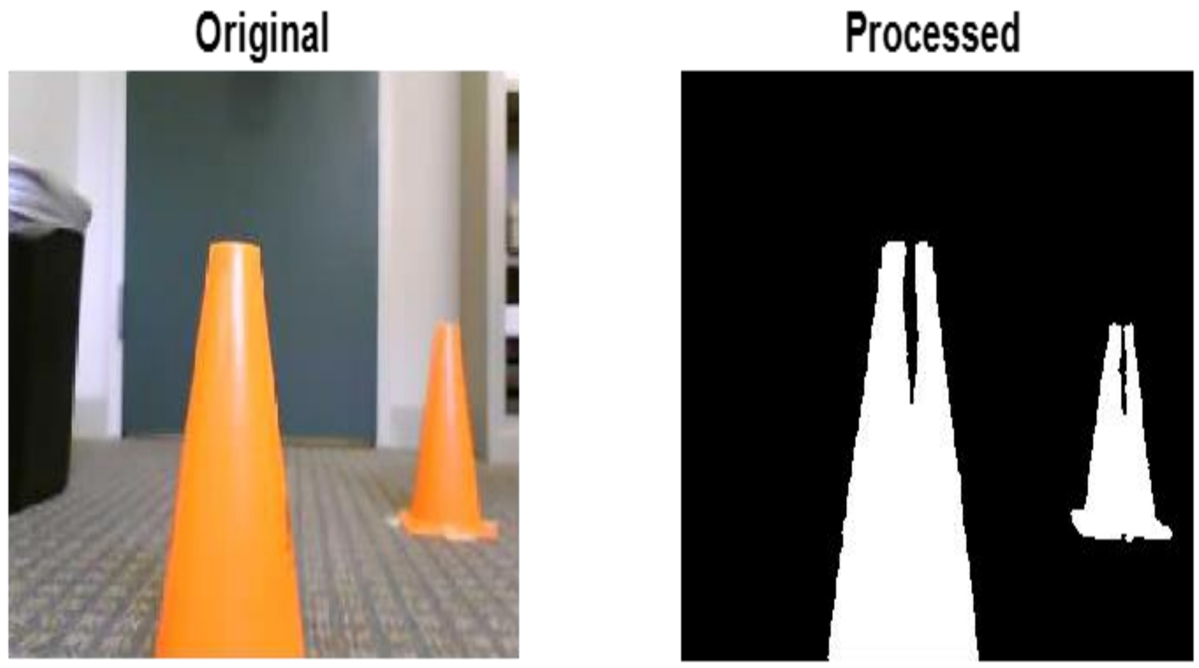


Image 9



Part 3

The code processes a set of images to detect, localize, and map orange traffic cones in the robot's field of view. Key tasks include color detection, blob analysis, centroid extraction, and calculating the angular position and proximity of cones relative to the robot.

The code is designed to detect and localize cones in a set of images by leveraging HSV color-based segmentation and blob analysis. It processes the images to identify the centroids of detected cones, calculates their angular positions and proximities relative to the robot, and visualizes the results in both direct and overhead views.

Explanation of the Code

The code is designed to detect and localize cones in a set of images by leveraging HSV color-based segmentation and blob analysis. It processes the images to identify the centroids of detected cones, calculates their angular positions and proximities relative to the robot, and visualizes the results in both direct and overhead views.

Code Workflow

1. **Image Input and Preprocessing**:
   - Reads images from the specified folder.
   - Converts RGB images to HSV color space to isolate orange cones using predefined thresholds.
2. **Binary Mask Creation**:
   i. Applies HSV thresholds to create a binary mask for orange color detection.
   ii. Cleans the binary mask using morphological operations (e.g., noise removal and hole filling).
3. **Blob Analysis**:
   i. It identifies connected components in the binary mask and extracts their properties such as area and centroid.
   ii. Filters valid regions based on a minimum area threshold.
4. **Angle and Proximity Calculation**:
   i. It calculates the angular offset of cones from the robot's centerline using their centroid positions.
   ii. Estimating proximity based on the vertical location of centroids in the image.
5. **Visualization**:
   i. Marks centroids and displays angles on the processed images.
   ii. Generates an overhead view plot showing cone positions relative to the robot.
6. **Saving Results**:
   a. Outputs visualized images and overhead views for further analysis.

Copy of commented code

The MATLAB codes of centroid detection, color threshold and overhead view of the cones are provided in the **Appendix D and Appendix E** for your reference.

Results

Finding the centroids of the cone:

Centroid for Train Image 1:

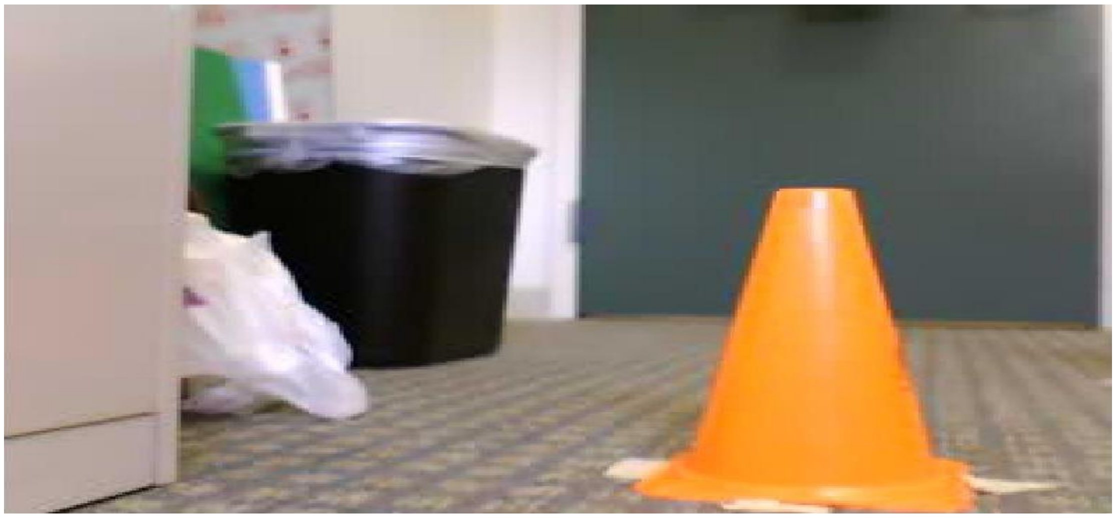


Figure 1 – Train Image 1

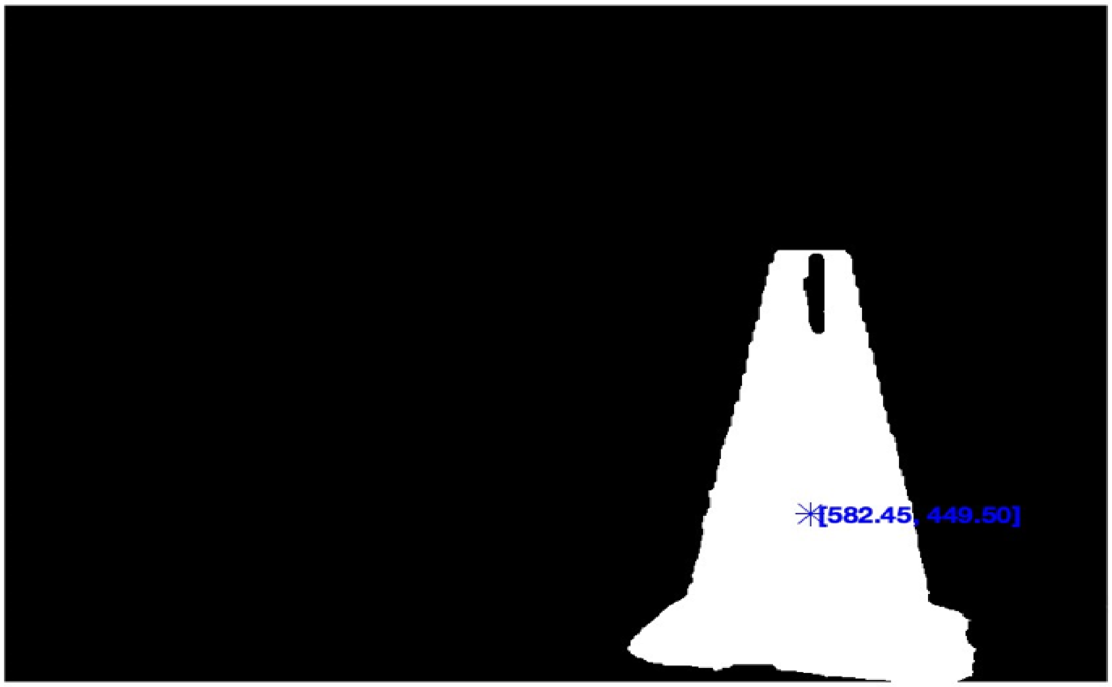


Figure 2 – Processed Train Image 1

Centroid coordinates:
[582.45, 449.50]

Centroid for Train Image 2:

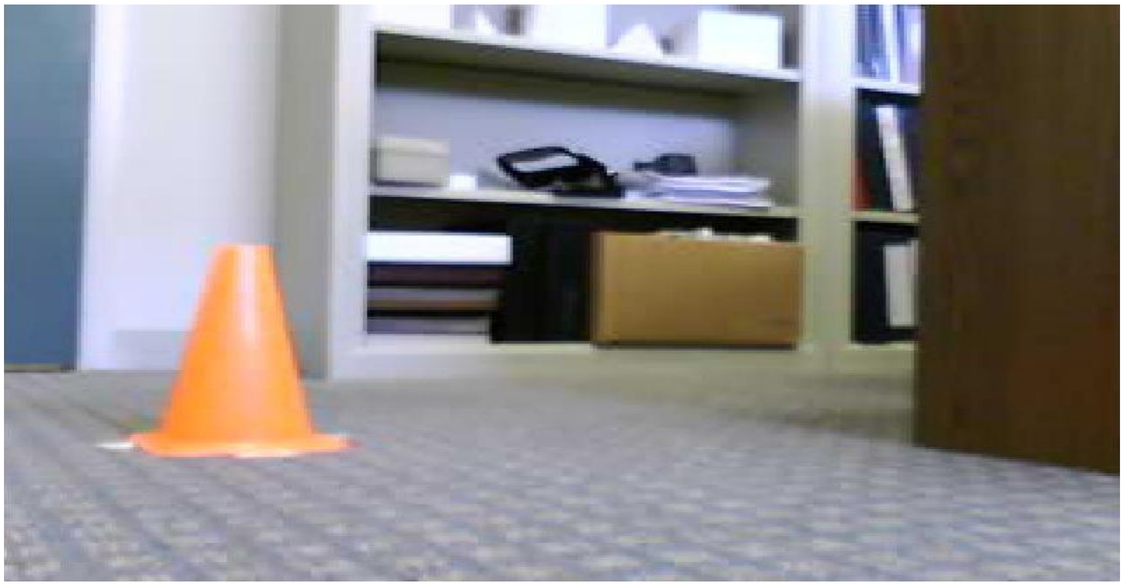


Figure 3 – Train Image 2

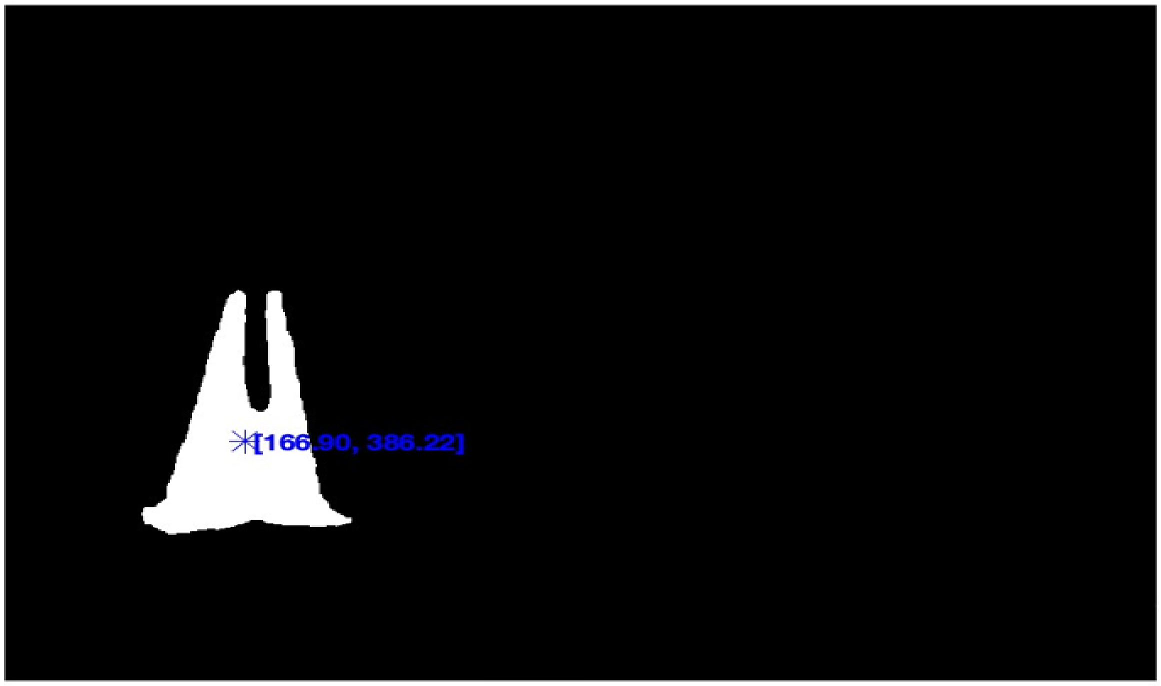


Figure 4 – Processed Train Image 2

Centroid coordinates:
[166.90, 386.22]

Centroid for Train Image 3:

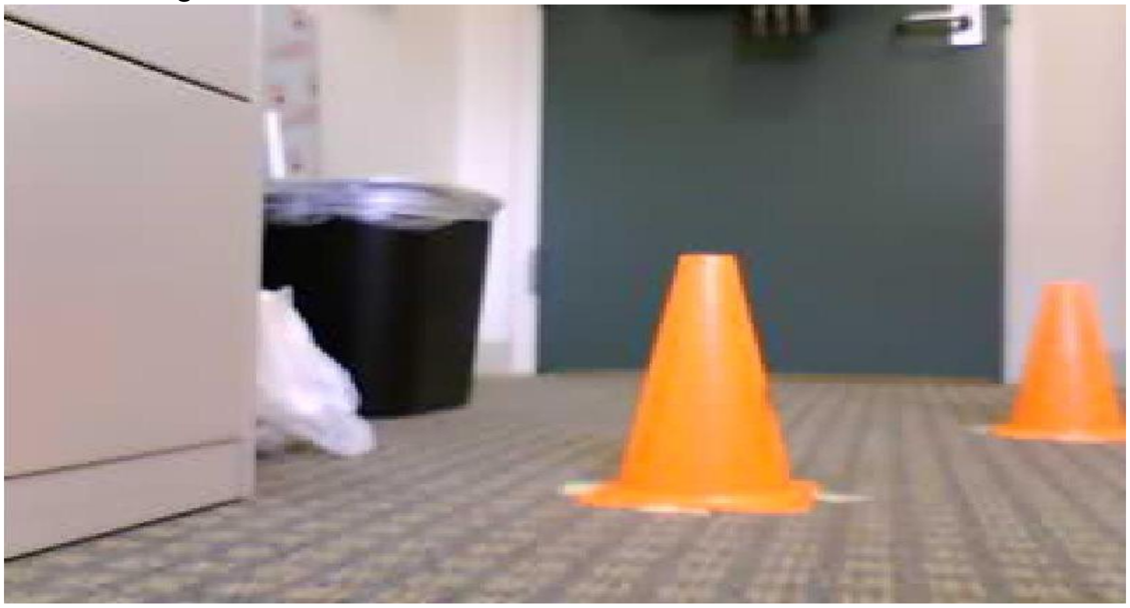


Figure 5 – Train image 3

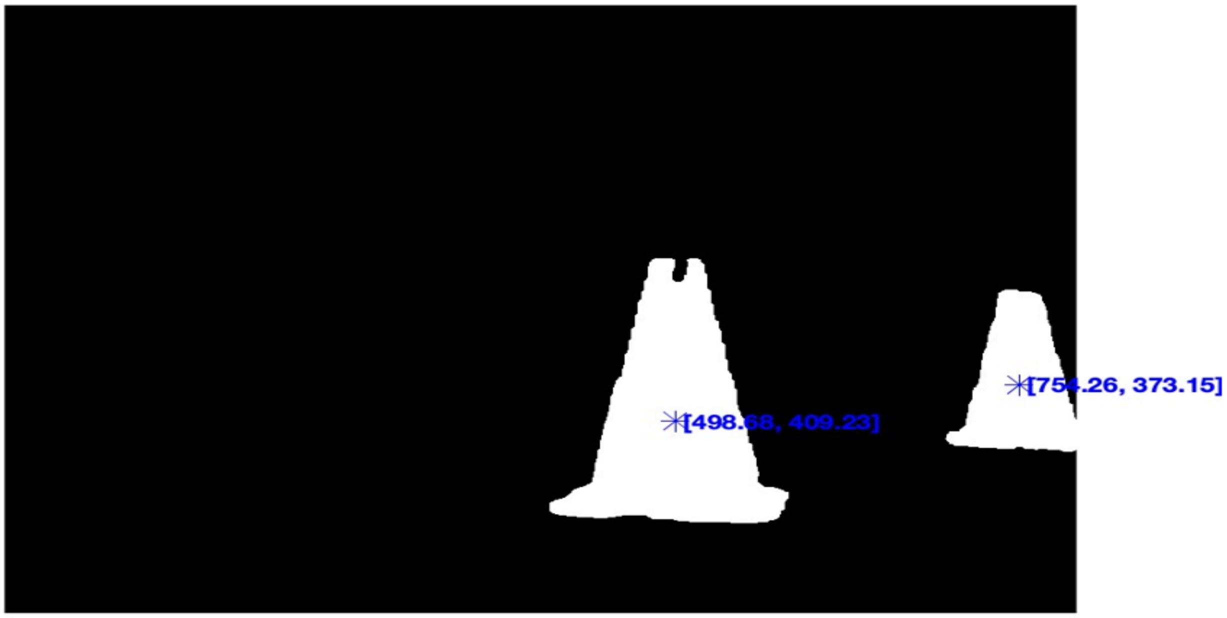


Figure 6 – Processed Train Image 3

Centroid coordinates:
[498.68, 409.23]
[754.26, 373.15]

Centroid for Train Image 4:

