HW6 Solution

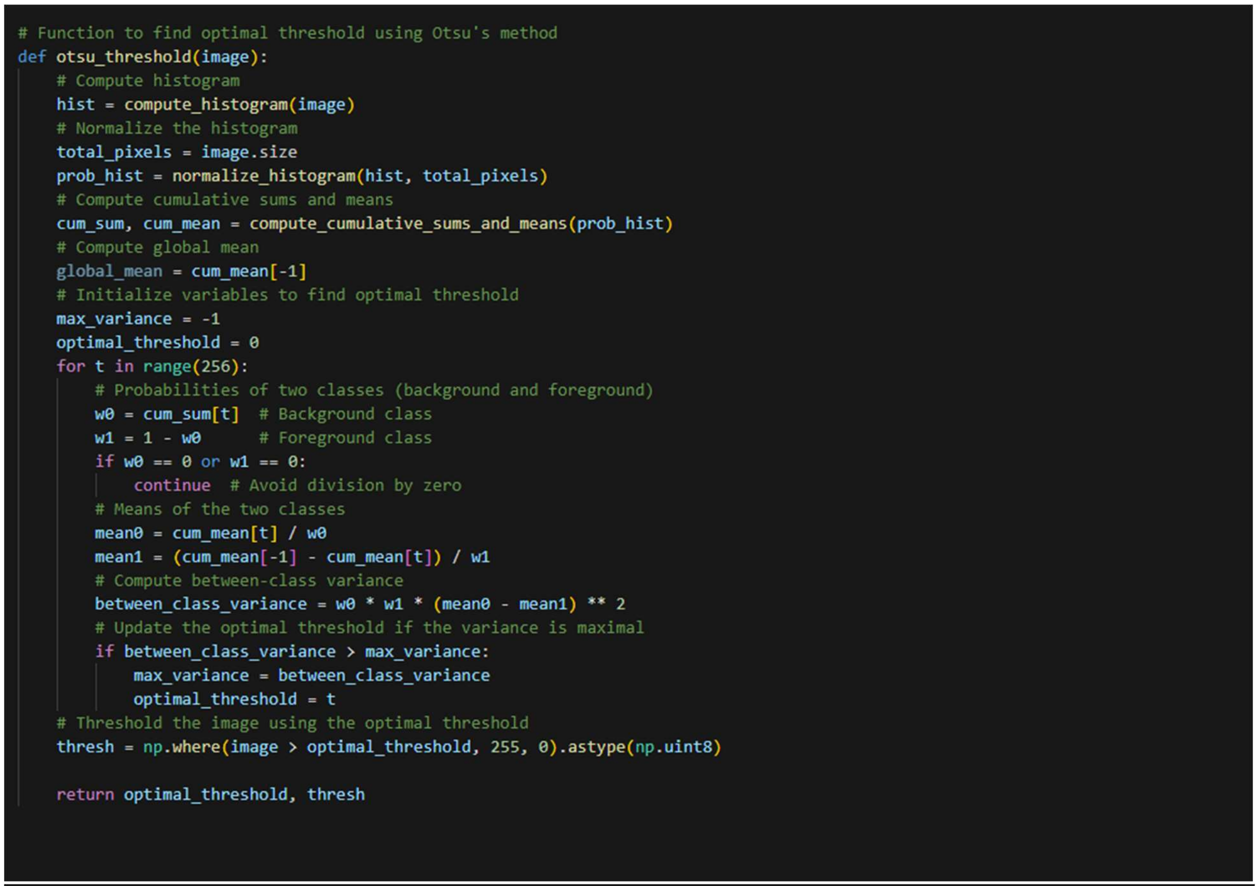
Theory question

Otsu's Algorithm is efficient for simple, bimodal images as it automatically determines the optimal threshold that separates the foreground from the background in grayscale images. Otsu algorithm struggles with images that have overlapping intensity distributions, resulting in poor segmentation. The Watershed Algorithm, on the other hand, detecting object boundaries by treating the image like a topographic map and identifying ridge lines, making it suitable for segmenting touching or overlapping objects. Its weakness lies in its sensitivity to noise and initial markers, which can lead to over-segmentation if not properly pre-processed or guided.

