RAJALAKSHMI ENGINEERING COLLEGE
An AUTONOMOUS Institution
Affiliated to ANNA UNIVERSITY, Chennai

DEPARTMENT
OF
COMPUTER SCIENCE AND DESIGN

CD19P02

FUNDAMENTALS
OF
IMAGE PROCESSING

LABORATORY RECORD

CD19P02 - FUNDAMENTALS OF IMAGE PROCESSING

List of Experiments	
1.	Practice of important image processing commands – imread(), imwrite(), imshow(), plot() etc.
2.	Program to perform Arithmetic and logical operations
3.	Program to implement sets operations, local averaging using neighborhood processing.
4.	Program to implement Convolution operation.
5.	Program to implement Histogram Equalization.
6.	Program to implement Mean Filter.
7.	Program to implement Order Statistic Filters
8.	Program to remove various types of noise in an image
9.	Program to implement Sobel operator.

INDEX

INTRODUCTION TO MATLAB

MATLAB stands for MATrix LABoratory and the software is built up around vectors and matrices. It is a technical computing environment for high performance numeric computation and visualization. It integrates numerical analysis, matrix computation, signal processing and graphics in an easy-to-use environment, where problems and solutions are expressed just as they are written mathematically, without traditional programming. MATLAB is an interactive system whose basic data element is a matrix that does not require dimensioning. It enables us to solve many numerical problems in a fraction of the time that it would take to write a program and execute in a language such as FORTRAN, BASIC, or C. It also features a family of application specific solutions, called toolboxes. Areas in which toolboxes are available include signal processing, image processing, control systems design, dynamic systems simulation, systems identification, neural networks, wavelength communication and others. It can handle linear, non- linear, continuous-time, discretetime, multivariable and multirate systems.

How to start MATLAB

Choose the submenu "Programs" from the "Start" menu. From the "Programs" menu, open the "MATLAB" submenu. From the "MATLAB" submenu, choose "MATLAB".

Procedure :

1. Open Matlab.
2. File New Script.
3. Type the program in untitled window
4. File Save type filename.m in Matlab workspace path.
5. Debug Run.
6. Output will be displayed at Figure dialog box.

Library Functions

clc:

Clear command window
Clears the command window and homes the cursor.

clear all:

Removes all variables from the workspace.

close all:

Closes all the open figure windows.

exp:

Y = exp(X) returns the exponential e x for each element in array X.

imread	Read image from graphics file
imwrite	Write image to graphics file
imfinfo	Information about graphics file
imshow	Display Image
Implay	Play movies, videos or image sequences
gray2ind	Convert grayscale to indexed image
ind2gray	Convert indexed image to grayscale image
mat2gray	Convert matrix to grayscale image
rgb2gray	Convert RGB image or colormap to grayscale
imbinarize	Binarize image by thresholding
adapthresh	Adaptive image threshold using local firstorder statistics
otsuthresh	Global histogram threshold using Otsu's method
im2uint16	Convert image to 16-bit unsigned integers
im2uint8	Convert image to 8-bit unsigned integers
imcrop	Crop image
imresize	Resize image
imrotate	Rotate image
imadjust	Adjust image intensity values or colormap
imcontrast	Adjust Contrast tool
imsharpen	Sharpen image using unsharp masking
histeq	Enhance contrast using histogram equalization
adapthisteq	Contrast-limited adaptive histogram equalization (CLAHE)
imhistmatch	Adjust histogram of image to match N-bin histogram of reference image
imnoise	Add noise to image
imfilter	N-D filtering of multidimensional images
fspecial	Create predefined 2-D filter
weiner2	2-D adaptive noise-removal filtering
medfilt2	2-D median filtering
ordfilt2	2-D order-statistic filtering
imfill	Fill image regions and holes
imclose	Morphologically close image
imdilate	Dilate image
imerode	Erode image
imopen	Morphologically open image
imreconstruct	Morphological reconstruction
watershed	Watershed transform
dct2	2-D discrete cosine transform

hough	Hough transform
graydist	Gray-weighted distance transform of grayscale image

linespace:

y = linspace(x1,x2) returns a row vector of 100 evenly spaced points between x1 and x2.

rand:

X = rand returns a single uniformly distributed random number in the interval (0,1).

ones:

X = ones(n) returns an n-by-n matrix of ones.

zeros:

X = zeros(n) returns an n-by-n matrix of zeros.

plot:

plot(X,Y) creates a 2-D line plot of the data in Y versus the corresponding values in X.

subplot:

subplot(m,n,p) divides the current figure into an m-by-n grid and creates an axes for a subplot in the position specified by p.

stem:

stem(Y) plots the data sequence, Y, as stems that extend from a baseline along the x-axis. The data values are indicated by circles terminating each stem.

title:

title(str) adds the title consisting of a string, str, at the top and in the center of the current axes.

xlabel:

xlabel(str) labels the x-axis of the current axes with the text specified by str.

ylabel:

ylabel(str) labels the y-axis of the current axes with the string, str.

A Summary of Matlab Commands Used

imread	Read image from graphics file
imwrite	Write image to graphics file
imfinfo	Information about graphics file
imshow	Display Image
Implay	Play movies, videos or image sequences
gray2ind	Convert grayscale to indexed image
ind2gray	Convert indexed image to grayscale image
mat2gray	Convert matrix to grayscale image
rgb2gray	Convert RGB image or colormap to grayscale
imbinarize	Binarize image by thresholding
adapthresh	Adaptive image threshold using local firstorder statistics
otsuthresh	Global histogram threshold using Otsu's method
im2uint16	Convert image to 16-bit unsigned integers
im2uint8	Convert image to 8-bit unsigned integers
imcrop	Crop image
imresize	Resize image
imrotate	Rotate image
imadjust	Adjust image intensity values or colormap
imcontrast	Adjust Contrast tool
imsharpen	Sharpen image using unsharp masking
histeq	Enhance contrast using histogram equalization
adapthisteq	Contrast-limited adaptive histogram equalization (CLAHE)
imhistmatch	Adjust histogram of image to match N-bin histogram of reference image
imnoise	Add noise to image
imfilter	N-D filtering of multidimensional images
fspecial	Create predefined 2-D filter
weiner2	2-D adaptive noise-removal filtering
medfilt2	2-D median filtering
ordfilt2	2-D order-statistic filtering
imfill	Fill image regions and holes
imclose	Morphologically close image
imdilate	Dilate image
imerode	Erode image
imopen	Morphologically open image
imreconstruct	Morphological reconstruction

watershed	Watershed transform
dct2	2-D discrete cosine transform
hough	Hough transform
graydist	Gray-weighted distance transform of grayscale image

Ex.No:1 Date:

IMPLEMENTATION OF IMAGE PROCESSING COMMANDS

Aim:

To Perform important image processing commands using Matlab.

Software Used:

MATLAB

Theory:

Basic Image Processing with MATLAB:

MATLAB is a very simple software for coding. All data variable in MATLAB are thought a matrix and matrix operations are used for analyzing them. MATLAB has the different toolboxes according to application areas. In this section, MATLAB Image Processing Toolbox is presented and the use of its basic functions for digital image is explained.

Read, write, show image and plot:

imread()

It is the function is used for reading image. If we run this function with requiring data, image is converted to a two-dimensional matrix (gray image is two-dimensional, but, color image is three-dimensional) with rows and columns including gray value in the each cell.

I = imread(‘path/filename.fileextension');

imread() function only needs an image file. If the result of imread() function is equal to a variable, a matrix variable (I) is created. File name, extension, and directory path that contains image must be written between two single quotes. If script and image file are in the same folder,path is not necessary.

imshow()

The matrix variable of image is showed using imshow() function. If many images show with sequence on the different figure windows, we use “figure” function for opening new window.

imwrite()

It is the function is used to create an image. This function only requires a new image file name with extension. If the new image is saved to a specific directory, the path of directory is necessary.

Subplot

Subplot divides the current figure into rectangular panes that are numbered rowwise. Each pane contains an axes object which you can manipulate using Axes Properties. Subsequent plots are output to the current pane. h = subplot(m,n,p) or subplot(mnp) breaks the figure window into an m-by-n matrix of small axes, selects the pth axes object for the current plot, and returns the axes handle. The axes are counted along the top row of the figure window, then the second row, etc.

impixelinfo

The function impixelinfo creates a Pixel Information tool in the current figure. The Pixel Information tool
displays information about the pixel in an image that the pointer is positioned over. The tool can display pixel information for all the images in a figure.