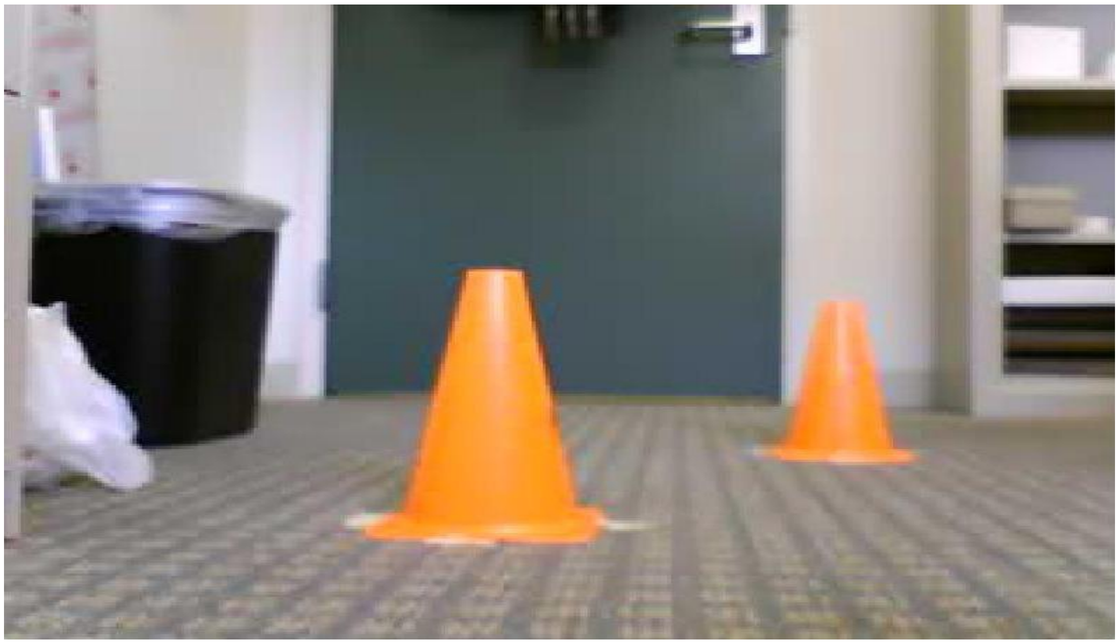


Figure 7 – Train Image 4

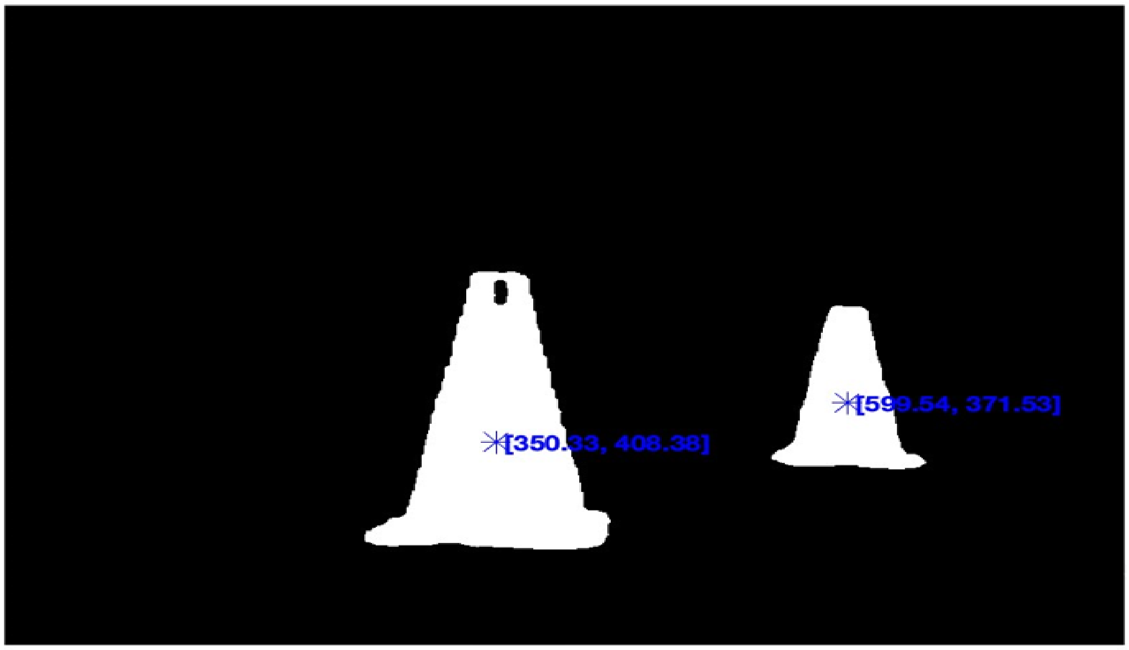


Figure 8 – Processed Train Image 4

Centroid coordinates:
[350.33, 408.38]
[599.54, 371.53]

Centroid for Train Image 5:

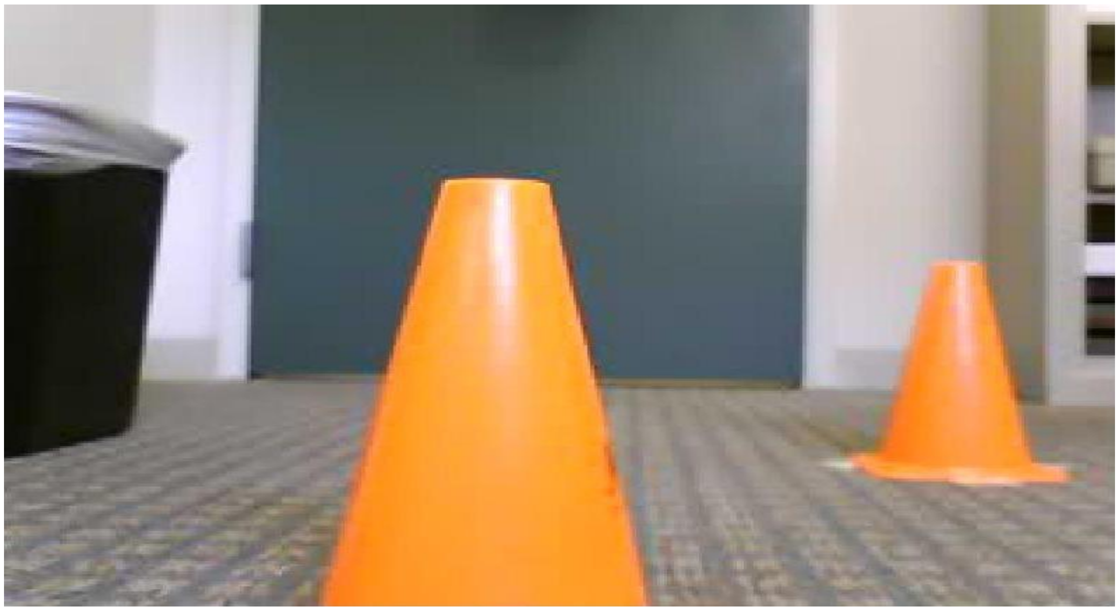


Figure 9 – Train Image 5

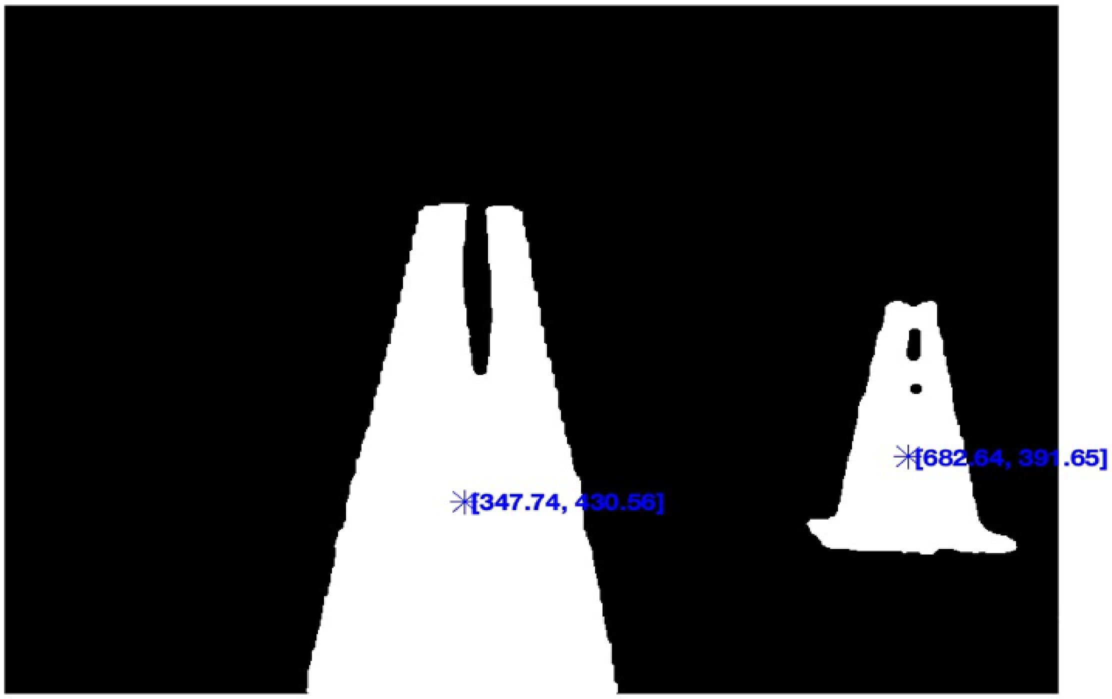


Figure 10 – Processed Train Image 5

Centroid coordinates:
[347.74, 430.56]
[682.64, 391.65]

Image with no cones:

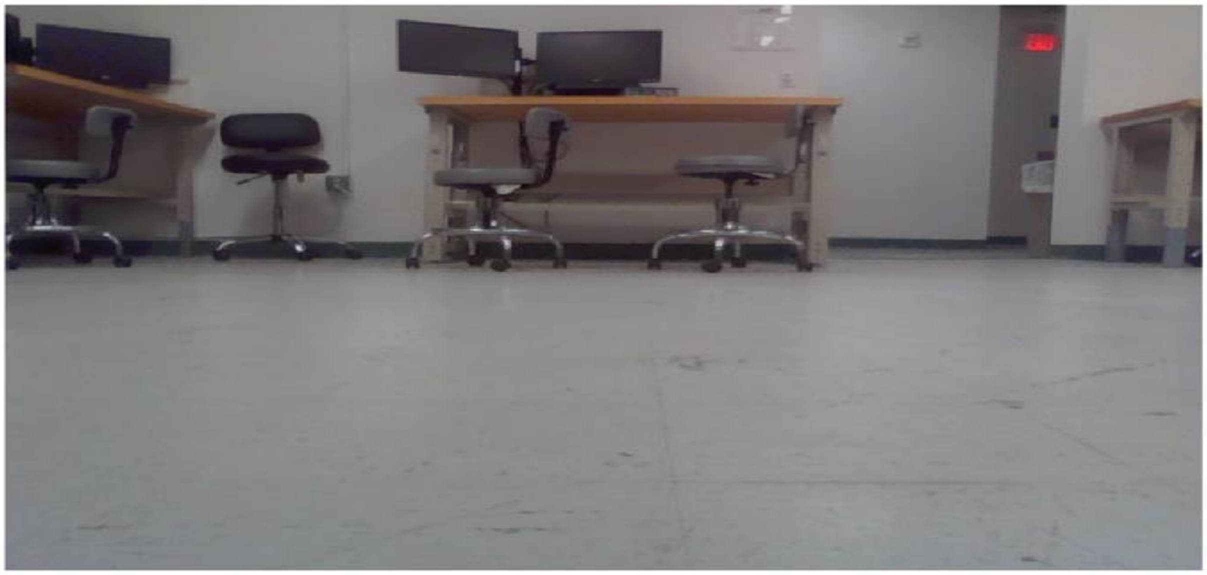


Figure 11 – No Cone Image

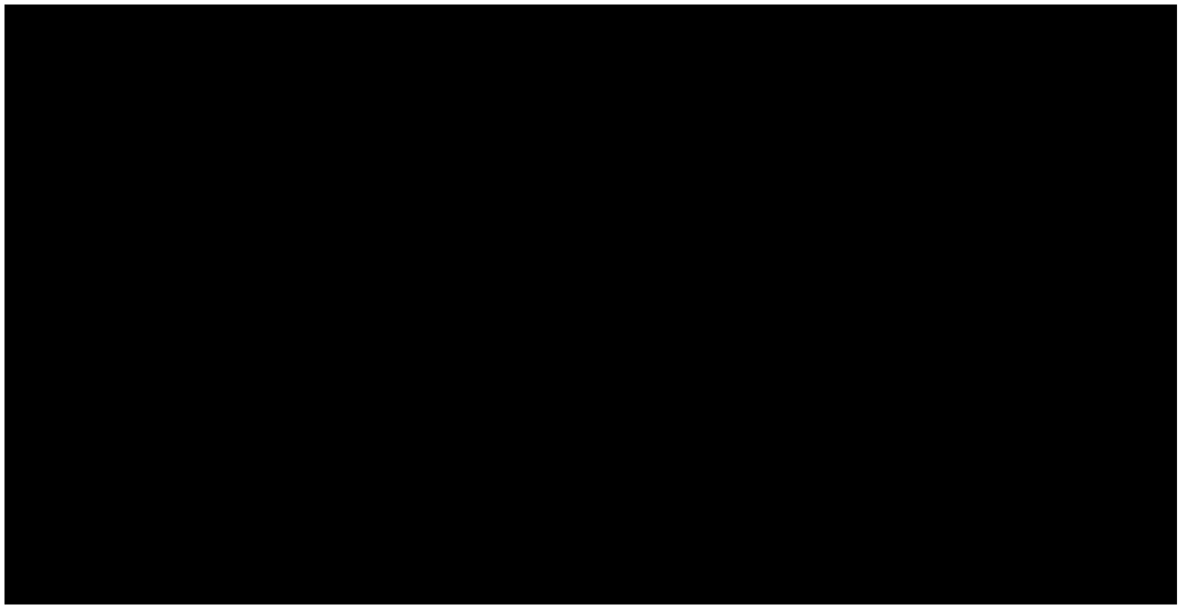


Figure 12 – Processed image for no Cone image

```
>>> Centroid coordinates: []
```

How we adapted the algorithm to determine cone angle and relative proximity:

To determine the angle and proximity of cones relative to the robot, the algorithm was carefully designed and adapted as follows:

1. Calculating Cone Angle

The angle of each cone is computed based on the horizontal position of its centroid in the image frame relative to the center of the image.

a) **Image Center as Reference**:

- The horizontal center of the image (ImageCenterX) is calculated as half the number of columns in the image.
- This serves as the reference point for determining the angular offset.

b) **Horizontal Offset**:

- The horizontal distance of the centroid (CentroidX) from ImageCenterX is measured.
- This difference (Xdiff = CentroidX - ImageCenterX) determines the position of the cone relative to the robot's forward direction.

2. Calculating Relative Proximity

The proximity of each cone is estimated based on its vertical position in the image, as cones farther from the camera tend to appear higher (closer to the top) in the image frame.

a) **Image Height as Reference**:

- The total number of rows (nr) in the image is used to determine the relative vertical position of the centroid.

b) **Vertical Normalization**:

- The vertical centroid position (CentroidY) is scaled to a range between 1 (near) and 3 (far).

c) **Interpretation**:

- A proximity value of 1 indicates that the cone is close to the robot.
- A value of 3 indicates that the cone is farther away.

3. Handling Multiple Cones

For images with multiple cones:

- The algorithm calculates angles and proximities for each detected cone independently.
- Results are stored in arrays to represent the positions and distances of all visible cones.

4. Adaptations for Robustness

- **Image Resolution Independence**: The calculations for angle and proximity dynamically adapt to the image resolution by using normalized values of the image width and height.

- **Threshold Adjustments**: The proximity calculation was scaled to account for varying sizes of cones in images, ensuring consistent interpretation across different scenarios.

Overhead view:
Overhead view of train cones image 1:

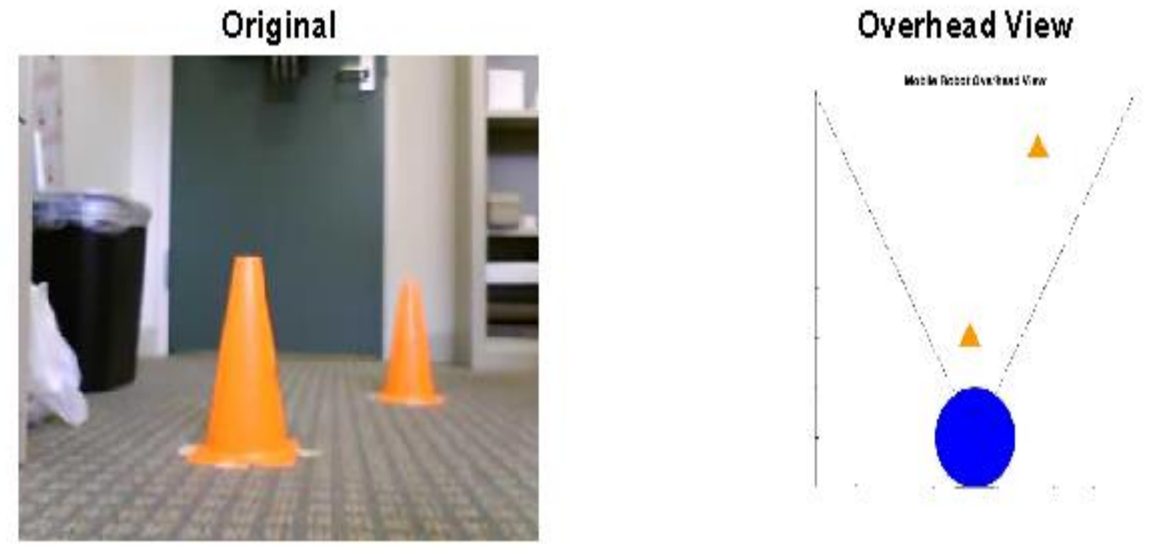


Figure 13 - Overhead_View_Train_cone_image1

>>>Centroid coordinates:

Cone 1: [350.33, 408.38]

Cone 2: [599.54, 371.53]

Orientation angles (degrees):

Cone 1: 3.59°

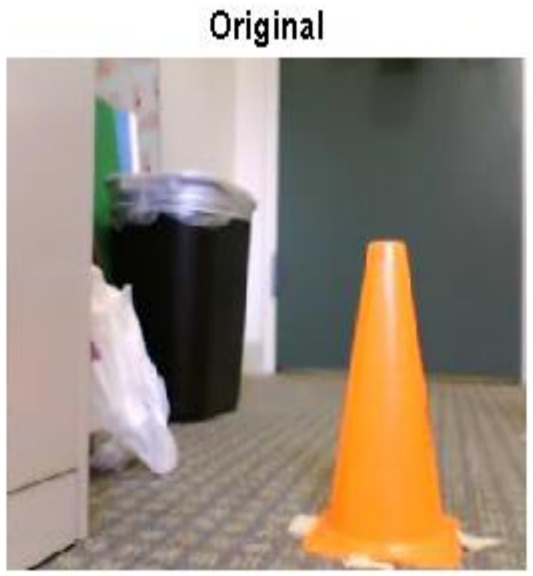
Cone 2: -15.19°

Proximity levels:

Cone 1: 1

Cone 2: 3

Overhead view of train cones image 2:



