**INDIAN INSTITUTE OF INFORMATION TECHNOLOGY SURAT**

**IIIT SURAT**

****

**Image Processing & Computer Vision(EC703)**

**LAB RECORD**

**Submitted by: -**

**Dev Yadav (UI22EC17)**

**3rd Year ECE**

**IIIT Surat**

# Table of Contents

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **S. no.** | **Title** | **Page no.** | **Date** | **signature** |
| **1.** | **Lab 1: Color Image Conversions and 2D Function Plotting** |  |  |  |
| **2.** | **Lab 2: Basic Arithmetic and Brightness/Contrast Manipulation on RGB Images** |  |  |  |
| **3.** | **Lab 3: Grayscale Image Transformations (Log, Power-Law, Contrast Stretching)** |  |  |  |
| **4.** | **Lab 4: Gray Level Slicing and Bit Plane Slicing of Grayscale Images** |  |  |  |
| **5.** | **Lab 5: Image Restoration with Spatial Filters on Noisy Images** |  |  |  |
| **6.** | **Lab 6: High-Pass Filtering Techniques for Enhanced Spatial Filtering** |  |  |  |
| **7.** | **Lab 7: Edge Detection Using Prewitt and Sobel Operators** |  |  |  |
| **8.** | **Lab 8: Edge Detection Using Sobel, Canny, and Marr-Hildreth (LOG) Operators** |  |  |  |
| **9.** | **Lab 9: Image Sharpening with Laplacian, Unsharp Masking, and Highboost Filtering** |  |  |  |
| **10.** | **Lab 10: Discrete Fourier Transform (DFT) Analysis and Reconstruction** |  |  |  |
| **11.** | **Lab 11: Morphological Operations (Erosion, Dilation, Opening, Closing) on Images** |  |  |  |
| **12.** |  |  |  |  |
| **13.** |  |  |  |  |
| **14.** |  |  |  |  |
| **15.** |  |  |  |  |

**Lab 1**

**Take a RGB image and perform following operations 1. RGB to Gray 2. RGB to index 3. RGB to binary with and without using function .**

**Create an Black image a white image and a black in white image.**  
  
IN MATLAB

%% Dev Yadav UI22EC17 6 Aug 2024

%take a RGB image and perform following operations

%RGB to Gray

%RGB to index

% RGB to binary

clc;

clear

RGB = imread('lena.jpg');

figure(1);

subplot(2,2,1)

imshow(RGB)

title('original image UI22EC17')

x= rgb2gray(RGB); % function

subplot(2,2,2)

imshow(x)

title('gray image (function)')

% RGB TO GRAY MANUALLY

R = RGB(:,:,1); % Red channel

G = RGB(:,:,2); % Green channel

B = RGB(:,:,3); % Blue channel

gray = 0.2989 \* R + 0.5870 \* G + 0.1140 \* B;

subplot(2,2,3)

imshow(gray)

title('gray image manually')

% RGB to index

RGB2 = imread('lotus.jpg');

figure , subplot(2,2,1)

imshow(RGB2)

title('original image UI22EC17')

[IND,map] = rgb2ind(RGB2,32);

subplot(2,2,2)

imagesc(IND)

colormap(map)

axis image

title('index image (function)')

zoom(1)

RGB3 =imread('messi.jpg')

% RGB IN BINARY USING FUNCTION

BiW = im2bw(RGB3,map,0.4); % function

figure,subplot(2,2,1)

imshow(RGB3)

title('original image UI22EC17')

subplot(2,2,2)

imshow(BiW);

title('binary image (function)')

t= 128 ; % manually

R = RGB3(:,:,1); %Red

G = RGB3(:,:,2); % Green

B = RGB3(:,:,3); % Blue

grayimage = 0.2989 \* R + 0.5870 \* G + 0.1140 \* B;

bin = grayimage > t ;

subplot(2,2,3)

imshow(bin)

title('binary image without function')

figure , subplot(2,2,1)

Black=zeros(255,255,'uint8');

imshow(Black)

title('Black image ')

subplot(2,2,2)

White=ones(255,0,'uint8');

imshow(White)

title('White Image')

subplot(2,2,3)

BW=checkerboard(32,4,4)>0.5 ;

imshow(BW)

title('Black in white image')

**IN PYTHON**

import numpy as np

import matplotlib.pyplot as plt

from skimage import io, color

from skimage.util import img\_as\_ubyte

from skimage.color import rgb2gray, rgb2hed

RGB = io.imread('lena.jpg')

# Displaying the original image

plt.figure(figsize=(10, 10))

plt.subplot(2, 2, 1)

plt.imshow(RGB)

plt.title('Original Image UI22EC17')

# RGB to Gray using function

gray\_function = rgb2gray(RGB)

plt.subplot(2, 2, 2)

plt.imshow(gray\_function, cmap='gray')

plt.title('Gray Image Using Function')

# RGB to Gray Manually

R = RGB[:, :, 0]

G = RGB[:, :, 1]

B = RGB[:, :, 2]

gray\_manual = 0.2989 \* R + 0.5870 \* G + 0.1140 \* B

plt.subplot(2, 2, 3)

plt.imshow(gray\_manual, cmap='gray')

plt.title('Gray Image Manually')

# Loading the second image for the next operations

RGB2 = io.imread('messi.jpg')

# RGB to Index using function

IND, map = rgb2hed(RGB2), None # Simulating index image with rgb2hed

plt.figure()

plt.subplot(2, 2, 1)

plt.imshow(RGB2)

plt.title('Original Image UI22EC17')

plt.subplot(2, 2, 2)

plt.imshow(IND)

plt.title('Index Image Using Function')

# RGB to Binary using function

binary\_function = gray\_function > 0.4

plt.figure()

plt.subplot(2, 2, 1)

plt.imshow(RGB)

plt.title('Original Image UI22EC17')

plt.subplot(2, 2, 2)

plt.imshow(binary\_function, cmap='gray')

plt.title('Binary Image Using Function')

# RGB to Binary Manually

threshold = 128

grayimage = img\_as\_ubyte(gray\_function)

binary\_manual = grayimage > threshold

plt.subplot(2, 2, 3)

plt.imshow(binary\_manual, cmap='gray')

plt.title('Binary Image Without Using Function')

# Displaying Black, White, and Checkerboard Images

plt.figure()

plt.subplot(2, 2, 1)

black\_image = np.zeros((255, 255), dtype=np.uint8)

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

plt.title('Black Image')

plt.subplot(2, 2, 2)

white\_image = np.ones((0 , 0), dtype=np.uint8)

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

plt.title('White Image')

plt.subplot(2, 2, 3)

checkerboard = np.kron([[1, 0] \* 4, [0, 1] \* 4] \* 4, np.ones((32, 32)))

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

plt.title('Checkerboard Image')

plt.show()

import numpy as np

import matplotlib.pyplot as plt

# linspace(start, stop, number of evenly spaced values)

x = np.linspace(-10, 10, 400)

y = np.linspace(-10, 10, 400)

X, Y = np.meshgrid(x, y)

u = np.random.rand()

v = np.random.rand()

# Ensure values for arcsin are within the valid range [-1, 1]

Z = np.clip(u \* X + v \* Y, -1, 1)

Z = np.arcsin(Z)

plt.imshow(Z, extent=[-10, 10, -10, 10], origin='lower')

plt.title(f'2D function f(x, y) = asin({u:.2f}x + {v:.2f}y)')

plt.colorbar(label='f(x, y)')

plt.show()

# Without using the built-in function

Z = np.zeros(X.shape)

for i in range(X.shape[0]):

for j in range(X.shape[1]):

value = u \* X[i, j] + v \* Y[i, j]

Z[i, j] = np.arcsin(np.clip(value, -1, 1))

plt.imshow(Z, extent=[-10, 10, -10, 10], origin='lower')

plt.title(f'2D function f(x, y) = asin({u:.2f}x + {v:.2f}y)')

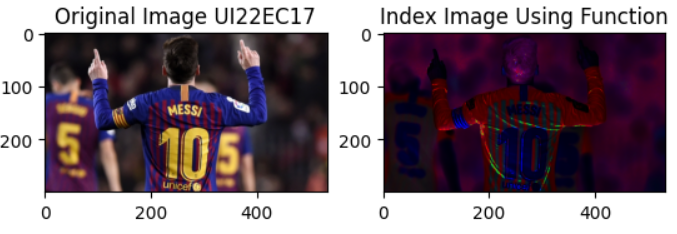
plt.colorbar(label='f(x, y)')

plt.show()

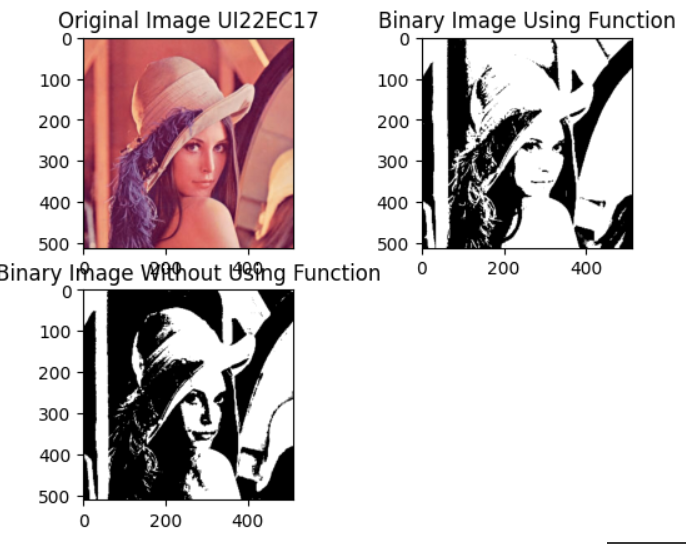
**Figure 1**

****

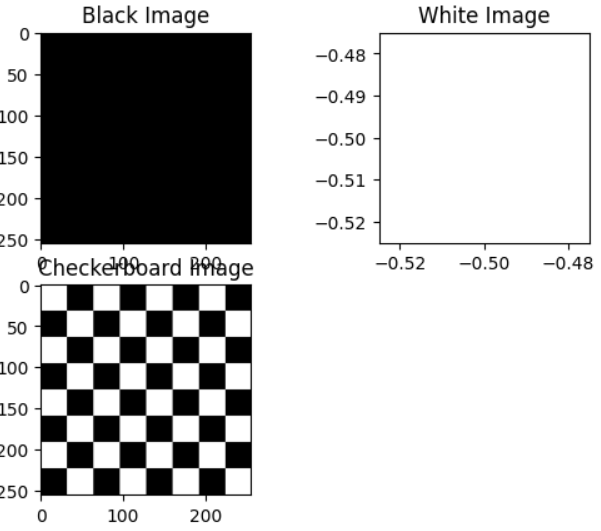
**Figure 2**

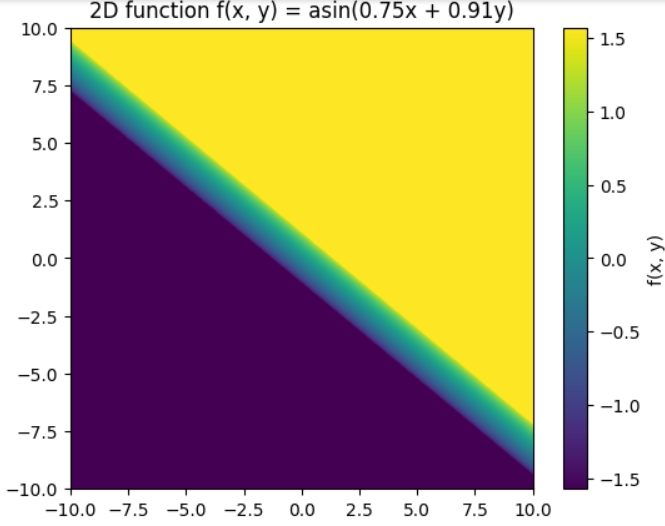
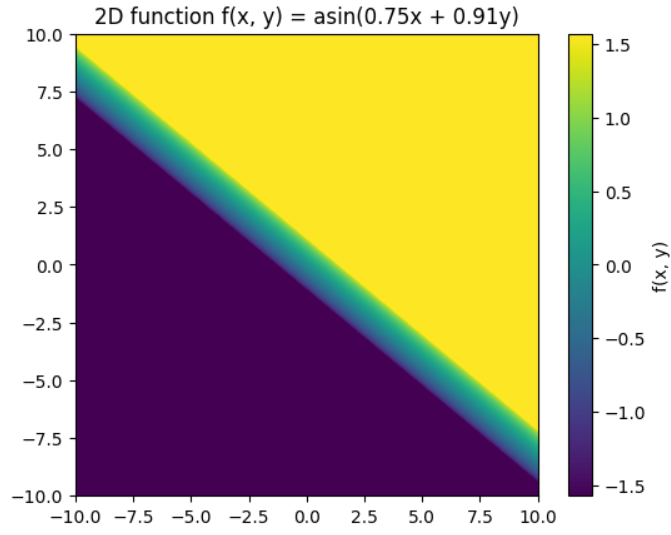
****