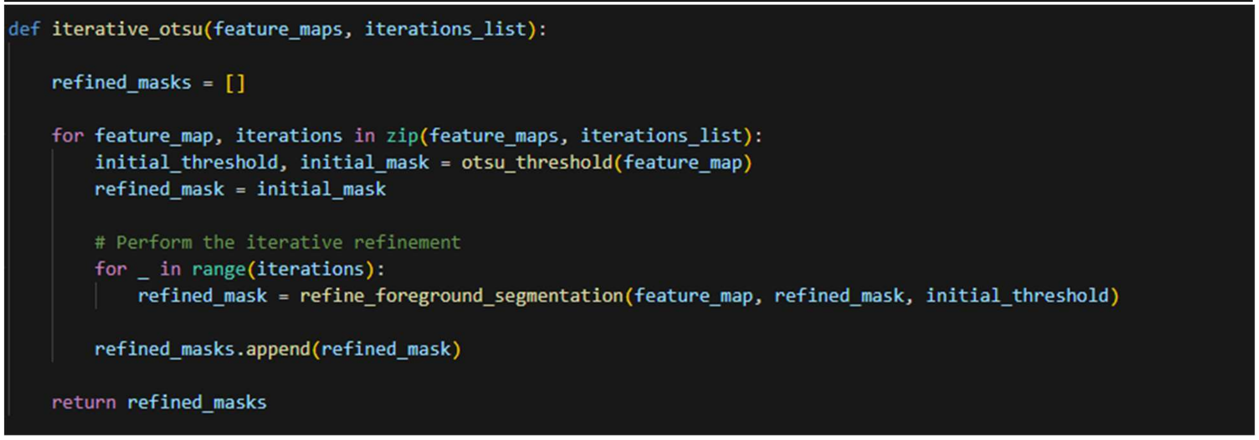
Task 1

Otsu implementation using RGB

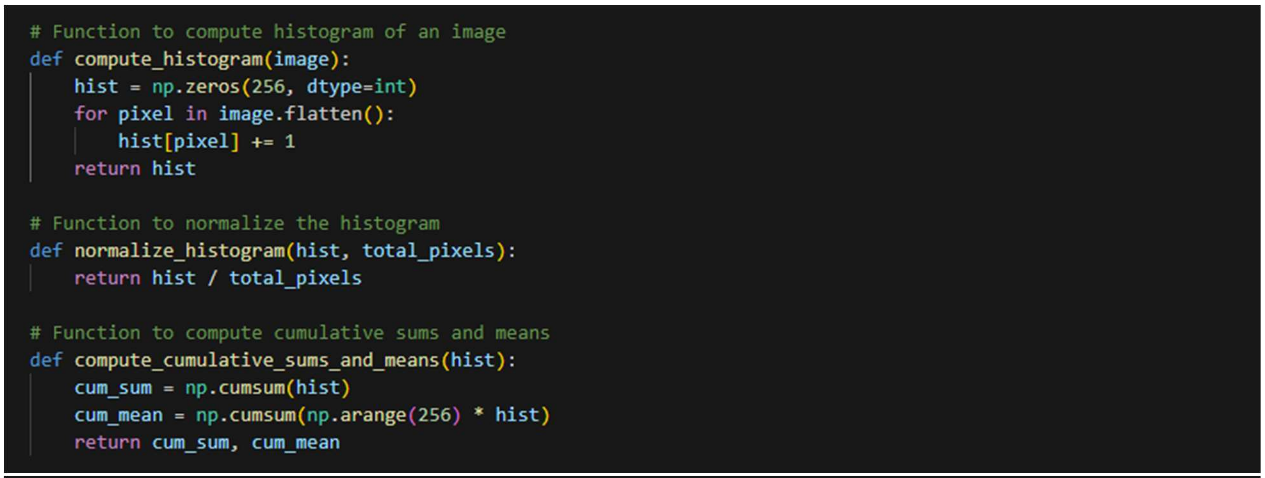
The implementation of Otsu's method aims to determine an optimal threshold for separating foreground and background pixels using its red, green and blue channels. First, a histogram of pixel intensities is computed and normalized to obtain the probability distribution of each intensity value. The cumulative sums and means are calculated to evaluate between-class variance, which measures the separation between the foreground and background classes. For each potential threshold, the algorithm computes the probabilities and means of the background and foreground classes and calculates the between-class variance. The threshold that maximizes this variance is selected as the optimal threshold. The image is then thresholded using this value, resulting in a binary mask. To further enhance the segmentation result, an iterative approach is applied to refine the foreground threshold over a specified independent number of iterations for each channel. The resulting segmentation masks were combined using the logical ‘AND’ operator and saved.



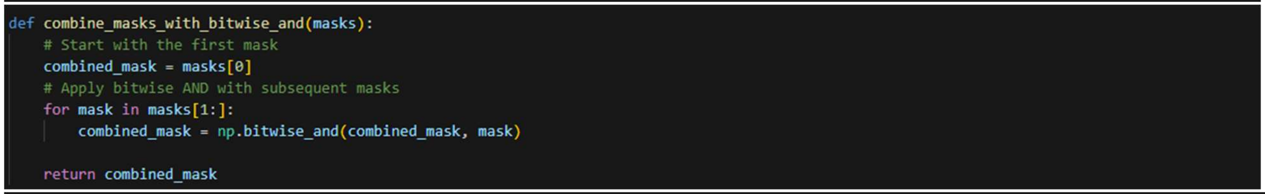
```python
# Function to find optimal threshold using Otsu's method
def otsu_threshold(image):
    # Compute histogram
    hist = compute_histogram(image)
    # Normalize the histogram
    total_pixels = image.size
    prob_hist = normalize_histogram(hist, total_pixels)
    # Compute cumulative sums and means
    cum_sum, cum_mean = compute_cumulative_sums_and_means(prob_hist)
    # Compute global mean
    global_mean = cum_mean[-1]
    # Initialize variables to find optimal threshold
    max_variance = -1
    optimal_threshold = 0
    for t in range(256):
        # Probabilities of two classes (background and foreground)
        w0 = cum_sum[t]  # Background class
        w1 = 1 - w0      # Foreground class
        if w0 == 0 or w1 == 0:
            continue  # Avoid division by zero
        # Means of the two classes
        mean0 = cum_mean[t] / w0
        mean1 = (cum_mean[-1] - cum_mean[t]) / w1
        # Compute between-class variance
        between_class_variance = w0 * w1 * (mean0 - mean1) ** 2
        # Update the optimal threshold if the variance is maximal
        if between_class_variance > max_variance:
            max_variance = between_class_variance
            optimal_threshold = t
    # Threshold the image using the optimal threshold
    thresh = np.where(image > optimal_threshold, 255, 0).astype(np.uint8)

    return optimal_threshold, thresh
```



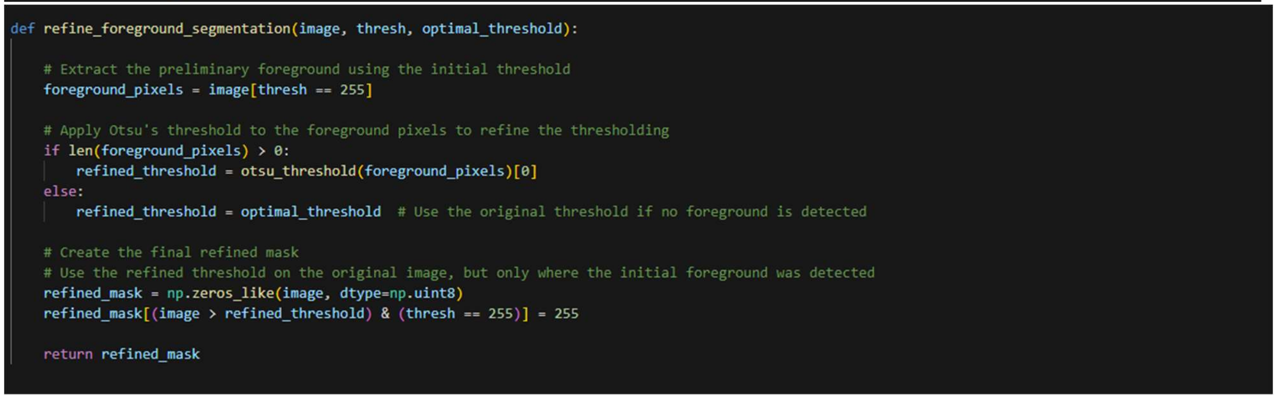
```python
def iterative_otsu(feature_maps, iterations_list):

    refined_masks = []

    for feature_map, iterations in zip(feature_maps, iterations_list):
        initial_threshold, initial_mask = otsu_threshold(feature_map)
        refined_mask = initial_mask

        # Perform the iterative refinement
        for _ in range(iterations):
            refined_mask = refine_foreground_segmentation(feature_map, refined_mask, initial_threshold)

        refined_masks.append(refined_mask)

    return refined_masks
```



```python
# Function to compute histogram of an image
def compute_histogram(image):
    hist = np.zeros(256, dtype=int)
    for pixel in image.flatten():
        hist[pixel] += 1
    return hist

# Function to normalize the histogram
def normalize_histogram(hist, total_pixels):
    return hist / total_pixels

# Function to compute cumulative sums and means
def compute_cumulative_sums_and_means(hist):
    cum_sum = np.cumsum(hist)
    cum_mean = np.cumsum(np.arange(256) * hist)
    return cum_sum, cum_mean
```