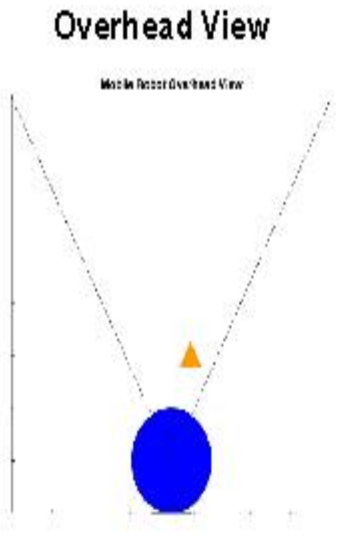


Figure 14 - Overhead_View_Train_cone_image2

>>>Centroid coordinates:

Cone 1: [582.45, 449.50]

Orientation angles (degrees):

Cone 1: -13.90°

Proximity levels:

Cone 1: 1

Overhead view of train cones image 3:

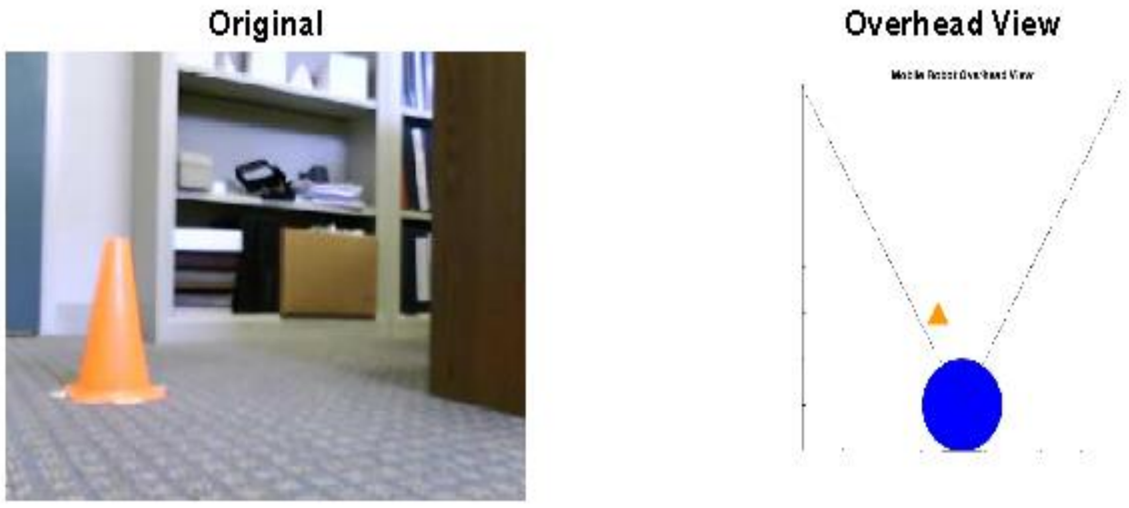


Figure 15 - Overhead_View_Train_cone_image3

```
>>>Centroid coordinates:
    Cone 1: [166.90, 386.22]
    Orientation angles (degrees):
    Cone 1: 17.42°
    Proximity levels:
    Cone 1: 1
```

Overhead view of train cones image 4:

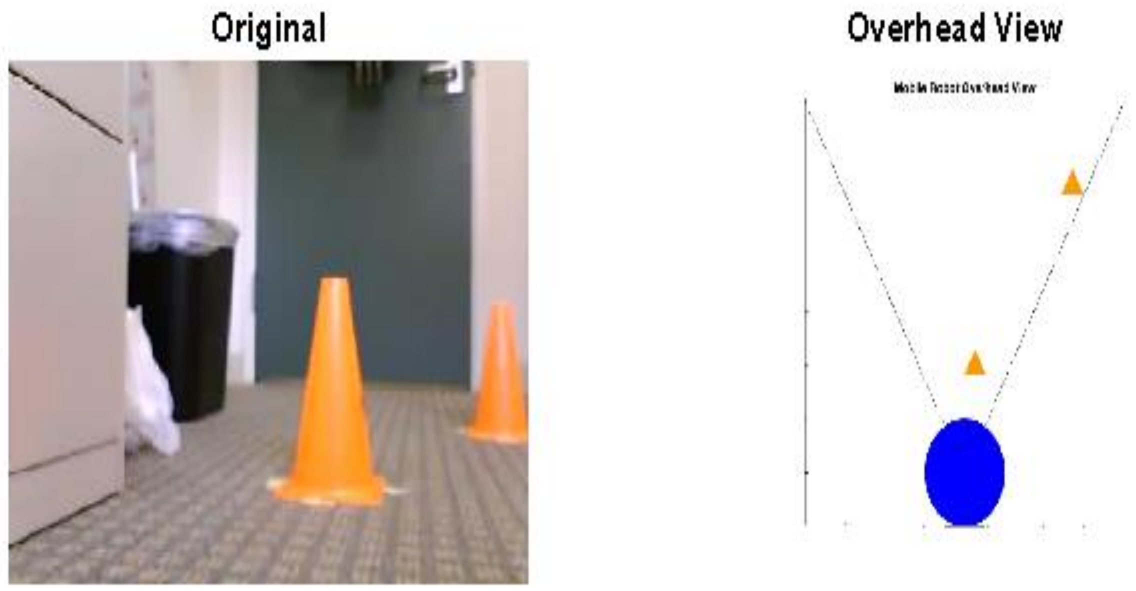


Figure 16 - Overhead_View_Train_cone_image4

>>>Centroid coordinates:

Cone 1: [498.68, 409.23]

Cone 2: [754.26, 373.15]

Orientation angles (degrees):

Cone 1: -7.59°

Cone 2: -26.85°

Proximity levels:

Cone 1: 1

Cone 2: 3

Overhead view of train cones image 5:

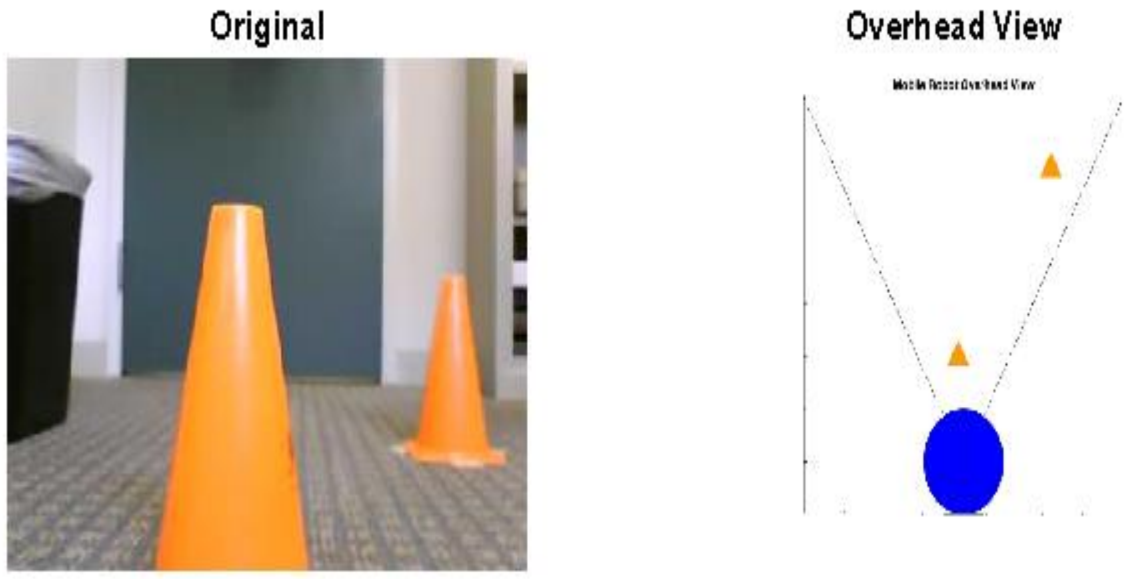


Figure 17 - Overhead_View_Train_cone_image5

>>>Centroid coordinates:

Cone 1: [347.74, 430.56]

Cone 2: [682.64, 391.65]

Orientation angles (degrees):

Cone 1: 3.79°

Cone 2: -21.46°

Proximity levels:

Cone 1: 1

Cone 2: 3

Part 4: Testing Blob Analysis & Target Detection:

The code processes a set of test images to detect, localize, and map orange traffic cones within the robot's field of view. The key functionalities include colour detection using HSV segmentation, blob analysis for identifying connected regions, centroid extraction, and calculating the angular positions and proximities of cones relative to the robot.

Specifically, the code employs HSV colour-based segmentation to isolate orange cones, followed by blob analysis to identify valid regions corresponding to cones based on size thresholds. For each detected cone, it extracts the centroid and calculates its angular offset and proximity relative to the robot's centre line. The results are visualized through direct image annotations highlighting centroids and angular positions, as well as an overhead view illustrating the cones' relative positions in the robot's operating environment.

Explanation of the Code

The code is designed to detect and localize cones in a set of test images by leveraging HSV color-based segmentation and blob analysis. It processes the test images to identify the centroids of detected cones, calculates their angular positions and proximities relative to the robot, and visualizes the results in both direct and overhead views.

Code Workflow

1. **Image Input and Preprocessing**:
   - The system reads images from the \ME598_ImgProcLab_MATLAB\TestConeImages\ folder.
   - Converts RGB images to HSV color space to isolate orange cones using predefined thresholds obtained in earlier parts.
2. **Binary Mask Creation:**
   - Applies the HSV thresholds to create a binary mask for orange cone detection.
   - Refines the binary mask through morphological operations to remove noise and fill holes.
3. **Blob Analysis:**
   - Identifies connected components in the binary mask.
   - Extracts properties such as area and centroid of each component.
   - Filters regions based on a minimum area threshold to exclude irrelevant detections.
4. **Angle and Proximity Calculation:**
   - Calculates angular offsets of cones from the robot's centerline using the centroids' horizontal positions relative to the image center.
   - Estimates proximity based on the centroids' vertical positions within the image, categorizing distances as near, medium, or far.
5. **Testing and Adjustments:**
   - The code is tested on new images from the test set, ensuring it generalizes well to previously unseen data.
   - Adjustments to HSV thresholds and morphological operations are made as needed to improve accuracy.

- Robustness to variations, such as different cone orientations and lighting conditions, is evaluated.

6. **Visualization:**
   - Marks cone centroids and overlays angle annotations on the processed images.
   - Uses the overheadView() function to generate an overhead view showing the relative position of detected cones with respect to the robot.
7. **Saving Results:**
   - Saves the processed images with visualized centroids and the overhead view plots for analysis.

Copy of commented code

The Matlab codes of centroid detection, color threshold and overhead view of the cones are provided in the **Appendix D and Appendix E** for your reference.

Results

Color-detected images with computed centroids:

i. **Centroid for test image 1:**

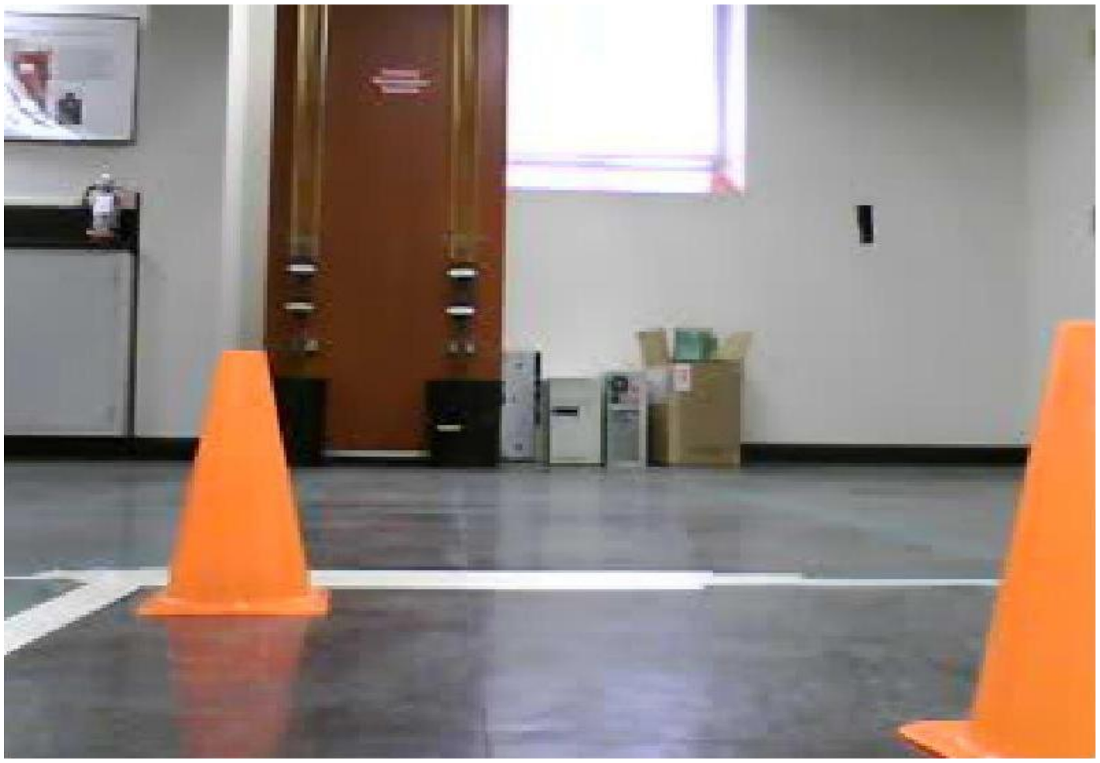


Figure 18- Test Cone Image 1

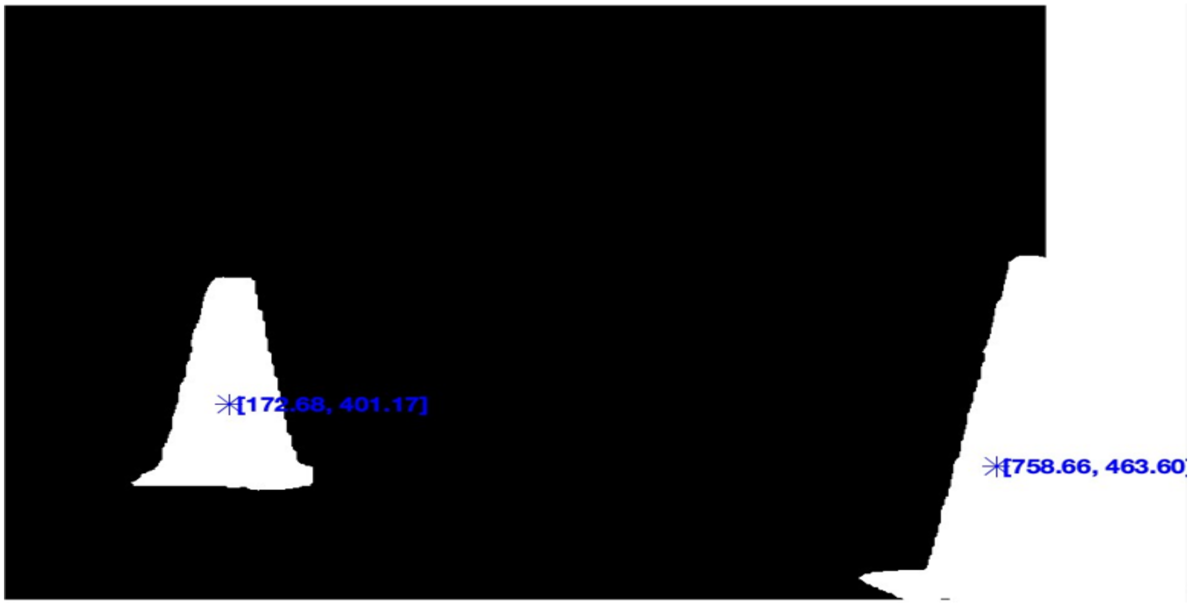


Figure 19-Processed Test Cone Image 1 with centroid

Centroid coordinates:

[758.66, 463.60]

[172.68, 401.17]

ii. Centroid for test image 2:

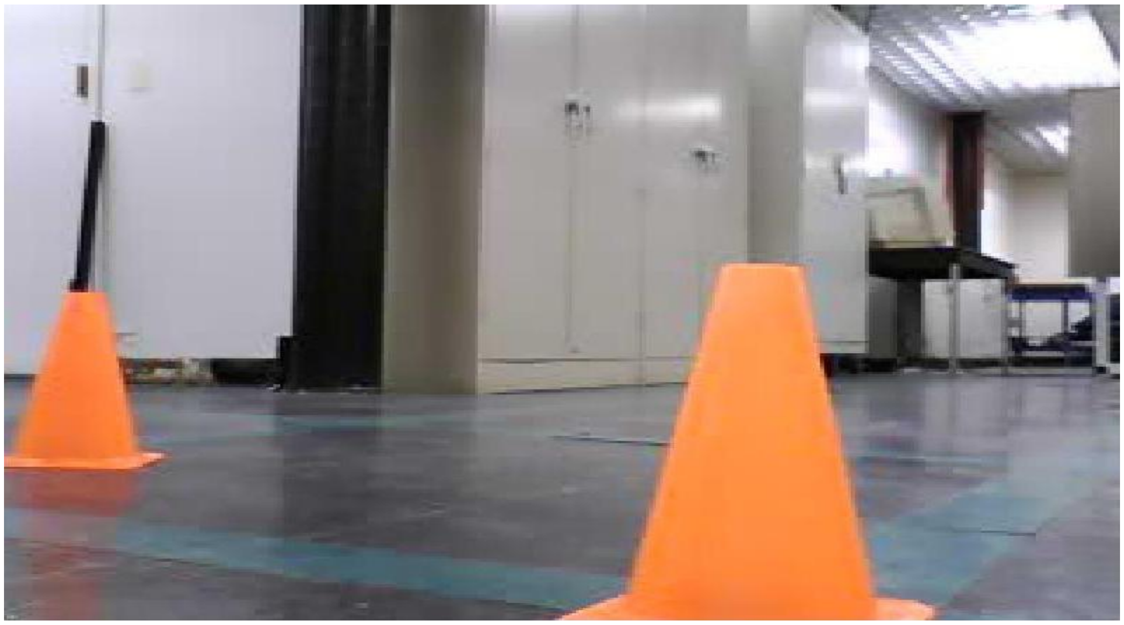


Figure 20-Test Cone Image 2

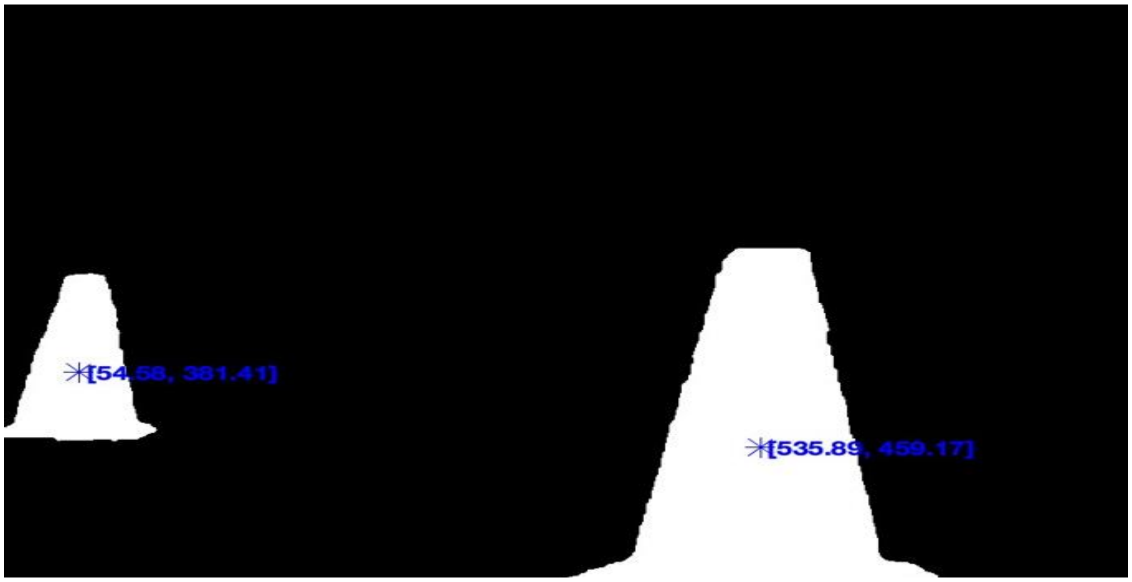


Figure 21-Processed Test Cone Image 2 with centroids

Centroid coordinates:
[535.89, 459.17]
[54.58, 381.41]

