Indian Institute of Information Technology Bhopal

Department of Computer Science & Engineering

Semester-V

Course Code: CSE 312

Digital Image Processing

Submitted By:

Labhanshu Gupta

22U02007

Submitted To:

Dr. Bhupendra Singh Kirar

Index

	• NOT Operation of Images • Calculate Intersection of Two Images □				
7	Write and Execute Programs for Geometric Transformation of Images • Translation • Rotation • Scaling • Reflection • Shrinking • Zooming	11/09/2024	5-11-2024	22-27	
8	Write and Execute Programs for Image Frequency Domain Filtering • Spatial Resolution • Intensity Resolution	11/09/2024	5-11-2024	28-36	
9	Write and Execute Programs for Image Frequency Domain Filtering • Apply FFT on Given Image • Perform Low Pass and High Pass Filtering in Frequency Domain • Apply IFFT to Reconstruct Image	09/10/2024	5-11-2024	33-37	
10	Write a Program in MATLAB for Edge Detection Using Different Edge Detection Masks	09/10/2024	5-11-2024	38-41	
11	1. Write and Execute Programs to Remove Noise Using Spatial Filters • Understand 1-D and 2-D Convolution Process • Use 3x3 Mask for Low Pass Filter and High Pass Filter	16/10/2024	5-11-2024	42-44	

ASSIGNMENT 1

Objective: Introduction to MATLAB commands and functions used in Digital Image Processing

Theory:

S No.	Matlab Cmd and functions	Description	Example Usage
1	imread	Reads an image from a file and stores it in a matrix.	I = imread('image.jpg');
2	imshow	Displays an image in a MATLAB figure window.	imshow(I);
3	rgb2gray	Converts an RGB image to a grayscale image.	I_gray = rgb2gray(I);
4	imhist	Displays the histogram of a grayscale image.	imhist(I_gray);
5	histeq	Enhances the contrast of an image using histogram equalization.	I_eq = histeq(I_gray);
6	imresize	Resizes an image to a specified size.	I_resized = imresize(I, [256 256]);
7	imadd	Adds two images or adds a constant to an image.	I_add = imadd(I1, I2);
8	imsubtract	Subtracts one image from another.	I_sub = imsubtract(I1, I2);
9	immultiply	Multiplies two images or an image by a constant.	I_mul = immultiply(I1, I2);
10	imdivide	Divides one image by another or divides an image by a constant.	I_div = imdivide(I1, I2);
11	imfilter	Applies a filter to an image (e.g., for smoothing or edge detection).	I_filt = imfilter(I, fspecial('gaussian', [3 3], 0.5));
12	fspecial	Creates a predefined 2D filter, such as Gaussian, Sobel, etc.	h = fspecial('sobel');
13	edge	Detects edges in an image using different methods such as Sobel, Canny, etc.	I_edge = edge(I_gray, 'canny');

14	subplot	Displays multiple plots or images in a single figure window.	subplot(1, 2, 1);
			imshow(I1);
			subplot(1, 2, 2);
			imshow(I2);

15	imshowpair	Displays two images side by side for comparison.	imshowpair(I 1, I2, 'montage');

ASSIGNMENT 2

Objective: Write programs to read and display images using subplot function in MATLAB

Theory:

- **imread**: Loads an image file from disk and stores it in a variable as a 2D matrix for grayscale images or a 3D matrix for RGB images.
- **imshow**: Displays the loaded image within a figure window.
- **subplot**: Splits the figure window into a grid, allowing the visualization of multiple images at once. This is particularly useful for comparing original and processed images side by side.

These functions are essential for efficiently handling image processing tasks and enabling direct visual comparisons.

Example Functions:

- **imread**: Imports image files (like .jpg or .png) into the MATLAB workspace.
- **imshow**: Shows the image in a figure window for easy viewing.

Code:

```
clear; clc;

filename = "labhanshu.jpg";
img = imread(filename);
figure;
imshow(img);
title("Displayed Image");
[rows, cols, channels] = size(img);
fprintf('Image size: %d x %d x %d\n', rows, cols, channels); fprintf('Image data type: %s\n',class(img));
```

Output:

Displayed Image-

ASSIGNMENT 3

Objective: Write programs to gray scale images and display images using subplot function

Theory:

Images can be represented in various formats, such as RGB (Red, Green, Blue) or grayscale. An RGB image consists of three separate color channels, while a grayscale image uses different shades of gray, simplifying many image processing tasks. Converting RGB images to grayscale reduces computational demands and is often a preliminary step in advanced techniques like edge detection or thresholding.

In MATLAB, the **rgb2gray** function is used for this conversion. The luminance of each pixel is calculated from its RGB values using the formula:

$Y=0.2989\times R+0.5870\times G+0.1140\times B$

This process removes color data, preserving only the intensity, which simplifies algorithms that rely on intensity variations. The **subplot** function allows side-by-side comparisons of grayscale and RGB images in one figure, making it easier to visualize differences.

Example Functions:

- rgb2gray: Converts an RGB image to a grayscale image.
- subplot: Displays grayscale and RGB images side by side for comparison.

Code:

```
clear;
clc;

filename = "lahbanshu.jpg";

color_img = imread(filename);
gray_img = rgb2gray(color_img);
figure;

subplot(1, 2, 1); imshow(color_img);
title('Original Color Image');
subplot(1, 2, 2); imshow(gray_img);
title('Grayscale Image');

fprintf('Original Image size: %d x %d x %d\n', size(color_img, 1), size(color_img, 2),
```

```
size(color_img, 3));
fprintf('Grayscale Image size: %d x %d\n', size(gray_img, 1), size(gray_img, 2));
```

Output:

ASSIGNMENT 4

Objective: Write a program for histogram calculation and equalization

- Standard MATLAB function
- Enhancing Contrast using histogram equalization

Theory:

Histograms graphically represent the distribution of pixel intensities in an image, revealing details about its brightness and contrast by showing the number of pixels for each intensity level.

- In an 8-bit grayscale image, the histogram spans intensity values from 0 (black) to 255 (white), illustrating how frequently each value occurs.
- **Histogram equalization** enhances image contrast by redistributing pixel intensities more evenly across the full range. This technique brightens dark areas and darkens bright ones, creating a more balanced image with improved visibility.

In MATLAB, the **imhist** function calculates and displays an image's histogram, while **histeq** performs histogram equalization to enhance contrast, particularly useful for images with poor lighting conditions.

Advantages of Histogram Equalization:

1. Enhances the overall contrast of images.
2. Beneficial for images with uneven lighting or brightness variations.
3. Commonly used to improve the visibility of features in medical or satellite imagery.