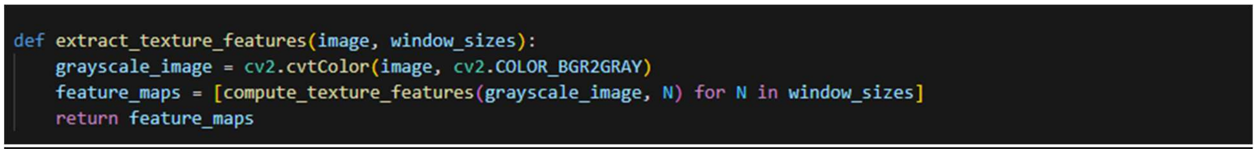
```python
def combine_masks_with_bitwise_and(masks):
    # Start with the first mask
    combined_mask = masks[0]
    # Apply bitwise AND with subsequent masks
    for mask in masks[1:]:
        combined_mask = np.bitwise_and(combined_mask, mask)

    return combined_mask
```



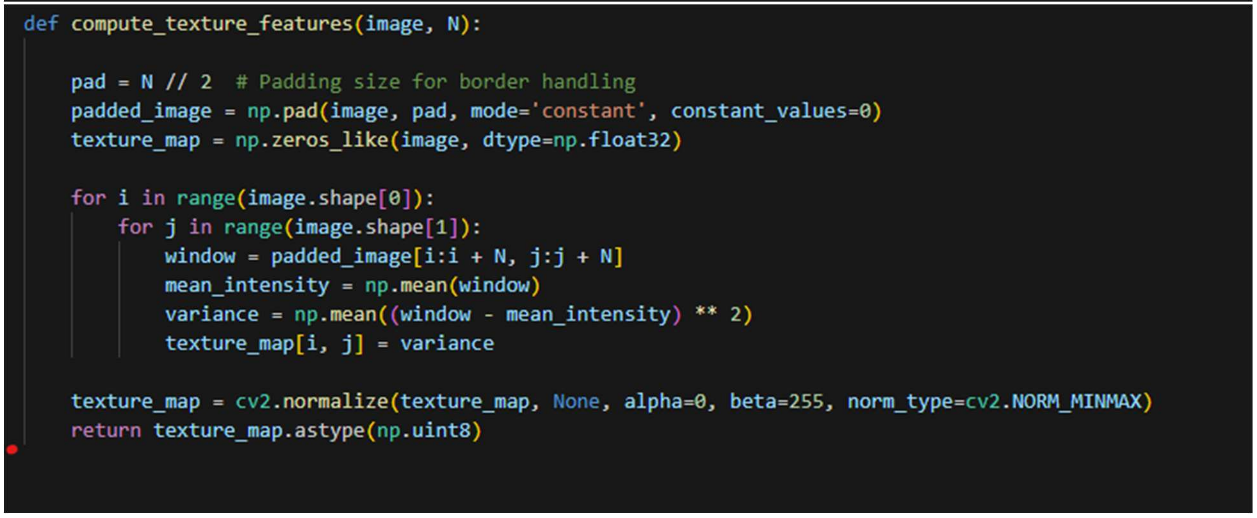
```python
def refine_foreground_segmentation(image, thresh, optimal_threshold):

    # Extract the preliminary foreground using the initial threshold
    foreground_pixels = image[thresh == 255]

    # Apply Otsu's threshold to the foreground pixels to refine the thresholding
    if len(foreground_pixels) > 0:
        refined_threshold = otsu_threshold(foreground_pixels)[0]
    else:
        refined_threshold = optimal_threshold  # Use the original threshold if no foreground is detected

    # Create the final refined mask
    # Use the refined threshold on the original image, but only where the initial foreground was detected
    refined_mask = np.zeros_like(image, dtype=np.uint8)
    refined_mask[(image > refined_threshold) & (thresh == 255)] = 255

    return refined_mask
```

Texture-based segmentation

The texture-based segmentation approach implemented is used to extract texture features from an image by computing the intensity variance within sliding windows of different sizes. To do this, the input image is converted to grayscale, and then texture feature maps are generated for specified window size N × N where N takes the value of 3, 5, 7, 9 …etc. For each pixel, the window is centered around it, and the mean intensity within this window is calculated. The variance, which captures the intensity variation within the window, is then computed and stored in a texture map. This variance serves as a measure of texture, where higher values indicate greater variation in intensity, suggesting a more textured region. The extracted textures were subsequently refined using the iterative otsu mentioned in the previous section.