**Figure 3**

****

**Figure 4**

****

****

**Lab 2**

**In matlab**

clc;

clear all;

% DEV YADAV UI22EC17 12 AUG 2024

% LAB 2

% brightness contrast negative image

% multiplication division addition substraction of two images

% brightness change

color = imread('lotus.jpg');

figure(1), subplot(2,2,1)

imshow(color)

title('Original ')

subplot(2,2,2)

f = 50;

inc = color + f;

inc = min(max(inc, 0), 255);

imshow(inc)

title('High Brightness')

sgtitle(' UI22EC17')

subplot(2,2,3)

k = 50;

dec = color - k;

dec = min(max(dec, 0), 255);

imshow(dec)

title('Low Brightness')

% contrast change

color = imread('lena.jpg');

figure(2), subplot(2,2,1)

imshow(color)

title('Original')

subplot(2,2,2)

f = 1.7;

inc = color \* f;

inc = min(max(inc, 0), 255);

imshow(inc)

title('High Contrast')

subplot(2,2,3)

k = 0.6;

dec = color \* k;

dec = min(max(dec, 0), 255);

imshow(dec)

title('Low Contrast')

% negative image

color = imread('lena.jpg');

figure(3), subplot(2,2,1)

imshow(color)

title('Original')

subplot(2,2,2)

neg = 255 - color;

imshow(neg)

title('Negative Image')

% addition, subtraction, multiplication, division

image1 = imread('lena.jpg');

image2 = imread('lotus.jpg');

% Resize image2 to match the size of image1

image2\_resized = imresize(image2, [size(image1, 1), size(image1, 2)]);

figure(4),

sgtitle('arithmetic operations UI22EC17')

subplot(3,3,1)

imshow(image1)

title('Image 1')

subplot(3,3,2)

imshow(image2\_resized)

title('Image 2 (Resized)')

img1 = im2gray(image1);

img2 = im2gray(image2\_resized);

subplot(3,3,3)

imshow(img1)

title('Gray Image 1')

subplot(3,3,4)

imshow(img2)

title('Gray Image 2')

subplot(3,3,5)

% Addition

add = double(img1) + double(img2); % Convert to double for arithmetic

add = uint8(min(max(add, 0), 255)); % Clip to range and convert back to uint8

imshow(add)

title('Addition')

subplot(3,3,6)

% Subtraction

sub = double(img1) - double(img2);

sub = uint8(min(max(sub, 0), 255));

imshow(sub)

title('Subtraction')

subplot(3,3,7)

% Multiplication

mul = double(img1) .\* double(img2);

mul = uint8(min(max(mul, 0), 255));

imshow(mul)

title('Multiplication')

subplot(3,3,8)

% Division (Avoid division by zero)

div = double(img1) ./ (double(img2) + eps); % Adding eps to avoid division by zero

div = uint8(min(max(div, 0), 255));

imshow(div)

title('Division')

**In python**

import numpy as np

from PIL import Image

import matplotlib.pyplot as plt

# DEV YADAV UI22EC17 12 AUG 2024

# LAB 2

# Load images

color = Image.open('lotus.jpg')

# Convert to numpy array

color = np.array(color)

# Brightness change

fig1, axs = plt.subplots(2, 2)

axs[0, 0].imshow(color)

axs[0, 0].set\_title('Original')

# Increase brightness

f = 50

inc = np.clip(color + f, 0, 255).astype(np.uint8)

axs[0, 1].imshow(inc)

axs[0, 1].set\_title('High Brightness')

# Decrease brightness

k = 50

dec = np.clip(color - k, 0, 255).astype(np.uint8)

axs[1, 0].imshow(dec)

axs[1, 0].set\_title('Low Brightness')

# Title for the figure

fig1.suptitle('UI22EC17')

plt.tight\_layout()

plt.show()

# Contrast change

color = Image.open('lena.jpg')

color = np.array(color)

fig2, axs = plt.subplots(2, 2)

axs[0, 0].imshow(color)

axs[0, 0].set\_title('Original')

# Increase contrast

f = 1.7

inc = np.clip(color \* f, 0, 255).astype(np.uint8)

axs[0, 1].imshow(inc)

axs[0, 1].set\_title('High Contrast')

# Decrease contrast

k = 0.6

dec = np.clip(color \* k, 0, 255).astype(np.uint8)

axs[1, 0].imshow(dec)

axs[1, 0].set\_title('Low Contrast')

plt.tight\_layout()

plt.show()

# Negative image

fig3, axs = plt.subplots(1, 2)

axs[0].imshow(color)

axs[0].set\_title('Original')

# Negative image

neg = 255 - color

axs[1].imshow(neg)

axs[1].set\_title('Negative Image')

plt.tight\_layout()

plt.show()

# Arithmetic operations

image1 = Image.open('lena.jpg')

image2 = Image.open('lotus.jpg')

# Resize image2 to match the size of image1

image2\_resized = image2.resize(image1.size)

# Convert to numpy array and grayscale

img1 = np.array(image1.convert('L'))

img2 = np.array(image2\_resized.convert('L'))

fig4, axs = plt.subplots(3, 3)

fig4.suptitle('Arithmetic Operations UI22EC17')

axs[0, 0].imshow(image1)

axs[0, 0].set\_title('Image 1')

axs[0, 1].imshow(image2\_resized)

axs[0, 1].set\_title('Image 2 (Resized)')

axs[0, 2].imshow(img1, cmap='gray')

axs[0, 2].set\_title('Gray Image 1')

axs[1, 0].imshow(img2, cmap='gray')

axs[1, 0].set\_title('Gray Image 2')

# Addition

add = np.clip(img1.astype(np.float32) + img2.astype(np.float32), 0, 255).astype(np.uint8)

axs[1, 1].imshow(add, cmap='gray')

axs[1, 1].set\_title('Addition')

# Subtraction

sub = np.clip(img1.astype(np.float32) - img2.astype(np.float32), 0, 255).astype(np.uint8)

axs[1, 2].imshow(sub, cmap='gray')

axs[1, 2].set\_title('Subtraction')

# Multiplication

mul = np.clip(img1.astype(np.float32) \* img2.astype(np.float32) / 255, 0, 255).astype(np.uint8)

axs[2, 0].imshow(mul, cmap='gray')

axs[2, 0].set\_title('Multiplication')

# Division (Avoid division by zero)

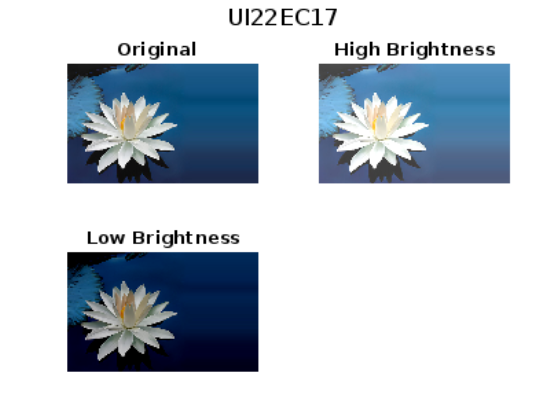
div = np.clip(img1.astype(np.float32) / (img2.astype(np.float32) + 1e-10) \* 255, 0, 255).astype(np.uint8)

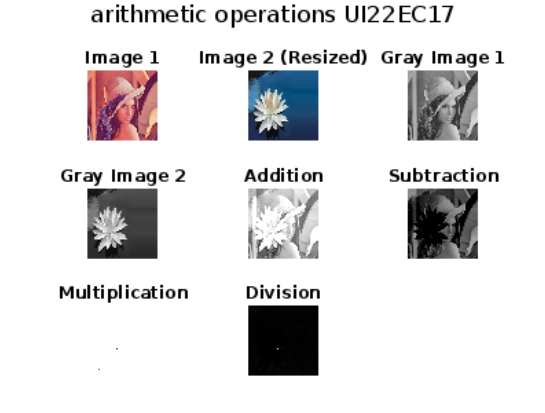
axs[2, 1].imshow(div, cmap='gray')

axs[2, 1].set\_title('Division')

plt.tight\_layout()

plt.show()

****

****

**Lab 3**

% take one particular colour image convert it into gray scale image and

% perform the following operations

%log transformationn (for c= 10 , c= 30 , c= 50

% power law transformation (c=1 ) (for dark image lambda = 0.6, 0.2 )(for brighter image lambda= 4,8)

%contrast streching

clc

clear all

close all

img = imread('lotus.jpg');

gray\_img = rgb2gray(img);

% Display the original grayscale image

figure;

imshow(gray\_img);

title('Original Grayscale Image UI22EC17');

% A) Log Transform

c1 = 10;

c2 = 30;

c3 = 500;

temp=log(1 + double(gray\_img));

log\_img1 = c1 \*temp ;

log\_img2 = c2 \*temp;

log\_img3 = c3 \*temp;

% Normalize to the range [0, 255]

log\_img1 = uint8(255 \* mat2gray(log\_img1));

log\_img2 = uint8(255 \* mat2gray(log\_img2));

log\_img3 = uint8(255 \* mat2gray(log\_img3));

% Display the log transformed images

figure;

subplot(2,2,1);

imshow(log\_img1);

sgtitle('log transform')

title('c=10');

subplot(2,2,2);

imshow(log\_img2);

title('c=30');

subplot(2,2,3);

imshow(log\_img3);

title('c=50');

% B) Power Law Transform (Gamma Correction)

gamma\_dark1 = 0.6;

gamma\_dark2 = 0.2;

gamma\_bright1 = 4;

gamma\_bright2 = 8;

power\_dark1 = c1 \* (double(gray\_img).^gamma\_dark1);

power\_dark2 = c1 \* (double(gray\_img).^gamma\_dark2);

power\_bright1 = c1 \* (double(gray\_img).^gamma\_bright1);

power\_bright2 = c1 \* (double(gray\_img).^gamma\_bright2);

% Normalize to the range [0, 255]

power\_dark1 = uint8(255 \* mat2gray(power\_dark1));

power\_dark2 = uint8(255 \* mat2gray(power\_dark2));

power\_bright1 = uint8(255 \* mat2gray(power\_bright1));

power\_bright2 = uint8(255 \* mat2gray(power\_bright2));

% Display the power law transformed images

figure;

subplot(2,2,1);

imshow(power\_dark1);

sgtitle('Power law transform UI22EC17')

title('Gamma=0.6');

subplot(2,2,2);

imshow(power\_dark2);

title('Gamma=0.2');

subplot(2,2,3);

imshow(power\_bright1);

title('Gamma=4');

subplot(2,2,4);

imshow(power\_bright2);

title('Gamma=8');

% C) Contrast Stretching

min\_intensity = double(min(gray\_img(:)));

max\_intensity = double(max(gray\_img(:)));

stretch = (double(gray\_img) - min\_intensity) \* (255 / (max\_intensity - min\_intensity));

stretch = uint8(stretch);

% Display the contrast stretched image

figure;

imshow(stretch);

title('Contrast Stretching UI22EC17');

**in python**

import cv2

import numpy as np

import matplotlib.pyplot as plt

# Load the image

img = cv2.imread('lotus.jpg')

# Convert the image to grayscale

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

# Display the original grayscale image

plt.figure()

plt.imshow(gray\_img, cmap='gray')

plt.title('Original Grayscale Image UI22EC17')

plt.show()

# A) Log Transform

c1 = 10

c2 = 30

c3 = 50

# Apply log transformation

temp = np.log1p(gray\_img.astype(float)) # log(1 + pixel\_value)

log\_img1 = c1 \* temp

log\_img2 = c2 \* temp

log\_img3 = c3 \* temp

# Normalize the images to the range [0, 255]

log\_img1 = np.uint8(255 \* (log\_img1 / np.max(log\_img1)))

log\_img2 = np.uint8(255 \* (log\_img2 / np.max(log\_img2)))

log\_img3 = np.uint8(255 \* (log\_img3 / np.max(log\_img3)))

# Display the log transformed images

plt.figure(figsize=(10, 8))

plt.subplot(2, 2, 1)

plt.imshow(log\_img1, cmap='gray')

plt.title('c=10')

plt.subplot(2, 2, 2)

plt.imshow(log\_img2, cmap='gray')

plt.title('c=30')

plt.subplot(2, 2, 3)

plt.imshow(log\_img3, cmap='gray')

plt.title('c=50')

plt.suptitle('Log Transform UI22EC17')

plt.show()

# B) Power Law Transform (Gamma Correction)

gamma\_dark1 = 0.6

gamma\_dark2 = 0.2

gamma\_bright1 = 4

gamma\_bright2 = 8

# Apply power law transformation for different gamma values

power\_dark1 = c1 \* (gray\_img.astype(float) \*\* gamma\_dark1)

power\_dark2 = c1 \* (gray\_img.astype(float) \*\* gamma\_dark2)

power\_bright1 = c1 \* (gray\_img.astype(float) \*\* gamma\_bright1)

power\_bright2 = c1 \* (gray\_img.astype(float) \*\* gamma\_bright2)