iii. **Centroid for test image 3:**

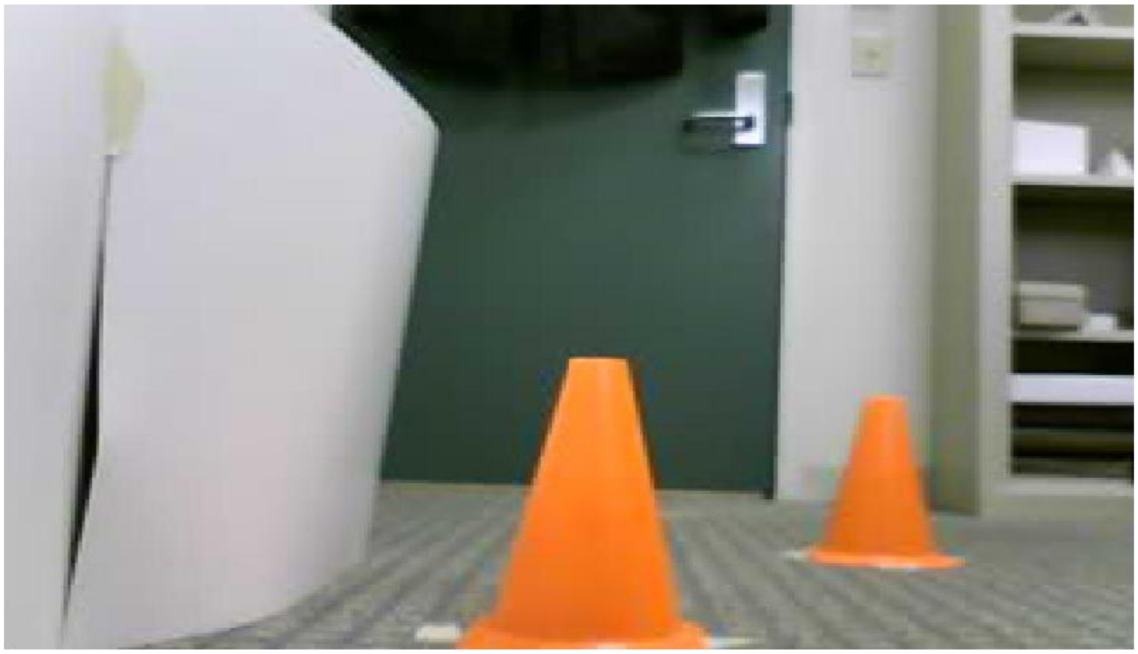


Figure 22-Test Cone Image 3

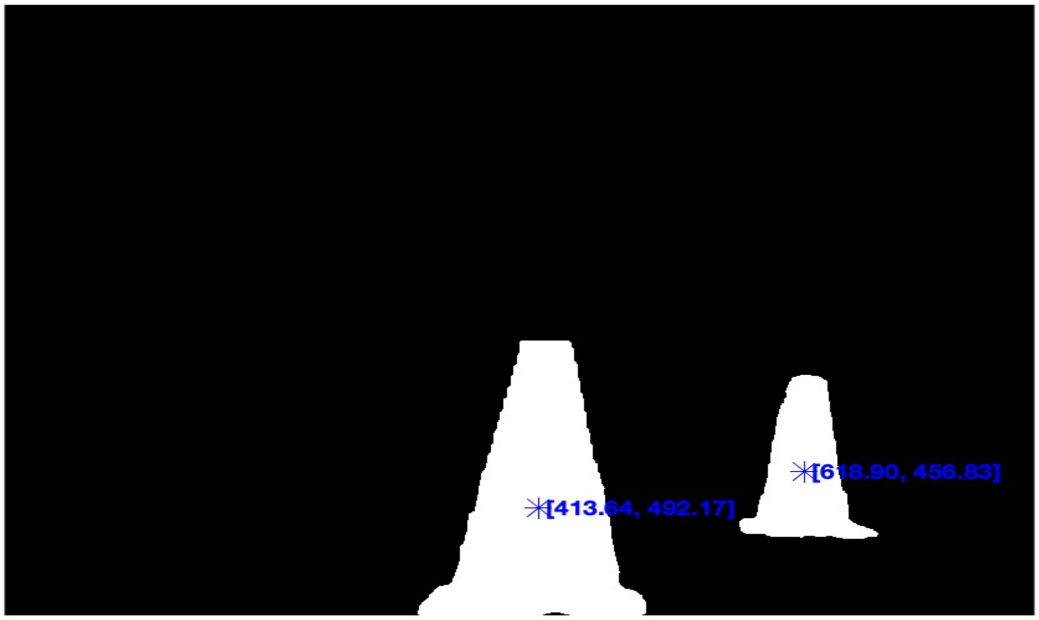


Figure 23-Processed Test Cone Image 3 with Centroids

Centroid coordinates:

[413.64, 492.17][618.90, 456.83]

iv. **Centroid for test image 4:**

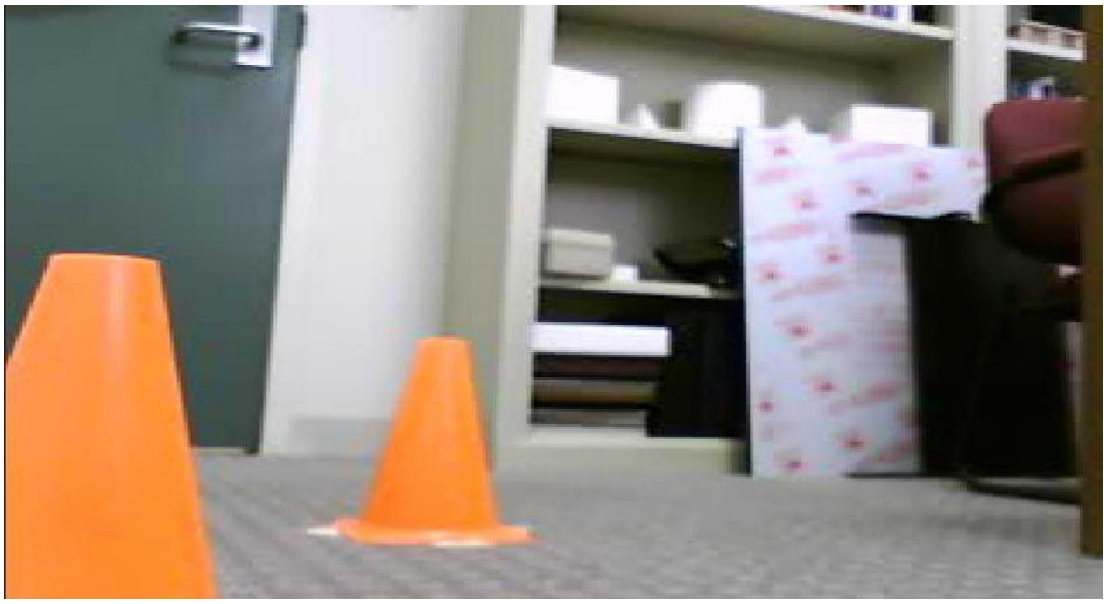


Figure 24-Test Cone Image 4

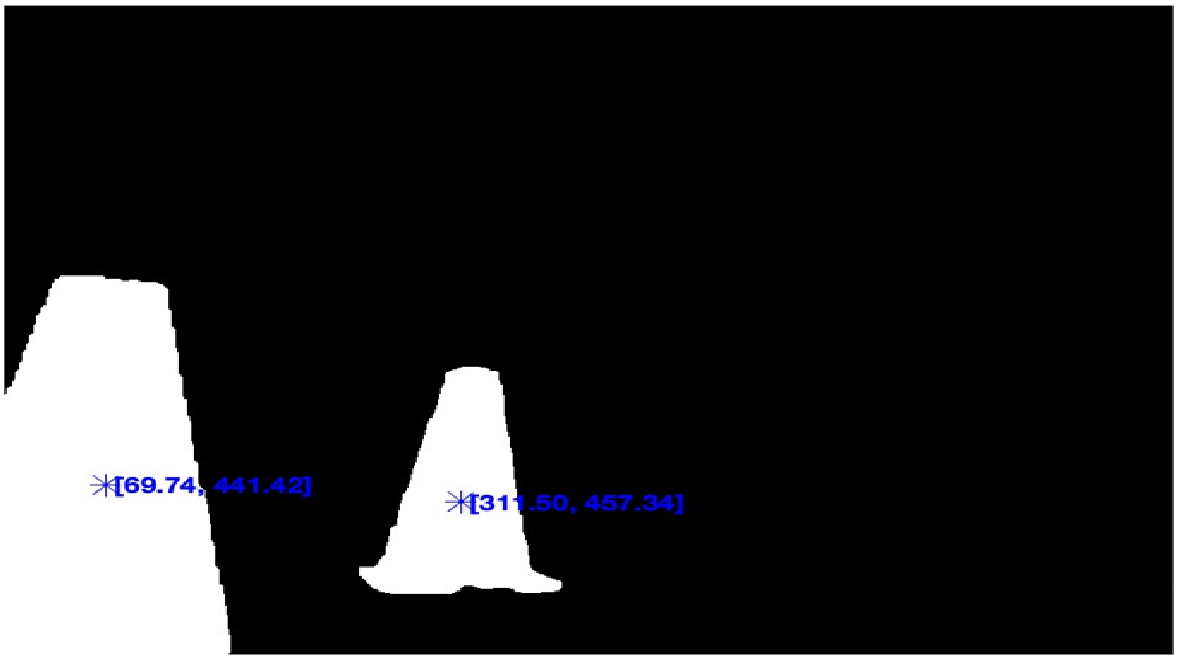


Figure 25-Processed Test Cone Image 4

Centroid coordinates:
[69.74, 441.42]
[311.50, 457.34]

v. **Centroid for test image 5:**

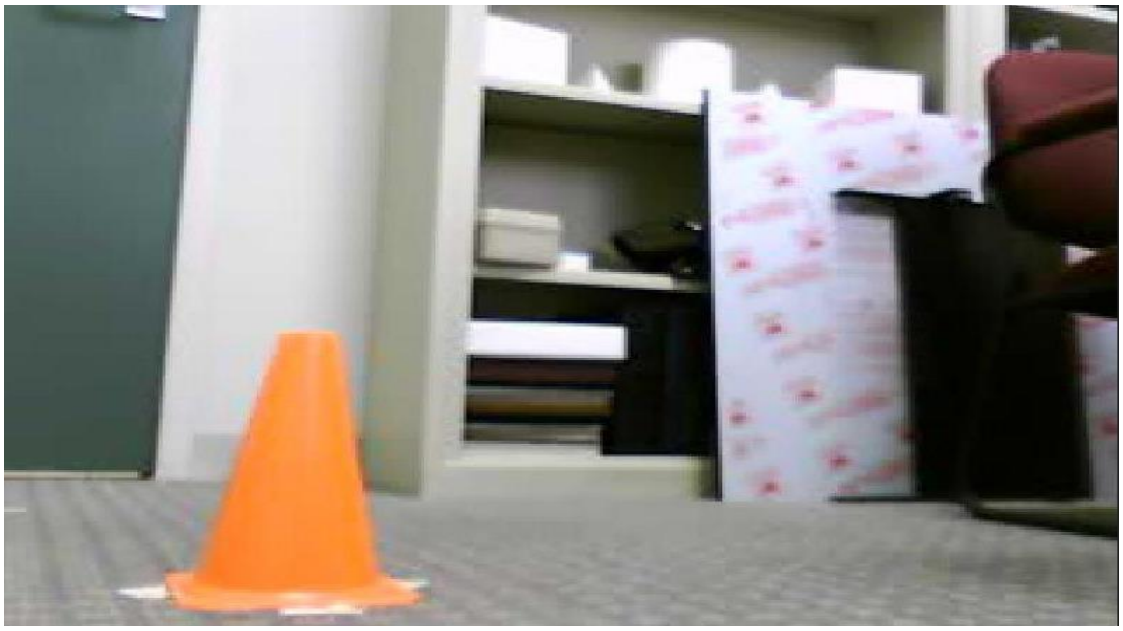


Figure 26-Test Cone Image 5

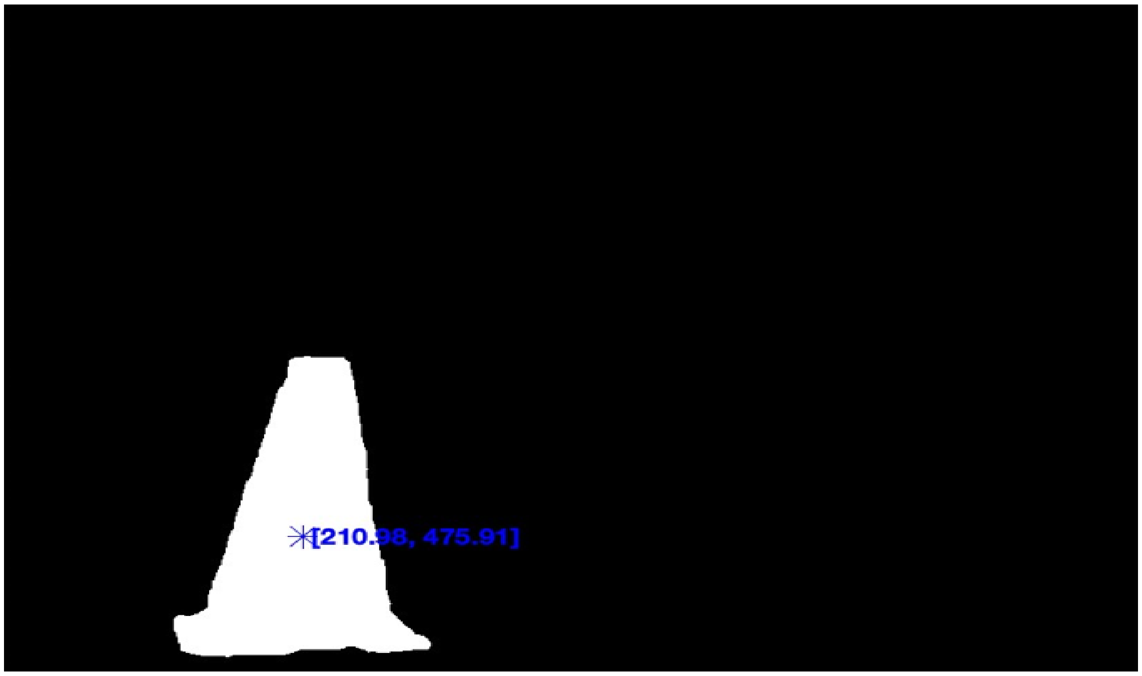


Figure 27-Processed Test Cone Image 5 with Centroids

Centroid coordinates:
[210.98, 475.91]

How we adapted the algorithm to determine cone angle and relative proximity:

<u>Calculating Cone Angle</u>:

- The angular position of each cone was calculated based on its horizontal centroid position relative to the image center.

a) **Using Image Centre as a Reference:**

- The horizontal centre of the image (ImageCenterX) was computed as half the total number of columns in the image.
- This value served as the reference point for calculating angular offsets.

b) **Computing Horizontal Offset:**

- The horizontal distance of the cone's centroid (CentroidX) from the image centre was measured.
- The offset (Xdiff = CentroidX - ImageCenterX) was used to determine the angular position of the cone.
- Given the camera's 60° horizontal field of view (FOV), the offset was scaled linearly to map Xdiff to angles ranging from +30° (leftmost pixel) to -30° (rightmost pixel).

<u>Calculating Relative Proximity:</u>

- The relative proximity of cones was estimated based on their vertical centroid positions in the image, with cones farther from the robot appearing higher in the image.

a) **Image Height as a Reference:**

- The total number of rows (nr) in the image was used to determine the cone's vertical position relative to the camera.

b.) Vertical Normalization:

- The vertical centroid position (CentroidY) is scaled to a range between 1 (near) and 3 (far).

c.) Interpretation:

- A proximity value of 1 indicates that the cone is close to the robot.
- A value of 3 indicates that the cone is farther away.

Handling Multiple Cones:

- The algorithm processed each cone independently, calculating its angle and proximity.
- Results were stored in arrays, capturing the angular positions and proximities of all detected cones.

Testing Phase Refinement For Robustness:

Dynamic Resolution Adaptation:

- Calculations for angles and proximities dynamically adjusted to the resolution of each test image, ensuring consistent performance regardless of image dimensions.

Threshold Optimization:

- Proximity calculations were fine-tuned to account for variations in cone sizes across different test images, ensuring accurate distance estimation.

Robust Detection Across Test Cases:

- The algorithm handled diverse scenarios, including images with one or two cones, as well as varying cone orientations and lighting conditions, with minimal need for manual parameter adjustments.