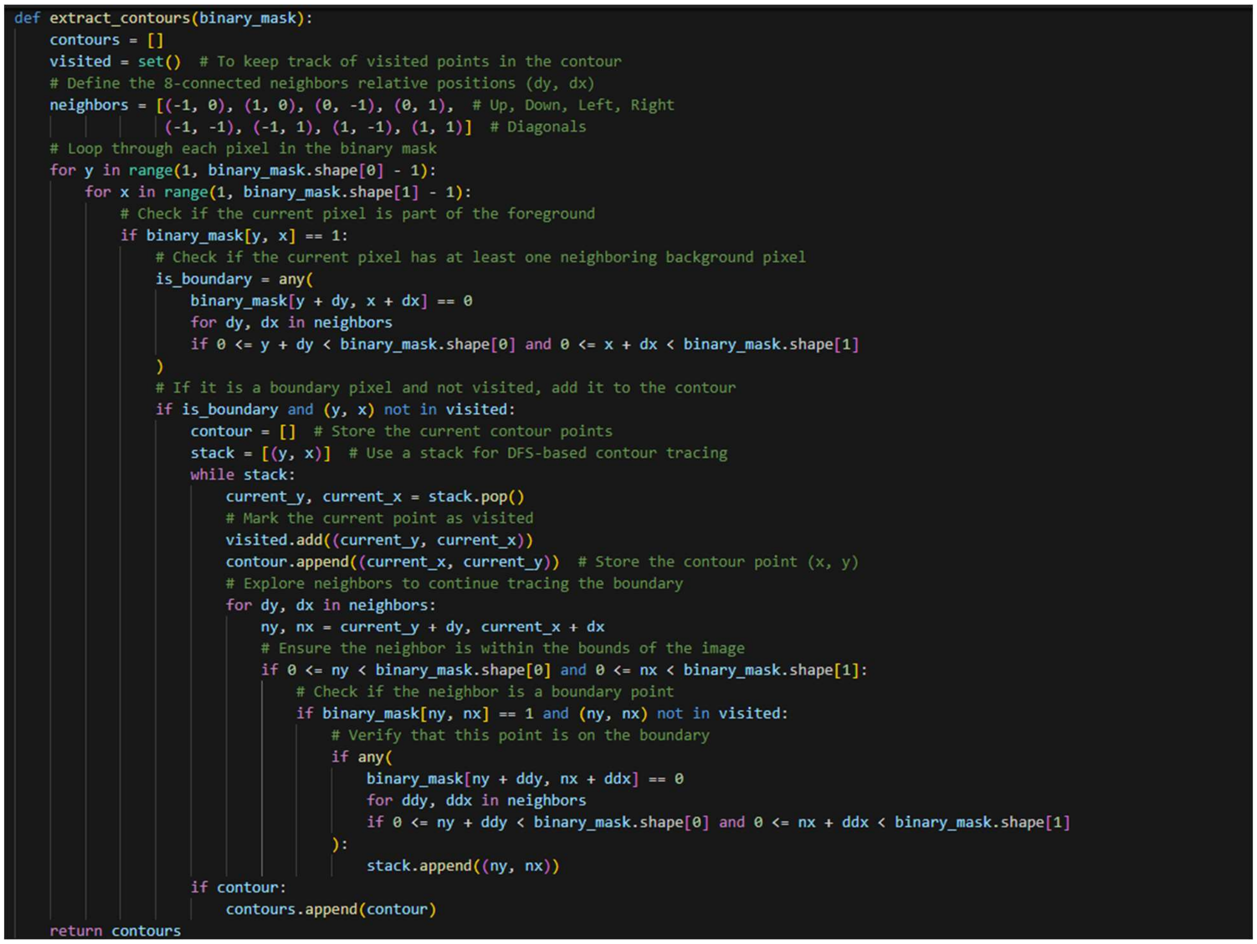
```
def extract_texture_features(image, window_sizes):
    grayscale_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    feature_maps = [compute_texture_features(grayscale_image, N) for N in window_sizes]
    return feature_maps
```



```
def compute_texture_features(image, N):

    pad = N // 2  # Padding size for border handling
    padded_image = np.pad(image, pad, mode='constant', constant_values=0)
    texture_map = np.zeros_like(image, dtype=np.float32)

    for i in range(image.shape[0]):
        for j in range(image.shape[1]):
            window = padded_image[i:i + N, j:j + N]
            mean_intensity = np.mean(window)
            variance = np.mean((window - mean_intensity) ** 2)
            texture_map[i, j] = variance

    texture_map = cv2.normalize(texture_map, None, alpha=0, beta=255, norm_type=cv2.NORM_MINMAX)
    return texture_map.astype(np.uint8)
```

Contour Extraction

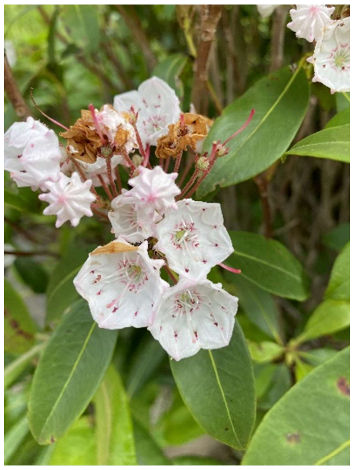
The contour extraction algorithm identifies and traces the boundaries of objects within a binary mask, where foreground pixels have a value of 1 and background pixels are 0. It uses an 8-connected neighborhood approach, which considers all adjacent pixels (including diagonals) to determine if a pixel lies on the boundary. For each unvisited foreground pixel, the algorithm checks if it has at least one neighboring background pixel, which would make it a boundary point. When such a point is found, a depth-first search (DFS) method is used to trace the boundary, storing each boundary point in a list called a contour. As the DFS explores the neighboring points, it only includes those that are also on the boundary and have not been visited before. This process continues until all boundary points of the object are traced, and the resulting contours are stored in a list, where each contour represents a set of boundary coordinates.



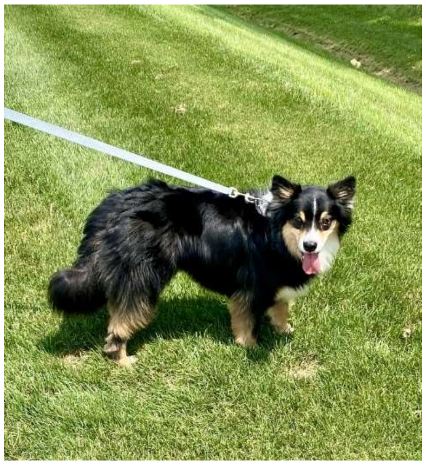
```python
def extract_contours(binary_mask):
    contours = []
    visited = set()  # To keep track of visited points in the contour
    # Define the 8-connected neighbors relative positions (dy, dx)
    neighbors = [(-1, 0), (1, 0), (0, -1), (0, 1),  # Up, Down, Left, Right
                 (-1, -1), (-1, 1), (1, -1), (1, 1)]  # Diagonals
    # Loop through each pixel in the binary mask
    for y in range(1, binary_mask.shape[0] - 1):
        for x in range(1, binary_mask.shape[1] - 1):
            # Check if the current pixel is part of the foreground
            if binary_mask[y, x] == 1:
                # Check if the current pixel has at least one neighboring background pixel
                is_boundary = any(
                    binary_mask[y + dy, x + dx] == 0
                    for dy, dx in neighbors
                    if 0 <= y + dy < binary_mask.shape[0] and 0 <= x + dx < binary_mask.shape[1]
                )
                # If it is a boundary pixel and not visited, add it to the contour
                if is_boundary and (y, x) not in visited:
                    contour = []  # Store the current contour points
                    stack = [(y, x)]  # Use a stack for DFS-based contour tracing
                    while stack:
                        current_y, current_x = stack.pop()
                        # Mark the current point as visited
                        visited.add((current_y, current_x))
                        contour.append((current_x, current_y))  # Store the contour point (x, y)
                        # Explore neighbors to continue tracing the boundary
                        for dy, dx in neighbors:
                            ny, nx = current_y + dy, current_x + dx
                            # Ensure the neighbor is within the bounds of the image
                            if 0 <= ny < binary_mask.shape[0] and 0 <= nx < binary_mask.shape[1]:
                                # Check if the neighbor is a boundary point
                                if binary_mask[ny, nx] == 1 and (ny, nx) not in visited:
                                    # Verify that this point is on the boundary
                                    if any(
                                        binary_mask[ny + ddy, nx + ddx] == 0
                                        for ddy, ddx in neighbors
                                        if 0 <= ny + ddy < binary_mask.shape[0] and 0 <= nx + ddx < binary_mask.shape[1]
                                    ):
                                        stack.append((ny, nx))
                    if contour:
                        contours.append(contour)
    return contours
```