Example Functions:

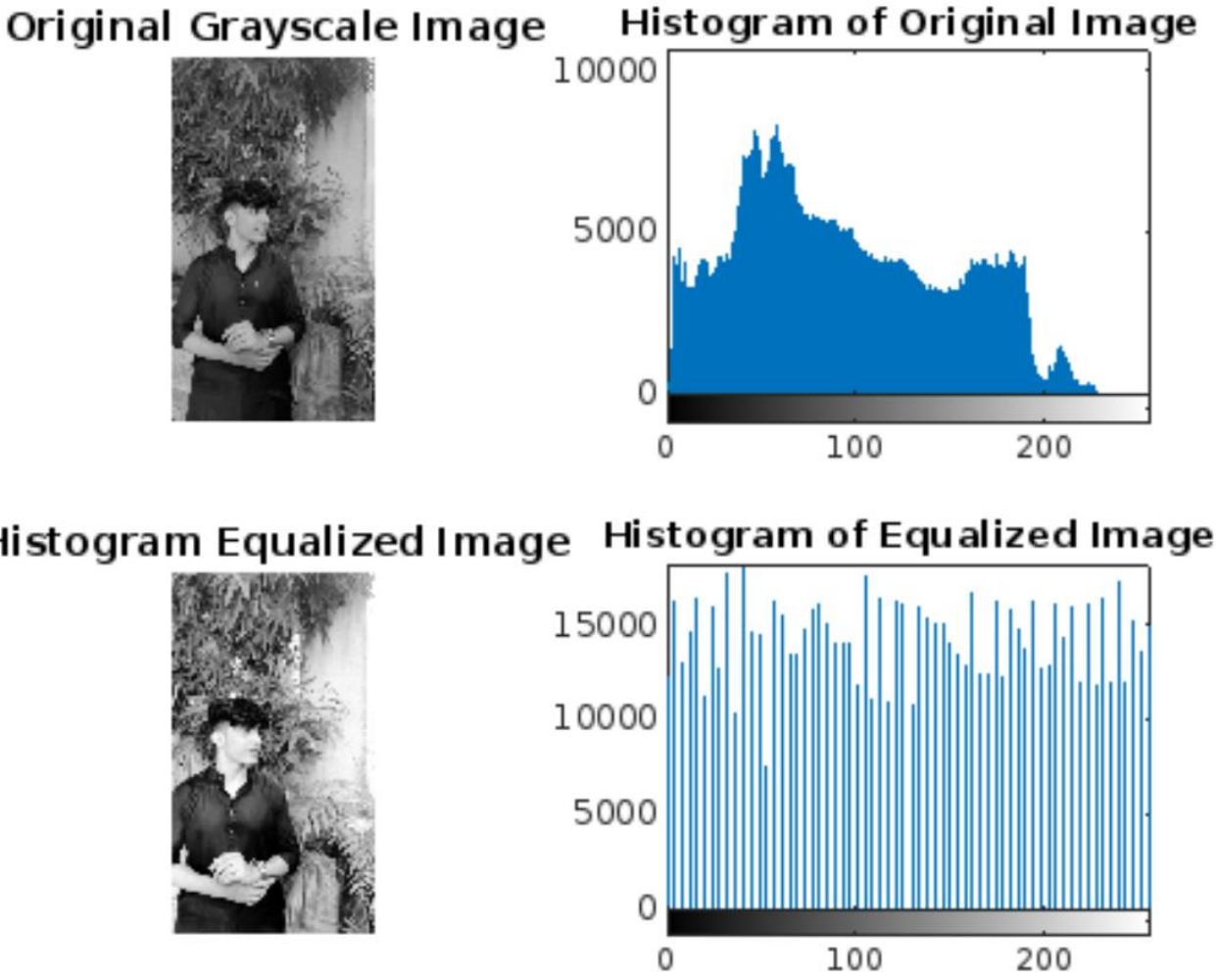
- **imhist**: Generates the histogram of an image.
- **histeq**: Performs histogram equalization to improve image contrast.

Code:

```
I = imread("labhanshu.jpg");
if size(I, 3) == 3
I = rgb2gray(I);
end

I_eq = histeq(I);
```

```
figure; subplot(1, 2, 1); imshow(I);
title('Original Image');

subplot(1, 2, 2); imshow(I_eq);
title('Histogram Equalized Image');

figure; subplot(1, 2, 1); imhist(I);
title('Histogram of Original Image');

subplot(1, 2, 2); imhist(I_eq);
title('Histogram of Equalized Image');
```

Output:

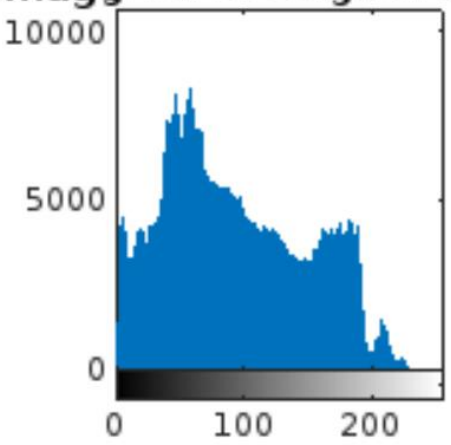


- **Enhancing contrast using histogram equalization**

```
clear;
clc;
close all;

imageFile = 'example.jpg'; % Replace with your image file name
img = imread(imageFile);

gray_img = rgb2gray(img);

subplot(2, 3, 1);
imshow(gray_img);
title('Original Grayscale Image');

subplot(2, 3, 2);
imhist(gray_img);
title('Histogram of Original Image');

equalized_img = histeq(gray_img);

subplot(2, 3, 4);
imshow(equalized_img);
title('Histogram Equalized Image');

subplot(2, 3, 5);
imhist(equalized_img);
title('Histogram of Equalized Image');

subplot(2, 3, 3);
imshow(gray_img);
title('Original Image');

subplot(2, 3, 6);
imshow(equalized_img);
title('Enhanced Image');
```

Original Grayscale Image

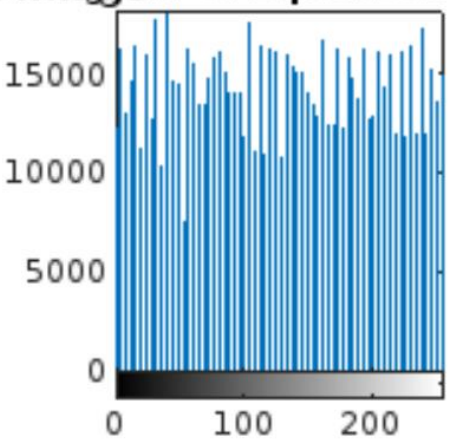
Histogram of Original Image



Original Image

Histogram Equalized Image

Histogram of Equalized Image



Enhanced Image

ASSIGNMENT 5

Objective: Write and execute the program of addition of two images

Theory:

Image arithmetic operations are fundamental techniques used in image processing to combine or manipulate images in various ways. These operations can help enhance image features, adjust brightness, and create composite effects. MATLAB provides a variety of built-in functions to perform these operations efficiently.

