**CO543: Image Processing**

**Lab 6 – Feature Extraction**

Ranage R.D.P.R. - E/19/310

The following is the original image that was used to implement the functions in the lab tasks.



The following functions are used throughout the lab to plot each figure to show the results.

import cv2

import numpy as np

import matplotlib.pyplot as plt

def show\_2\_images(image1, image2, title1, title2):

    fig, ax = plt.subplots(1, 2, figsize=(10, 5))

    ax[0].imshow(image1, cmap='gray')

    ax[0].set\_title(title1)

    ax[1].imshow( image2, cmap='gray')

    ax[1].set\_title(title2)

    plt.show()

def show\_3\_images(image1, image2, image3, title1, title2, title3):

    fig, ax = plt.subplots(1, 3, figsize=(15, 5))

    ax[0].imshow(image1, cmap='gray')

    ax[0].set\_title(title1)

    ax[1].imshow( image2, cmap='gray')

    ax[1].set\_title(title2)

    ax[2].imshow( image3, cmap='gray')

    ax[2].set\_title(title3)

    plt.show()

**Lab Task 1: Edge Detection**

**1.1) Identify the different edges present in an image using Sobel, Laplacian, and Canny edge detection algorithms, and discuss the differences in their outputs.**

import cv2

import numpy as np

import matplotlib.pyplot as plt

# Load the image

img = plt.imread('image.jpg')

# Sobel Edge Detection

def sobel\_edge\_detection(img):

    # Convert the image to grayscale

    gray = cv2.cvtColor(img, cv2.COLOR\_BGR2GRAY)

    # Apply Gaussian Blur

    blur = cv2.GaussianBlur(gray, (3, 3), 0)

    # Apply Sobel operator in X direction

    sobelx = cv2.Sobel(blur, cv2.CV\_64F, 1, 0, ksize=5)

    # Apply Sobel operator in Y direction

    sobely = cv2.Sobel(blur, cv2.CV\_64F, 0, 1, ksize=5)

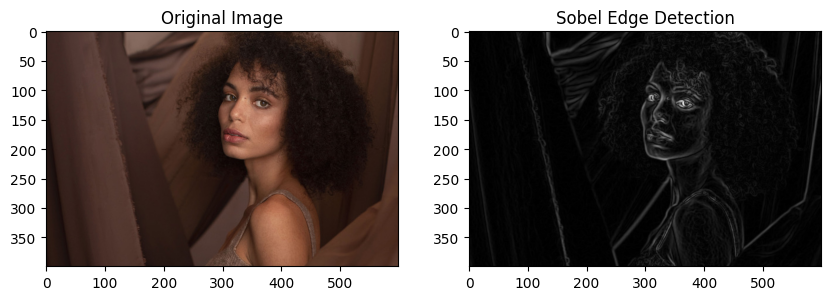
    # Calculate the gradient magnitude

    magnitude = np.sqrt(sobelx\*\*2 + sobely\*\*2)

    return magnitude

sobel\_img = sobel\_edge\_detection(img)

show\_2\_images(img, sobel\_img, 'Original Image', 'Sobel Edge Detection')

****

# Laplacian Edge Detection

def laplacian\_edge\_detection(img):

    # Convert the image to grayscale

    gray = cv2.cvtColor(img, cv2.COLOR\_BGR2GRAY)

    # Apply Gaussian Blur

    blur = cv2.GaussianBlur(gray, (3, 3), 0)

    # Apply Laplacian operator

    laplacian = cv2.Laplacian(blur, cv2.CV\_64F)

    return laplacian

laplacian\_img = laplacian\_edge\_detection(img)

show\_2\_images(img, laplacian\_img, 'Original Image', 'Laplacian Edge Detection')

**A person with curly hair and a graph

Description automatically generated**

# Canny Edge Detection

def canny\_edge\_detection(img):

    # Apply Gaussian Blur

    blur = cv2.GaussianBlur(img, (3, 3), 0)

    # Apply Canny Edge Detection

    canny = cv2.Canny(blur, 100, 150)

    return canny

canny\_img = canny\_edge\_detection(img)

show\_2\_images(img, canny\_img, 'Original Image', 'Canny Edge Detection')

**A person with curly hair and a graph

Description automatically generated**

**Sobel Edge Detection:** Calculates gradients in both x and y directions to capture edge information, particularly in specific orientations. It is sensitive to noise, resulting in less clean edges compared to other methods like Canny.

