**EX.NO:7 DATE: 2.02.2025**

**EDGE DETECTION TECHNIQUES**

**Aim:**  
To apply edge detection using techniques like Canny, Prewitt and Robert.

**Algorithm:**

1. Canny Edge Detection

* Gaussian Blur: Apply Gaussian blur to the input image to reduce noise using a Gaussian kernel.
* Compute Gradients: Compute the gradient magnitude and orientation using Sobel kernels. The code supports kernel sizes of 3 or 5.
* Non-Maximum Suppression: Suppress non-maximum pixel values to thin out edges.
* Edge Tracking by Hysteresis: Track edges by hysteresis using a double threshold.

2. Prewitt Edge Detection

* Apply Prewitt Operator: Convolve the image with Prewitt operator masks in both horizontal (Mx) and vertical (My) directions.
* Calculate Gradient Magnitude: Compute the gradient magnitude using the formula sqrt(Gx^2 + Gy^2).
* Thresholding: Apply a threshold value to the gradient magnitude.
* Binarization: Binarize the image based on the thresholded values to produce a final edge map.

3. Roberts Cross Edge Detection

* Apply Roberts Cross Operator: Convolve the image with Roberts Cross kernels (kernel\_x and kernel\_y).
* Calculate Gradient Magnitude: Compute the gradient magnitude using the formula sqrt(Gx^2 + Gy^2).
* Normalization: Normalize the resulting image to a 0-255 range for proper display.

**Code:**

import cv2  
import numpy as np  
from PIL import Image  
import matplotlib.pyplot as plt

image = cv2.imread('lena.jpg',0)  
plt.imshow(image, cmap='gray')  
plt.title("Original Image")

Text(0.5, 1.0, 'Original Image')



def apply\_gaussian\_blur(image, kernel\_size):  
 def gaussian\_kernel(size, sigma):  
 kernel = np.fromfunction(  
 lambda x, y: (1/ (2 \* np.pi \* sigma \*\* 2)) \*   
 np.exp(- ((x - (size-1)/2) \*\* 2 + (y - (size-1)/2) \*\* 2) / (2 \* sigma \*\* 2)),  
 (size, size)  
 )  
 return kernel / np.sum(kernel)  
  
   
 if kernel\_size % 2 == 0:  
 kernel\_size += 1  
  
   
 kernel = gaussian\_kernel(kernel\_size, sigma=1.0)  
  
   
 rows, cols = image.shape  
  
   
 k\_half = kernel\_size // 2  
  
   
 output = np.zeros\_like(image)   
  
 for i in range(k\_half, rows - k\_half):  
 for j in range(k\_half, cols - k\_half):  
 output[i, j] = np.sum(image[i - k\_half: i + k\_half + 1, j - k\_half: j + k\_half + 1] \* kernel)  
  
 return output[k\_half:rows-k\_half, k\_half:cols-k\_half]

def compute\_gradient\_magnitude\_and\_orientation(image, sobel\_kernel\_size):  
  
 if sobel\_kernel\_size == 3:  
 sobel\_x = np.array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]])  
 sobel\_y = np.array([[-1, -2, -1], [0, 0, 0], [1, 2, 1]])  
  
 elif sobel\_kernel\_size == 5:  
 sobel\_x = np.array([[-1, -2, 0, 2, 1], [-2, -3, 0, 3, 2], [-3, -5, 0, 5, 3], [-2, -3, 0, 3, 2], [-1, -2, 0, 2, 1]])  
 sobel\_y = np.array([[-1, -2, -3, -2, -1], [-2, -3, -5, -3, -2], [0, 0, 0, 0, 0], [2, 3, 5, 3, 2], [1, 2, 3, 2, 1]])  
 else:  
 sys.exit("Sobel kernel size should be 3 or 5!")  
  