1. Addition of Two Images

Adding two images combines their pixel values, which can be used to merge features from both images or increase overall brightness. In MATLAB, this is typically done using the **imadd** function. If the sum of pixel values exceeds 255 (the maximum intensity value for 8-bit images), the result is clipped to avoid overflow.

2. Subtraction of One Image from Another

Subtracting the pixel values of one image from another highlights differences between the images. This operation is useful for detecting changes or extracting specific features. MATLAB uses the **imsubtract** function for this, and values are clipped to remain within valid intensity ranges (0 to 255).

3. Multiplication of Two Images

Multiplying two images combines their pixel values, which can be used for blending or masking operations. This can also affect image brightness and contrast. The **immultiply** function in MATLAB performs element-wise multiplication, scaling pixel values appropriately.

4. Division of Two Images

Dividing one image by another can enhance contrast or suppress certain features in an image. The **imdivide** function performs pixel-wise division, ensuring the resulting values stay within valid intensity ranges. Care must be taken to avoid division by zero, which can lead to undefined values.

5. Image Blending

Image blending combines two images smoothly by adjusting their relative contributions. This technique is often used in image composition or creating fade effects. In MATLAB, blending is typically achieved using weighted addition:

$$\text{Blended Image} = \alpha \times \text{Image1} + (1 - \alpha) \times \text{Image2}$$

where **α** is a blending factor between 0 and 1.

6. Calculate Mean Value of an Image

The mean value of an image represents the average intensity of its pixels, providing a measure of overall brightness. This value is useful for analyzing image characteristics or performing brightness adjustments. MATLAB's **mean** function, combined with **mean2** for 2D arrays, calculates the mean intensity.

7. Adjusting Brightness by Changing Mean Value

Brightness can be adjusted by adding or subtracting a constant value to all pixels or by scaling pixel values relative to the mean intensity. Increasing the mean value brightens the image, while decreasing it darkens the image. MATLAB allows for this adjustment using basic arithmetic operations on image matrices.

These arithmetic operations are essential in image analysis, enhancement, and manipulation, forming the basis for more complex image processing tasks.

Code:

```
image1 = imread(labhanshu.jpg');
image2 = imread(image2.jpg');

if size(image1) ~= size(image2)
   error('Images must be of the same size!');
end

figure;
subplot(3, 3, 1);
imshow(image1);
title('Image 1');

subplot(3, 3, 2);
imshow(image2);
title('Image 2');

added_image = imadd(image1, image2);
subplot(3, 3, 3);
imshow(added_image);
title('Addition of Two Images');

subtracted_image = imsubtract(image1, image2);
subplot(3, 3, 4);
imshow(subtracted_image);
```

```
title('Subtraction of Images');

multiplied_image = immultiply(image1, image2);
subplot(3, 3, 5);
imshow(multiplied_image);
title('Multiplication of Images');

small_constant = 1e-10;
divided_image = imdivide(double(image1), double(image2) + small_constant);
divided_image = uint8(divided_image);
subplot(3, 3, 6);
imshow(divided_image);
title('Division of Images');

alpha = 0.5;
blended_image = imadd(immultiply(image1, alpha), immultiply(image2, 1 - alpha));
subplot(3, 3, 7);
imshow(blended_image);
title('Image Blending');

mean_value = mean(image1(:));
disp(['Mean value of image1: ', num2str(mean_value)]);

desired_mean = 150;
current_mean = mean(image1(:));
brightness_adjustment = desired_mean - current_mean;
brightened_image = image1 + brightness_adjustment;
brightened_image = uint8(min(double(brightened_image), 255));
subplot(3, 3, 8);
imshow(brightened_image);
title('Brightness Adjusted Image');
```

Output:

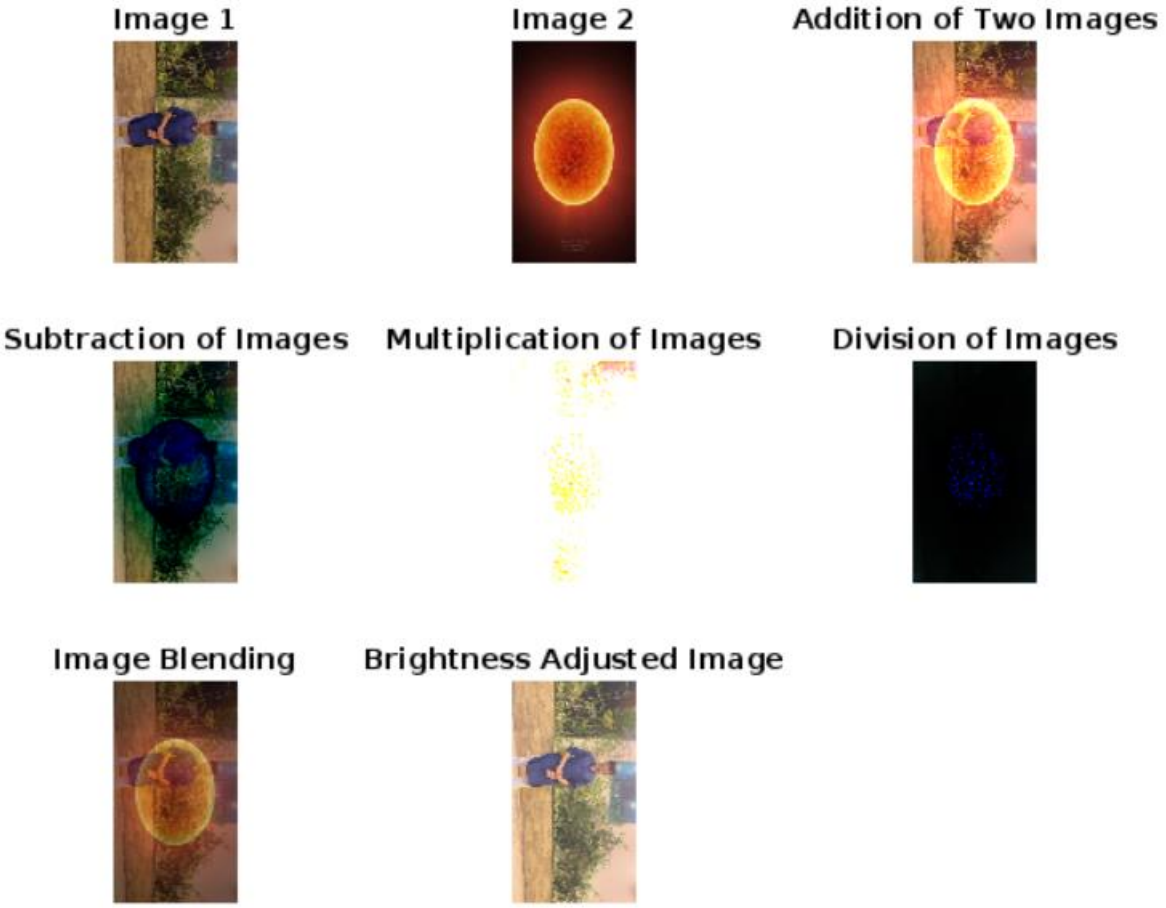


Image 1
Image 2
Addition of Two Images
Subtraction of Images
Multiplication of Images
Division of Images
Image Blending
Brightness Adjusted Image