Overhead view Figures for test images:

i. Overhead view of test cone image 5:

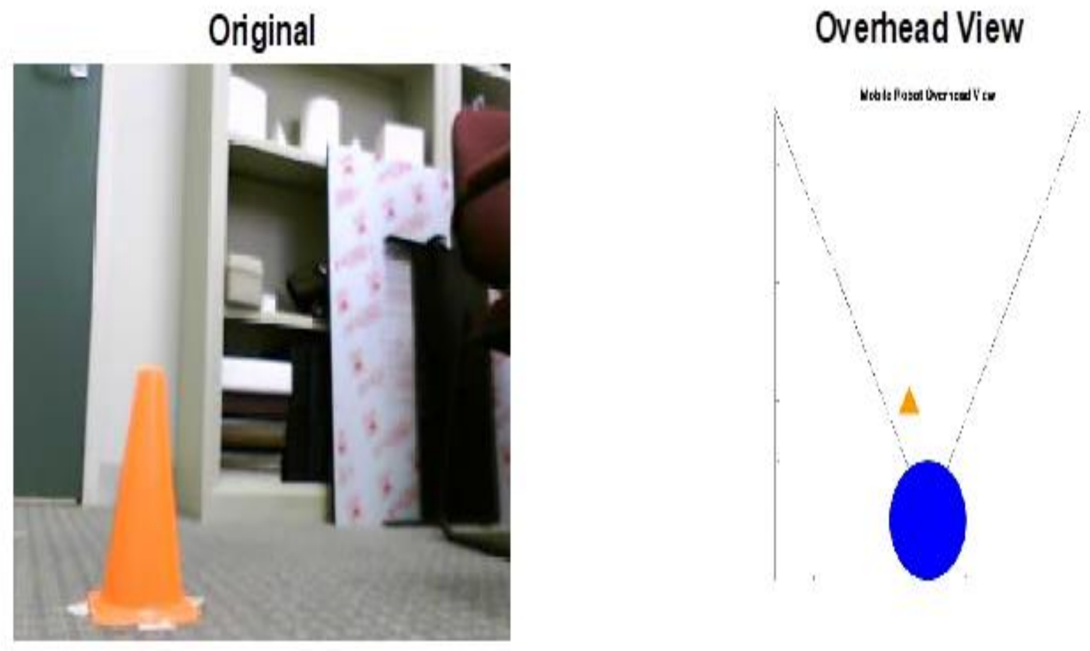


Figure 28-Overhead view of Test Cone Image 5

>>>Centroid coordinates:

Cone 1: [210.98, 475.91]

Orientation angles (degrees):

Cone 1: 14.10°

Proximity levels:

Cone 1: 1

Overhead view of test cone image 4:

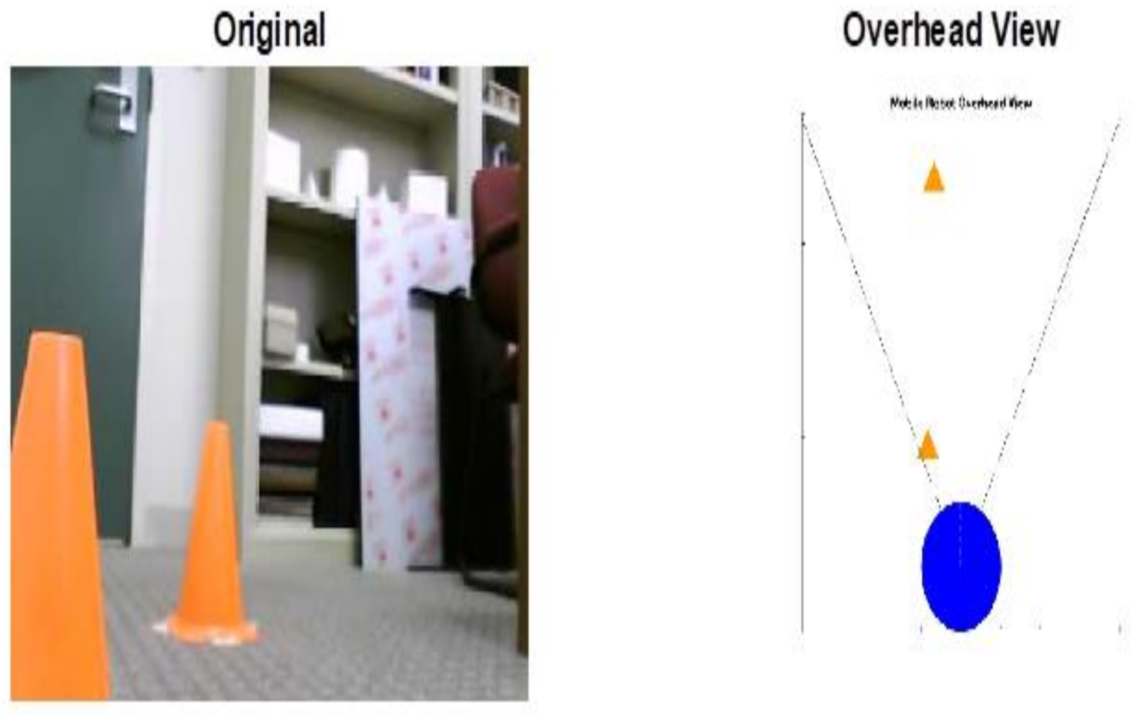


Figure 29-Overhead view of Test Cone Image 4

```
>>>Centroid coordinates:
    Cone 1: [69.74, 441.42]
    Cone 2: [311.50, 457.34]
    Orientation angles (degrees):
    Cone 1: 24.74°
    Cone 2: 6.52°
    Proximity levels:
    Cone 1: 1
    Cone 2: 3
```

Overhead view of test cone image 3:

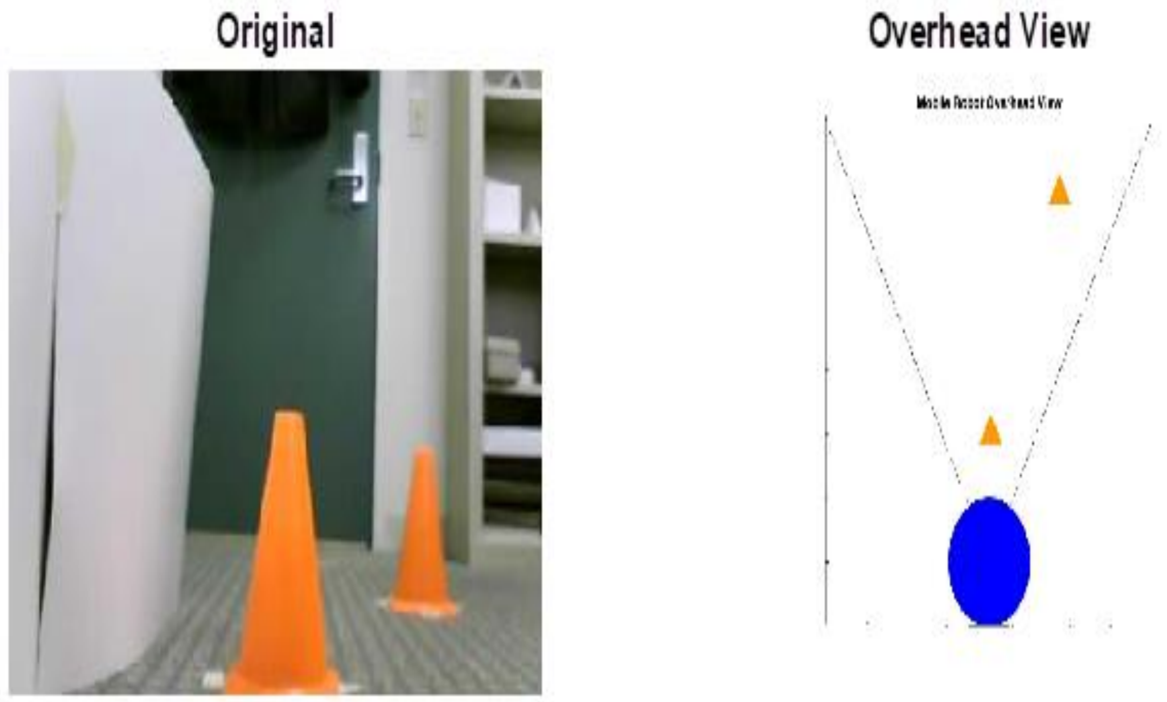


Figure 30- Overhead View of Test Cone Image 3

```
>>>Centroid coordinates:
    Cone 1: [413.64, 492.17]
    Cone 2: [618.90, 456.83]
    Orientation angles (degrees):
    Cone 1: -1.18°
    Cone 2: -16.65°
    Proximity levels:
    Cone 1: 1
    Cone 2: 3
```

Overhead view of test cone image 2:

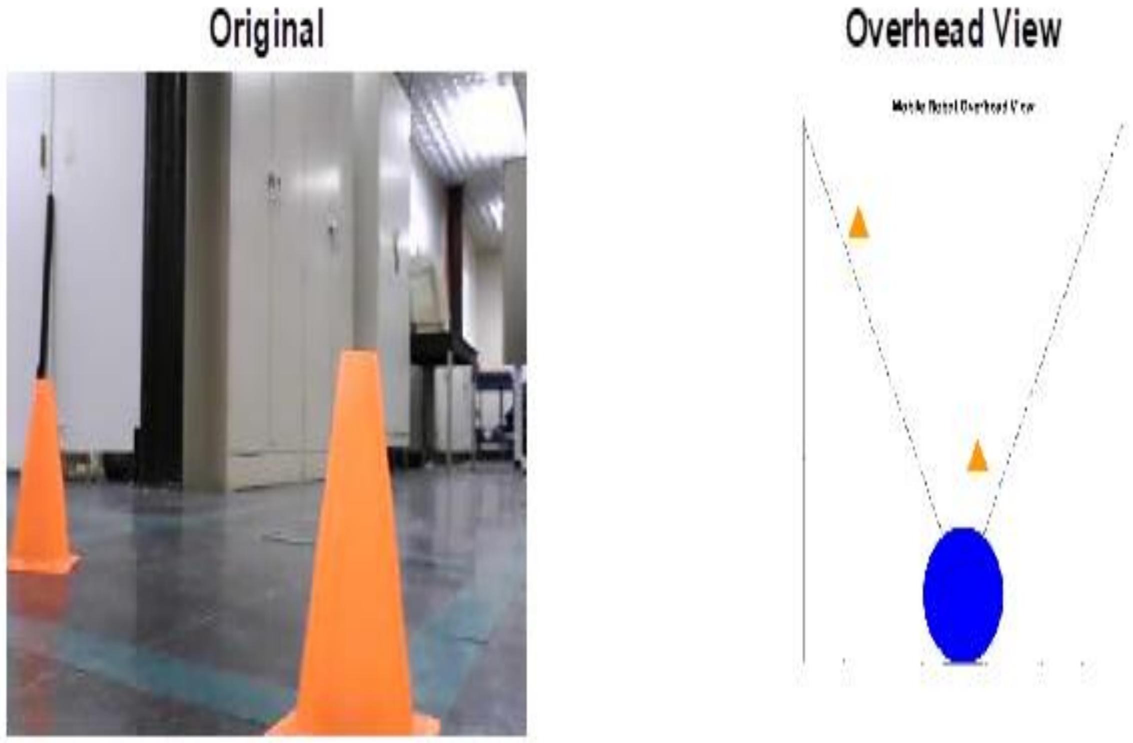


Figure 31- Overhead View of Test Cone Image 2

```
>>>Centroid coordinates:
    Cone 1: [535.89, 459.17]
    Cone 2: [54.58, 381.41]
    Orientation angles (degrees):
    Cone 1: -10.39°
    Cone 2: 25.89°
    Proximity levels:
    Cone 1: 1
    Cone 2: 3
```

Overhead view of test cone image 1:

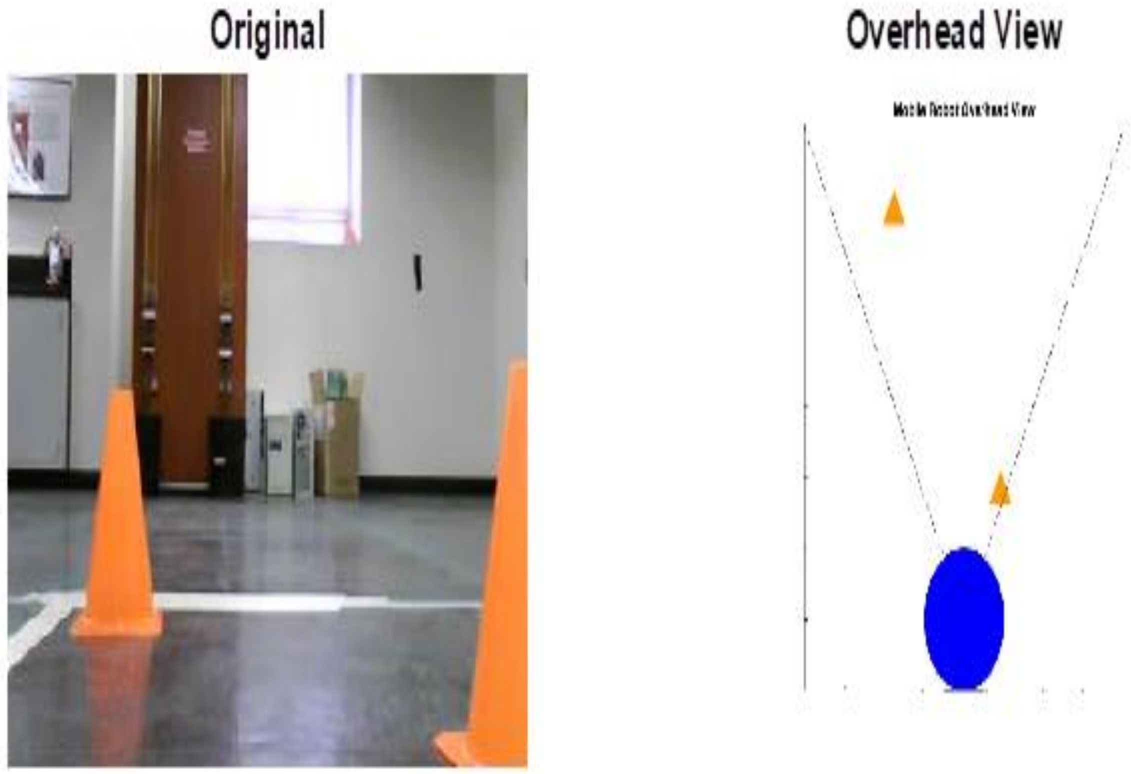


Figure 32- Overhead View of Test Cone Image 1

```
>>>Centroid coordinates:
    Cone 1: [758.66, 463.60]
    Cone 2: [172.68, 401.17]
    Orientation angles (degrees):
    Cone 1: -27.19°
    Cone 2: 16.98°
    Proximity levels:
    Cone 1: 1
    Cone 2: 3
```

Part 5: Vision-Based Detection with Newly Captured Images:

a) Capturing original images:

We successfully captured images of orange traffic cones using my laptop webcam and saved them to the computer via MATLAB. And the images were successfully saved in the current working directory of MATLAB and are ready for further processing or analysis.

```
>>> figure(1)
>>> saveas(gcf,'Image1','jpg')
```

The captured images are saved with unique names, so that the Matlab code runs smoothly.

b)

Description of the Code: Vision-Based Detection with Newly - Captured Images

1. **Image Acquisition**:

- Using the laptop webcam, images with one or two cones in view were captured and saved in MATLAB as .jpg files with unique filenames. These images were moved into a designated directory for further processing.

2. **Processing the Images**:

- The same HSV-based color detection function (coneThreshold) developed earlier was applied to the new test images to isolate the cones.
- Steps include:
  I. Converting images from RGB to HSV color space.
  II. Applying predefined HSV thresholds to create binary masks for detecting orange regions corresponding to cones.
  III. Using morphological operations (e.g., noise removal, filling holes) to refine the binary mask.

3. **Blob Analysis**:

- Detected regions in the binary masks were analyzed to identify connected components.
- Properties such as centroid, area, and bounding box of the connected components were extracted.
- Regions with areas below the threshold were ignored to eliminate noise.

4. **Cone Localization**:

- For valid detected regions:

  I. The horizontal position of the centroid relative to the image's center was used to compute the angle of the cone with respect to the robot's centerline.
  II. The vertical position of the centroid was used to estimate proximity (values 1 for near, 3 for far).

5. **Visualization**:
   - Each processed image showed the detected cones, marked centroids, and calculated angular offsets.
   - The overhead view (`overheadView`) function plots cone positions relative to the robot, illustrating their angular orientation and proximity in a 60° field of view.

6. **Adjustments**:
   - Parameters such as HSV thresholds and area thresholds were fine-tuned to handle variations in lighting and cone visibility in the new images. Its manually done for the clear visualization.

- **The MATLAB code used for this task is provided in Appendix A, Appendix D, and Appendix E for your reference.**

Results

Figures of color-detected images with computed centroids

Captured Image 1:

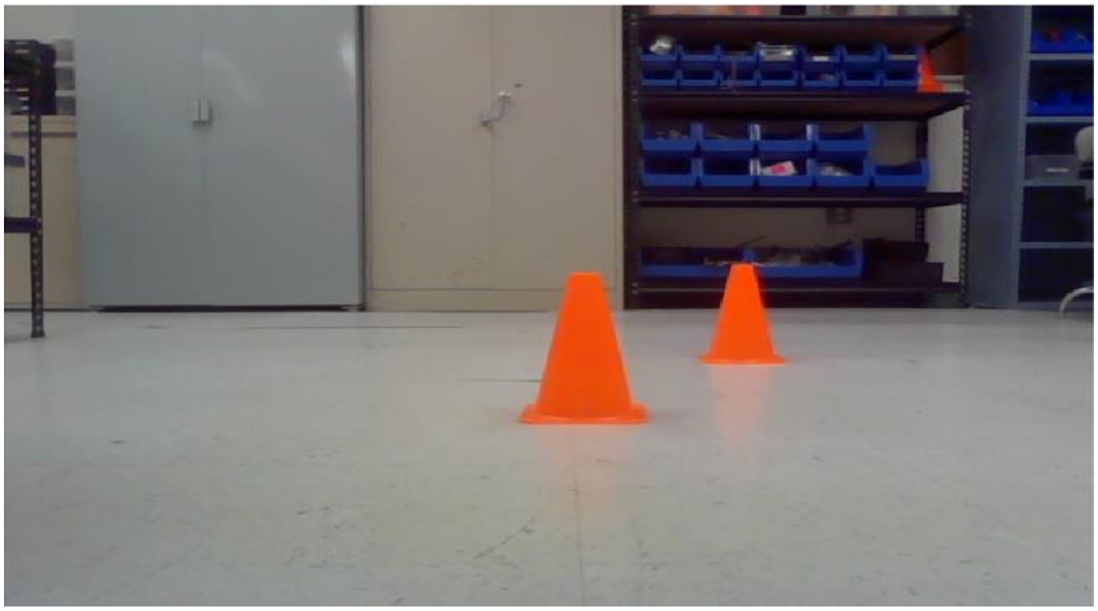


Figure 33 – Captured Image 1

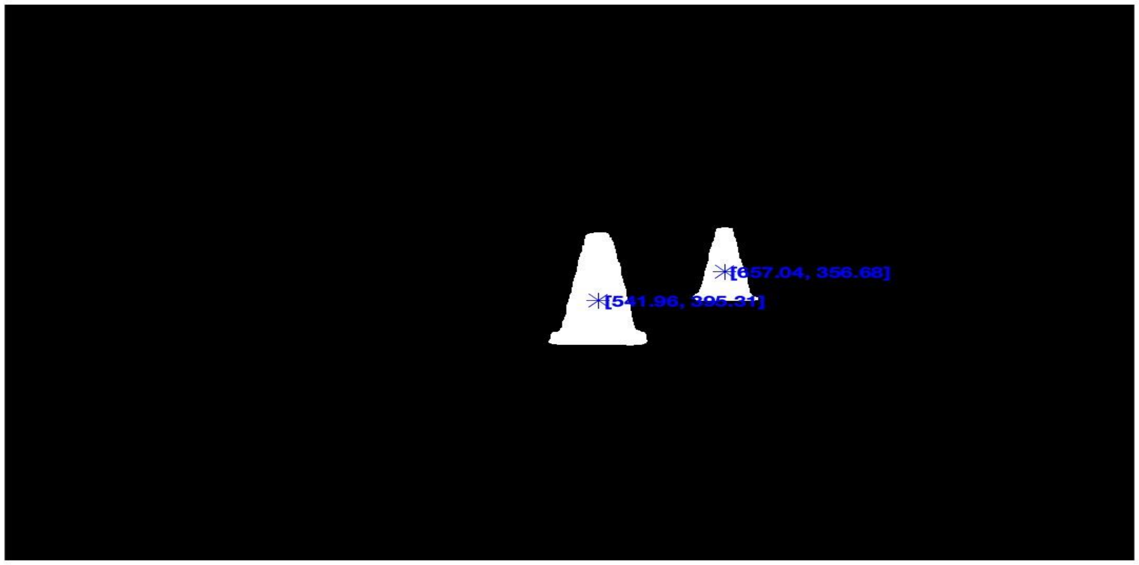


Figure 34 – Processed Image 1 with centroid points

Centroid coordinates:
[514.96, 395.31] [657.04, 356.68]