Input images

The input images for Task 1 can be seen in Figure 1



(a) Flower



(b) Dog

Figure 1. Input images for Task 1

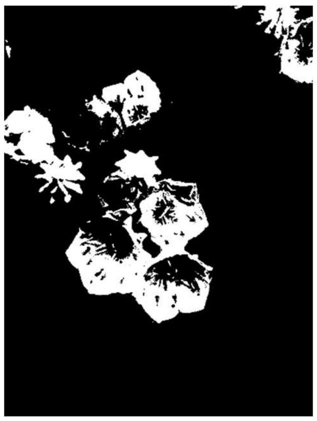
Output

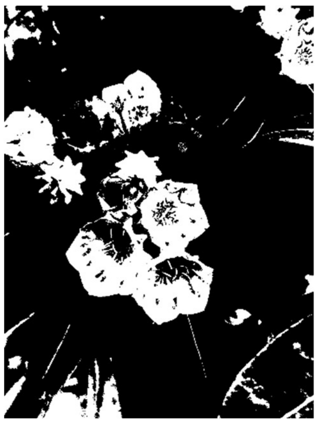
The parameters used for Task 1 Can be seen in Table1. Figure 2 shows the output of the R, G and B masks using otsu, the combination of these masks and the contour. Figure 3 is the shows the mirror results for the flower images obtained using texture-based segmentation. Figure 4 and Figure 5 and otsu and feature texture segmentation method outputs for dog image.

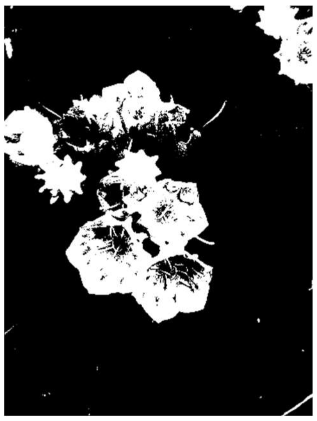
Table 1. Parameters used for Task 1

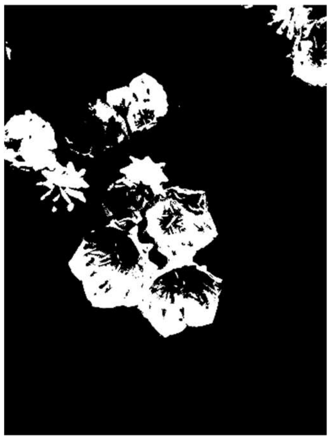
Image	method	parameter	Value
Flower	RGB channels	No of iterations	(1,1,1)
	Texture features	Window size	(15, 17, 21)
		No of iterations	(1,1,1)
Dog	RGB Channel	No of iterations	(3,5,5)
	Texture features	Window size	(11, 13, 17)
		No of iterations	(1,1,1)