EXPERIMENT NO. 6

Objective: To write and execute programs for image logical operations

Theory:

Image logical operations are essential in image processing, especially for tasks that involve masking, feature extraction, segmentation, and combining binary images. These operations apply logical rules to pixel values, which are often represented as binary images where pixels have values of 0 (black) or 1 (white). MATLAB provides built-in functions to perform these logical operations efficiently.

1. AND Operation Between Two Images

The AND operation compares corresponding pixels from two images. The result is 1 if both pixels are 1; otherwise, it is 0. This operation is useful for masking images, where only the overlapping regions are highlighted. In MATLAB, the **bitand** function performs this operation.

2. OR Operation Between Two Images

The OR operation compares corresponding pixels from two images. The result is 1 if at least one of the pixels is 1. This operation is often used to combine regions of interest from different images. MATLAB provides the **bitor** function for this purpose.

3. NAND Operation Between Two Images

The NAND operation is the negation of the AND operation. It outputs 0 if both input pixels are 1 and 1 otherwise. This operation is useful for highlighting areas where at least one of the conditions is not met. MATLAB can implement this using the **bitand** function followed by a negation operation.

4. NOR Operation Between Two Images

The NOR operation is the negation of the OR operation. It outputs 1 only if both input pixels are 0 and 0 otherwise. This operation is commonly used in scenarios where you want to emphasize areas without overlap. MATLAB can achieve this using the **bitor** function followed by negation.

5. EXOR (XOR) Operation Between Two Images

The EXOR operation (exclusive OR) outputs 1 if the input pixels differ (one is 1 and the other is 0) and 0 if they are the same. This operation is useful for detecting differences between two images. In MATLAB, the **bitxor** function is used for this purpose.

6. EXNOR (XNOR) Operation Between Two Images

The EXNOR operation (exclusive NOR) is the negation of the EXOR operation. It outputs 1 if the input pixels are the same (both 0 or both 1) and 0 if they differ. This operation can be used to highlight similarities between images. It can be performed using the **bitxor** function followed by negation in MATLAB.

7. NOT Operation on Images

The NOT operation inverts the pixel values of an image, changing 0 to 1 and 1 to 0. This operation is useful for creating negative images or inverting masks. MATLAB uses the **~** operator for this operation.

8. Calculating the Intersection of Two Images

The intersection of two images can be determined using the AND operation, which highlights the regions common to both images. This technique is widely used in applications such as image masking and region analysis. The **bitand** function in MATLAB helps calculate this intersection.

MATLAB Code:

```
image1 = imread(labhanshu.jpg');
image2 = imread(image2.jpg');

if size(image1) ~= size(image2)
   error('Images must be of the same size!');
end

figure;

subplot(3, 3, 1);
imshow(image1);
title('Image 1');

subplot(3, 3, 2);
imshow(image2);
title('Image 2');

and_image = bitand(image1, image2);
```

```
subplot(3, 3, 3);
imshow(and_image);
title('AND Operation');

or_image = bitor(image1, image2);
subplot(3, 3, 4);
imshow(or_image);
title('OR Operation');

nand_image = bitcmp(and_image);
subplot(3, 3, 5);
imshow(nand_image);
title('NAND Operation');

nor_image = bitcmp(or_image);
subplot(3, 3, 6);
imshow(nor_image);
title('NOR Operation');

xor_image = bitxor(image1, image2);
subplot(3, 3, 7);
imshow(xor_image);
title('EXOR Operation');

exnor_image = bitcmp(xor_image);
subplot(3, 3, 8);
imshow(exnor_image);
title('EXNOR Operation');

not_image = bitcmp(image1);
subplot(3, 3, 9);
imshow(not_image);
title('NOT Operation');
```

Image 1

Image 2

AND Operation

OR Operation

NAND Operation

NOR Operation

EXOR Operation

EXNOR Operation

NOT Operation

Conclusion:

MATLAB program to implement logical operations is implemented..

EXPERIMENT NO. 7

Objective: To write and execute program for geometric transformation of image

Theory:
We will perform following geometric transformations on the image in this experiment

Translation: Translation is movement of image to new position. Mathematically translation is represented as:

$$x' = x + \delta x \quad \text{and } y' = y + \delta y$$

In matrix form translation is represented by:

$$[x' \quad y' \quad 1] = \begin{bmatrix} 1 & 0 & \delta x \\ 0 & 1 & \delta y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Scaling: Scaling means enlarging or shrinking. Mathematically scaling can be represented by:

$$x' = x \times S_x \quad \text{and } y' = y \times S_y$$

In matrix form scaling is represented by:

$$[x' \quad y' \quad 1] = \begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Rotation: Image can be rotated by an angle θ, in matrix form it can be represented asIfθ is substituted with -θ, this matrix rotates the image in clockwise direction.

$$x' = x\cos\theta - y\sin\theta \quad \text{and} \quad y' = x\sin\theta + y\cos\theta$$

In matrix form rotation is represented by:

$$\begin{bmatrix} x' & y' & 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Shearing: Image can be distorted (sheared) either in x direction or y direction. Shearing can be represented as:

$$x' = sh_x \times y \qquad y' = y$$

In matrix form rotation is represented by:

$$X_{shear} = \begin{bmatrix} 1 & 0 & 0 \\ sh_x & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Shearing in Y direction can be given by:

$$x' = x \qquad y' = y \times sh_y$$

$$Y_{shear} = \begin{bmatrix} 1 & sh_y & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Zooming : zooming of image can be done by process called pixel replication or interpolation. Linear interpolation or some non-linear interpolation like cubic interpolation can be performed for better result.

Reflection in Image Processing

Reflection, also known as image flipping, is a geometric transformation that flips an image across a specified axis. This transformation can be applied either horizontally or vertically:

- **Horizontal Reflection**: Flips the image along the vertical axis, creating a mirror image from left to right. In MATLAB, this can be achieved using the **flip** function with the appropriate axis argument.
- **Vertical Reflection**: Flips the image along the horizontal axis, generating a mirror image from top to bottom. The **flip** function can also be used for this transformation.