# Normalize the images to the range [0, 255]

power\_dark1 = np.uint8(255 \* (power\_dark1 / np.max(power\_dark1)))

power\_dark2 = np.uint8(255 \* (power\_dark2 / np.max(power\_dark2)))

power\_bright1 = np.uint8(255 \* (power\_bright1 / np.max(power\_bright1)))

power\_bright2 = np.uint8(255 \* (power\_bright2 / np.max(power\_bright2)))

# Display the power law transformed images

plt.figure(figsize=(10, 8))

plt.subplot(2, 2, 1)

plt.imshow(power\_dark1, cmap='gray')

plt.title('Gamma=0.6')

plt.subplot(2, 2, 2)

plt.imshow(power\_dark2, cmap='gray')

plt.title('Gamma=0.2')

plt.subplot(2, 2, 3)

plt.imshow(power\_bright1, cmap='gray')

plt.title('Gamma=4')

plt.subplot(2, 2, 4)

plt.imshow(power\_bright2, cmap='gray')

plt.title('Gamma=8')

plt.suptitle('Power Law Transform UI22EC17')

plt.show()

# C) Contrast Stretching

min\_intensity = np.min(gray\_img)

max\_intensity = np.max(gray\_img)

# Apply contrast stretching

stretch = (gray\_img.astype(float) - min\_intensity) \* (255 / (max\_intensity - min\_intensity))

stretch = np.uint8(stretch)

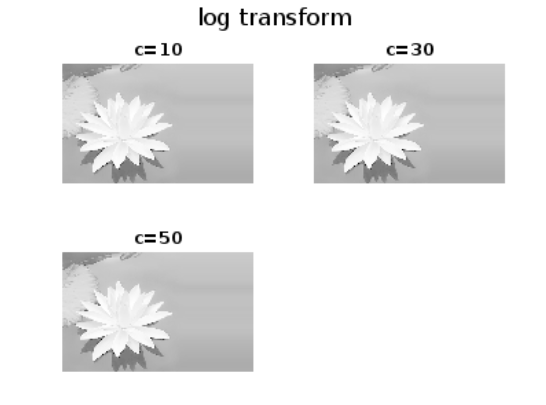
# Display the contrast stretched image

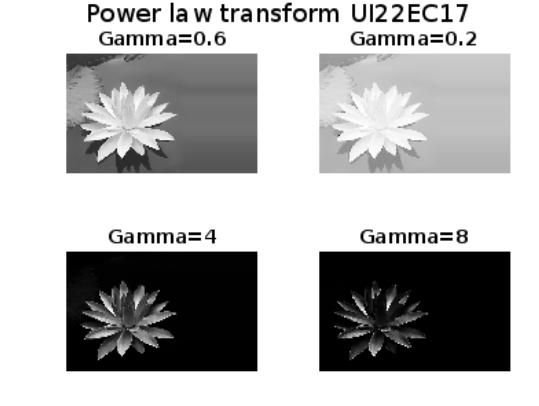
plt.figure()

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

plt.title('Contrast Stretching UI22EC17')

plt.show()

****

****

**Lab 4**

%%Lab4- Take one particular colour image, convert it into a gray-scale image and perform the following transformation operations:

%% 1-Gray level slicing (with and without background).

%% 2-Bit Plane slicing.

clc;

clear all;

close all;

%% Read and Convert Image

img = imread('fractal-iris.tif'); % Load the color image % Convert to grayscale

grayImg = double(img); % Convert to double precision for processing

%% 1. Gray Level Slicing

% Define thresholds

lowThr = 100;

highThr = 150;

% Slicing with background

slicedWithBg = (grayImg >= lowThr & grayImg <= highThr) \* 255;

slicedWithBg = uint8(slicedWithBg);

% Slicing without background

slicedNoBg = grayImg;

slicedNoBg(grayImg < lowThr | grayImg > highThr) = 0;

slicedNoBg = uint8(slicedNoBg);

% Display results in separate figures

figure;

imshow(slicedWithBg);

title('Gray Level Slicing with Background - UI22EC17');

figure;

imshow(slicedNoBg);

title('Gray Level Slicing without Background - UI22EC17');

%% 2. Bit Plane Slicing

% Convert grayscale image to uint8

grayImgUint8 = uint8(grayImg);

% Number of bit planes

nPlanes = 8;

% Display bit planes in separate figures

for i = 1:nPlanes

% Extract bit plane

bitPlane = bitget(grayImgUint8, i);

bitPlaneImg = uint8(bitPlane) \* 255; % Convert for display

% Display each bit plane in a new figure

figure;

imshow(bitPlaneImg);

title(['Bit Plane ', num2str(i), ' - UI22EC17']);

end

**in python**

import cv2

import numpy as np

import matplotlib.pyplot as plt

# Read and Convert Image

img = cv2.imread('fractal-iris.tif', cv2.IMREAD\_GRAYSCALE) # Load the grayscale image

grayImg = img.astype(float) # Convert to float for processing

# 1. Gray Level Slicing

# Define thresholds

lowThr = 100

highThr = 150

# Slicing with background

slicedWithBg = np.where((grayImg >= lowThr) & (grayImg <= highThr), 255, 0)

slicedWithBg = slicedWithBg.astype(np.uint8)

# Slicing without background

slicedNoBg = np.where((grayImg >= lowThr) & (grayImg <= highThr), grayImg, 0)

slicedNoBg = slicedNoBg.astype(np.uint8)

# Display results

plt.figure()

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

plt.title('Gray Level Slicing with Background - UI22EC17')

plt.show()

plt.figure()

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

plt.title('Gray Level Slicing without Background - UI22EC17')

plt.show()

# 2. Bit Plane Slicing

# Convert grayscale image to uint8

grayImgUint8 = grayImg.astype(np.uint8)

# Number of bit planes

nPlanes = 8

# Display bit planes

for i in range(1, nPlanes + 1):

# Extract bit plane

bitPlane = np.bitwise\_and(grayImgUint8, 1 << (i - 1)) >> (i - 1)

bitPlaneImg = bitPlane \* 255 # Convert for display

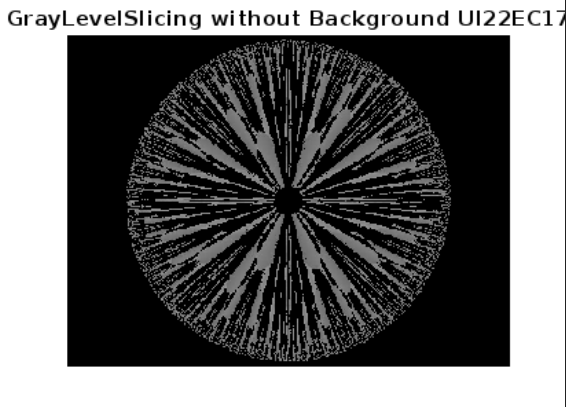
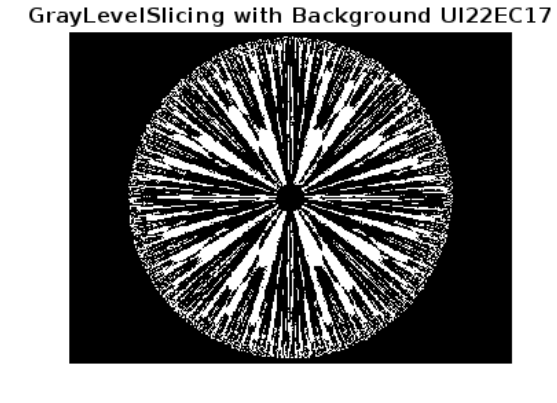
# Display each bit plane

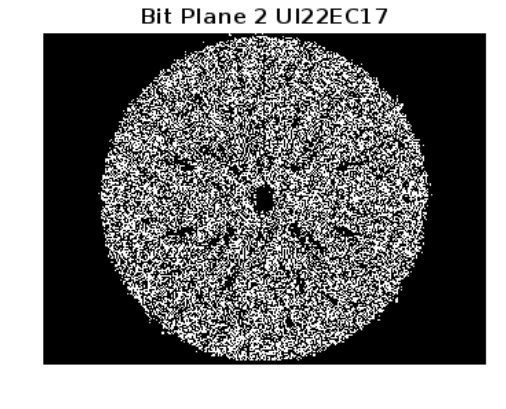
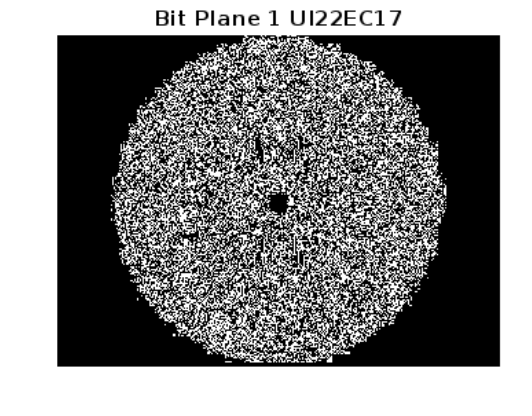
plt.figure()

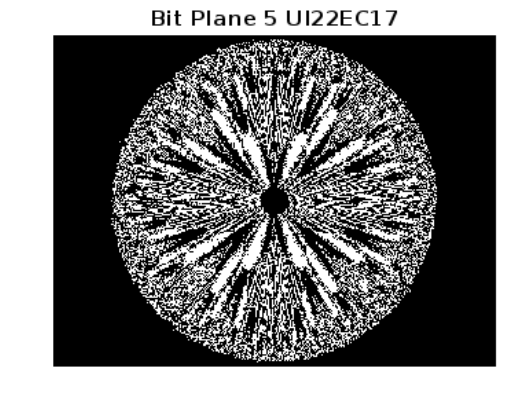
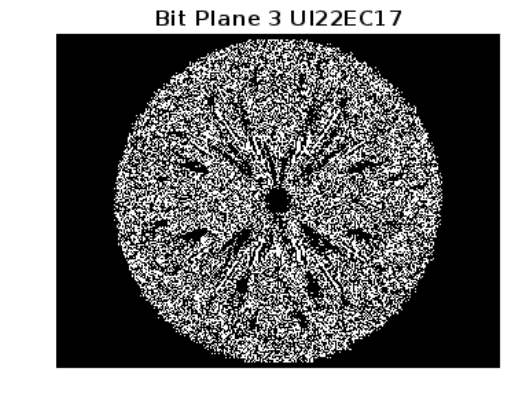
plt.imshow(bitPlaneImg, cmap='gray')

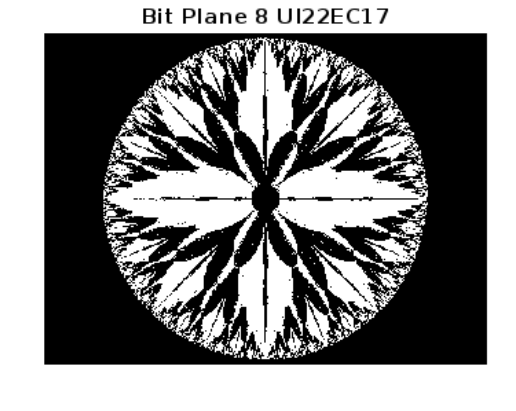
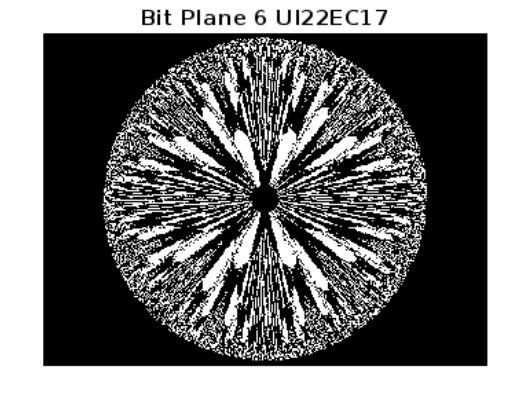
plt.title(f'Bit Plane {i} - UI22EC17')

plt.show()

****

****

****

****

**LAB 5**

%% LAB-5

% dev yadav

% UI22EC17

%% Objective:

% (A) Add Gaussian noise, salt noise, pepper noise, and salt & pepper noise

% to the original image (ckt\_board.tif).