 rows, cols = image.shape  
  
 gradient\_x = np.zeros\_like(image, dtype=np.float64)  
 gradient\_y = np.zeros\_like(image, dtype=np.float64)  
  
 half\_size = sobel\_kernel\_size // 2  
 for i in range(half\_size, rows - half\_size):  
 for j in range(half\_size, cols - half\_size):  
 window = image[i - half\_size:i + half\_size + 1, j - half\_size:j + half\_size + 1]  
 gradient\_x[i, j] = np.sum(window \* sobel\_x)  
 gradient\_y[i, j] = np.sum(window \* sobel\_y)  
  
 magnitude = np.sqrt(gradient\_x \*\* 2 + gradient\_y \*\* 2)  
   
 orientation = np.arctan2(gradient\_y, gradient\_x)  
  
 return magnitude, orientation

def apply\_non\_max\_suppression(magnitude, orientation):  
 suppressed\_magnitude = np.copy(magnitude)  
 rows, cols = magnitude.shape  
   
 for i in range(1, rows - 1):  
 for j in range(1, cols - 1):  
 angle = orientation[i][j]  
 q = [0, 0]  
 if (-np.pi/8 <= angle < np.pi/8) or (7\*np.pi/8 <= angle):  
 q[0] = magnitude[i][j+1]  
 q[1] = magnitude[i][j-1]  
 elif (np.pi/8 <= angle < 3\*np.pi/8):  
 q[0] = magnitude[i+1][j+1]  
 q[1] = magnitude[i-1][j-1]  
 elif (3\*np.pi/8 <= angle < 5\*np.pi/8):  
 q[0] = magnitude[i+1][j]  
 q[1] = magnitude[i-1][j]  
 else:  
 q[0] = magnitude[i-1][j+1]  
 q[1] = magnitude[i+1][j-1]  
   
 if magnitude[i][j] < max(q[0], q[1]):  
 suppressed\_magnitude[i][j] = 0  
   
 return suppressed\_magnitude

def apply\_edge\_tracking\_by\_hysteresis(magnitude, low\_threshold, high\_threshold):  
 rows, cols = magnitude.shape  
 edge\_map = np.zeros((rows, cols), dtype=np.uint8)  
   
 strong\_edge\_i, strong\_edge\_j = np.where(magnitude >= high\_threshold)  
 weak\_edge\_i, weak\_edge\_j = np.where((magnitude >= low\_threshold) & (magnitude < high\_threshold))  
   
 edge\_map[strong\_edge\_i, strong\_edge\_j] = 255  
  
 for i, j in zip(weak\_edge\_i, weak\_edge\_j):  
 if (edge\_map[i-1:i+2, j-1:j+2] == 255).any():  
 edge\_map[i, j] = 255  
   
 return edge\_map

smoothed\_image = apply\_gaussian\_blur(image, 5)  
  
magnitude, orientation = compute\_gradient\_magnitude\_and\_orientation(smoothed\_image, 3)  
  
mx = apply\_non\_max\_suppression(magnitude, orientation)  
  
low\_threshold, high\_threshold = 70, 100  
emap = apply\_edge\_tracking\_by\_hysteresis(mx, low\_threshold, high\_threshold)  
  
plt.figure(figsize=(10,6))  
plt.subplot(1,2,1)  
plt.imshow(image,cmap='gray')  
plt.title("Original Image")  
plt.subplot(1,2,2)  
plt.imshow(emap, cmap='gray')  
plt.title("Canny Image")

Text(0.5, 1.0, 'Canny Image')