Reflection is useful in applications like image symmetry analysis, creating mirror effects, or preparing images for data augmentation in machine learning.

Shrinking in Image Processing

Shrinking, also known as image downscaling or resizing, reduces the dimensions of an image. This operation is useful for various purposes, such as compressing image data, reducing the computational cost of image analysis, or fitting images into smaller displays.

- Shrinking can be performed using MATLAB's **imresize** function. The function allows for specifying a scaling factor or the desired output size.
- When an image is shrunk, the number of pixels is reduced, which may lead to a loss of detail. Various interpolation methods (such as nearest-neighbor, bilinear, or bicubic) can be used to determine how pixel values are calculated in the resized image.

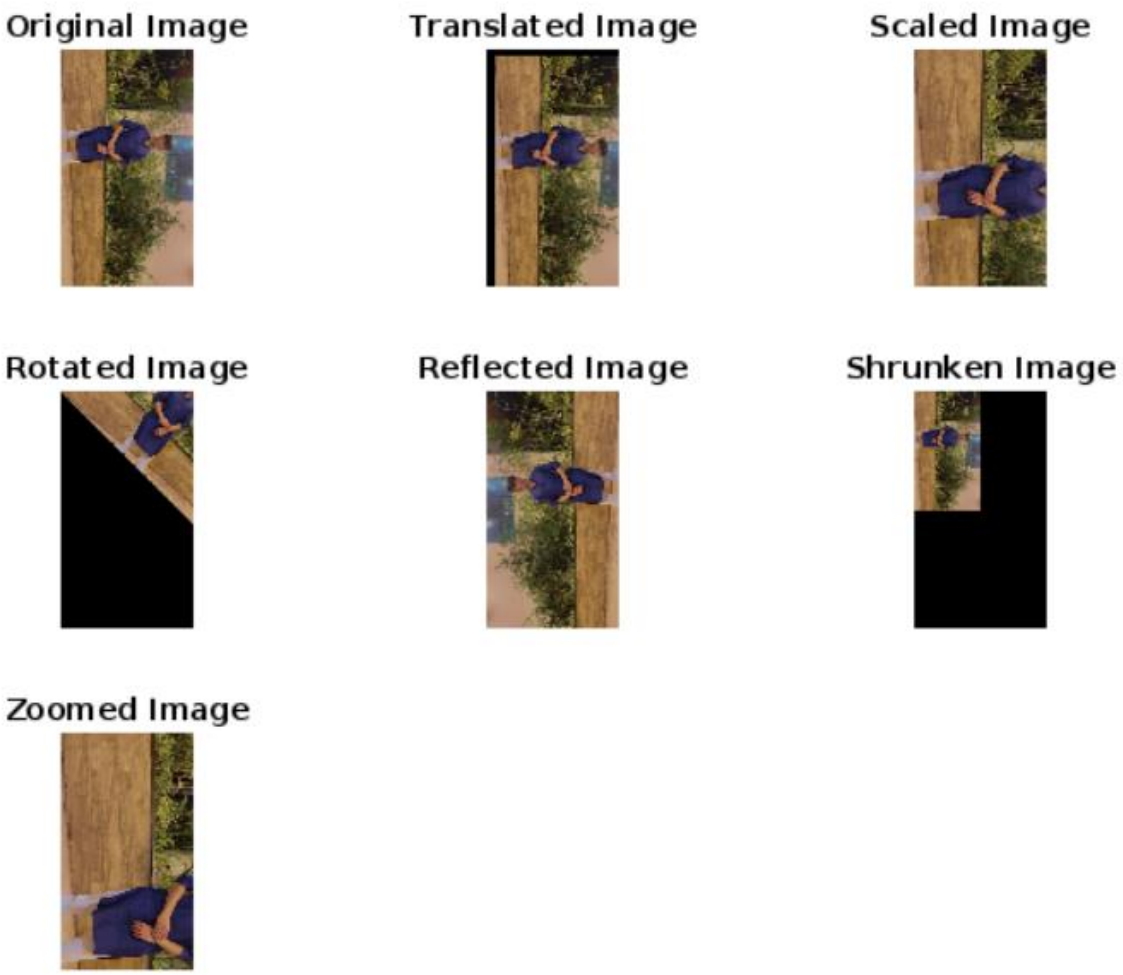
MATLAB Code:

```
image = imread(labhanshu.jpg');


figure;


subplot(3, 3, 1);
imshow(image);
title('Original Image');


tform_translate = affine2d([1 0 0; 0 1 0; 50 30 1]);
translated_image = imwarp(image, tform_translate, 'OutputView',
imref2d(size(image)));
subplot(3, 3, 2);
```

```
imshow(translated_image);
title('Translated Image');

tform_scale = affine2d([1.5 0 0; 0 1.5 0; 0 0 1]);
scaled_image = imwarp(image, tform_scale, 'OutputView',
imref2d(size(image)));
subplot(3, 3, 3);
imshow(scaled_image);
title('Scaled Image');

angle = 45;
tform_rotate = affine2d([cosd(angle) -sind(angle) 0; sind(angle)
cosd(angle) 0; 0 0 1]);
rotated_image = imwarp(image, tform_rotate, 'OutputView',
imref2d(size(image)));
subplot(3, 3, 4);
imshow(rotated_image);
title('Rotated Image');

tform_reflection = affine2d([-1 0 0; 0 1 0; size(image, 2) 0 1]);
reflected_image = imwarp(image, tform_reflection, 'OutputView',
imref2d(size(image)));
subplot(3, 3, 5);
imshow(reflected_image);
title('Reflected Image');

tform_shrink = affine2d([0.5 0 0; 0 0.5 0; 0 0 1]);
```

```
shrunk_image = imwarp(image, tform_shrink, 'OutputView',
imref2d(size(image)));
subplot(3, 3, 6);
imshow(shrunk_image);
title('Shrunken Image');

tform_zoom = affine2d([2 0 0; 0 2 0; 0 0 1]);
zoomed_image = imwarp(image, tform_zoom, 'OutputView',
imref2d(size(image)));
subplot(3, 3, 7);
imshow(zoomed_image);
title('Zoomed Image');
```

Output:



Conclusion

Different geometrical transforms are applied successfully

EXPERIMENT NO. 8

Objective:

Write and Exceute MATLAB programs for image frequency domain filtering, specifically focusing on:

1) **Spatial Resolution**: Understanding and manipulating the level of detail or sharpness in an image.
2) **Intensity Resolution**: Managing the range of intensity levels that an image can represent.

Theory:

Image Frequency Domain Filtering

Frequency domain filtering involves transforming an image from the spatial domain to the frequency domain using techniques such as the Fourier Transform. This method allows for the manipulation of specific frequency components to enhance or suppress image features. Filtering in the frequency domain is often used for image enhancement, noise reduction, and feature extraction.