Flower


(a) R


(b) G


(c) B


(d) RGB

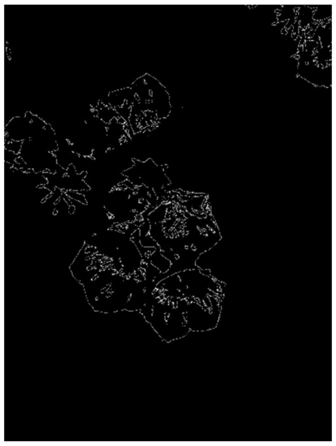

(e) Contour

Figure 2. Flower otsu segmentation output

(a) N = 15

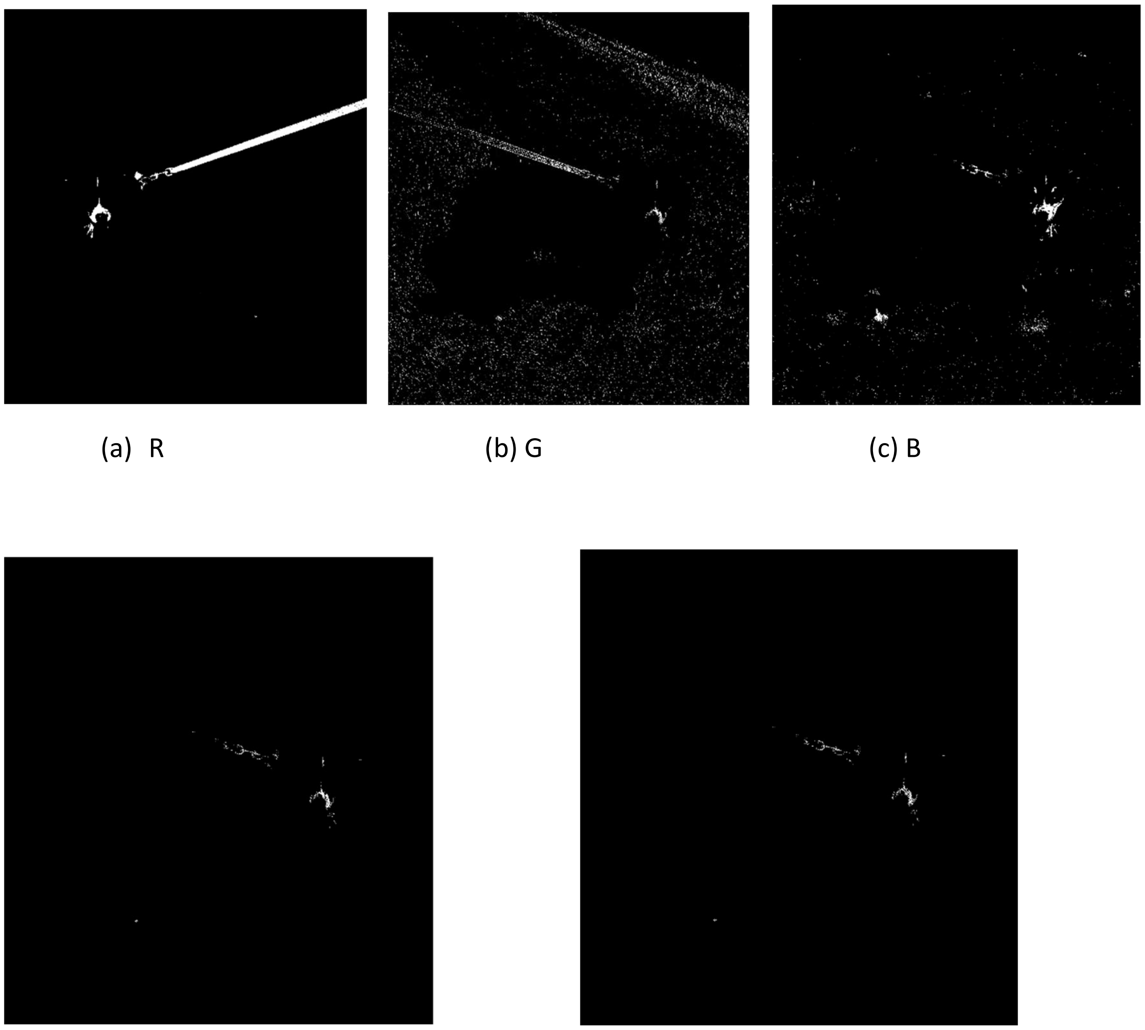
(B) N = 17

(c) N = 21

(d) Combination

(e) Contour

Figure 3. Flower texture feature segmentation output



(a) R (b) G (c) B

(d) RGB (e) Contour

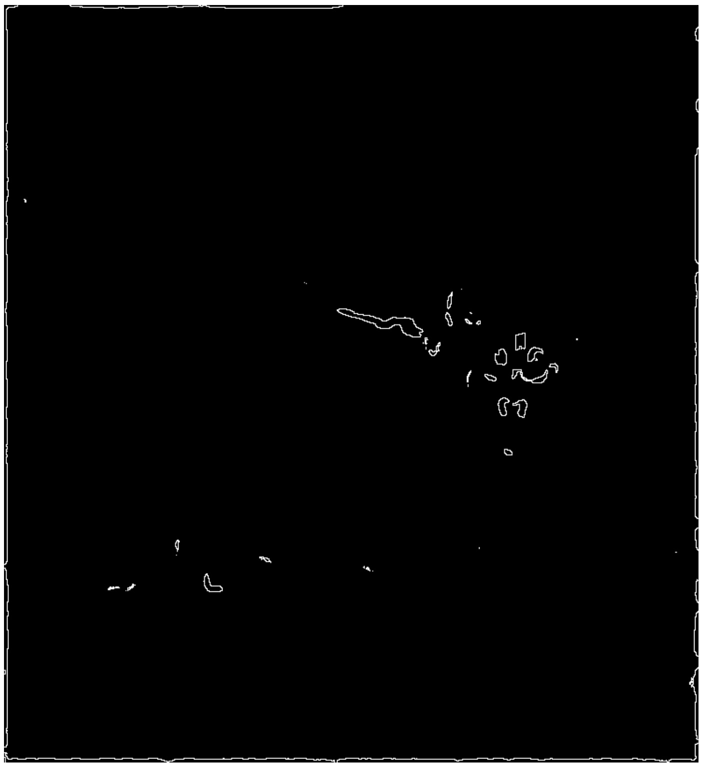
Figure 4. Dog otsu segmentation output

(a) N = 11

(b) N = 13

(c) N = 17

(d) Combined mask from texture-based masks of dog image.

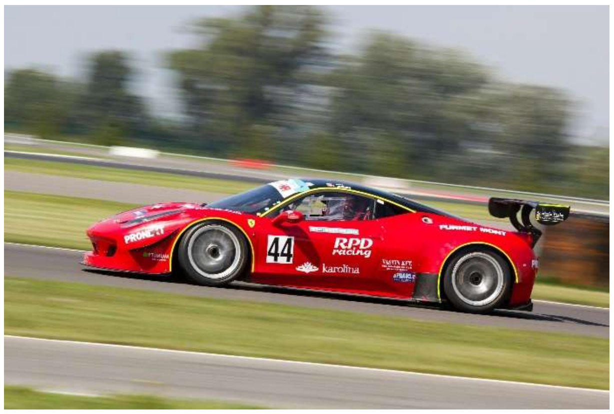


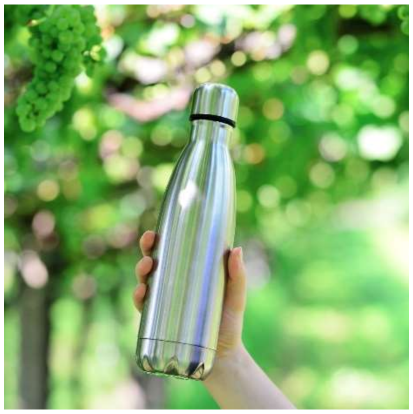
(e) Contour

Figure 5. Dog texture feature segmentation output

Task 2

The input images for Task 2 can be seen in Figure 6





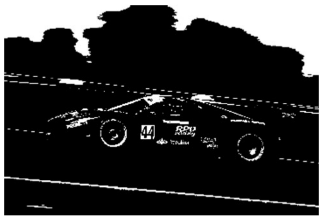
(a) Car

(b) Bottle

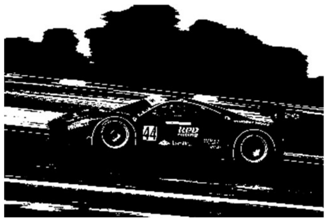
Figure 6. Input images for Task 2

Table 2. Parameters used for Task 2

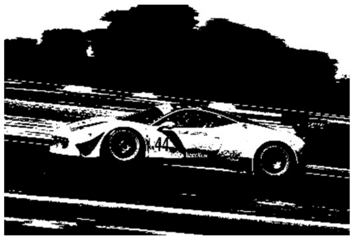
Image	method	parameter	Value
Car	RGB channels	No of iterations	(1,1,1)
	Texture features	Window size	(7,9,13)
		No of iterations	(1,2,1)
bottle	RGB Channel	No of iterations	(1,1,1)
	Texture features	Window size	(21, 25, 29)
		No of iterations	(1,1,1)



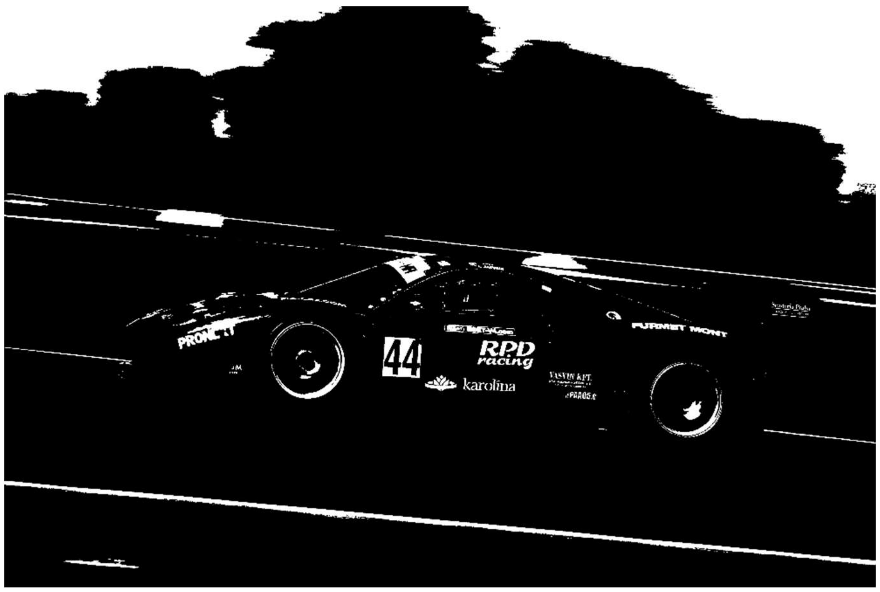
(a) R



(b) G



(c) B



(d) Combined mask from RGB-based masks of car image.