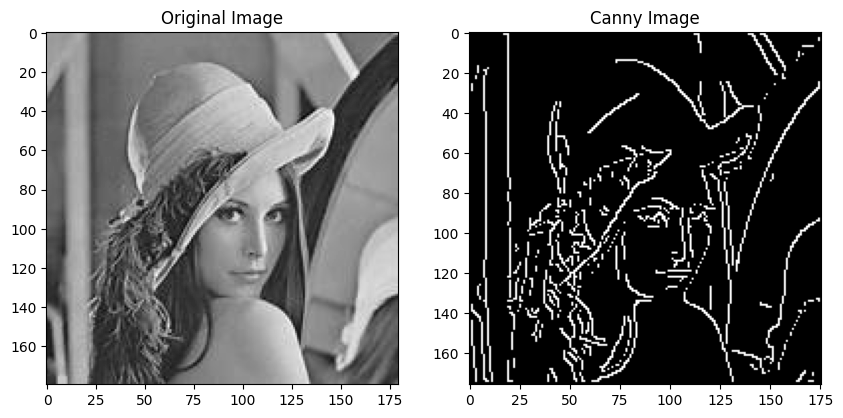
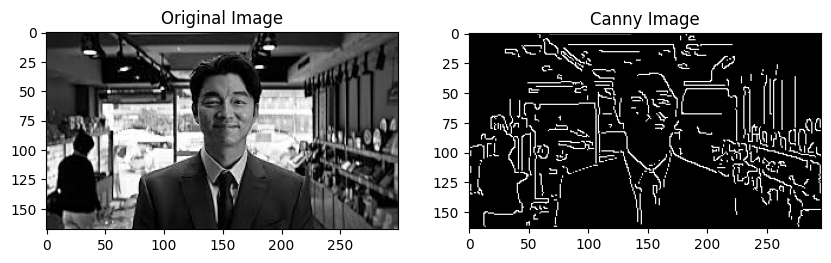
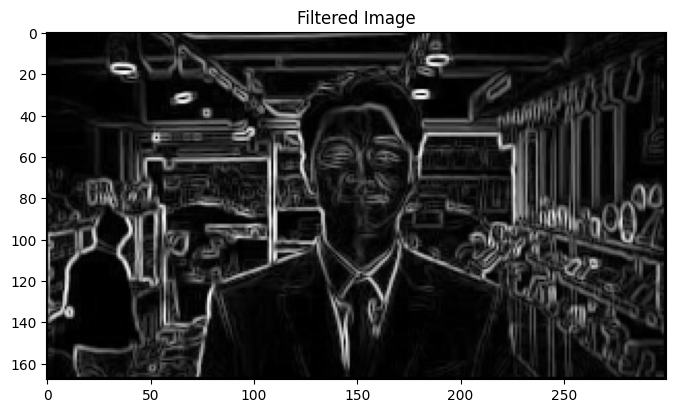


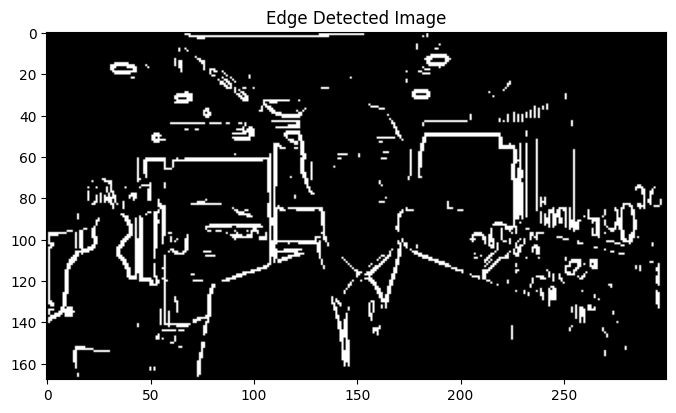
image = cv2.imread("download.jpeg", 0)  
smoothed\_image = apply\_gaussian\_blur(image, 5)  
  
magnitude, orientation = compute\_gradient\_magnitude\_and\_orientation(smoothed\_image, 3)  
  
mx = apply\_non\_max\_suppression(magnitude, orientation)  
  
low\_threshold, high\_threshold = 125, 200  
emap = apply\_edge\_tracking\_by\_hysteresis(mx, low\_threshold, high\_threshold)  
  
plt.figure(figsize=(10,6))  
plt.subplot(1,2,1)  
plt.imshow(image,cmap='gray')  
plt.title("Original Image")  
plt.subplot(1,2,2)  
plt.imshow(emap, cmap='gray')  
plt.title("Canny Image")

Text(0.5, 1.0, 'Canny Image')