1. Spatial Resolution

- Definition: Spatial resolution refers to the level of detail present in an image and is determined by the number of pixels used to represent it. Higher spatial resolution means more detail, while lower resolution results in a blurred or less detailed image.
- Frequency Domain Aspect: In the frequency domain, spatial resolution is linked to the high-frequency components, which correspond to edges and fine details in the image. Filtering operations can be designed to enhance or reduce these components to adjust the spatial resolution.

2. Intensity Resolution

- Definition: Intensity resolution refers to the number of distinct intensity levels that an image can represent, affecting how well subtle variations in brightness can be depicted.
- Frequency Domain Aspect: By altering the distribution of intensity levels or the dynamic range in the frequency domain, we can enhance or reduce the visibility of specific features. This approach is crucial for applications requiring precise contrast adjustments.

Implementation Approach

1. Transform the Image: Use the Fourier Transform (e.g., fft2 in MATLAB) to convert the image to the frequency domain.
2. Apply Filters: Design and apply appropriate filters to adjust spatial or intensity resolution as needed.
3. Inverse Transform: Use the Inverse Fourier Transform (e.g., ifft2) to return the modified image to the spatial domain for visualization.

Code:

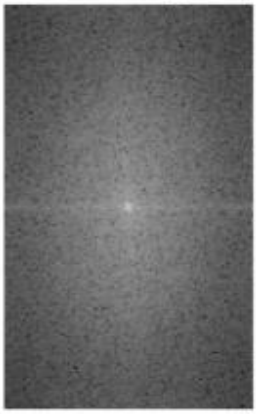
```
image = imread(labhanshu.jpg');
gray_image = rgb2gray(image);

fft_image = fft2(double(gray_image));
fft_shifted = fftshift(fft_image);
magnitude_spectrum = log(1 + abs(fft_shifted));

[m, n] = size(gray_image);
[u, v] = meshgrid(1:n, 1:m);
D0 = 50;
D = sqrt((u - n/2).^2 + (v - m/2).^2);

low_pass_filter = double(D <= D0);
low_pass_result = fft_shifted .* low_pass_filter;
low_pass_image = real(ifft2(ifftshift(low_pass_result)));

high_pass_filter = double(D > D0);
high_pass_result = fft_shifted .* high_pass_filter;
high_pass_image = real(ifft2(ifftshift(high_pass_result)));

figure;
subplot(2, 2, 1);
imshow(gray_image);
title('Original Grayscale Image');

subplot(2, 2, 2);
imshow(magnitude_spectrum, []);
title('Magnitude Spectrum');

subplot(2, 2, 3);
imshow(uint8(low_pass_image));
```

```
title('Low-Pass Filtered Image');

subplot(2, 2, 4);
imshow(uint8(high_pass_image));
title('High-Pass Filtered Image');

image = imread('punya.jpg');
gray_image = rgb2gray(image);

low_intensity_image = gray_image / 2;
high_intensity_image = min(gray_image * 2, 255);
bit_depth_image = bitshift(gray_image, -2);
bit_depth_image = bitshift(bit_depth_image, 2);

figure;
subplot(2, 2, 1);
imshow(gray_image);
title('Original Grayscale Image');

subplot(2, 2, 2);
imshow(low_intensity_image);
title('Reduced Intensity Resolution (1/2)');

subplot(2, 2, 3);
imshow(high_intensity_image);
title('Increased Intensity Resolution (x2)');

subplot(2, 2, 4);
imshow(bit_depth_image);
title('Reduced Bit Depth');
```

Output-:

Original Grayscale Image

Magnitude Spectrum



Low-Pass Filtered Image

High-Pass Filtered Image

Conclusion:

This assignment demonstrated how to use frequency domain filtering in MATLAB to adjust spatial and intensity resolution, enhancing or suppressing image features using Fourier Transforms and filters.

EXPERIMENT NO. 9

Objective:: Write and execute programs for image frequency domain filtering

Theory:

In spatial domain, we perform convolution of filter mask with image data. In frequency domain, we perform multiplication of Fourier transform of image data with filter transfer function.

Fourier transform of image f(x,y) of size MxN can be given by:

$$F(u,v) = \frac{1}{MN}\sum_{x=0}^{M-1}\sum_{y=0}^{N-1} f(x,y)e^{-j2\pi(\frac{ux}{M}+\frac{vy}{N})}$$

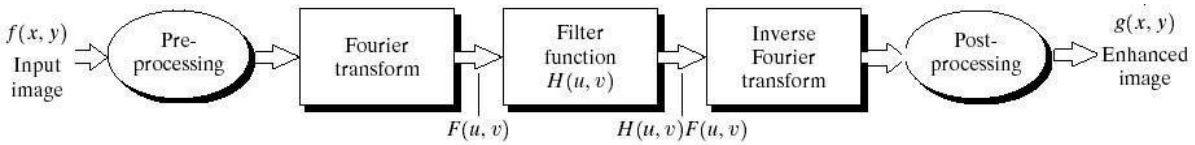
Where, u = 0,1,2 ..... M-1 and v = 0,1,2.....N-1 Inverse

Fourier transform is given by:

$$f(x,y) = \sum_{u=0}^{M-1}\sum_{v=0}^{N-1} F(u,v)e^{j2\pi(\frac{ux}{M}+\frac{vy}{N})}$$

Where, x = 0,1,2 ..... M-1 and y = 0,1,2.....N-1

Basic steps for filtering in frequency domain:



Pre-processing: Multiply input image f(x,y)by $(-1)^{x+y}$ to center the

transform

Computer Discrete Fourier Transform F(u,v) of input image f(x,y)

Multiply F(u,v) by filter function H(u,v) Result: H(u,v)F(u,v) Computer inverse DFT of the result

Obtain real part of the result

Post-Processing: Multiply the result by $(-1)^{x+y}$

MATLAB Code:

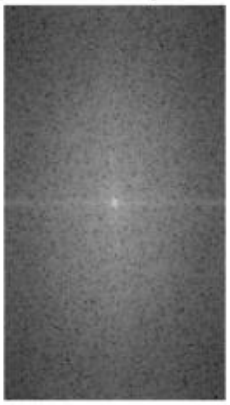
```
image = imread(labhanshu.jpg');
gray_image = rgb2gray(image);

fft_image = fft2(double(gray_image));
fft_shifted = fftshift(fft_image);
magnitude_spectrum = log(1 + abs(fft_shifted));

[m, n] = size(gray_image);
[u, v] = meshgrid(1:n, 1:m);
D0 = 50;
D = sqrt((u - n/2).^2 + (v - m/2).^2);
low_pass_filter = double(D <= D0);
low_pass_result = fft_shifted .* low_pass_filter;
low_pass_image = real(ifft2(ifftshift(low_pass_result)));

high_pass_filter = double(D > D0);
high_pass_result = fft_shifted .* high_pass_filter;
high_pass_image = real(ifft2(ifftshift(high_pass_result)));

figure;
subplot(2, 3, 1);
imshow(gray_image);
title('Original Grayscale Image');
```

```
subplot(2, 3, 2);
imshow(magnitude_spectrum, []);
title('Magnitude Spectrum');

subplot(2, 3, 3);
imshow(uint8(low_pass_image));
title('Low-Pass Filtered Image');

subplot(2, 3, 4);
imshow(uint8(high_pass_image));
title('High-Pass Filtered Image');

reconstructed_image = real(ifft2(ifftshift(fft_shifted)));
subplot(2, 3, 5);
imshow(uint8(reconstructed_image));
title('Reconstructed Image (IFFT)');
```

Result:

Original Grayscale Image

Magnitude Spectrum

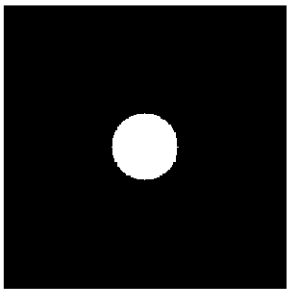


Low-Pass Filtered Image

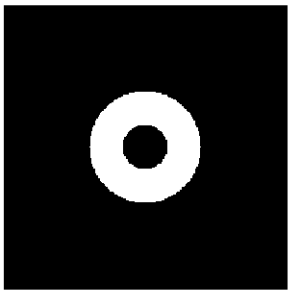
High-Pass Filtered Image

Reconstructed Image (IFFT)

Low Pass Mask



Band Pass Mask



Conclusion:

MATLAB program for frequency domain filtering is executed.

EXPERIMENT NO. 10

Aim:Write MATLAB code for edge detection using different edge detection mask.

Theory:

Image segmentation involves dividing an image into its constituent regions or objects. Segmentation should conclude when the objects of interest for a specific application are effectively isolated. The primary goal is to partition an image into meaningful regions tailored to the requirements of the application. This segmentation relies on measurements from the image, which may include gray levels, color, texture, depth, or motion.

There are two main types of image segmentation methods:

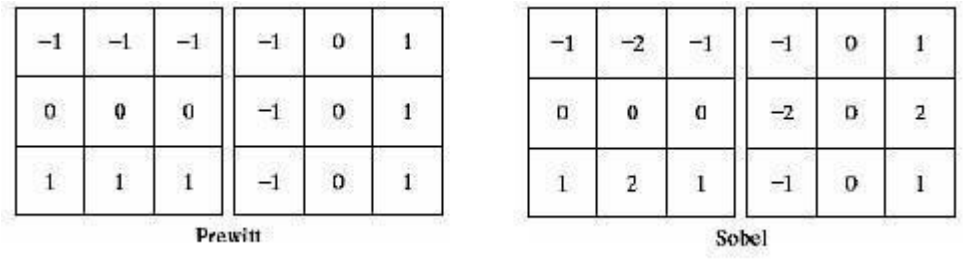
1. **Discontinuity-Based Segmentation**: Focuses on identifying abrupt changes in the image, such as isolated points, lines, or edges.
2. **Similarity-Based Segmentation**: Groups pixels with similar characteristics using methods like thresholding, region growing, and region splitting and merging.

Edge Detection is a popular discontinuity-based segmentation method. Edges are critical in image processing as they outline objects and indicate transitions in material properties, intensity, or depth. Pixels along these transitions are known as edge points. Edge detection techniques identify these gray level transitions using first-order or second-order derivative operators.

First order line detection 3x3 mask are:

−1	−1	−1	−1	−1	2	−1	2	−1	2	−1	−1	−1	−1	2	−1	2	−1
2	2	2	−1	2	−1	−1	2	−1	−1	2	−1	−1	2	−1	−1	2	−1
−1	−1	−1	2	−1	−1	−1	2	−1	−1	−1	2	−1	−1	2	−1	−1	2
Horizontal			+45°			Vertical			−45°								

Popular edge detection masks:



−1	−1	−1	−1	0	1
0	0	0	−1	0	1
1	1	1	−1	0	1

Prewitt

−1	−2	−1	−1	0	1
0	0	0	−2	0	2
1	2	1	−1	0	1

Sobel

The Sobel operator is more effective for edge detection in images with noise compared to the Prewitt operator because it incorporates averaging, which helps smooth the image while detecting edges. This smoothing effect reduces the likelihood of detecting spurious edges caused by noise.

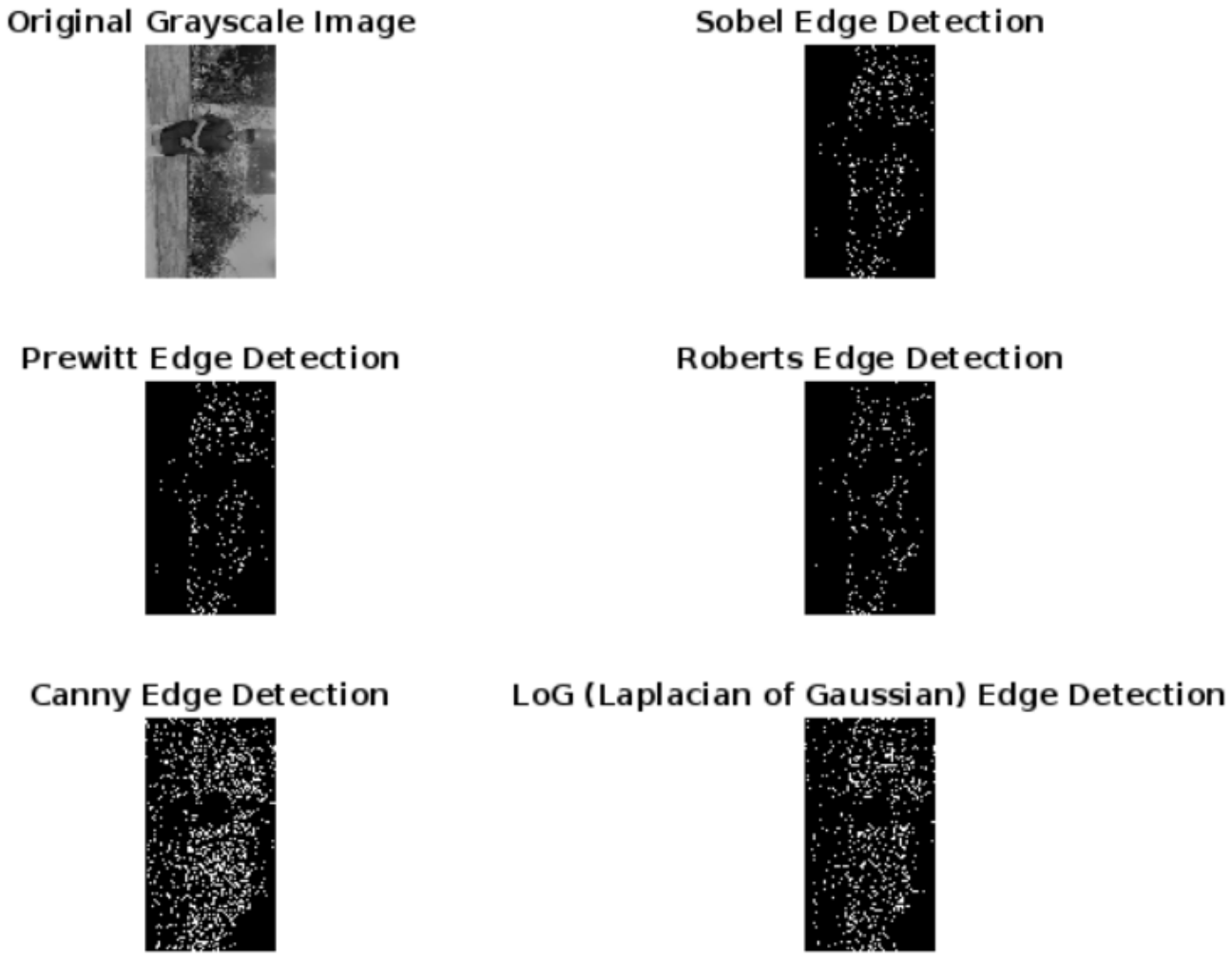
Second-order derivative operators, while highly sensitive to noise, are not typically used for direct edge detection. Instead, they are useful for extracting additional information, such as determining whether a point lies on the darker or brighter side of an edge based on the sign of the result. Additionally, zero-crossings in the second derivative can pinpoint the exact location of edges, especially in cases where the image transition is gradual.