Captured Image 2:

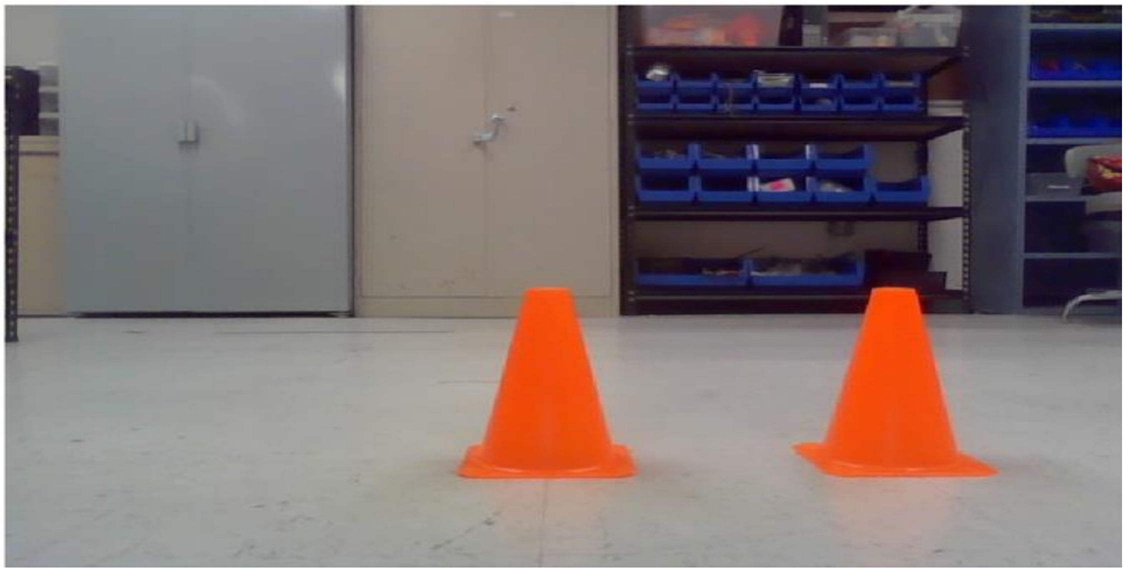


Figure 35 – Captured Image 2

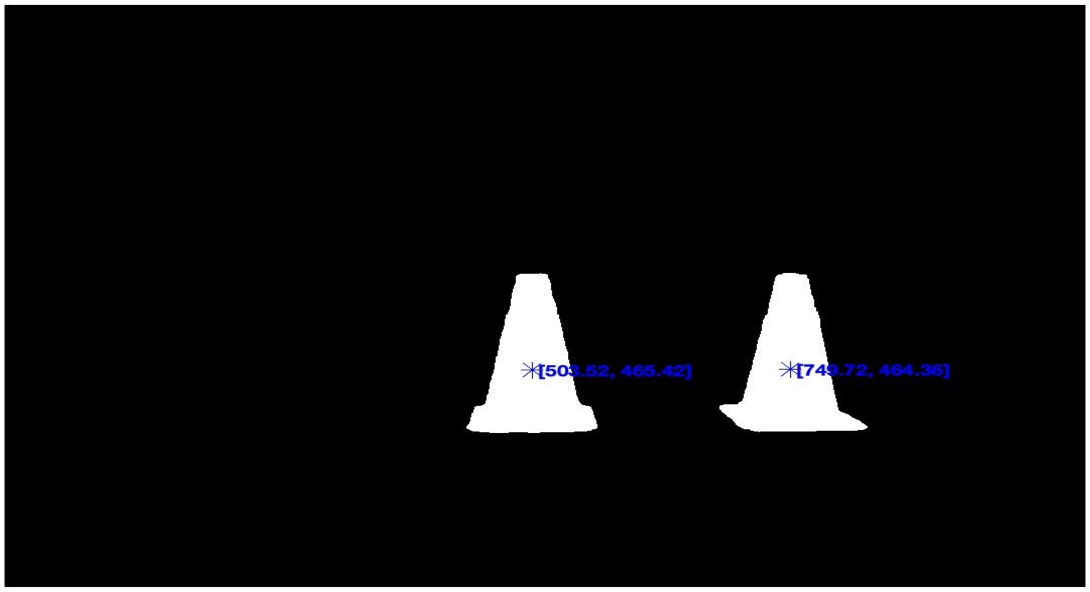


Figure 36 – Processed Image 2 with centroid points

Centroid coordinates:
[503.52, 465.42]
[749.72, 464.36]

Captured Image 3:

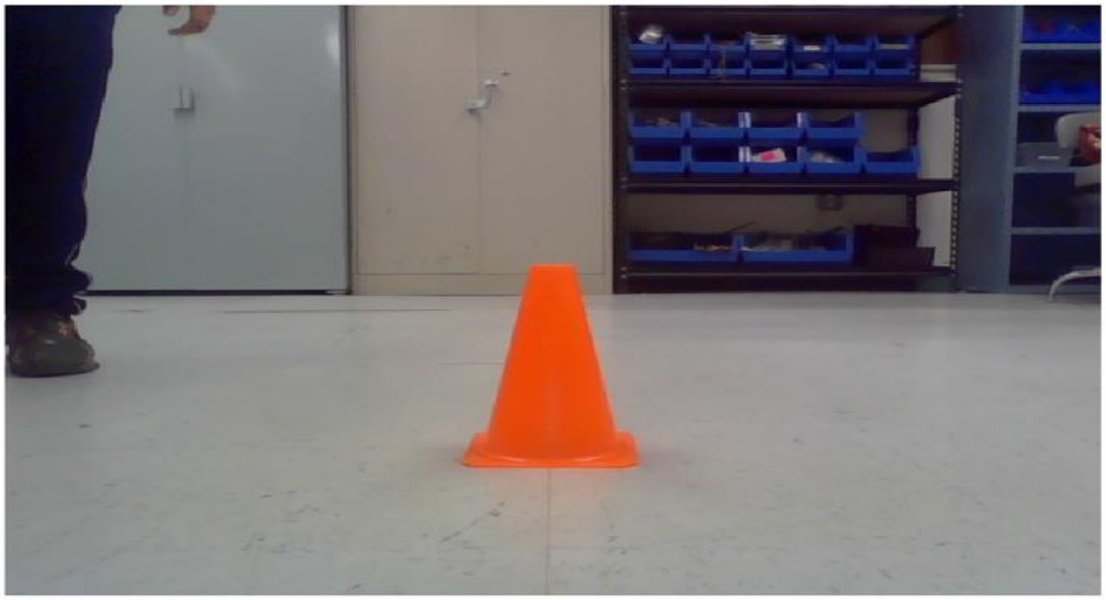


Figure 37 – Captured Image 3

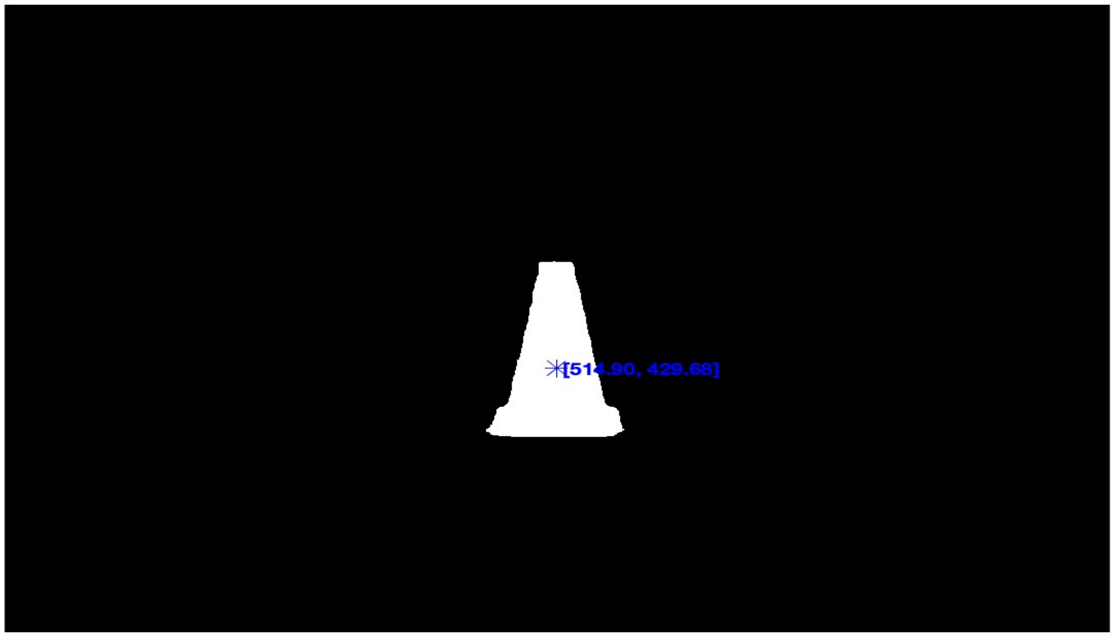


Figure 38 – Processed Image 3 with centroid points

Centroid coordinates:
[514.90, 429.68]

Captured Image 4:

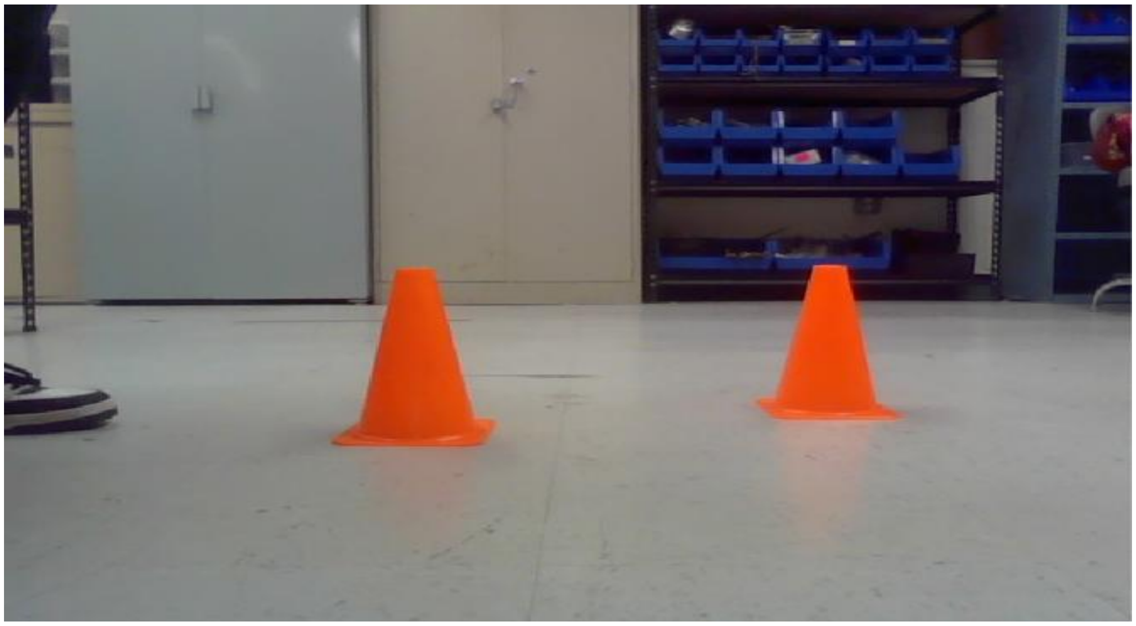


Figure 39 – Captured Image 4

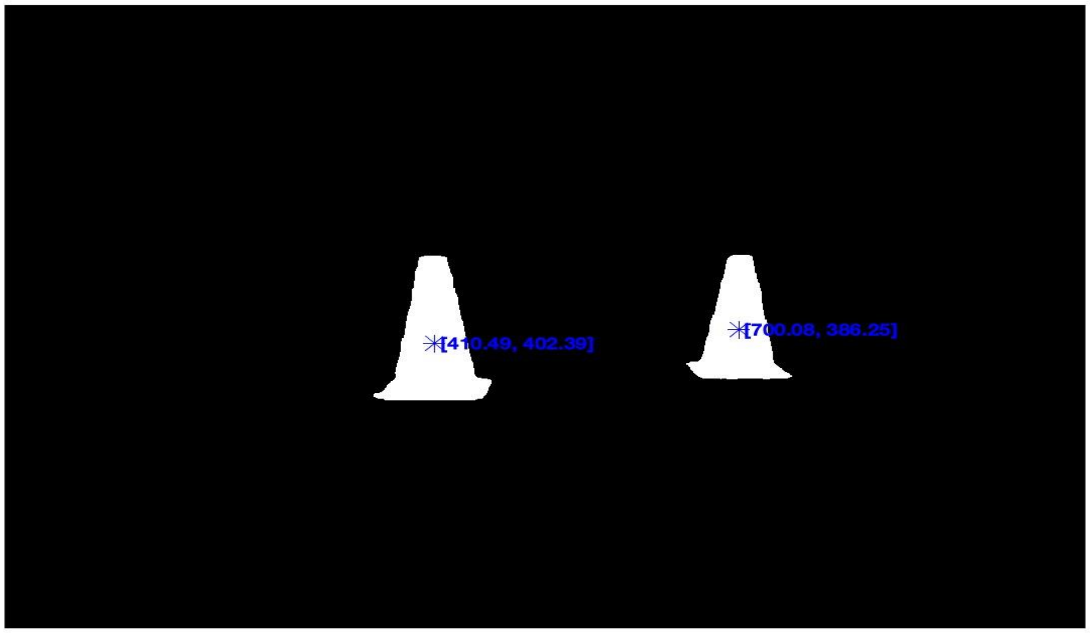


Figure 40 – Processed Image 4 with centroid points

Centroid coordinates:
[410.49, 402.39]
[700.08, 386.25]

Captured Image 5:

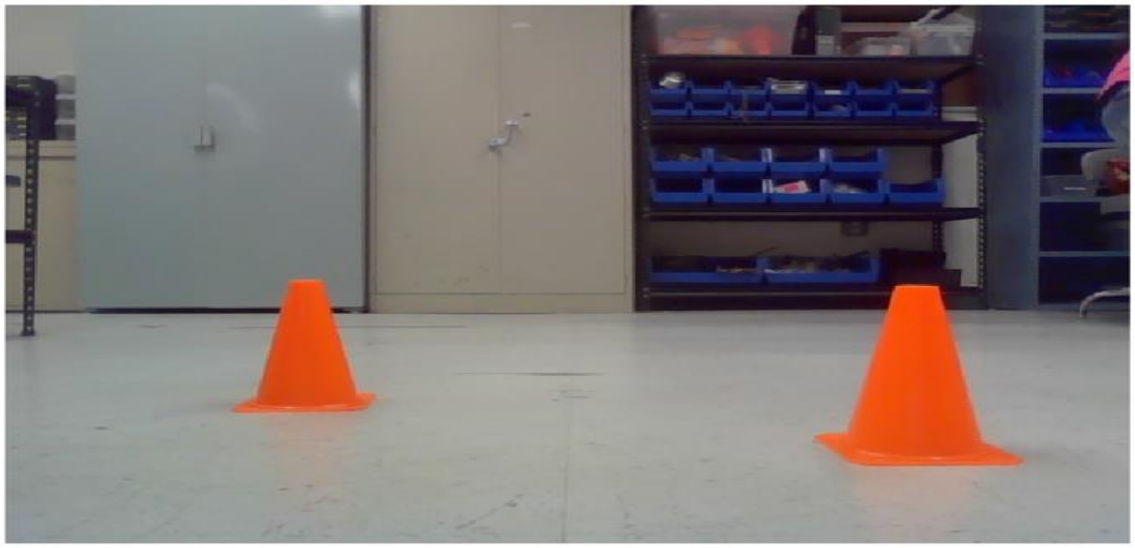


Figure 41 – Captured Image 5

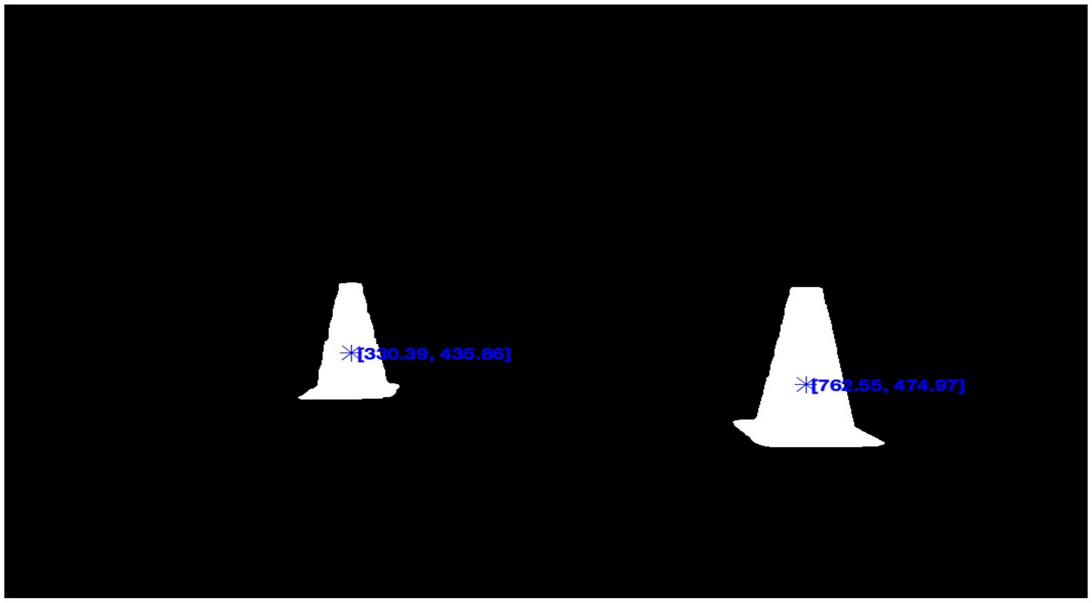


Figure 42 – Processed Image 5 with centroid points

Centroid coordinates:
[330.39, 435.86]
[732.55, 474.97]