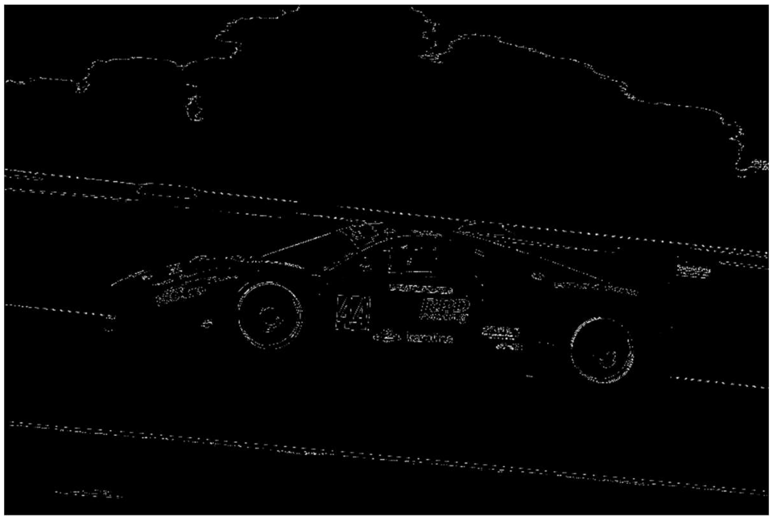
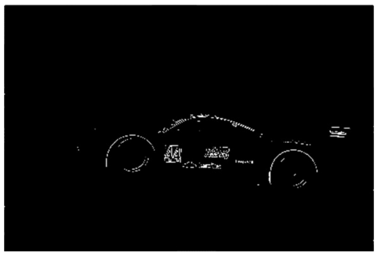
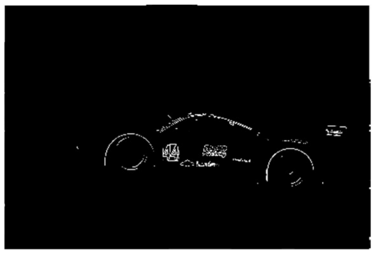
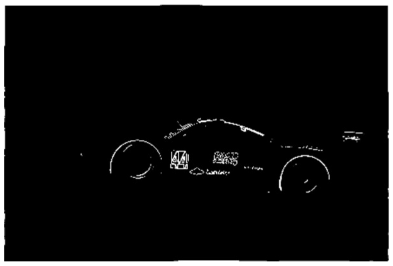
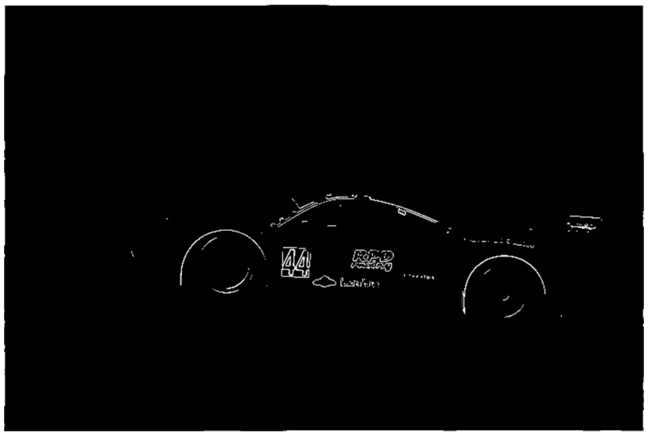

(e) Contour

Figure 7. Car otsu segmentation output


(a) N = 7


(b) N = 9


(c) N = 13


(d) Combined mask from texture-based masks of car image.

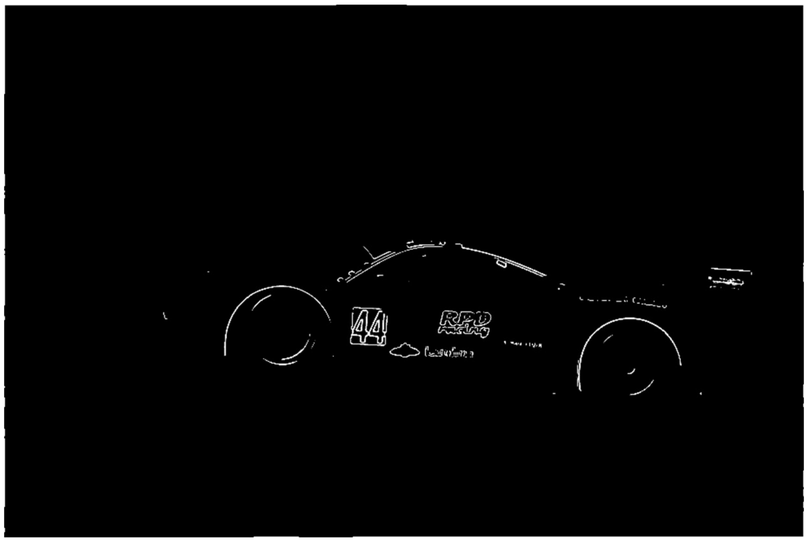
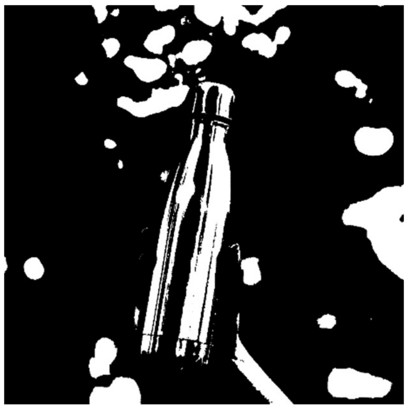
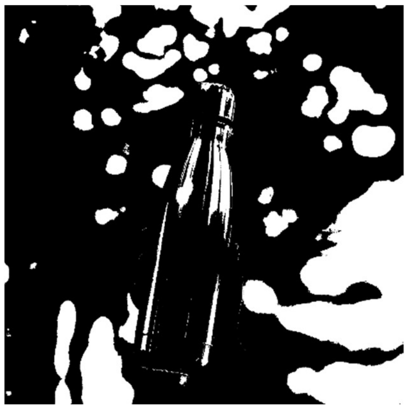
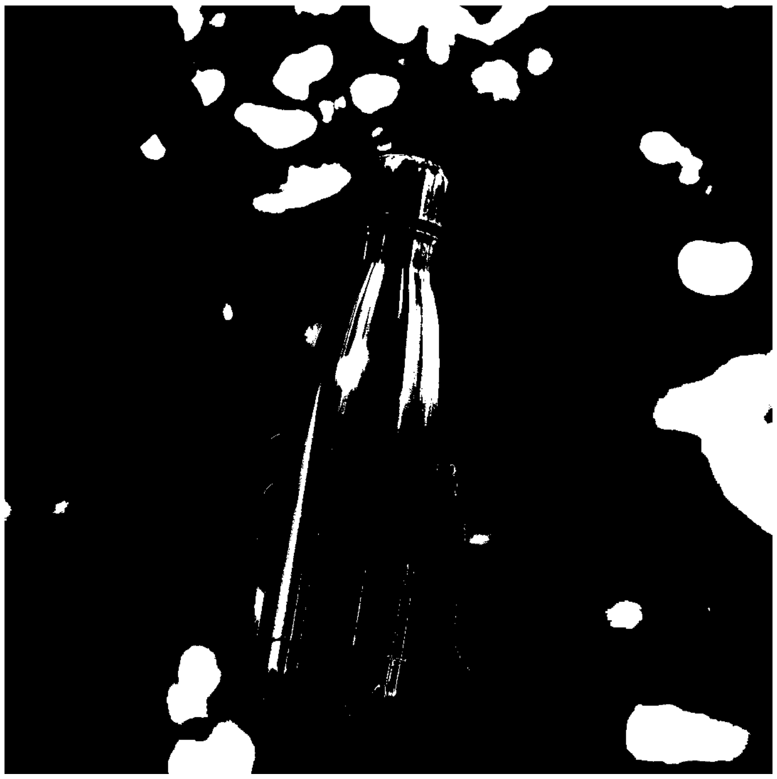