**Laplacian Edge Detection:** Highlights regions of rapid intensity change and detects edges regardless of orientation. However, it is more sensitive to noise, which can lead to the detection of spurious edges.

**Canny Edge Detection:** A sophisticated method that reduces noise, resulting in cleaner edges. It uses steps like Gaussian blur, gradient calculation, non-maximum suppression, and edge tracking by hysteresis, making it accurate and widely preferred for detecting true edges.

**1.2) Using the provided image jigsaw.jpg, identify the boundary lines of the puzzle piece.**

**Follow the below steps to obtain the lines:**

**i. Crop the region containing the puzzle piece using a simple python matrix manipulation.**

# Load the image

image = cv2.imread('jigsaw.jpg')

image\_rgb = cv2.cvtColor(image, cv2.COLOR\_BGR2RGB)

# Define the cropping coordinates

x\_start, y\_start, x\_end, y\_end = 1250, 2000, 1750, 2350

# Crop the image

cropped\_image = image\_rgb[y\_start:y\_end, x\_start:x\_end]

**A close-up of a puzzle piece

Description automatically generated**

**ii. Apply simple preprocessing techniques to convert the cropped image to grayscale and remove noises (e.g: Blurring, Thresholding).**

gray\_image = cropped\_image.mean(axis=2)

# Apply Gaussian Blurring

blurred\_image = cv2.GaussianBlur(gray\_image, (5, 5), 0)

# Apply Thresholding

\_, thresholded\_image = cv2.threshold(blurred\_image, 127, 255, cv2.THRESH\_BINARY)

thresholded\_image = thresholded\_image.astype('uint8')

show\_3\_images(gray\_image, blurred\_image, thresholded\_image, 'Gray Image', 'Blurred Image', 'Thresholded Image')

**A piece of a puzzle

Description automatically generated**

**iii. Perform edge detection on the binarized image using a suitable edge detection algorithm (e.g: Canny Edge Detection).**

cannyImg = canny\_edge\_detection(thresholded\_image)

plt.imshow(cannyImg, cmap='gray')

**A black and white puzzle piece

Description automatically generated**

**iv. Apply the Hough Transform (cv2.HoughLines)to detect lines that bound the four main corners of the jigsaw piece.**

def hough\_line\_detection(edges, img):

    lines = cv2.HoughLines(edges, rho=2, theta=np.pi/360, threshold=80)

    img\_copy = img.copy()

    if lines is not None:

        for line in lines:

            rho, theta = line[0]

            a = np.cos(theta)

            b = np.sin(theta)

            x0 = a \* rho

            y0 = b \* rho

            x1 = int(x0 + 1000 \* (-b))

            y1 = int(y0 + 1000 \* (a))

            x2 = int(x0 - 1000 \* (-b))

            y2 = int(y0 - 1000 \* (a))

            cv2.line(img\_copy, (x1, y1), (x2, y2), (0, 0, 255), 2)

    return img\_copy

img\_with\_lines = hough\_line\_detection(cannyImg, cropped\_image)

plt.imshow(img\_with\_lines, cmap='gray')

plt.title('Detected Lines (Hough Transform)')

plt.axis('off')

plt.show()

**A piece of a puzzle

Description automatically generated**

**v. Explain the impact of the rho, theta, and threshold parameters of Hough transformation in detecting lines.**

Rho (ρ) determines how finely distances are measured from the origin to detect lines. Smaller rho values give more precise results but require more computation. Larger rho values are faster but might miss details.

Theta (θ) decides the angular resolution for detecting lines of different orientations. Smaller theta values detect lines more accurately but are slower. Larger theta values are faster but might confuse lines that are nearly parallel or perpendicular.