Imageinfo

The function imageinfo creates an Image Information tool associated with the image in the current figure. The tool displays information about the basic attributes of the target image in a separate figure. title – The function title('string') outputs the string at the top and in the center of the current axes.

Program:

To read and show the image

```
clear
close all
clc
I = imread('a.png');
imshow(I);
```

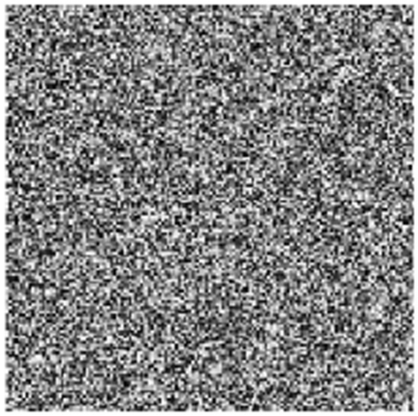
output :

Program :

```
clc;
clear all;
close all;
```

```
A = rand(150);
imwrite(A,'tn.png');
imshow('my.png')
```

Output :



Program :

```
clc;
clear all;
close all;
load clown.mat
newmap = copper(81);
imwrite(X,newmap,'copperclown.png');
imshow('copperclown.png);
```

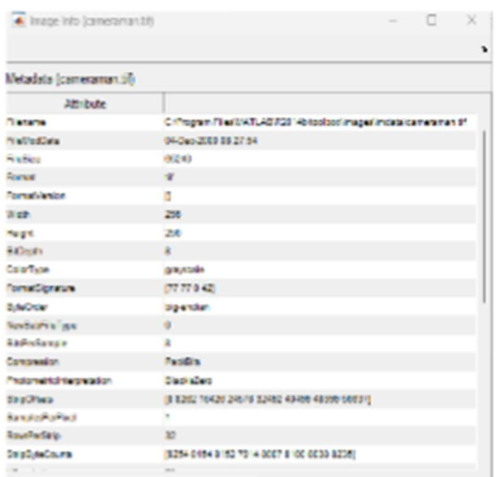
Output :

Program :

```
Clc;
Clear all;
```

```
Close all;
Subplot(2,2,1), imshow(‘a.png’),title(‘a.png’);
Subplot(2,2,2), imshow(‘b.png’),title(‘b.png’);
Subplot(2,2,3), imshow(‘c.png’),title(‘c.png’);
Subplot(2,2,4), imshow(‘d.png’),title(‘d.png’);
impixelinfo;
imageinfo(‘a.png’)
imageinfo(‘b.png’)
imageinfo(‘c.png’)
imageinfo(‘d.png’)
```

Output :



RESULT :

The important image commands have been displayed and studied.

Ex.No:2a Date:

IMPLEMENTATION OF ARTHIMETIC OPERATIONS

Aim:

To Perform arithmetic operations using Matlab.

Software Used:

MATLAB

Theory:

Imadd

Add two images or add constant to image

Syntax:

Z = imadd(X,Y)

Description:

Z = imadd(X,Y) adds each element in array X with the corresponding element in array Y and returns the sum
in the corresponding element of the output array Z. X and Y are real, nonsparse numeric arrays with the same size and class, or Y is a scalar double. Z has the same size and class as X, unless X is logical, in which case Z is double.
If X and Y are integer arrays, elements in the output that exceed the range of the integer type are truncated, and fractional values are rounded.

Example

Add two uint8 arrays. Note the truncation that occurs when the values exceed 255.
X = uint8([255 0 75; 44 225 100]);
Y = uint8([50 50 50; 50 50 50]);
Z = imadd(X,Y)
Z =
255 50 125
94 255 150

Imsubtract

Subtract one image from another or subtract constant from image

Syntax

Z = imsubtract(X,Y)

Description

Z = imsubtract(X,Y) subtracts each element in array Y from the corresponding element in array X and returns
the difference in the corresponding element of the output array Z. X and Y are real, nonsparse numeric arrays
of the same size and class, or Y is a double scalar. The array returned, Z, has the same size and class as X unless X is logical, in which case Z is double.
If X is an integer array, elements of the output that exceed the range of the integer type are truncated, and fractional values are rounded.