% (B) Restore the original image using various spatial filtering operations:

% Arithmetic mean, Geometric mean, Harmonic, Contra-Harmonic, Median, Min, Max, and Mid-point filters.

clc;

clear;

close all;

% Load the image

img = imread('ckt\_board.tif');

% Display original image

figure;

imshow(img);

title('Original Image (UI22EC17)');

%% (A) Generate degraded images by adding various types of noise

% Add Gaussian noise

gaussianNoiseImg = imnoise(img, 'gaussian', 0, 0.01);

figure;

imshow(gaussianNoiseImg);

title('Image with Gaussian Noise (UI22EC17)');

% Add Salt noise

saltNoiseImg = imnoise(img, 'salt & pepper', 0.05);

figure;

imshow(saltNoiseImg);

title('Image with Salt & Pepper Noise (UI22EC17)');

% Add Pepper noise (using salt & pepper with low salt probability)

pepperNoiseImg = imnoise(img, 'salt & pepper', 0.05);

figure;

imshow(pepperNoiseImg);

title('Image with Pepper Noise (UI22EC17)');

% Add Salt & Pepper noise

saltPepperNoiseImg = imnoise(img, 'salt & pepper', 0.1);

figure;

imshow(saltPepperNoiseImg);

title('Image with Salt & Pepper Noise (UI22EC17)');

%% (B) Apply various spatial filter operations to restore the images

% Arithmetic Mean Filter

arithMeanFilter = fspecial('average', [3, 3]);

amRestored = imfilter(gaussianNoiseImg, arithMeanFilter);

figure;

imshow(amRestored);

title('Restored with Arithmetic Mean Filter (UI22EC17)');

% Geometric Mean Filter

gmRestored = exp(imfilter(log(double(gaussianNoiseImg) + 1), ones(3, 3), 'replicate')) .^ (1/9);

gmRestored = uint8(gmRestored - 1);

figure;

imshow(gmRestored);

title('Restored with Geometric Mean Filter (UI22EC17)');

% Harmonic Mean Filter

hmRestored = 9 ./ (imfilter(1 ./ (double(gaussianNoiseImg) + eps), ones(3, 3), 'replicate'));

hmRestored = uint8(hmRestored);

figure;

imshow(hmRestored);

title('Restored with Harmonic Mean Filter (UI22EC17)');

% Contra-Harmonic Mean Filter

Q = 1.5; % You can change the Q value to control the restoration

chmRestored = imfilter(double(saltPepperNoiseImg).^(Q+1), ones(3,3)) ./ imfilter(double(saltPepperNoiseImg).^Q, ones(3,3));

chmRestored = uint8(chmRestored);

figure;

imshow(chmRestored);

title(['Restored with Contra-Harmonic Mean Filter, Q = ', num2str(Q), ' (UI22EC17)']);

% Median Filter

medianRestored = medfilt2(saltPepperNoiseImg, [3, 3]);

figure;

imshow(medianRestored);

title('Restored with Median Filter (UI22EC17)');

% Min Filter

minRestored = ordfilt2(pepperNoiseImg, 1, ones(3, 3));

figure;

imshow(minRestored);

title('Restored with Min Filter (UI22EC17)');

% Max Filter

maxRestored = ordfilt2(saltNoiseImg, 9, ones(3, 3));

figure;

imshow(maxRestored);

title('Restored with Max Filter (UI22EC17)');

% Mid-point Filter

mpRestored = 0.5 \* (minRestored + maxRestored);

figure;

imshow(mpRestored);

title('Restored with Mid-point Filter (UI22EC17)');

**in python**

import cv2

import numpy as np

from skimage.util import random\_noise

from scipy.ndimage import generic\_filter

import matplotlib.pyplot as plt

# Load the image

img = cv2.imread('ckt\_board.tif', cv2.IMREAD\_GRAYSCALE)

# Display function

def show\_image(title, image):

plt.figure()

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

plt.title(title)

plt.axis('off')

show\_image('Original Image (UI22EC17)', img)

# (A) Generate degraded images by adding various types of noise

# Add Gaussian noise

gaussian\_noise\_img = random\_noise(img, mode='gaussian', mean=0, var=0.01)

gaussian\_noise\_img = (255 \* gaussian\_noise\_img).astype(np.uint8)

show\_image('Image with Gaussian Noise (UI22EC17)', gaussian\_noise\_img)

# Add Salt noise (using salt & pepper noise but only salt component)

salt\_noise\_img = random\_noise(img, mode='s&p', salt\_vs\_pepper=1, amount=0.05)

salt\_noise\_img = (255 \* salt\_noise\_img).astype(np.uint8)

show\_image('Image with Salt Noise (UI22EC17)', salt\_noise\_img)

# Add Pepper noise (using salt & pepper noise but only pepper component)

pepper\_noise\_img = random\_noise(img, mode='s&p', salt\_vs\_pepper=0, amount=0.05)

pepper\_noise\_img = (255 \* pepper\_noise\_img).astype(np.uint8)

show\_image('Image with Pepper Noise (UI22EC17)', pepper\_noise\_img)

# Add Salt & Pepper noise

salt\_pepper\_noise\_img = random\_noise(img, mode='s&p', amount=0.1)

salt\_pepper\_noise\_img = (255 \* salt\_pepper\_noise\_img).astype(np.uint8)

show\_image('Image with Salt & Pepper Noise (UI22EC17)', salt\_pepper\_noise\_img)

# (B) Apply various spatial filter operations to restore the images

# Arithmetic Mean Filter

arith\_mean\_filter = cv2.blur(gaussian\_noise\_img, (3, 3))

show\_image('Restored with Arithmetic Mean Filter (UI22EC17)', arith\_mean\_filter)

# Geometric Mean Filter

def geometric\_mean\_filter(image):

return np.exp(cv2.blur(np.log(image + 1), (3, 3))) - 1

gm\_restored = geometric\_mean\_filter(gaussian\_noise\_img).astype(np.uint8)

show\_image('Restored with Geometric Mean Filter (UI22EC17)', gm\_restored)

# Harmonic Mean Filter

def harmonic\_mean\_filter(image):

return 9 / cv2.blur(1 / (image + 1e-5), (3, 3))

hm\_restored = harmonic\_mean\_filter(gaussian\_noise\_img).astype(np.uint8)

show\_image('Restored with Harmonic Mean Filter (UI22EC17)', hm\_restored)

# Contra-Harmonic Mean Filter

Q = 1.5 # You can change Q to control the restoration

def contra\_harmonic\_mean\_filter(image, Q):

num = cv2.blur(np.power(image, Q + 1), (3, 3))

denom = cv2.blur(np.power(image, Q), (3, 3))

return num / (denom + 1e-5)

chm\_restored = contra\_harmonic\_mean\_filter(salt\_pepper\_noise\_img.astype(float), Q).astype(np.uint8)

show\_image(f'Restored with Contra-Harmonic Mean Filter, Q = {Q} (UI22EC17)', chm\_restored)

# Median Filter

median\_restored = cv2.medianBlur(salt\_pepper\_noise\_img, 3)

show\_image('Restored with Median Filter (UI22EC17)', median\_restored)

# Min Filter

min\_restored = generic\_filter(pepper\_noise\_img, np.min, size=(3, 3))

show\_image('Restored with Min Filter (UI22EC17)', min\_restored)

# Max Filter

max\_restored = generic\_filter(salt\_noise\_img, np.max, size=(3, 3))

show\_image('Restored with Max Filter (UI22EC17)', max\_restored)

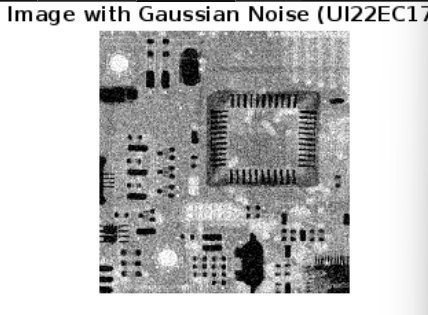
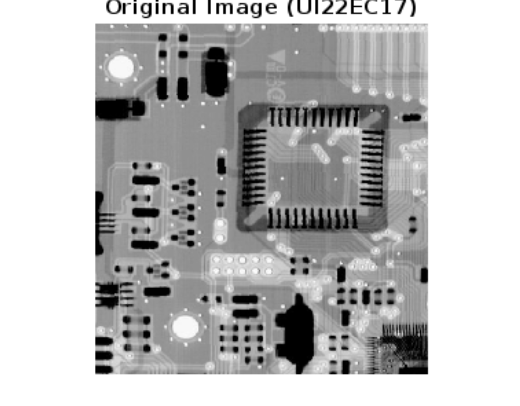
# Mid-point Filter

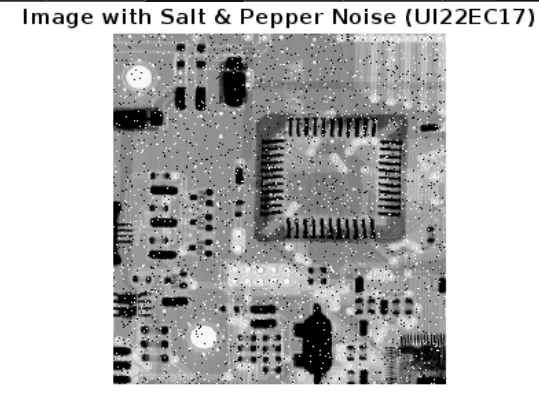
mp\_restored = ((min\_restored + max\_restored) / 2).astype(np.uint8)

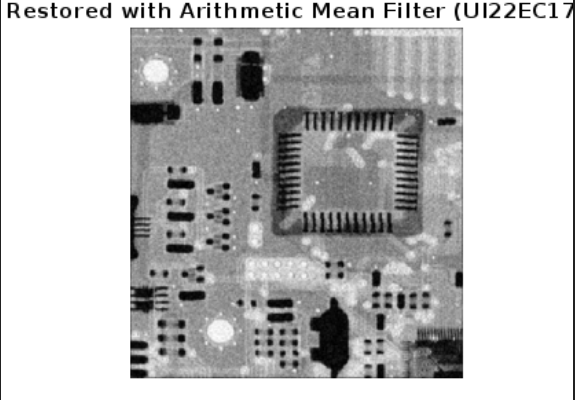
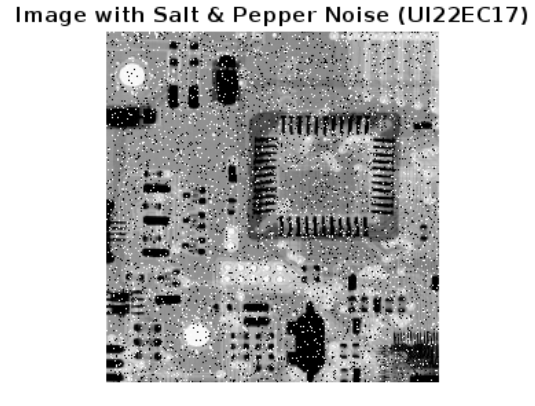
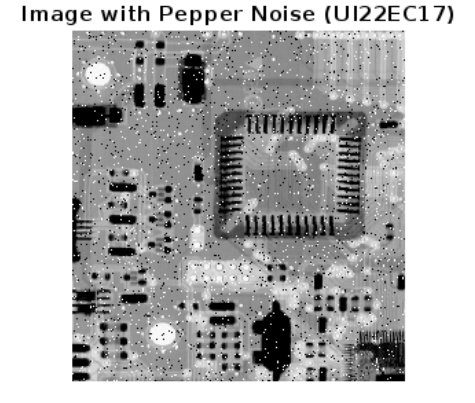
show\_image('Restored with Mid-point Filter (UI22EC17)', mp\_restored)

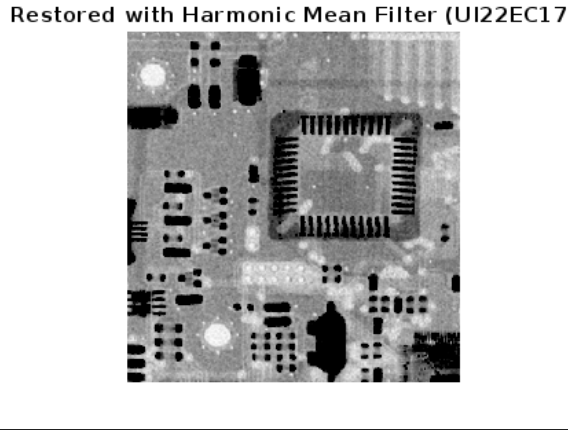
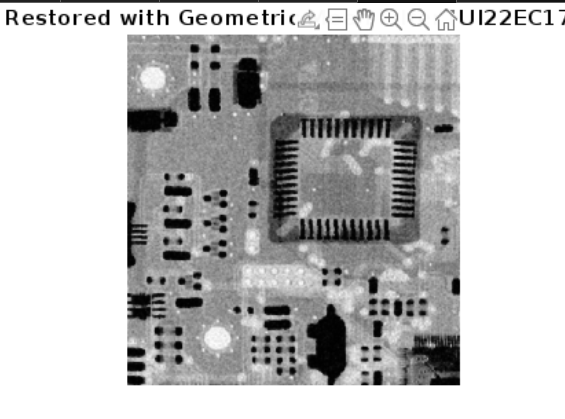
# Show all images