(e) Contour

Figure 8. Car texture feature segmentation output


(a) R


(b) G

(c) B


(d) Combined mask from RGB-based masks of dog image.

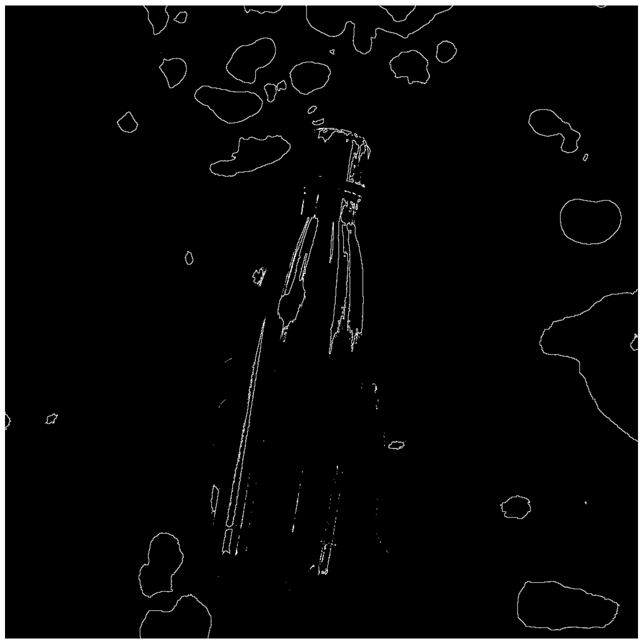
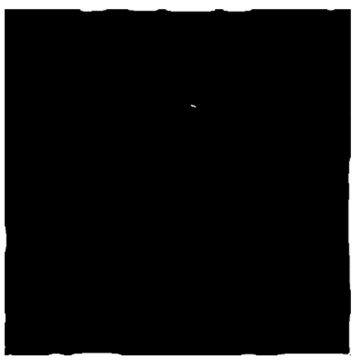
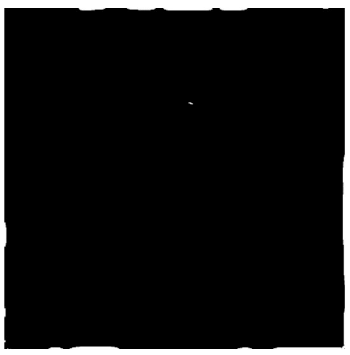

(e) Contour

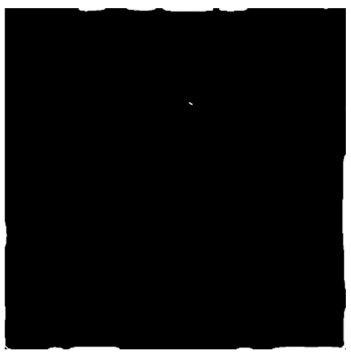
Figure 9. Bottle otsu segmentation output



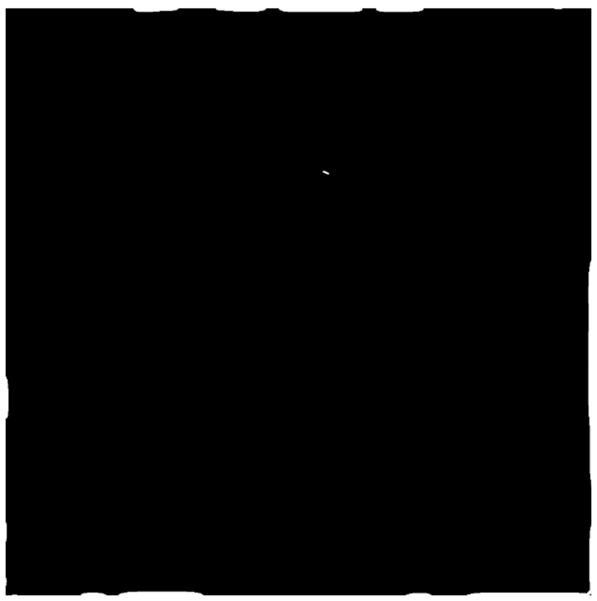
(a) N = 21



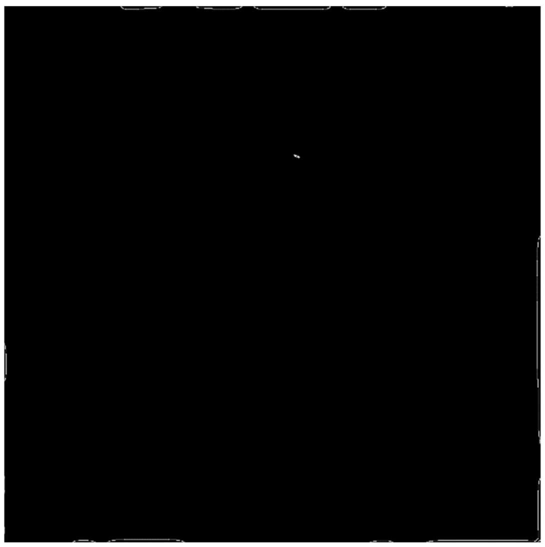
(b) N = 25



(c) N = 29



(d) Combined mask from texture-based masks of bottle image.



(e) Contour

Figure 10. Bottle texture feature segmentation output

Concluding notes

1. In some cases (flower and bottle image), the Otsu RBG performed better than the texture method
2. Different iterations and sliding window was tried for the bottle but the outputs were poor
3. Both algorithms performed poorly on the Dog image. The best output was in the car image