Example

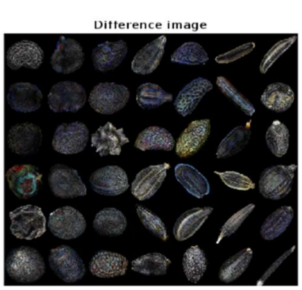
Subtract two uint8 arrays. Note that negative results are rounded to 0.

```
X = uint8([ 255 10 75; 44 225 100]);
Y = uint8([ 50 50 50; 50 50 50 ]);
Z = imsubtract(X,Y)
Z =
205 0 25
0 175 50
```

Program :

```
close all;
clear;
I = imread('gp.png');
background = imopen(I, strel('disk',15));
Ip = imsubtract(I, background);
imshow(Ip, []), title('Difference Image');
Iq = imsubtract(I,50);
figure
subplot(1,2,1), imshow(I), title('Original Image');
subplot(1,2,2), imshow(Iq), title('Subtracted Image');
```

Output :



immultiply

Multiply two images or multiply image by constant

Syntax

Z = immultiply(X,Y)

Description

Z = immultiply(X,Y) multiplies each element in array X by the corresponding element in array Y and returns the

product in the corresponding element of the output array Z.

If X and Y are real numeric arrays with the same size and class, then Z has the same size and class as X. If X is a numeric array and Y is a scalar double, then Z has the same size and class as X. If X is logical and Y is numeric, then Z has the same size and class as Y. If X is numeric and Y is logical, then Z has the same size and class as X.

immultiply

computes each element of Z individually in double-precision floating point. If X is an integer array,

then elements of Z exceeding the range of the integer type are truncated, and fractional values are rounded. If X and Y are numeric arrays of the same size and class, you can use the expression X.*Y instead of immultiply.

Example

%Scale an image by a constant factor:

I = imread('moon.tif');

J = immultiply(I,0.5);

subplot(1,2,1), imshow(I)

subplot(1,2,2), imshow(J)

imdivide

Divide one image into another or divide image by constant

Syntax

Z = imdivide(X,Y)

Description

Z = imdivide(X,Y) divides each element in the array X by the corresponding element in array Y and returns the result in the corresponding element of the output array Z. X and Y are real, nonsparse numeric arrays with the same size and class, or Y can be a scalar double. Z has the same size and class as X and Y, unless X is logical, in which case Z is double. If X is an integer array, elements in the output that exceed the range of integer type are truncated, and fractional values are rounded. If X and Y are numeric arrays of the same size and class, you can use the expression X./Y instead of imdivide.

Example

```
%Divide two uint8 arrays. Note that fractional values greater than or equal to 0.5 are rounded up to the nearest integer.
X = uint8([ 255 10 75; 44 225 100]);
Y = uint8([ 50 20 50; 50 50 50 ]);
Z = imdivide(X,Y)
Z =
5 1 2
1 5 2
%Estimate and divide out the background of the rice image.
I = imread(&#39;rice.png&#39;);
background = imopen(I,strel(&#39;disk&#39;,15));
Ip = imdivide(I,background);
imshow(Ip,[])
```

program :

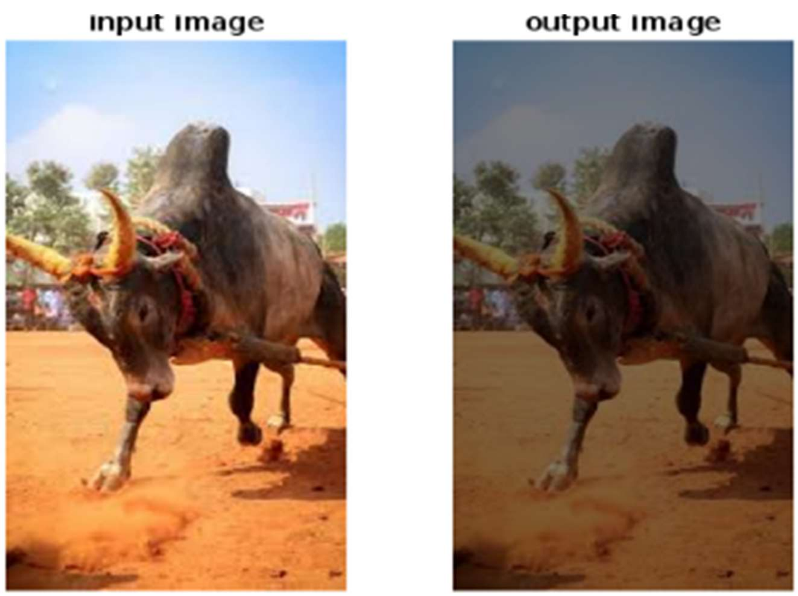
```
clc;
close all;
clear all;
I = imread('tn.jpg');
I16 = uint16(I);
J = immultiply(I16,I16);
imshow(I), title('input image'), figure, imshow(J), title('multiplied image');
```