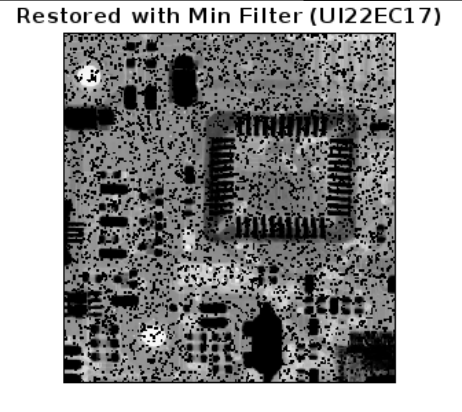
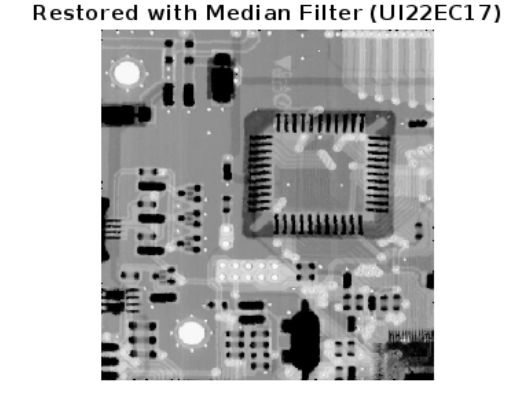
plt.show()

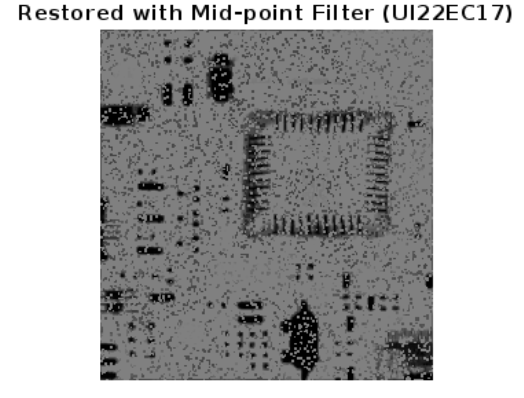
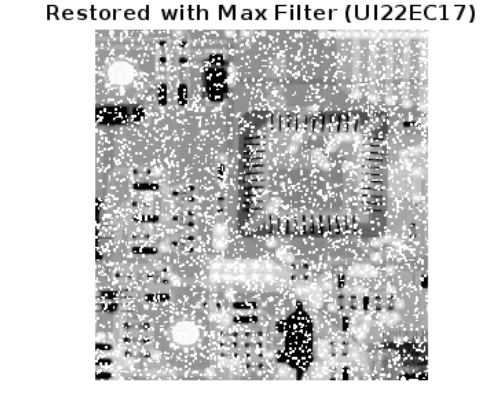
****

****

****

****

****

****

**Lab 6**

%% LAB6 UI22EC17 7/10/24

close all;

%% AIM

% Perform various Spatial domain filtering techniques on given image.

% Apply filter operation with a different kernel sizes of 3x3, 7x7,

% 11x11,15x15 and 21x21

% 1-Low pass filtering

% 2-High pass filtering

% 3-Weighted average filter

% Read the input image

img\_gray = imread('filter.tif');

%img\_gray = rgb2gray(img\_gray); % Convert to grayscale if it's a color image

figure(1),imshow(img\_gray);

title('Original Image');

% Add Gaussian noise

gaussian\_noisy\_img = imnoise(img\_gray, 'gaussian', 0, 0.01); % 0 is mean, 0.01 is variance

figure(2);

imshow(gaussian\_noisy\_img);

title('Image with Gaussian Noise');

% Define kernel sizes

kernel\_sizes = [3, 7, 11, 15, 21];

% Preallocate arrays to store filtered images

low\_pass\_images = cell(length(kernel\_sizes), 1);

high\_pass\_images = cell(length(kernel\_sizes), 1);

weighted\_avg\_images = cell(length(kernel\_sizes), 1);

for i = 1:length(kernel\_sizes)

k = kernel\_sizes(i);

% Create low-pass filter (Gaussian)

h\_lp = fspecial('gaussian', k, 1); % 1 is the standard deviation

% Apply low-pass filter

low\_pass\_images{i} = imfilter(img\_gray, h\_lp, 'replicate');

% Create high-pass filter

h\_hp = -fspecial('gaussian', k, 1); % Negative Gaussian

h\_hp((k+1)/2, (k+1)/2) = h\_hp((k+1)/2, (k+1)/2) + 1; % Center coefficient adjustment

% Apply high-pass filter

high\_pass\_img = imfilter(img\_gray, h\_hp, 'replicate');

% Post-processing for high-pass filter output:

% 1. Take the absolute value to remove negative values

high\_pass\_img = abs(high\_pass\_img);

% 2. Normalize the image to the range [0, 1]

high\_pass\_img = mat2gray(high\_pass\_img);

% Store the result

high\_pass\_images{i} = high\_pass\_img;

% Create weighted average filter

weights = ones(k) / (k \* k); % Average weights

weighted\_avg\_images{i} = imfilter(img\_gray, weights, 'replicate');

end

% Display results for Low Pass Filter

figure(3);

for i = 1:length(kernel\_sizes)

% Display low-pass filtered images

subplot(3, 2, i); % 3 rows, 2 columns

imshow(low\_pass\_images{i});

title(['Low Pass - ', num2str(kernel\_sizes(i))]);

end

% Display results for High Pass Filter

figure(4);

for i = 1:length(kernel\_sizes)

% Display high-pass filtered images

subplot(3, 2, i); % 3 rows, 2 columns

imshow(high\_pass\_images{i});

title(['High Pass - ', num2str(kernel\_sizes(i))]);

end

% Display results for Weighted Average Filter

figure(5);

for i = 1:length(kernel\_sizes)

if i < 5

% Display first four images in the first two rows (2 images each)

subplot(3, 2, i); % 3 rows, 2 columns

else

% Display the last image in the last row, first column

subplot(3, 2, 5);

end

imshow(weighted\_avg\_images{i});

title(['Weighted Avg - ', num2str(kernel\_sizes(i))]);

end

**in python**import cv2

import numpy as np

import matplotlib.pyplot as plt

from scipy.ndimage import gaussian\_filter

# Load the image in grayscale

img\_gray = cv2.imread('filter.tif', cv2.IMREAD\_GRAYSCALE)

# Display original image

plt.figure(figsize=(10, 5))

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

plt.title('Original Image')

plt.axis('off')

plt.show()

# Add Gaussian noise

gaussian\_noisy\_img = img\_gray + np.random.normal(0, 25, img\_gray.shape).astype(np.uint8)

plt.figure()

plt.imshow(gaussian\_noisy\_img, cmap='gray')

plt.title('Image with Gaussian Noise')

plt.axis('off')

plt.show()

# Define kernel sizes

kernel\_sizes = [3, 7, 11, 15, 21]

# Preallocate lists to store filtered images

low\_pass\_images = []

high\_pass\_images = []

weighted\_avg\_images = []

# Apply filters with different kernel sizes

for k in kernel\_sizes:

# Low-pass filter (Gaussian)

low\_pass\_img = cv2.GaussianBlur(img\_gray, (k, k), sigmaX=1)

low\_pass\_images.append(low\_pass\_img)

# High-pass filter (by subtracting low-pass filter from original)

high\_pass\_img = img\_gray - low\_pass\_img

high\_pass\_img = cv2.normalize(np.abs(high\_pass\_img), None, 0, 255, cv2.NORM\_MINMAX)

high\_pass\_images.append(high\_pass\_img)

# Weighted average filter

weights = np.ones((k, k), np.float32) / (k \* k)

weighted\_avg\_img = cv2.filter2D(img\_gray, -1, weights)

weighted\_avg\_images.append(weighted\_avg\_img)

# Display results for Low Pass Filter

plt.figure(figsize=(10, 8))

for i, lp\_img in enumerate(low\_pass\_images):

plt.subplot(3, 2, i + 1)

plt.imshow(lp\_img, cmap='gray')

plt.title(f'Low Pass - {kernel\_sizes[i]}x{kernel\_sizes[i]}')

plt.axis('off')

plt.tight\_layout()

plt.show()

# Display results for High Pass Filter

plt.figure(figsize=(10, 8))

for i, hp\_img in enumerate(high\_pass\_images):

plt.subplot(3, 2, i + 1)

plt.imshow(hp\_img, cmap='gray')

plt.title(f'High Pass - {kernel\_sizes[i]}x{kernel\_sizes[i]}')

plt.axis('off')

plt.tight\_layout()

plt.show()

# Display results for Weighted Average Filter

plt.figure(figsize=(10, 8))

for i, wa\_img in enumerate(weighted\_avg\_images):

plt.subplot(3, 2, i + 1)

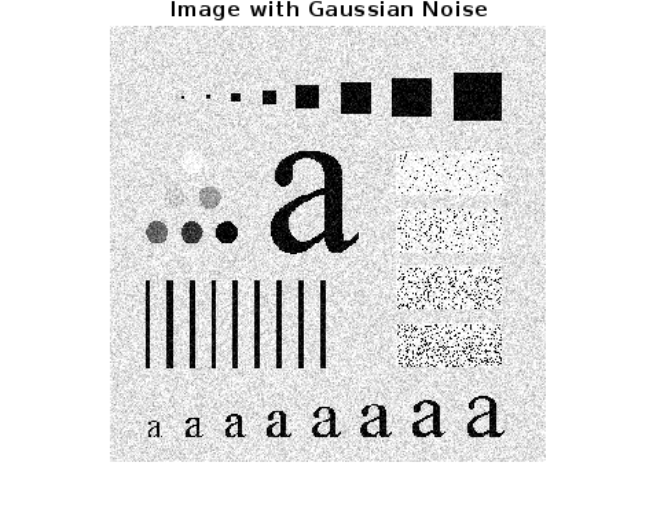
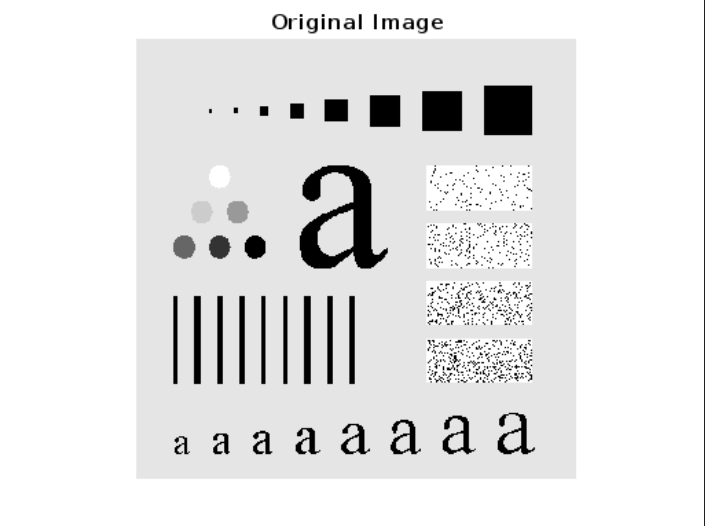
plt.imshow(wa\_img, cmap='gray')

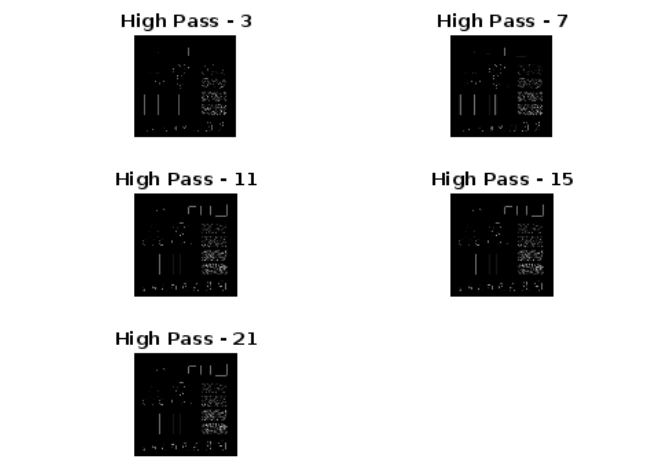
plt.title(f'Weighted Avg - {kernel\_sizes[i]}x{kernel\_sizes[i]}')

plt.axis('off')

plt.tight\_layout()

plt.show()

****

****

****

**Lab 7**

close all ;

% Read the color image

img = imread('lotus.jpg');

% Convert the image to grayscale

grayImg = rgb2gray(img);

% Display the grayscale image

figure, imshow(grayImg), title('Grayscale Image');

% Apply Prewitt edge detection

prewittEdges = edge(grayImg, 'Prewitt');

% Display the result

figure, imshow(prewittEdges), title('Prewitt Edge Detection');

% Apply Sobel edge detection

sobelEdges = edge(grayImg, 'Sobel');

% Display the result

figure, imshow(sobelEdges), title('Sobel Edge Detection');

% Apply an averaging filter (smoothing filter)

h = fspecial('average', [5 5]); % 5x5 averaging filter

smoothedImg = imfilter(grayImg, h);

% Apply Sobel edge detection on the smoothed image

smoothedSobelEdges = edge(smoothedImg, 'Sobel');