MATLAB CODE:

```
image = imread(labhanshu.jpg');
gray_image = rgb2gray(image);

figure;

subplot(3, 2, 1);
imshow(gray_image);
title('Original Grayscale Image');

sobel_edges = edge(gray_image, 'Sobel');
subplot(3, 2, 2);
imshow(sobel_edges);
title('Sobel Edge Detection');

prewitt_edges = edge(gray_image, 'Prewitt');
subplot(3, 2, 3);
imshow(prewitt_edges);
title('Prewitt Edge Detection');
```

```
roberts_edges = edge(gray_image, 'Roberts');
subplot(3, 2, 4);
imshow(roberts_edges);
title('Roberts Edge Detection');

canny_edges = edge(gray_image, 'Canny');
subplot(3, 2, 5);
imshow(canny_edges);
title('Canny Edge Detection');

log_edges = edge(gray_image, 'log');
subplot(3, 2, 6);
imshow(log_edges);
title('LoG (Laplacian of Gaussian) Edge Detection');
```

Result:



Conclusion

MATLAB program for edge detection is execute

EXPERIMENT NO. 11

Aim:

Write and execute programs to remove noise using spatial filters

- Understand 1-D and 2-D convolution process
- Use 3x3 Mask for low pass filter and high pass filter

Theory:

Spatial Filtering, also known as neighborhood processing, involves defining a central point and applying a filter operation to the surrounding pixels within a specified neighborhood. The operation results in a single value, which is assigned to the central point in the modified image. This process is repeated for each pixel, using its neighbors to determine the output.

The concept works like a "sliding filter" that moves across the image, computing values for the central location. In spatial filtering, convolution is performed between the image data and the filter coefficients. For image processing, this involves convolving a set of 3x3 filter coefficients with 2D image data, while in signal processing, convolution is applied between 1D data and a set of filter coefficients.

MATLAB CODE:

Program for 2D convolution:

1D convolution (Useful for 1-D Signal Processing):

```
clc;
close all;
clear all;
L1=[1 1 1;1 1 1;1 1 1];
L2=[0 1 0;1 2 1;0 1 0];
L3=[1 2 1;2 4 2;1 2 1];
H1=[-1 -1 -1;-1 9 -1;-1 -1 -1];
H2=[0 -1 0;-1 5 -1;-0 -1 0];
H3=[1 -2 1;-2 5 -2;1 -2 1];

Filename=”labhanshu.jpg”
myimage = imread(filename);
if(size(myimage,3)==3)
myimage=rgb2gray(myimage);
```

```
end
subplot(3,2,1);
imshow(myimage); title('Original Image');

L1 = L1/sum(L1);
filt_image= conv2(double(myimage),double(L1)); subplot(3,2,2); imshow(filt_image,[]);
title('Filtered image with mask L1');

L2 = L2/sum(L2);
filt_image= conv2(double(myimage),double(L2));
 subplot(3,2,3);
 imshow(filt_image,[]); title('Filtered image with mask L2');

L3 = L3/sum(L3);
filt_image= conv2(double(myimage),double(L3));
subplot(3,2,4); imshow(filt_image,[]); title('Filtered image with mask L3');

filt_image= conv2(double(myimage),H1);
subplot(3,2,5); imshow(filt_image,[]); title('Filtered image with mask H1');
filt_image= conv2(double(myimage),H2);
subplot(3,2,6); imshow(filt_image,[]); title('Filtered image with mask H1');

figure; subplot(2,2,1); imshow(myimage); title('Original Image');
fspecial() is used to create mask

gaussmask = fspecial('gaussian',3);
filtimg = imfilter(myimage,gaussmask);
subplot(2,2,2); imshow(filtimg,[]),title('Output of Gaussian filter 3 X 3');

avgfilt = [ 1 1 1 1 1 1 1;
        1 1 1 1 1 1 1;
        1 1 1 1 1 1 1;
        1 1 1 1 1 1 1;
```

```
1 1 1 1 1 1 1;
1 1 1 1 1 1 1;
1 1 1 1 1 1 1];

avgfiltmask = avgfilt/sum(avgfilt);
convimage= conv2(double(myimage),double(avgfiltmask));

subplot(2,2,3); imshow(convimage,[]); title('Average filter with conv2()');

filt_image= conv2(double(myimage),H3);
subplot(3,2,6); imshow(filt_image,[]); title('Filtered image with mask H3');
```

Result:

Original Image

Filtered image with mask L1

Filtered image with mask L2

Filtered image with mask L3

Filtered image with mask H1

Filtered image with mask H1

Original Image

Output of Gaussian filter 3 X 3

Average filter with conv2()

Filtered image with mask H3

Conclusion:

MATLAB program spatial filtering is executed.