Threshold sets the minimum votes needed in the Hough space to confirm a line. A higher threshold avoids detecting noise but might miss faint lines. A lower threshold detects more lines, including noise.

**Lab Task 2: Corner Detection**

**2.1) Apply Harris, Shi-Tomasi, and SIFT algorithms on an image to identify corners and discuss the differences in these algorithms.**

def apply\_harris(img):

    gray = cv2.cvtColor(img, cv2.COLOR\_BGR2GRAY)

    gray = np.float32(gray)

    harris\_corners = cv2.cornerHarris(gray, blockSize=2, ksize=3, k=0.04)

    harris\_corners = cv2.dilate(harris\_corners, None)

    img\_copy = img.copy()  # Create a writable copy of the image

    img\_copy[harris\_corners > 0.01 \* harris\_corners.max()] = [0, 0, 255]

    return img\_copy

img = plt.imread('blox.jpg')

harris\_img = apply\_harris(img)

show\_2\_images(img, harris\_img, 'Original Image', 'Harris Corners')

**A collage of different shapes

Description automatically generated**

def apply\_shi\_tomasi(img):

    gray = cv2.cvtColor(img, cv2.COLOR\_BGR2GRAY)

    corners = cv2.goodFeaturesToTrack(gray, maxCorners=100, qualityLevel=0.01, minDistance=10)

    corners = np.intp(corners)

    img\_copy = img.copy()  # Create a writable copy of the image

    for corner in corners:

        x, y = corner.ravel()

        cv2.circle(img\_copy, (x, y), 3, (0, 255, 0), -1)

    return img\_copy

shi\_tomasi\_img = apply\_shi\_tomasi(img)

show\_2\_images(img, shi\_tomasi\_img, 'Original Image', 'Shi-Tomasi Corners')

**A collage of different shapes

Description automatically generated**

def apply\_sift(img):

    gray = cv2.cvtColor(img, cv2.COLOR\_BGR2GRAY)

    sift = cv2.SIFT\_create()

    keypoints, descriptors = sift.detectAndCompute(gray, None)

    img\_copy = img.copy()  # Create a writable copy of the image

    img\_copy = cv2.drawKeypoints(img\_copy, keypoints, None, color=(255, 0, 0))

    return img\_copy

sift\_img = apply\_sift(img)

show\_2\_images(img, sift\_img, 'Original Image', 'SIFT Keypoints')

A collage of images of different shapes

Description automatically generated

**2.2) Using the provided image jigsaw.jpg, identify the corners present in the puzzle piece.**

gray = cv2.cvtColor(cropped\_image, cv2.COLOR\_BGR2GRAY)

# Shi-Tomasi Corner Detection

shi\_tomasi\_corners = cv2.goodFeaturesToTrack(gray, 10, 0.01, 10)

shi\_tomasi\_corners = np.int0(shi\_tomasi\_corners)

# Plot the results

for corner in shi\_tomasi\_corners:

    x = corner[0][0]

    y = corner[0][1]

    #filtration to get the required points only based on observation

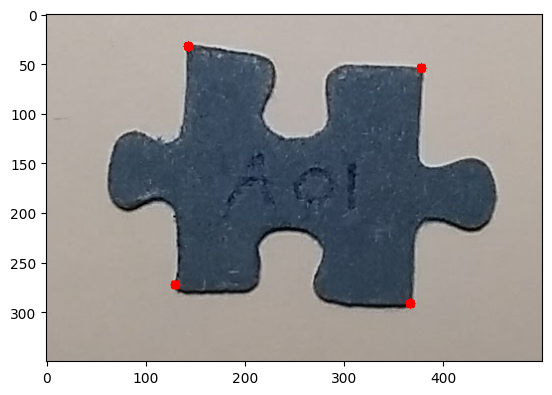
    if x < 120 or (x>145 and x<360) or (y>100 and y<250) :

        continue

    x, y = corner.ravel()

    cv2.circle(cropped\_image, (x, y), 5, 255, -1)

plt.imshow(cropped\_image, cmap='gray')

****