% Display the result

figure, imshow(smoothedSobelEdges), title('Sobel Edge Detection with Smoothing Filter');

**in python**# LAB - UI22EC17

import cv2

import numpy as np

from skimage import filters

import matplotlib.pyplot as plt

# Read the color image

img = cv2.imread('lotus.jpg')

img\_rgb = cv2.cvtColor(img, cv2.COLOR\_BGR2RGB) # Convert from BGR to RGB for display in matplotlib

# Convert the image to grayscale

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

# Display the grayscale image

plt.figure()

plt.imshow(gray\_img, cmap='gray')

plt.title('Grayscale Image')

plt.axis('off')

plt.show()

# Apply Prewitt edge detection

prewitt\_edges = filters.prewitt(gray\_img)

# Display the result for Prewitt Edge Detection

plt.figure()

plt.imshow(prewitt\_edges, cmap='gray')

plt.title('Prewitt Edge Detection')

plt.axis('off')

plt.show()

# Apply Sobel edge detection

sobel\_edges = cv2.Sobel(gray\_img, cv2.CV\_64F, 1, 0, ksize=3) + cv2.Sobel(gray\_img, cv2.CV\_64F, 0, 1, ksize=3)

sobel\_edges = np.hypot(sobel\_edges, sobel\_edges).astype(np.uint8)

# Display the result for Sobel Edge Detection

plt.figure()

plt.imshow(sobel\_edges, cmap='gray')

plt.title('Sobel Edge Detection')

plt.axis('off')

plt.show()

# Apply an averaging filter (smoothing filter)

smoothed\_img = cv2.blur(gray\_img, (5, 5))

# Apply Sobel edge detection on the smoothed image

smoothed\_sobel\_edges = cv2.Sobel(smoothed\_img, cv2.CV\_64F, 1, 0, ksize=3) + cv2.Sobel(smoothed\_img, cv2.CV\_64F, 0, 1, ksize=3)

smoothed\_sobel\_edges = np.hypot(smoothed\_sobel\_edges, smoothed\_sobel\_edges).astype(np.uint8)

# Display the result for Sobel Edge Detection with Smoothing Filter

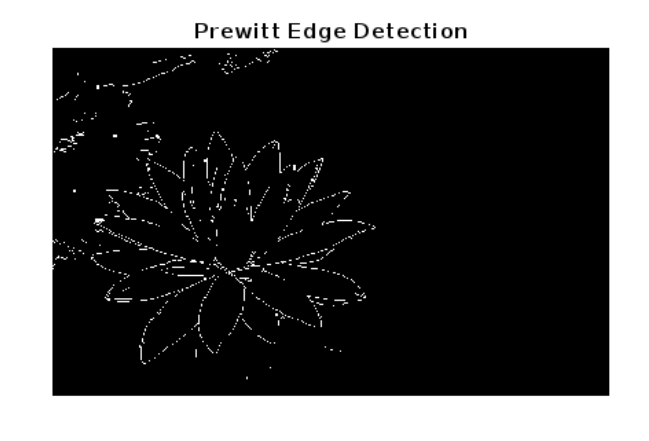
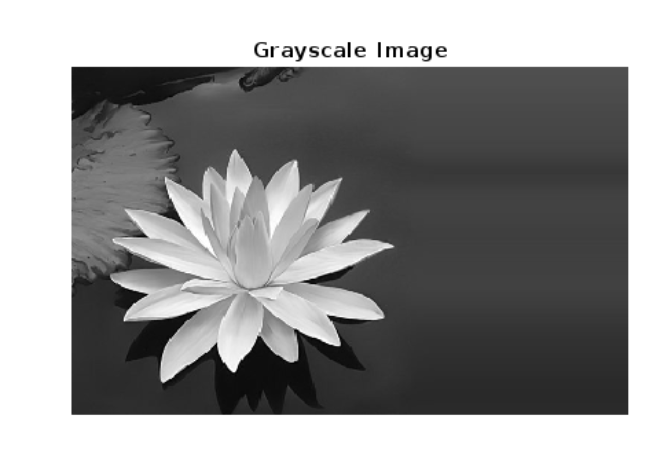
plt.figure()

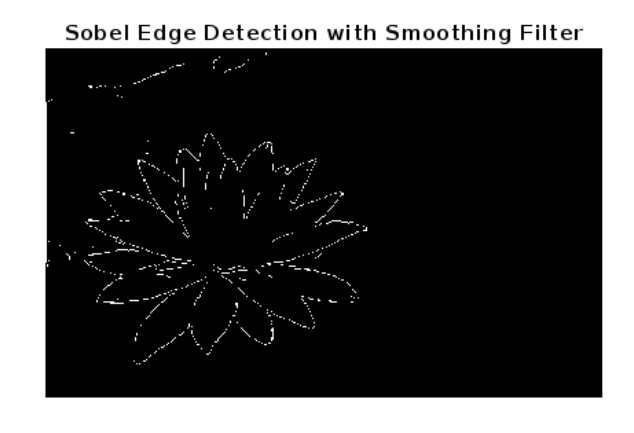
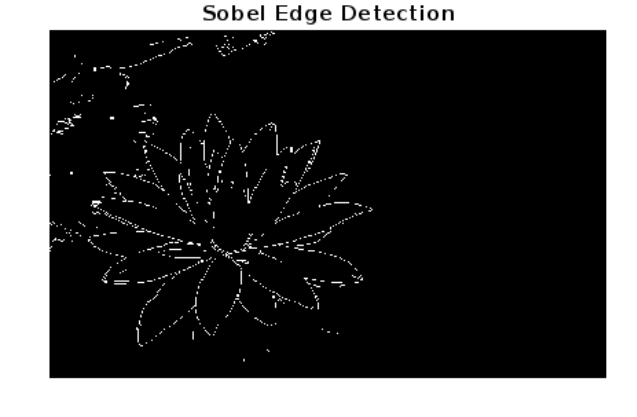
plt.imshow(smoothed\_sobel\_edges, cmap='gray')

plt.title('Sobel Edge Detection with Smoothing Filter')

plt.axis('off')

plt.show()

****

****

**Lab 8**

%For the color image, convert to RGB to Gray and finding the edges

%using following operators:

%1) Use of thresholding with Sobel operator.

%2) Canny operators.

%3) Laplace of Gaussian (LOG) Marr-Hilderth transform.

%Compare the results with Threshold gradient, LOG and canny operators.

close all;

%ui22ec17

% Read the color image

image = imread('messi.jpg');

figure; imshow(image); title('Original Color Image');

% Convert to grayscale

gray = rgb2gray(image);

figure; imshow(gray); title('Grayscale Image');

% 1) Sobel operator with thresholding

sobelX = fspecial('sobel'); % Sobel operator in X direction

sobelY = sobelX'; % Sobel operator in Y direction

% Apply Sobel operator

gradientX = imfilter(double(gray), sobelX);

gradientY = imfilter(double(gray), sobelY);

gradientMagnitude = sqrt(gradientX.^2 + gradientY.^2);

% Thresholding

threshold = 0.1 \* max(gradientMagnitude(:));

edgesSobel = gradientMagnitude > threshold;

figure; imshow(edgesSobel); title('Edges using Sobel Operator');

% 2) Canny operator

edgesCanny = edge(gray, 'Canny');

figure; imshow(edgesCanny); title('Edges using Canny Operator');

% 3) Laplacian of Gaussian (LoG) using Marr-Hildreth transform

sigma = 2; % Standard deviation for Gaussian

hLoG = fspecial('log', 5, sigma); % Create LoG filter

edgesLoG = imfilter(double(gray), hLoG); % Apply LoG filter

% Thresholding for LoG results

thresholdLoG = 0.1;

edgesLoGBinary = edgesLoG > thresholdLoG;

figure; imshow(edgesLoGBinary); title('Edges using LoG (Marr-Hildreth)');

% Compare the results

figure;

subplot(2, 2, 1); imshow(gray); title('gray image ui22ec17');

subplot(2, 2, 2); imshow(edgesSobel); title('Sobel Edges');

subplot(2, 2, 3); imshow(edgesCanny); title('Canny Edges');

subplot(2, 2, 4); imshow(edgesLoGBinary); title('LoG Edges');

**in python**# Edge Detection Using Sobel, Canny, and Laplacian of Gaussian (LoG)

# UI22EC17

import cv2

import numpy as np

from scipy.ndimage import gaussian\_laplace

import matplotlib.pyplot as plt

# Load the color image

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

image\_rgb = cv2.cvtColor(image, cv2.COLOR\_BGR2RGB) # Convert BGR to RGB for display

plt.figure()

plt.imshow(image\_rgb)

plt.title('Original Color Image')

plt.axis('off')

plt.show()

# Convert to grayscale

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

plt.figure()

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

plt.title('Grayscale Image')

plt.axis('off')

plt.show()

# 1) Sobel operator with thresholding

sobel\_x = cv2.Sobel(gray, cv2.CV\_64F, 1, 0, ksize=3) # Sobel in X direction

sobel\_y = cv2.Sobel(gray, cv2.CV\_64F, 0, 1, ksize=3) # Sobel in Y direction

gradient\_magnitude = np.sqrt(sobel\_x\*\*2 + sobel\_y\*\*2)

# Thresholding

threshold = 0.1 \* gradient\_magnitude.max()

edges\_sobel = gradient\_magnitude > threshold

# Display Sobel edges

plt.figure()

plt.imshow(edges\_sobel, cmap='gray')

plt.title('Edges using Sobel Operator')

plt.axis('off')

plt.show()

# 2) Canny operator

edges\_canny = cv2.Canny(gray, 100, 200)

plt.figure()

plt.imshow(edges\_canny, cmap='gray')

plt.title('Edges using Canny Operator')

plt.axis('off')

plt.show()

# 3) Laplacian of Gaussian (LoG) using Marr-Hildreth transform

sigma = 2 # Standard deviation for Gaussian

edges\_log = gaussian\_laplace(gray, sigma=sigma)

threshold\_log = 0.1

edges\_log\_binary = edges\_log > threshold\_log

# Display LoG (Marr-Hildreth) edges

plt.figure()

plt.imshow(edges\_log\_binary, cmap='gray')

plt.title('Edges using LoG (Marr-Hildreth)')

plt.axis('off')

plt.show()

# Compare the results

plt.figure(figsize=(10, 8))

plt.subplot(2, 2, 1)

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

plt.title('Grayscale Image (UI22EC17)')

plt.axis('off')

plt.subplot(2, 2, 2)

plt.imshow(edges\_sobel, cmap='gray')

plt.title('Sobel Edges')

plt.axis('off')

plt.subplot(2, 2, 3)

plt.imshow(edges\_canny, cmap='gray')

plt.title('Canny Edges')

plt.axis('off')

plt.subplot(2, 2, 4)

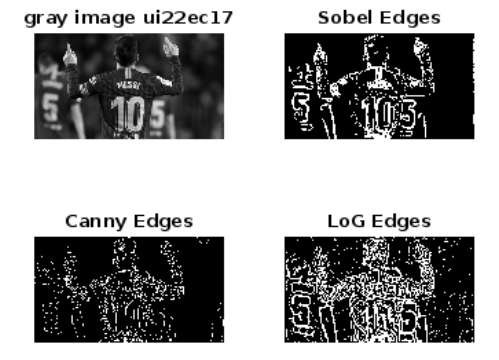
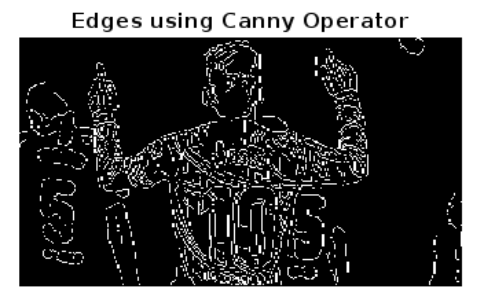
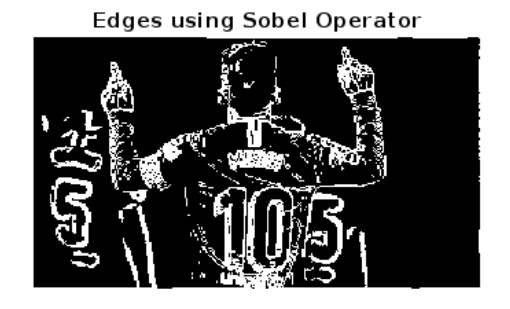
plt.imshow(edges\_log\_binary, cmap='gray')

plt.title('LoG Edges')

plt.axis('off')

plt.tight\_layout()

plt.show()

****

**Lab 9**

%{

AIM---

Task1:Laplacian operator filter:Apply Laplacian mask with +4 as center coefficient [[0,-1,0],[-1,4,1],[0,-1,0]] •Apply Laplacian mask with -4 as center coefficient [[0,1,0],[1,4,1],[0,1,0]]

73-77

•Apply Laplacian mask with -8 as center coefficient [[1,1,1],[1,-8,1],[1,1,1]] •Apply Laplacian mask with +8 as center coefficient [[-1,-1,-1],[-1,8,-1],[-1,-1,- 1]]

Task2:Unsharping mask: •Blur original input image using LPF filter having kernel size 3 X 3. •Generate mask image via subtracting it from original image •Add them asked image with the original one to get the final output image •Repeat the same for different LPF kernel size and observe the effect

Task3:Highboost filtering: •Generalization of Unsharp masking operation •Blur original input image using LPF filter having 3x3 kernel size •Generate mask image via subtracting it from original image •Multiply them ask image with constant integervalue(>1) and then add it with the original one to get the final output image.