Figures of overhead views:

Overhead View for Image 1 of 2 Cones:

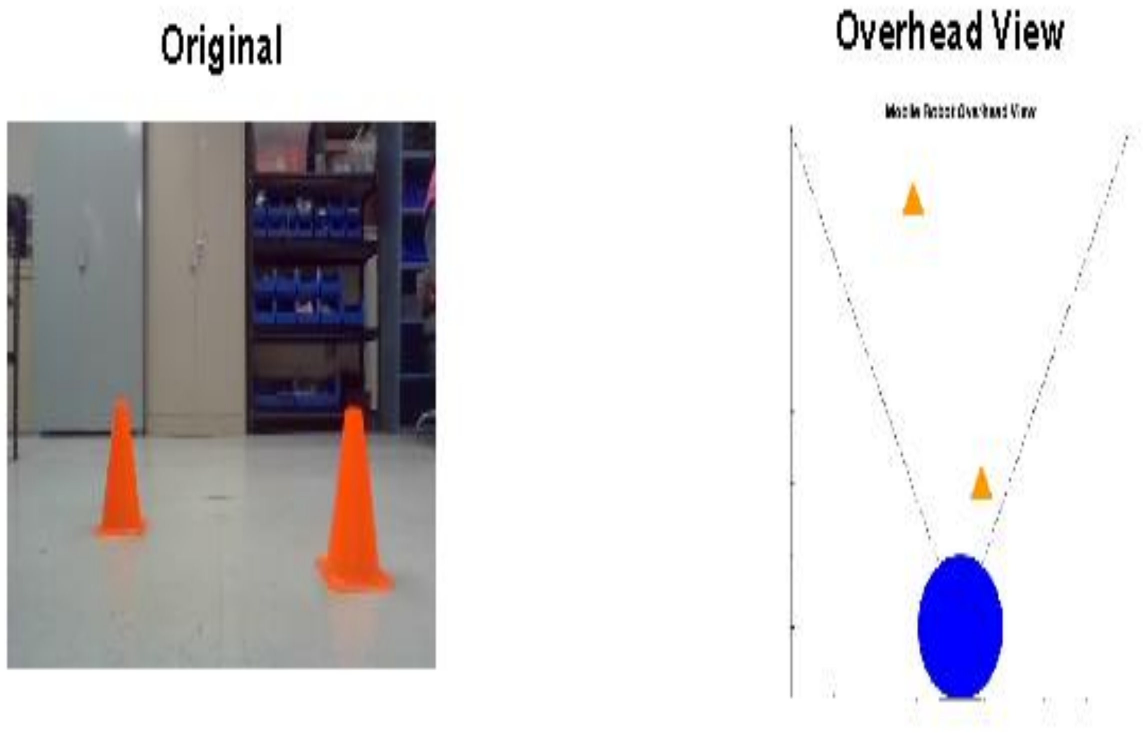


Figure 43 – Overhead View captured for image 1

```
>>>Centroid coordinates:
    Cone 1: [762.55, 474.97]
    Cone 2: [330.39, 435.86]
    Orientation angles (degrees):
    Cone 1: -14.42°
    Cone 2: 10.75°
    Proximity levels:
    Cone 1: 1
    Cone 2: 3
```

Overhead View for Image 2 of 2 Cones:

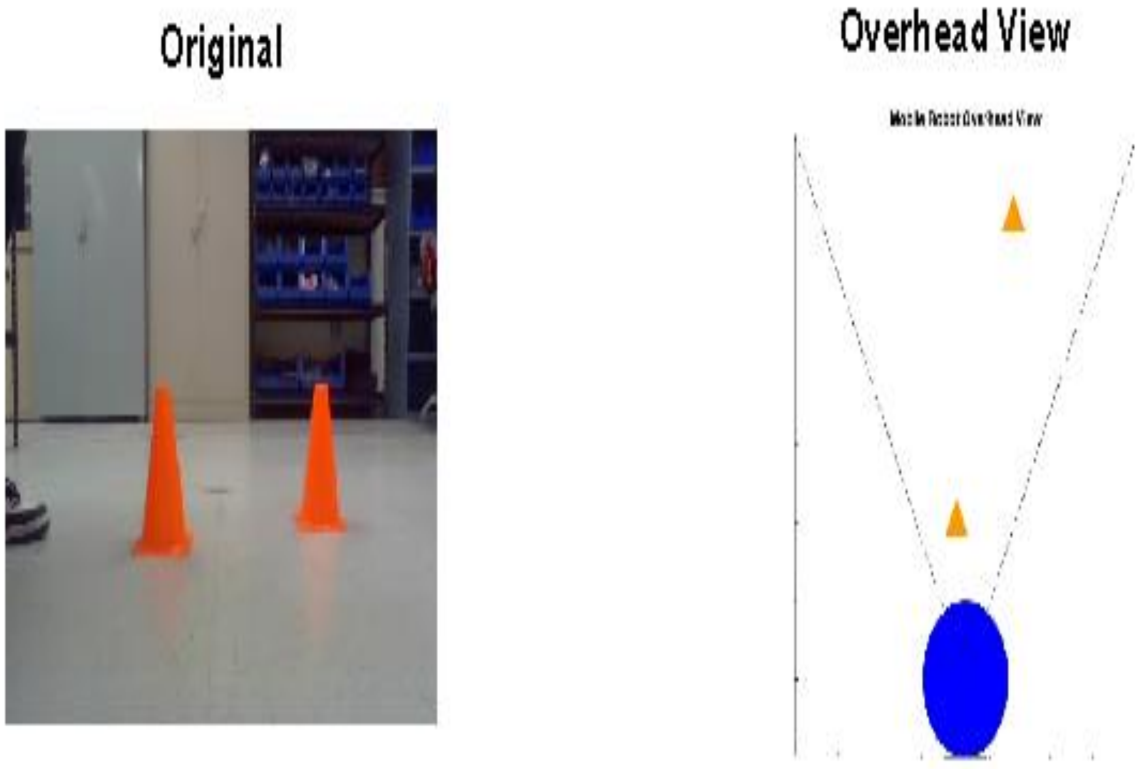


Figure 44 – Overhead View captured for image 2

```
>>>Centroid coordinates:
    Cone 1: [410.49, 402.39]
    Cone 2: [700.08, 386.25]
    Orientation angles (degrees):
    Cone 1: 6.09°
    Cone 2: -10.78°
    Proximity levels:
    Cone 1: 1
    Cone 2: 3
```

Overhead View for Image 3 of 1 Cone:

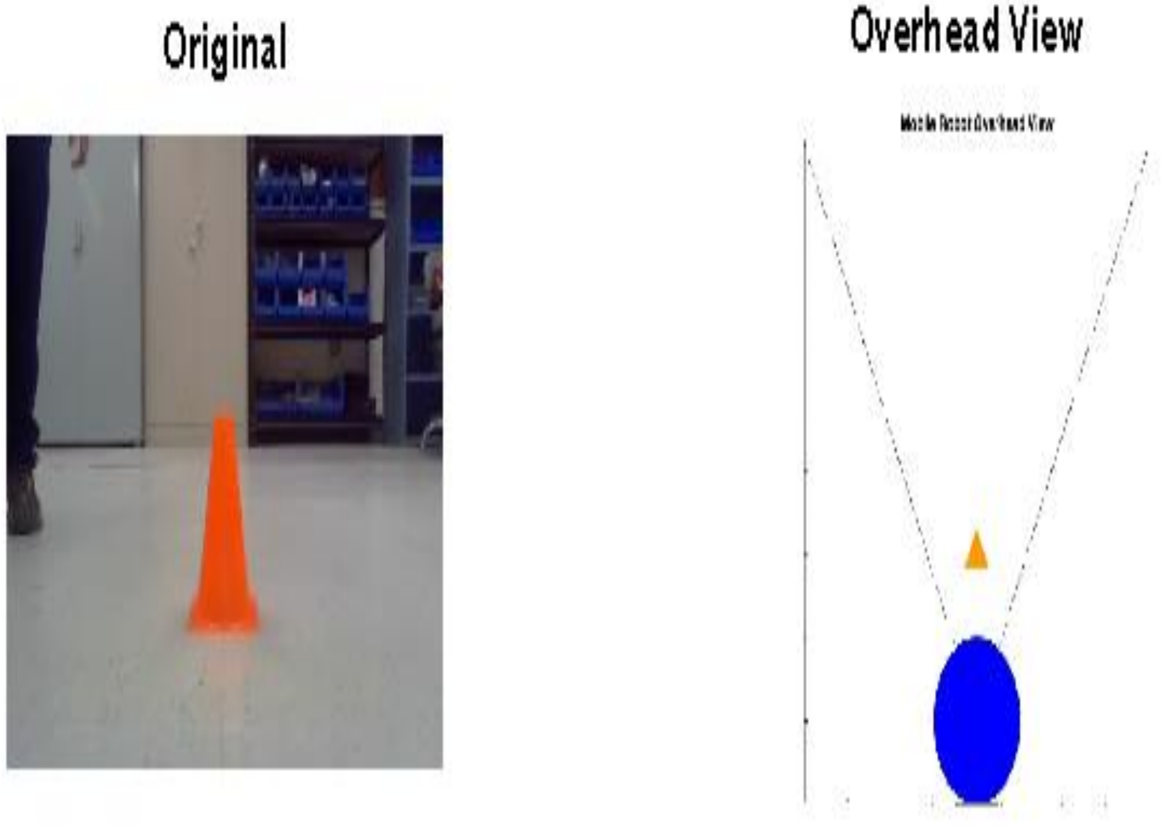
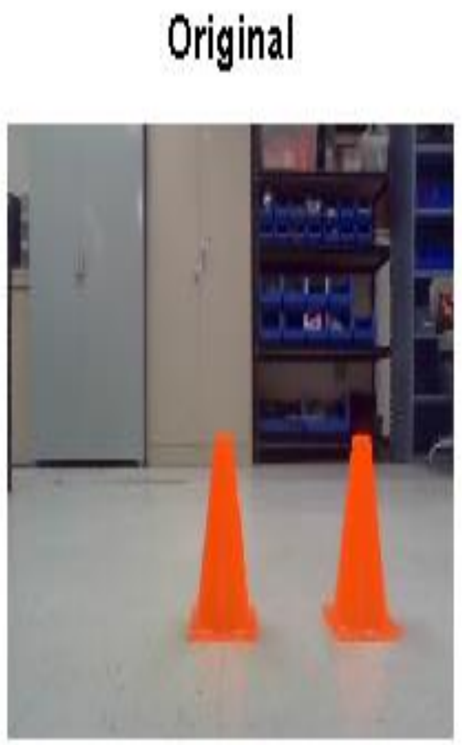


Figure 45 – Overhead View captured for image 3

>>>Centroid coordinates:
  Cone 1: [514.90, 429.68]
  Orientation angles (degrees):
  Cone 1: 0.01°
  Proximity levels:
  Cone 1: 1

Overhead View for Image 4 of 2 Cones



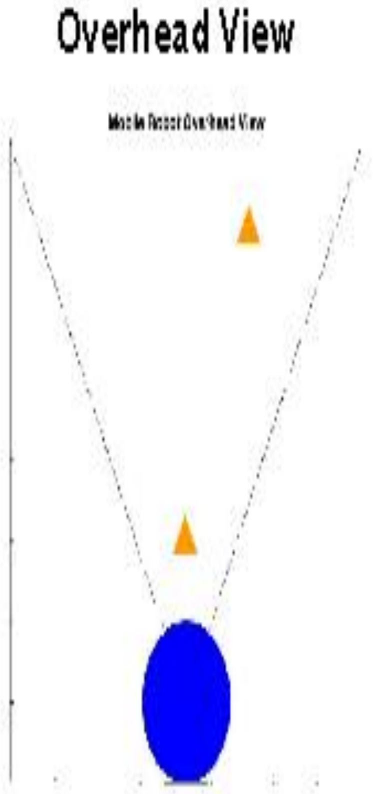


Figure 46 – Overhead View captured for image 4

```
>>>Centroid coordinates:
    Cone 1: [503.52, 465.42]
    Cone 2: [749.72, 464.36]
    Orientation angles (degrees):
    Cone 1: 0.67°
    Cone 2: -13.67°
    Proximity levels:
    Cone 1: 1
    Cone 2: 3
```

Overhead View for Image 5 of 2 Cones

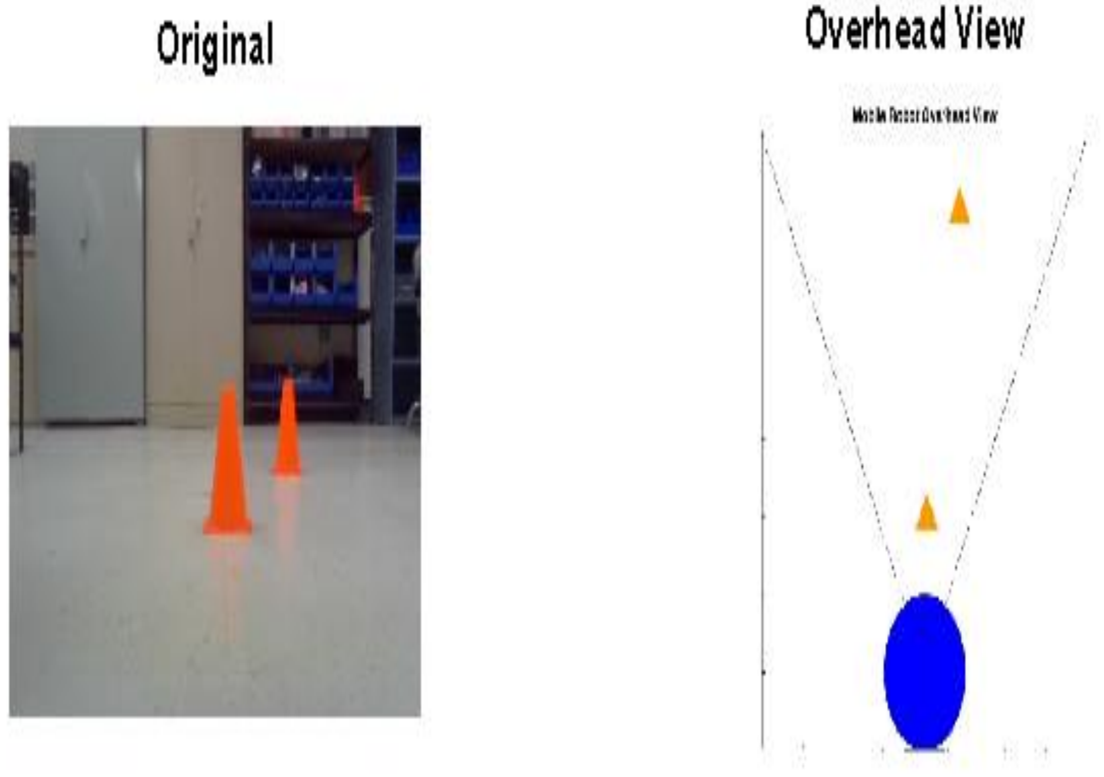


Figure 47 – Overhead View captured for image 5

```
>>> Centroid coordinates:
    Cone 1: [541.96, 395.31]
    Cone 2: [657.04, 356.68]
    Orientation angles (degrees):
    Cone 1: -1.57°
    Cone 2: -8.27°
    Proximity levels:
    Cone 1: 1
    Cone 2: 3
```

The code and tuned parameters were successful in processing the newly captured images, though minor adjustments were required. Below are the details:

- The code successfully detected cones in most of the newly captured images, accurately identifying centroids and calculating angular positions. The cone Threshold function and blob analysis worked effectively in varying positions and distances of the cones.

1. **Adjustments to Tolerances**:
   Adjustments were necessary for the area threshold during blob analysis to filter out smaller noise regions caused by variations in the background. Morphological operations, such as noise removal and hole filling, were refined slightly to handle lighting inconsistencies in the captured images.
2. **Retraining HSV Values**:
   The HSV thresholds established during initial training were mostly sufficient. However, the lower bound of the V-channel was increased to improve performance under varying lighting conditions in the new environment. This adjustment minimized false positives and improved cone detection accuracy.
3. **Difficulties that we encountered**:
   - **Lighting Issues**: Inconsistent lighting caused shadows and glare, which occasionally led to false positives or missed detections. This was addressed by refining the HSV thresholds and using morphological operations.
   - **Partial Cones in the Field of View**: Images with cones partially outside the camera's field of view sometimes created small blob regions, which were ignored using the adjusted area threshold.
4. **The overall performance**:
   The code demonstrated reliable performance in detecting and localizing cones. With minor adjustments, it was able to adapt to the new images and maintained accuracy in various scenarios.

Discussion:

Part 1: Discussion of Camera Setup

Setting up a camera in MATLAB to facilitate image acquisition—a prerequisite for every image processing task—was the main goal of this lab section. We demonstrated the ability to take real-time photographs by building a webcam object. For the best balance between processing speed and image quality, we started with a resolution of 640x480. By taking pictures with the `snapshot` function, we were able to show the data in a MATLAB figure and visually confirm it. The saved photos served as a point of reference for additional examination. This configuration is essential because it guarantees that we can reliably obtain images for processing from a regulated source, which promotes repeatability in the subsequent stages of image detection.

Part 2: Configuring the Color Detector

The setup of a color detection system to recognize particular objects—in this case, orange traffic cones based on color was covered in this section. We were able to experiment with various HSV values that best separated the orange hue from other aspects of the image by using training photos. To attain accurate detection, we adjusted the hue, saturation, and value parameters using MATLAB's Color Thresholder App. The end product was a MATLAB code that guaranteed constant color segmentation by automatically detecting these cones in a variety of images. Iterative refinement was required to calibrate the color detection such that it would generalize well across all the training photos that were supplied. This demonstrated the difficulty of modifying parameters to preserve accuracy in a variety of settings.

Part 3: Setting Up Target Detection & Blob Analysis

Blob analysis, a technique for locating and examining discrete areas in a picture, was added to the image processing in Part 3. Extracting the centroids of the cones found in Part 2 was the main objective. By removing tiny, pointless pixel clusters, filling in the blanks, identifying related parts, and extracting useful information like area and centroid coordinates, this procedure reduced noise. To visually confirm accuracy, the final centroid data was plotted over the color-detected images. Determining the spatial link between the detected cones and the camera was another goal of the landmark localization task. To evaluate the cones' orientation with respect to the camera's position, we used the computed centroids to generate angles and relative distances based on the horizontal field of view. Carefully chosen thresholds and strong parameter tweaking to accommodate different numbers of cones (from zero to two) are essential to the correctness of this study.