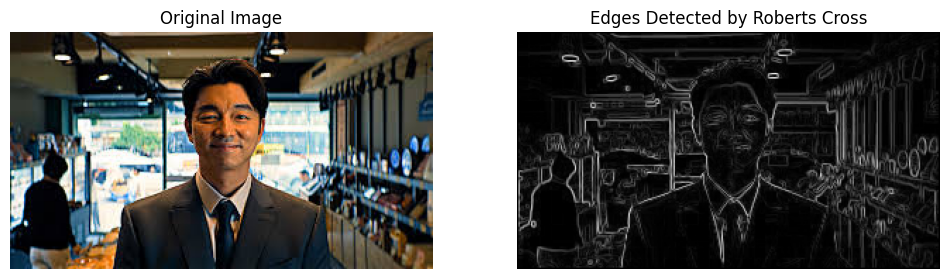


def apply\_prewitt\_edge\_detection(input\_image):  
   
 input\_image = input\_image.astype(np.float64)  
   
   
 filtered\_image = np.zeros(input\_image.shape)  
   
 Mx = np.array([-1, 0, 1, -1, 0, 1, -1, 0, 1]).reshape(3, 3)  
 My = np.array([-1, -1, -1, 0, 0, 0, 1, 1, 1]).reshape(3, 3)  
   
 for i in range(1, input\_image.shape[0] - 1):  
 for j in range(1, input\_image.shape[1] - 1):  
   
 Gx = np.sum(Mx \* input\_image[i-1:i+2, j-1:j+2])  
 Gy = np.sum(My \* input\_image[i-1:i+2, j-1:j+2])  
   
 filtered\_image[i, j] = np.sqrt(Gx\*\*2 + Gy\*\*2)  
   
 filtered\_image = (filtered\_image / filtered\_image.max() \* 255).astype(np.uint8)  
 plt.figure(figsize=(8, 6))  
 plt.imshow(filtered\_image, cmap='gray')  
 plt.title('Filtered Image')  
 plt.show()  
   
 threshold\_value = 100 # varies between [0 255]  
 output\_image = np.maximum(filtered\_image, threshold\_value)  
 output\_image[output\_image == threshold\_value] = 0  
 output\_image = (output\_image > 0).astype(np.uint8) \* 255  
   
 plt.figure(figsize=(8, 6))  
 plt.imshow(output\_image, cmap='gray')  
 plt.title('Edge Detected Image')  
 plt.show()  
   
input\_image = np.array(Image.open('download.jpeg').convert('L'))  
  
apply\_prewitt\_edge\_detection(input\_image)





def roberts\_cross\_edge\_detection(input\_image):  
 if len(input\_image.shape) == 3:  
 input\_image = np.mean(input\_image, axis=2).astype(np.uint8)  
   
 kernel\_x = np.array([[1, 0], [0, -1]])  
 kernel\_y = np.array([[0, 1], [-1, 0]])  
   
 output\_image = np.zeros(input\_image.shape)  
   
 for i in range(1, input\_image.shape[0] - 1):  
 for j in range(1, input\_image.shape[1] - 1):  
 patch = input\_image[i-1:i+1, j-1:j+1]  
 Gx = np.sum(patch \* kernel\_x)  
 Gy = np.sum(patch \* kernel\_y)  
 output\_image[i, j] = np.sqrt(Gx\*\*2 + Gy\*\*2)  
   
 output\_image = (output\_image / output\_image.max() \* 255).astype(np.uint8)  
   
 return output\_image  
  
input\_image = np.array(Image.open('download.jpeg'))  
output\_image = roberts\_cross\_edge\_detection(input\_image)  
  
plt.figure(figsize=(12, 6))  
plt.subplot(1, 2, 1)  
plt.imshow(input\_image, cmap='gray')  
plt.title('Original Image')  
plt.axis('off')  
  
plt.subplot(1, 2, 2)  
plt.imshow(output\_image, cmap='gray')  
plt.title('Edges Detected by Roberts Cross')  
plt.axis('off')  
  
plt.show()



**Inference:** The algorithms identify edges by detecting changes in pixel intensity. Higher gradient magnitudes correspond to stronger edge presence. Thresholding helps to isolate prominent edges.

**Result:** The output images highlight the detected edges, effectively representing boundaries and contours within the original image.