%}

clc;

clear all;

close all;

% Edge detection using Laplacian Filter.

img=imread('lotus.jpg');

% Convert rgb image to grayscale.

original\_img=rgb2gray(img);

% Convert into double format.

grey\_img=double(original\_img);

%% Task - 1

% Laplacian with different center coeffecient

Laplacian\_pos\_4=[0 -1 0; -1 4 -1; 0 -1 0];

Laplacian\_neg\_4=[0 1 0; 1 -4 1; 0 1 0];

Laplacian\_pos\_8=[-1 -1 -1; -1 8 -1; -1 -1 -1];

Laplacian\_neg\_8=[1 1 1; 1 -8 1; 1 1 1];

% Convolve the images using Laplacian Filter

final\_img\_pos\_4=conv2(grey\_img, Laplacian\_pos\_4, 'same');

final\_img\_pos\_4 = abs(final\_img\_pos\_4);

final\_img\_neg\_4=conv2(grey\_img, Laplacian\_neg\_4, 'same');

final\_img\_neg\_4 = abs(final\_img\_neg\_4);

final\_img\_pos\_8=conv2(grey\_img, Laplacian\_pos\_8, 'same');

final\_img\_pos\_8 = abs(final\_img\_pos\_8);

final\_img\_neg\_8=conv2(grey\_img, Laplacian\_neg\_8, 'same');

final\_img\_neg\_8 = abs(final\_img\_neg\_8);

% Display the images.

figure;

imshow(grey\_img , [] );

title("ORIGINAL IMAGE")

figure;

subplot(2,2,1);

imshow(final\_img\_pos\_4 , [] );

title("Laplacian mask with +4");

subplot(2,2,2);

imshow(final\_img\_neg\_4 , [] );

title("Laplacian mask with -4 ");

subplot(2,2,3);

imshow(final\_img\_pos\_8 , [] );

title("Laplacian mask with +8 ");

subplot(2,2,4);

imshow(final\_img\_neg\_8 , [] );

title("Laplacian mask with -8");

%% Task - 2

kernel\_sizes = [3, 15];

original\_img = rgb2gray(img); % Ensure the original image is grayscale

for k = kernel\_sizes

figure;

% Create an averaging filter (low-pass filter)

low\_pass\_kernel = fspecial('average', [k k]);

% Apply low-pass filter to the image

low\_pass\_filtered = imfilter(original\_img, low\_pass\_kernel, 'symmetric');

% Generate the mask and the sharpened image

mask = original\_img - low\_pass\_filtered;

final\_img = original\_img + mask;

% Display the results

subplot(2,2,1);

imshow(original\_img);

title("Original Image");

subplot(2,2,2);

imshow(low\_pass\_filtered);

title("Low-pass Filtered Image");

subplot(2,2,3);

imshow(mask);

title("Mask Image");

subplot(2,2,4);

imshow(final\_img);

title([num2str(k) 'x' num2str(k)]);

end

%% TASK - 3

% kernel\_sizes = [3 , 5 ];

original\_img=rgb2gray(img);

k = 3;

constant = 15;

%for k = kernel\_sizes

figure;

% Create an averaging filter (low-pass filter)

low\_pass\_kernel = fspecial('average', [k k]);

% Apply low-pass filter to the image

low\_pass\_filtered = imfilter(original\_img, low\_pass\_kernel, 'symmetric');

our\_mask = original\_img - low\_pass\_filtered;

multiplied\_mask = constant \* our\_mask;

final\_img = multiplied\_mask + original\_img ;

% Display result

subplot(2,2,1)

imshow(original\_img)

title("ORIGINAL IMAGE")

subplot(2,2,2)

imshow(low\_pass\_filtered)

title("LOW PASS FILTERED IMAGE")

subplot(2,2,3)

imshow(multiplied\_mask)

title("MULTIPLIED MASKED IMAGE")

subplot(2,2,4)

imshow(final\_img)

title("RESULTANT IMAGE")

%end

%{

CONCLUSION-

Task 1 applies various Laplacian masks with different central values to highlight edges differently.

Task 2 performs unsharp masking by subtracting a blurred version of the image from the original to create a mask, then adding this mask back to sharpen the image. Different kernel sizes affect the sharpness level.

Task 3 performs high-boost filtering by scaling the mask by a factor greater than 1 before adding it back, enhancing the edges and details more prominently than in unsharp masking.

%}

**In python**

# Edge Detection and Filtering in Python

# Tasks: Laplacian filtering, Unsharp Masking, and High-boost Filtering

import cv2

import numpy as np

import matplotlib.pyplot as plt

from scipy.ndimage import convolve

# Read the image

img = cv2.imread('lotus.jpg')

original\_img = cv2.cvtColor(img, cv2.COLOR\_BGR2GRAY) # Convert to grayscale

# Task 1: Laplacian filtering with different center coefficients

laplacian\_masks = {

"+4": np.array([[0, -1, 0], [-1, 4, -1], [0, -1, 0]]),

"-4": np.array([[0, 1, 0], [1, -4, 1], [0, 1, 0]]),

"+8": np.array([[-1, -1, -1], [-1, 8, -1], [-1, -1, -1]]),

"-8": np.array([[1, 1, 1], [1, -8, 1], [1, 1, 1]])

}

# Apply each Laplacian mask

laplacian\_results = {}

for name, mask in laplacian\_masks.items():

filtered\_img = convolve(original\_img.astype(np.float64), mask, mode='reflect')

laplacian\_results[name] = np.abs(filtered\_img)

# Display Laplacian results

plt.figure(figsize=(10, 8))

plt.subplot(2, 2, 1)

plt.imshow(laplacian\_results["+4"], cmap='gray')

plt.title("Laplacian mask with +4")

plt.axis('off')

plt.subplot(2, 2, 2)

plt.imshow(laplacian\_results["-4"], cmap='gray')

plt.title("Laplacian mask with -4")

plt.axis('off')

plt.subplot(2, 2, 3)

plt.imshow(laplacian\_results["+8"], cmap='gray')

plt.title("Laplacian mask with +8")

plt.axis('off')

plt.subplot(2, 2, 4)

plt.imshow(laplacian\_results["-8"], cmap='gray')

plt.title("Laplacian mask with -8")

plt.axis('off')

plt.show()

# Task 2: Unsharp Masking with different low-pass filter kernel sizes

kernel\_sizes = [3, 15]

for k in kernel\_sizes:

# Create an averaging filter (low-pass filter)

low\_pass\_kernel = np.ones((k, k), np.float32) / (k \* k)

# Apply low-pass filter

low\_pass\_filtered = cv2.filter2D(original\_img, -1, low\_pass\_kernel)

# Generate mask and sharpened image

mask = original\_img - low\_pass\_filtered

final\_img = original\_img + mask

# Display the results

plt.figure(figsize=(10, 8))

plt.subplot(2, 2, 1)

plt.imshow(original\_img, cmap='gray')

plt.title("Original Image")

plt.axis('off')

plt.subplot(2, 2, 2)

plt.imshow(low\_pass\_filtered, cmap='gray')

plt.title("Low-pass Filtered Image")

plt.axis('off')

plt.subplot(2, 2, 3)

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

plt.title("Mask Image")

plt.axis('off')

plt.subplot(2, 2, 4)

plt.imshow(final\_img, cmap='gray')

plt.title(f'Unsharp Masking ({k}x{k})')

plt.axis('off')

plt.show()

# Task 3: High-boost filtering

k = 3

constant = 15

# Create an averaging filter (low-pass filter)

low\_pass\_kernel = np.ones((k, k), np.float32) / (k \* k)

# Apply low-pass filter to the image

low\_pass\_filtered = cv2.filter2D(original\_img, -1, low\_pass\_kernel)

our\_mask = original\_img - low\_pass\_filtered

multiplied\_mask = constant \* our\_mask

final\_img = multiplied\_mask + original\_img

# Display results for High-boost filtering

plt.figure(figsize=(10, 8))

plt.subplot(2, 2, 1)

plt.imshow(original\_img, cmap='gray')

plt.title("Original Image")

plt.axis('off')

plt.subplot(2, 2, 2)

plt.imshow(low\_pass\_filtered, cmap='gray')

plt.title("Low-pass Filtered Image")

plt.axis('off')

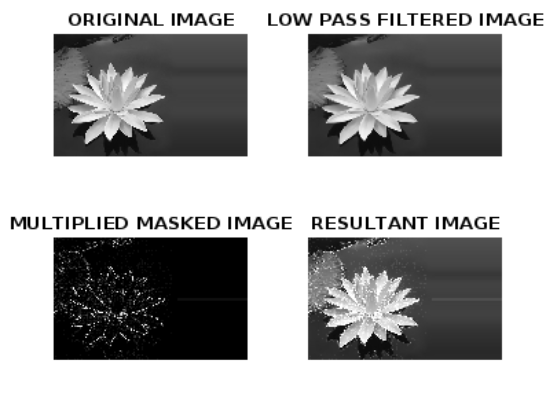
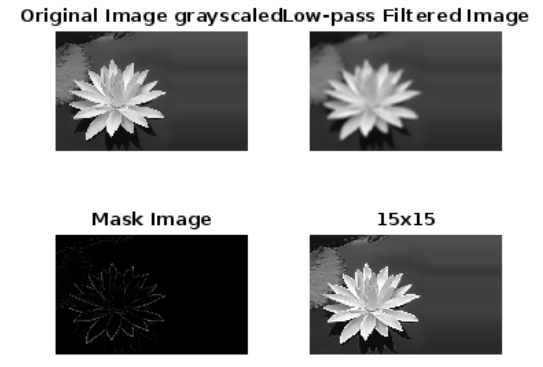
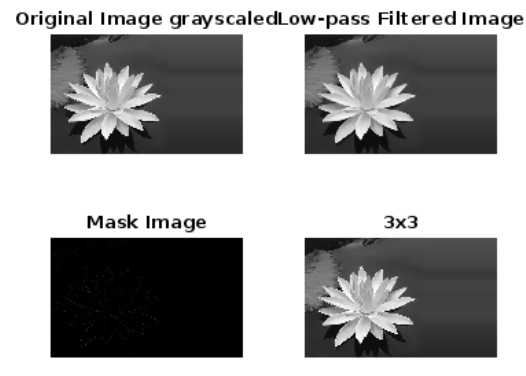
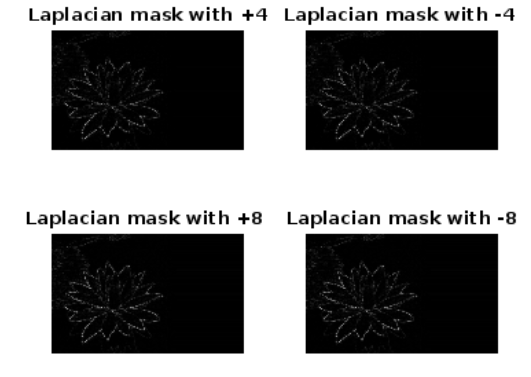
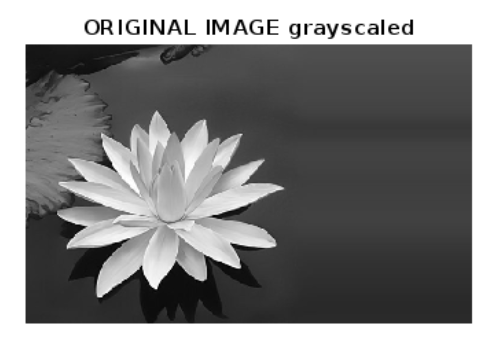
plt.subplot(2, 2, 3)

plt.imshow(multiplied\_mask, cmap='gray')

plt.title("Multiplied Mask Image")

plt.axis('off')

plt.subplot(2, 2, 4)

pl****

**Lab 10**

clc;

clear all;

close all;

% Function for Manual DFT

function F = manualDFT2D(image)

[M, N] = size(image);

F = zeros(M, N);

for u = 1:M

for v = 1:N

sum = 0;

for x = 1:M

for y = 1:N

sum = sum + double(image(x, y)) \* exp(-1i \* 2 \* pi \* ((u-1)\*(x-1)/M + (v-1)\*(y-1)/N));

end

end

F(u, v) = sum;

end

end

end

% Load and Preprocess Image

image = imread('lotus.jpg');

image\_gray = rgb2gray(image);

image\_gray = double(image\_gray); % Convert to double for DFT

% Resize the image to reduce computation time (e.g., 64x64)

image\_gray = imresize(image\_gray, [64, 64]);