Part 4: Discussing Target Detection & Testing Blob Analysis

The blob analysis and target detection algorithms' resilience were examined in Part 4. We used a fresh image dataset to test the created image-processing pipeline. In order to detect overfitting or the need for modifications, this step was essential for confirming whether the adjusted parameters from previous stages applied to unseen photos. To account for different backgrounds or lighting circumstances in the new photos, slight adjustments were made to the thresholds or HSV values. The significance of a well-generalized model that sustains detection accuracy after the initial training set was highlighted by this testing step, which calls for a balance between specificity and parameter setting flexibility.

Part 5: Discussion of Vision-Based Detection using Recently Captured Images

During this stage, we added more complexity by utilizing a webcam to take new pictures of orange traffic cones positioned at various angles. The built image-processing pipeline was tested in a real-world setting using these new photos. We assessed how well the prior calibration handled fresh data after using the same color detection and blob analysis techniques. This section evaluated our system's ability to adjust to a wider variety of visual conditions, maybe necessitating parameter adjustments if the initial configurations weren't sufficient to handle the newly recorded situations. Additionally, it strengthened the practical abilities needed to manage uncertain circumstances in robotics applications.

Part 6: Discussion of Student Feedback

A thoughtful evaluation of the lab experience was made possible via the feedback section. We talked about each task's clarity and degree of difficulty, identifying issues like managing image noise or adjusting HSV values for different lighting conditions. This step also included keeping note of how much time was spent on each lab component overall, which yields useful information for estimating the amount of work needed. Giving constructive criticism, concentrating on whether elements were interesting or too complicated for the situation, is essential to enhancing subsequent labs.

Using the MATLAB Color Thresholder App Discussion (Appendix A)

The MATLAB Color Thresholder App, a graphical user interface that greatly streamlines the process of adjusting HSV parameters for color detection, was used in detail in Appendix A. We could effectively isolate the target item by visually modifying the hue, saturation, and value sliders. This allowed us to convert the adjusted parameters into a reusable MATLAB code. By using a visual approach to thresholding rather than manually coding the changes, this strategy saves time.

Using the MATLAB Image Batch Processor App Discussion (Appendix B)

This section demonstrated how well MATLAB's Image Batch Processor App handles numerous images at once. By using the same image-processing algorithm on a directory with several images, the application enabled us to automate the processing of the directory. Large datasets benefit greatly from this automation since it minimizes human interaction and guarantees that algorithms are applied consistently to every image.

Copying a MATLAB Figure for Use in a Report Discussion (Appendix C)

Best techniques for duplicating MATLAB figures were discussed in Appendix C to guarantee excellent visual representation in reports. By doing this, figures are guaranteed to retain their clarity and resolution, preventing pixelation that may arise from using incorrect copying techniques. It underlined how crucial excellent documentation is, especially in technical reports where analytical results are supported by visual data.

Conclusion

The objective of this lab was effectively accomplished by employing vision-based techniques for orange traffic cone detection, localization, and mapping. The implementation of HSV-based color recognition, blob analysis, and centroid extraction were crucial milestones. The cones' angular locations and proximities with respect to the robot were then determined. Accurately recognizing cones at different positions, distances, and environmental conditions, the code showed resilience in processing both training and freshly taken photos. Despite obstacles including erratic lighting and background noise, dependable performance was guaranteed by making small parameter changes, such as fine-tuning area filters and HSV thresholds. All things considered; the lab laid the foundation for upcoming applications in mobile robot simulations by offering invaluable experience with image processing techniques for vision-based robotics tasks. This exercise improved both technical comprehension and robotics problem-solving abilities by highlighting the significance of algorithm flexibility and parameter tuning when working with real-world data.

References

[1] Gonzalez, R. C., & Woods, R. E. (2018). *Digital Image Processing* (4th ed.). Pearson: Referenced for theoretical understanding of HSV color space and its application in color segmentation.

[2] Corke, P. (2017). *Robotics, Vision and Control: Fundamental Algorithms in MATLAB®* (2nd ed.). Springer: Referenced for techniques in visual perception and blob analysis in robotic systems.

[3] Mechanical Engineering Department, ME 598A. *Lab 3: Image Processing for Vision-Based Tasks*: Provided detailed guidance on task requirements, code structure, and objectives for image-based cone detection.

[4] MathWorks. (n.d.). *Using the Color Thresholder App*. Retrieved from: https://www.mathworks.com/videos/webcam-support-89504.html

Student Feedback:

Although there were several difficulties in this lab, each one offered a chance to gain and hone critical abilities in vision-based robotics. Dealing with uneven illumination in the photos was one of the main challenges, as it impacted the cone detection accuracy. To guarantee dependable operation in various settings, the HSV thresholds had to be repeatedly adjusted due to variations in illumination. There were also difficulties in determining the proper blob analysis thresholds. To filter out noise without neglecting legitimate cones, it was essential to set a minimum area threshold, which required a great deal of trial and error. Cone identification was occasionally hampered by environmental noise, such as background clutter, which made the employment of sophisticated morphological techniques to clear the binary masks necessary. Understanding how to translate 2D picture data, like centroids, into angular locations and proximities for the overhead view representation was another difficulty. This stage emphasized the significance of exact geometric reasoning in robots.

Despite these difficulties, the lab was very instructive. It provided practical expertise with HSV-based color segmentation, a crucial computer vision method for color-based object detection. To improve detection accuracy, we also learned how to preprocess photos using morphological techniques. Another important area of study was blob analysis, which made it possible to extract helpful characteristics from identified objects, such as areas and centroids. A crucial component of robotic navigation and perception, the lab also presented the idea of mapping picture input to useful spatial information, Including angles and closeness.

Future robotics applications will greatly benefit from the abilities acquired in this lab. Similar methods for identifying landmarks and objects are used in vision-based navigation, therefore this lab serves as a basis for more complex tasks like SLAM (Simultaneous Localization and Mapping). The tracking and object detection techniques established here can also be applied to tasks like object sorting, obstacle avoidance, and robotic grasping. Crucially, the lab highlighted how crucial it is to modify algorithms to account for real-world differences like noise and sunlight, which are frequent problems in robots. All things considered, this lab offered both theoretical knowledge and real-world experience that will be crucial for creating sophisticated robotic systems.

Appendices

Appendix A: Cone Threshold for Part 2 & Part 4:

```
function [BW,maskedRGBImage] = coneThreshold(RGB)
% Convert RGB image to chosen color space
I = rgb2hsv(RGB);

% Define thresholds for channel 1 based on histogram settings
channel1Min = 0.024;
channel1Max = 0.135;

% Define thresholds for channel 2 based on histogram settings
channel2Min = 0.378;
channel2Max = 1.000;

% Define thresholds for channel 3 based on histogram settings
channel3Min = 0.741;
channel3Max = 1.000;

% Create mask based on chosen histogram thresholds
sliderBW = (I(:,:,1) >= channel1Min ) & (I(:,:,1) <= channel1Max) &
...
    (I(:,:,2) >= channel2Min ) & (I(:,:,2) <= channel2Max) & ...
    (I(:,:,3) >= channel3Min ) & (I(:,:,3) <= channel3Max);
BW = sliderBW;

% Initialize output masked image based on input image.
maskedRGBImage = RGB;
```

Appendix B: Cone Threshold for Part 3

```
function [BW,maskedRGBImage] = part5(RGB)
% Convert RGB image to chosen color space
I = rgb2hsv(RGB);

% Define thresholds for channel 1 based on histogram settings
channel1Min = 0.023;
channel1Max = 0.092;

% Define thresholds for channel 2 based on histogram settings
channel2Min = 0.546;
channel2Max = 1.000;

% Define thresholds for channel 3 based on histogram settings
channel3Min = 0.652;
channel3Max = 1.000;

% Create mask based on chosen histogram thresholds
sliderBW = (I(:,:,1) >= channel1Min ) & (I(:,:,1) <= channel1Max) &
...
    (I(:,:,2) >= channel2Min ) & (I(:,:,2) <= channel2Max) & ...
    (I(:,:,3) >= channel3Min ) & (I(:,:,3) <= channel3Max);
BW = sliderBW;

% Initialize output masked image based on input image.
maskedRGBImage = RGB;

% Set background pixels where BW is false to zero.
maskedRGBImage(repmat(~BW,[1 1 3])) = 0;
```

end

Appendix C: Creating figures

```
% Left / right
for iter = 1: 9
 figure
 subplot(1,2,1);
 imshow(allresults(iter).fileName);
 title ('Original')

 subplot(1,2,2);
 imshow(allresults(iter).output)
 title('Processed')
end
```

Appendix D: Matlab Code for detecting the cone centroid

```
function detect_cone_centroids(image_folder, area_threshold)
    % detect_cone_centroids - Detects and saves centroids of cones
in all images in a folder
    %
    % Parameters:
    %   image_folder (string): Path to the folder containing images
    %   area_threshold (int): Minimum area size to consider a region
as a cone

    % Get list of images in the folder
    image_files = dir(fullfile(image_folder, '*.jpg'));
    output_folder = 'output_centroids'; % Folder to save results

    if ~exist(output_folder, 'dir')
        mkdir(output_folder);
    end

    for i = 1:length(image_files)
        % Load the current image
        img_path = fullfile(image_folder, image_files(i).name);
        CD = imread(img_path);

        % Convert to binary if the image is grayscale
        if size(CD, 3) == 3
            CD = rgb2gray(CD);
        end
        CD = imbinarize(CD);

        % Step 1: Filtering out connected pixel regions smaller than
a user-defined threshold
        CDfiltered = bwareaopen(CD, area_threshold);

        % Step 2: Labeling the connected components
        L = bwlabel(CDfiltered);

        % Step 3: Extracting properties of each labeled region
        stats = regionprops(L, 'Area', 'Centroid');
        areas = [stats.Area];
        centroids = cat(1, stats.Centroid);
        % Step 4: Checking for valid regions and handle cases for 0,
1, or 2 cones
        disp(['Image: ', image_files(i).name]);
        if isempty(areas)
            disp('Centroid coordinates: []'); % No cones detected
            continue;
        end

        % Finding the two largest areas that meet the area threshold
        [sorted_areas, sort_idx] = sort(areas, 'descend');
        valid_centroids = [];
        for idx = 1:min(2, length(sorted_areas))
            if sorted_areas(idx) >= area_threshold
                valid_centroids = [valid_centroids;
centroids(sort_idx(idx), :)];
            end
        end

        % Printing centroid coordinates
        if isempty(valid_centroids)
            disp('Centroid coordinates: []');
            disp('Size: [0 0]');
            continue;
        else
            disp('Centroid coordinates:');
            for k = 1:size(valid_centroids, 1)
                fprintf('[%.2f, %.2f]\n', valid_centroids(k,1),
valid_centroids(k,2));
            end
        end

        % Step 5: Applying iterative closing to fill smaller
internal black regions
        se_small = strel('disk', 5); % Small disk for closing small
gaps
        CDfilled = CDfiltered;
        while true
            CDclosed = imclose(CDfilled, se_small);
            if isequal(CDclosed, CDfilled)
                break;
            end
            CDfilled = CDclosed;
        end
        % Step 6: Applying flood-fill from each centroid to capture
larger internal holes
        for k = 1:size(valid_centroids, 1)
            CDfilled = flood_fill(CDfilled, round(valid_centroids(k,
:)));
        end

        % Display the modified image with filled regions
        figure('Visible', 'off');
        imshow(CDfilled);
        hold on;

        % Step 7: Plot the centroids on the modified image
        for k = 1:size(valid_centroids, 1)
            plot(valid_centroids(k,1), valid_centroids(k,2), 'b*',
'MarkerSize', 10);

        % Display the coordinates next to the centroid
        text(valid_centroids(k,1) + 5, valid_centroids(k,2), ...
        sprintf('[%.2f, %.2f]', valid_centroids(k,1),
valid_centroids(k,2)), ...
        'Color', 'blue', 'FontSize', 10, 'FontWeight', 'bold');
        end


        % Save the output image
        [~, name, ext] = fileparts(image_files(i).name);
        saveas(gcf, fullfile(output_folder, ['centroids_' name
ext]));
        close(gcf);
    end
end

function filled_image = flood_fill(image, start_point)
    % flood_fill - Custom flood-fill to fill larger black areas
within white regions.
    %
    % Parameters:
    %   image (binary matrix): Binary image with black (0) and white
(1) pixels
    %   start_point (1x2 array): [x, y] coordinates of the starting
point (centroid)
```

```
    %
    % Returns:
    %   filled_image: Binary image with larger internal black pixels
filled

    filled_image = image;
    queue = [start_point];

    while ~isempty(queue)
        point = queue(1, :);
        queue(1, :) = []; % Pop from queue
        x = point(1);
        y = point(2);

        % Check bounds and only process black pixels
        if x > 1 && x < size(filled_image, 2) && y > 1 && y <
size(filled_image, 1)
            if filled_image(y, x) == 0 % Black pixel to be filled
                filled_image(y, x) = 1; % Fill with white

                % Add adjacent pixels to queue
                queue = [queue; x+1, y; x-1, y; x, y+1; x, y-1];
            end
        end
    end
end
```

Appendix E: Matlab Code for detecting Centroid & Overhead View of the cones:

```
function process_cone_images(processing_image_folder,
area_threshold)
    % process_cone_images - Processes images to detect and save
centroids of cones
    % Additionally calculates angles, proximity, and generates an
overhead view
    %
    % Parameters:
    %   processing_image_folder (string): Path to the folder
containing images for processing
    %   area_threshold (int): Minimum area size to consider a region
as a cone

    original_image_folder =
'/Users/prayashdas/Documents/MATLAB/ME598_ImgProcLab_MATLAB/TrainCon
eImages';

    % Get list of images in the folder
    image_files = dir(fullfile(processing_image_folder, '*.jpg'));
    output_folder = 'Processed'; % Folder to save processed images

    if ~exist(output_folder, 'dir')
        mkdir(output_folder);
    end

    % Initialize structure to store results for plotting
    allresults = struct();

    for i = 1:length(image_files)
        % Load the processed image
        img_path = fullfile(processing_image_folder,
image_files(i).name);
        CD = imread(img_path);

        % Load the original image for plotting
        original_path = fullfile(original_image_folder,
image_files(i).name);
        original_image = imread(original_path);
        % Convert to binary if the image is grayscale
        if size(CD, 3) == 3
            CD = rgb2gray(CD);
        end
        CD = imbinarize(CD);
        % Get the size of the image
        [nr, nc] = size(CD);
        ImageCenterX = nc / 2;

        % Step 1: Filtering out connected pixel regions smaller than
a user-defined threshold
        CDfiltered = bwareaopen(CD, area_threshold);

        % Step 2: Labeling the connected components
        L = bwlabel(CDfiltered);

        % Step 3: Extracting properties of each labeled region
        stats = regionprops(L, 'Area', 'Centroid');
        areas = [stats.Area];
        centroids = cat(1, stats.Centroid);

        % Step 4: Checking for valid regions and handle cases for 0,
1, or 2 cones
        disp(['Processing Image: ', image_files(i).name]);
        if isempty(areas)
            disp('No cones detected.');
            disp('--------------------------------------------');
            continue;
        end

        % Find the two largest areas that meet the area threshold
        [sorted_areas, sort_idx] = sort(areas, 'descend');
        valid_centroids = [];
        valid_areas = [];
        for idx = 1:min(2, length(sorted_areas))
            if sorted_areas(idx) >= area_threshold
                valid_centroids = [valid_centroids;
centroids(sort_idx(idx), :)];
                valid_areas = [valid_areas; sorted_areas(idx)];
            end
        end

        % Dynamically determine area thresholds for proximity
        max_area = max(valid_areas);
        min_area = min(valid_areas);
        mid_area = (max_area + min_area) / 2;

        % Calculating orientation angles and proximity for each
centroid
        angles = [];
        proximity = [];
        for k = 1:size(valid_centroids, 1)
            CentroidX = valid_centroids(k, 1);
            Xdiff = CentroidX - ImageCenterX;

            angle = -(Xdiff / ImageCenterX) * 30; % ±30° range
            angles = [angles; angle];

            % Determining proximity based on dynamically calculated
thresholds
            if valid_areas(k) >= mid_area
                prox = 1; % Near
            elseif valid_areas(k) > min_area
                prox = 2; % Medium
            else
                prox = 3; % Far
            end
            proximity = [proximity; prox];
        end

        % Display centroids, angles, and proximity in the command
window
        disp('Centroid coordinates:');
        for k = 1:size(valid_centroids, 1)
            fprintf('  Cone %d: [%.2f, %.2f]\n', k,
valid_centroids(k, 1), valid_centroids(k, 2));
        end
        disp('Orientation angles (degrees):');
        for k = 1:length(angles)
```

```
            fprintf('  Cone %d: %.2f°\n', k, angles(k));
        end
        disp('Proximity levels:');
        for k = 1:length(proximity)
            fprintf('  Cone %d: %d\n', k, proximity(k));
        end
        disp('----------------------------------------------');

        % Generating overhead view of cone positions
        if ~isempty(angles) && ~isempty(proximity)
            fignum = 100 + i; % Unique figure number
            figure(fignum);
            overheadView(angles, proximity, fignum);
            overhead_image_path = fullfile(output_folder,
['Overhead_', image_files(i).name]);
            saveas(gcf, overhead_image_path); % Save overhead figure
            close(gcf); % Close the figure after saving
        end

        % Store original and overhead images for plotting
        allresults(i).fileName = original_image;
        allresults(i).output = imread(overhead_image_path); % Load
the saved overhead image
    end

    % Plotting original and overhead images
    % Above / Below layout
    for iter = 1:size(allresults, 2)
        figure;
        subplot(2, 1, 1);
        imshow(allresults(iter).fileName);
        title('Original');
        subplot(2, 1, 2);
        imshow(allresults(iter).output);
        title('Overhead View');
    end

    % Left / Right layout
    for iter = 1:size(allresults, 2)
        figure;
        subplot(1, 2, 1);
        imshow(allresults(iter).fileName);
        title('Original');
        subplot(1, 2, 2);
        imshow(allresults(iter).output);
        title('Overhead View');
    end
end

function filled_image = flood_fill(image, start_point)
    % flood_fill - Custom flood-fill to fill larger black areas
within white regions.
    %
    % Parameters:
    %   image (binary matrix): Binary image with black (0) and white
(1) pixels
    %   start_point (1x2 array): [x, y] coordinates of the starting
point (centroid)
    %
    % Returns:
    %   filled_image: Binary image with larger internal black pixels
filled

    filled_image = image;
    queue = [start_point];

    while ~isempty(queue)
        point = queue(1, :);
        queue(1, :) = []; % Pop from queue
        x = point(1);
        y = point(2);

        % Check bounds and only process black pixels
        if x > 1 && x < size(filled_image, 2) && y > 1 && y <
size(filled_image, 1)
            if filled_image(y, x) == 0 % Black pixel to be filled
                filled_image(y, x) = 1; % Fill with white

                % Adding adjacent pixels to queue
                queue = [queue; x+1, y; x-1, y; x, y+1; x, y-1];
            end
        end
    end
end
```