[M, N] = size(image\_gray);

% Manual DFT

F\_manual = manualDFT2D(image\_gray);

% Built-in DFT

F1 = fftshift(fft2(image\_gray));

% Verify Difference

difference = norm(F\_manual - F1);

disp(['Difference between manual and built-in DFT: ', num2str(difference)]);

% Compute and Display Magnitude and Phase Spectra

magnitude\_spectrum = 20 \* log(abs(F1) + 1);

phase\_spectrum = angle(F1);

figure(1);

subplot(1, 2, 1);

imshow(magnitude\_spectrum, []);

title('Magnitude Spectrum');

subplot(1, 2, 2);

imshow(phase\_spectrum, []);

title('Phase Spectrum');

% Shift the Image and Compute DFT

shifted\_image = circshift(image\_gray, [10, 10]);

F\_shifted = fftshift(fft2(shifted\_image));

% Display Magnitude and Phase of Shifted Image

magnitude\_shifted = 20 \* log(abs(F\_shifted) + 1);

phase\_shifted = angle(F\_shifted);

figure(2);

subplot(1, 2, 1);

imshow(magnitude\_shifted, []);

title('Magnitude Spectrum (Shifted Image)');

subplot(1, 2, 2);

imshow(phase\_shifted, []);

title('Phase Spectrum (Shifted Image)');

% Rotate the Image and Compute DFT

rotated\_image = rot90(image\_gray);

F\_rotated = fftshift(fft2(rotated\_image));

% Display Magnitude and Phase of Rotated Image

magnitude\_rotated = 20 \* log(abs(F\_rotated) + 1);

phase\_rotated = angle(F\_rotated);

figure(3);

subplot(1, 2, 1);

imshow(magnitude\_rotated, []);

title('Magnitude Spectrum (Rotated Image)');

subplot(1, 2, 2);

imshow(phase\_rotated, []);

title('Phase Spectrum (Rotated Image)');

% Load and Preprocess Two Images for Reconstruction

image2 = imread('lena.jpg');

image2\_gray = rgb2gray(image2);

image2\_gray = double(image2\_gray);

image2\_gray = imresize(image2\_gray, [64, 64]); % Resize for faster computation

% Compute DFTs for Both Images

F2 = fftshift(fft2(image2\_gray));

% Separate Magnitude and Phase

magnitude1 = abs(F1);

phase1 = angle(F1);

magnitude2 = abs(F2);

phase2 = angle(F2);

% Reconstruct Images Using Mixed Magnitude and Phase

reconstructed1 = ifft2(ifftshift(magnitude1 .\* exp(1i \* phase2)));

reconstructed2 = ifft2(ifftshift(magnitude2 .\* exp(1i \* phase1)));

% Display Reconstructed Images

figure(4);

subplot(1, 2, 1);

imshow(abs(reconstructed1), []);

title('Magnitude of Image 1 + Phase of Image 2');

subplot(1, 2, 2);

imshow(abs(reconstructed2), []);

title('Magnitude of Image 2 + Phase of Image 1');

Difference between manual and built-in DFT: 392957.0292

**In python**

import numpy as np

import cv2

import matplotlib.pyplot as plt

# Function for Manual DFT

def manual\_dft2d(image):

M, N = image.shape

F = np.zeros((M, N), dtype=complex)

for u in range(M):

for v in range(N):

sum\_val = 0

for x in range(M):

for y in range(N):

sum\_val += image[x, y] \* np.exp(-1j \* 2 \* np.pi \* ((u \* x / M) + (v \* y / N)))

F[u, v] = sum\_val

return F

# Load and preprocess image

image = cv2.imread('lotus.jpg', cv2.IMREAD\_GRAYSCALE)

image = cv2.resize(image, (64, 64))

image\_gray = image.astype(np.float64)

# Manual DFT

F\_manual = manual\_dft2d(image\_gray)

# Built-in DFT

F\_builtin = np.fft.fftshift(np.fft.fft2(image\_gray))

# Verify difference

difference = np.linalg.norm(F\_manual - F\_builtin)

print(f'Difference between manual and built-in DFT: {difference}')

# Magnitude and Phase Spectra

magnitude\_spectrum = 20 \* np.log(np.abs(F\_builtin) + 1)

phase\_spectrum = np.angle(F\_builtin)

plt.figure(figsize=(10, 4))

plt.subplot(1, 2, 1)

plt.imshow(magnitude\_spectrum, cmap='gray')

plt.title('Magnitude Spectrum')

plt.axis('off')

plt.subplot(1, 2, 2)

plt.imshow(phase\_spectrum, cmap='gray')

plt.title('Phase Spectrum')

plt.axis('off')

plt.show()

# Shifted Image and DFT

shifted\_image = np.roll(image\_gray, shift=10, axis=(0, 1))

F\_shifted = np.fft.fftshift(np.fft.fft2(shifted\_image))

magnitude\_shifted = 20 \* np.log(np.abs(F\_shifted) + 1)

phase\_shifted = np.angle(F\_shifted)

plt.figure(figsize=(10, 4))

plt.subplot(1, 2, 1)

plt.imshow(magnitude\_shifted, cmap='gray')

plt.title('Magnitude Spectrum (Shifted Image)')

plt.axis('off')

plt.subplot(1, 2, 2)

plt.imshow(phase\_shifted, cmap='gray')

plt.title('Phase Spectrum (Shifted Image)')

plt.axis('off')

plt.show()

# Rotated Image and DFT

rotated\_image = np.rot90(image\_gray)

F\_rotated = np.fft.fftshift(np.fft.fft2(rotated\_image))

magnitude\_rotated = 20 \* np.log(np.abs(F\_rotated) + 1)

phase\_rotated = np.angle(F\_rotated)

plt.figure(figsize=(10, 4))

plt.subplot(1, 2, 1)

plt.imshow(magnitude\_rotated, cmap='gray')

plt.title('Magnitude Spectrum (Rotated Image)')

plt.axis('off')

plt.subplot(1, 2, 2)

plt.imshow(phase\_rotated, cmap='gray')

plt.title('Phase Spectrum (Rotated Image)')

plt.axis('off')

plt.show()

# Load second image and preprocess

image2 = cv2.imread('lena.jpg', cv2.IMREAD\_GRAYSCALE)

image2 = cv2.resize(image2, (64, 64))

image2\_gray = image2.astype(np.float64)

# Compute DFTs for both images

F1 = F\_builtin

F2 = np.fft.fftshift(np.fft.fft2(image2\_gray))

# Separate magnitude and phase

magnitude1 = np.abs(F1)

phase1 = np.angle(F1)

magnitude2 = np.abs(F2)

phase2 = np.angle(F2)

# Reconstruct images with mixed magnitude and phase

reconstructed1 = np.fft.ifft2(np.fft.ifftshift(magnitude1 \* np.exp(1j \* phase2)))

reconstructed2 = np.fft.ifft2(np.fft.ifftshift(magnitude2 \* np.exp(1j \* phase1)))

# Display reconstructed images

plt.figure(figsize=(10, 4))

plt.subplot(1, 2, 1)

plt.imshow(np.abs(reconstructed1), cmap='gray')

plt.title('Magnitude of Image 1 + Phase of Image 2')

plt.axis('off')

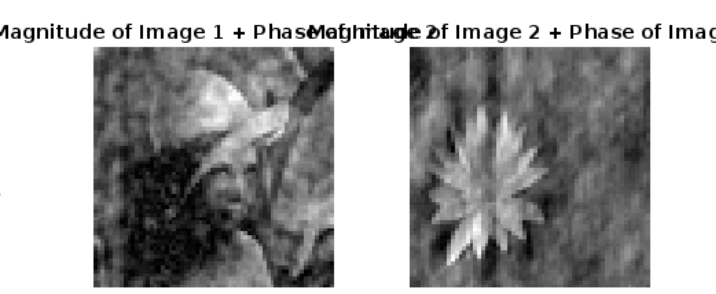
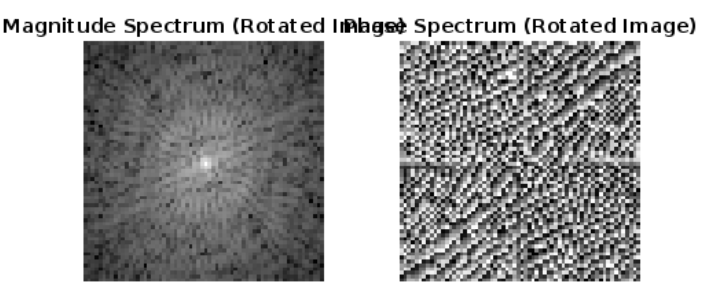
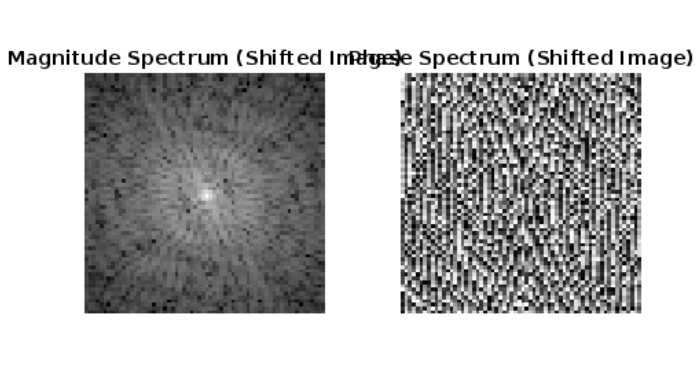
plt.subplot(1, 2, 2)

plt.imshow(np.abs(reconstructed2), cmap='gray')

plt.title('Magnitude of Image 2 + Phase of Image 1')

plt.axis('off')

plt.show()

****

**Lab 11**

close all;

%ui22ec17

img = imread('fractured\_spine.tif');

% Convert to binary if grayscale

if ~islogical(img)

img = imbinarize(img);

end

% Define a 5x5 Structuring Element

se = strel('square', 5);

% Morphological Operations

erosion = imerode(img, se);

dilation = imdilate(img, se);

opening = imopen(img, se);

closing = imclose(img, se);

% Display Results

figure;

subplot(2,3,1),

imshow(img), title('Original Image');

subplot(2,3,2),

imshow(erosion), title('Erosion');

subplot(2,3,3),

imshow(dilation), title('Dilation');

subplot(2,3,4),

imshow(opening), title('Opening');

subplot(2,3,5),

imshow(closing), title('Closing');

**in python**

import cv2

import numpy as np

from matplotlib import pyplot as plt

# Read the image in grayscale

img = cv2.imread('fractured\_spine.tif', cv2.IMREAD\_GRAYSCALE)

# Convert to binary

\_, img\_bin = cv2.threshold(img, 127, 255, cv2.THRESH\_BINARY)

# Define a 5x5 Structuring Element

se = np.ones((5,5), np.uint8)

# Morphological Operations

erosion = cv2.erode(img\_bin, se)

dilation = cv2.dilate(img\_bin, se)

opening = cv2.morphologyEx(img\_bin, cv2.MORPH\_OPEN, se)

closing = cv2.morphologyEx(img\_bin, cv2.MORPH\_CLOSE, se)

# Display Results

plt.figure(figsize=(10,6))

plt.subplot(2, 3, 1)

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

plt.title('Original Image')

plt.axis('off')

plt.subplot(2, 3, 2)

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

plt.title('Erosion')

plt.axis('off')

plt.subplot(2, 3, 3)

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

plt.title('Dilation')

plt.axis('off')

plt.subplot(2, 3, 4)

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

plt.title('Opening')

plt.axis('off')

plt.subplot(2, 3, 5)

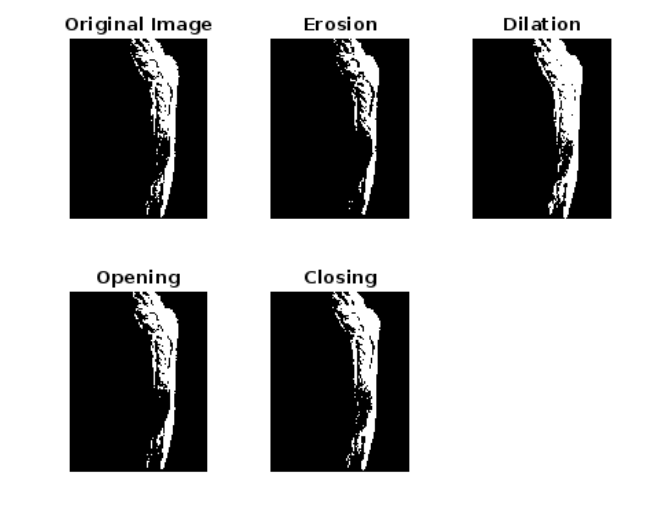
plt.imshow(closing, cmap='gray')

plt.title('Closing')

plt.axis('off')

plt.tight\_layout()

plt.show()

****