Output :

Program :

```
clc;
clear all;
close all;
I = imread('jaa.jpg');
J = imdivide(I,2);
subplot(1,2,1), imshow(I), title('input image');
subplot(1,2,2), imshow(J), title('output image');
```

Output :



RESULT :

Thus , the Implementation of Arthimetic Operation was done and studied.

Ex.No:2b Date:

IMPLEMENTATION OF LOGICAL OPERATIONS

Aim:

To implement logical operations of an image using Matlab.

Software Used:

MATLAB

Theory:

Logical operations apply only to binary images, whereas arithmetic operations apply to multi-valued pixels. Logical operations are basic tools in binary image processing, where they are used for tasks such as masking, feature detection, and shape analysis. Logical operations on entire image are performed pixel by pixel. Because the AND operation of two binary variables is 1 only when both variables are 1, the result at any location in a resulting AND image is 1 only if the corresponding pixels in the two input images are 1. As logical operation involve only one pixel location at a time, they can be done in place, as in the case of arithmetic operations. The XOR (exclusive OR) operation yields a 1 when one or other pixel (but not both) is 1, and it yields a 0 otherwise. The operation is unlike the OR operation, which is 1, when one or the other pixel is 1, or both pixels are 1. Logical AND & OR operations are useful for the masking and compositing of images. For example, if we compute the AND of a binary image with some other image, then pixels for which the corresponding value in the binary image is 1 will be preserved, but pixels for which the corresponding binary value is 0 will be set to 0 (erased) . Thus the binary image acts as a mask that removes information from certain parts of the image. On the other hand, if we compute the OR of a binary image with some other image , the pixels for which the corresponding value in the binary image is will be preserved, but pixels for which the corresponding binary value is 1, will be set to 1 (cleared).

Logical AND:

Syntax:

c = a & b;

Logical And is commonly used for detecting differences in images, highlighting target regions with a binarymask or producing bit-planes through an image.

Logical OR:

Syntax:

C = a | b;

It is useful for processing binary-valued images (0 or 1) to detect objects which have moved between frames.

Binary objects are typically produced through application of thresholding to a grey-scale image.

Logical NOT:

Syntax:

B = ~A

This inverts the image representation. In the simplest case of a binary image, the (black) background pixels become (white) and vice versa.

Logical X OR:

Syntax:

C = xor (a,b) ;

It is useful for processing binary-valued images (0 or 1) to detect objects which have moved between frames. Binary objects are typically produced through application of thresholding to a grey-scale image.

Program:

To perform OR operation in an image

```
imageSize = [200, 200];
i = zeros(imageSize);
rowStart = 50;
rowEnd = 150;
colStart = 80;
colEnd = 120;
i(rowStart:rowEnd, colStart:colEnd) = 1;
imageSize = [200, 200];
j = ones(imageSize);
resultImage = i | j;
subplot(1, 3, 1), imshow(i), title("Image 1");
subplot(1, 3, 2), imshow(j), title("image 2');
subplot(1, 3, 3), imshow(resultImage), title('Output Image');
```

Output :

Program:

To perform AND operation in an image

```
imageSize = [200, 200];
i = zeros(imageSize);
rowStart = 50;
rowEnd = 150;
colStart = 80;
colEnd = 120;
i(rowStart:rowEnd, colStart:colEnd) = 1;
imageSize = [200, 200];
j = ones(imageSize);
resultImage = i &amp; j;
subplot(1, 3, 1), imshow(i), title(&#39;Image 1&#39;);
subplot(1, 3, 2), imshow(j), title(&#39;Image 2&#39;);
subplot(1, 3, 3), imshow(resultImage), title(&#39;Output Image&#39;);
```

Output:

Program:

To perform NOT operation in an image

```
imageSize = [200, 200];
i = zeros(imageSize);
rowStart = 50;
rowEnd = 150;
colStart = 80;
colEnd = 120;
i(rowStart:rowEnd, colStart:colEnd) = 1;
```

```
resultImage = ~i ;
subplot(2, 2, 1), imshow(i), title(&#39;Input Image &#39;);
subplot(2, 2, 2), imshow(resultImage), title(&#39;Output Image&#39;);
```

Output :

Program:

To perform XOR operation in an image

```
imageSize = [200, 200];
i = zeros(imageSize);
rowStart = 50;
rowEnd = 150;
colStart = 80;
colEnd = 120;
i(rowStart:rowEnd, colStart:colEnd) = 1;
imageSize = [200, 200];
j = ones(imageSize);
resultImage = xor(i,j);
subplot(1, 3, 1), imshow(i), title(&#39;Image 1&#39;);
subplot(1, 3, 2), imshow(j), title(&#39;Image 2&#39;);
subplot(1, 3, 3), imshow(resultImage), title(&#39;Output Image&#39;);
```

Output:

Result:

Thus, the logical operations of an image have been implemented using MATLAB.

Ex.No:3a Date:

IMPLEMENTATION OF SET OPERATIONS

Aim:

To implement Set operations of an image using Matlab.

Software Used:

MATLAB

Theory:

Set operations in MATLAB refer to various mathematical operations performed on the pixel values of two or more images. These operations allow you to combine or manipulate the pixel values to achieve different effects. Here's an overview of some common set operations in MATLAB image processing.

Union:

Syntax:
unionImage = max(image A, image B);
The union of two images is obtained by taking the maximum pixel value at each corresponding pixel position from the input images. This operation can be used for merging images or enhancing certain features.

Interssection:

Syntax:
intersectionImage = min(image A, image B);
The intersection of two images is obtained by taking the minimum pixel value at each corresponding pixel position from the input images. This operation highlights common features between the images.

Complement:

Syntax:
ComplementImage = 255 – image;
The complement of an image is obtained by subtracting each pixel value from the maximum pixel value (often 255 for 8-bit images). This operation results in an image with inverted pixel values.

Difference:

Syntax:
differenceimage = abs (image A – image B) ;

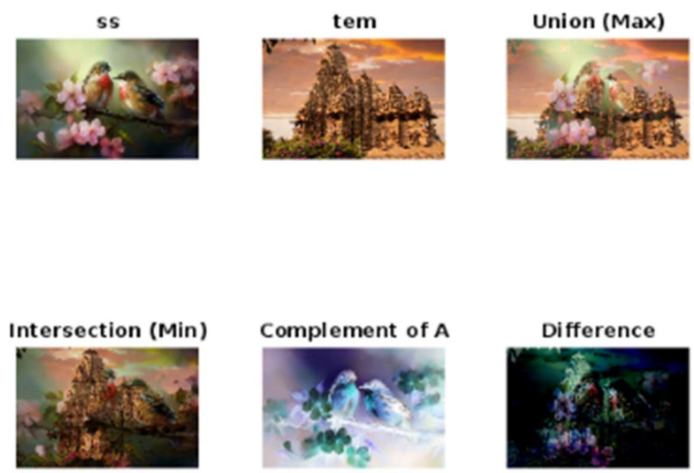
The difference between two images is obtained by taking the absolute difference between their pixel values. This operation can be used for highlighting dissimilarities between images.

Program:

To perform Set operation’s in an image

```
A = imread(&#39;Image A.jpeg&#39;);
imageB = imread(&#39;Image B.jpeg&#39;);
imageA = imresize(A, [225,225]);
if ~isequal(size(imageA), size(imageB))
error(&#39;Input images must have the same dimensions.&#39;);
end
unionImage = max(imageA, imageB);
intersectionImage = min(imageA, imageB);
complementImageA = 255 - imageA;
differenceImage = abs(imageA - imageB);
subplot(2, 3, 1);
imshow(imageA);
title(&#39;Image A&#39;);
subplot(2, 3, 2);
imshow(imageB);
title(&#39;Image B&#39;);
subplot(2, 3, 3);
imshow(unionImage);
title(&#39;Union (Max)&#39;);
subplot(2, 3, 4);
imshow(intersectionImage);
title(&#39;Intersection (Min)&#39;);
subplot(2, 3, 5);
imshow(complementImageA);
title(&#39;Complement of A&#39;);
subplot(2, 3, 6);
imshow(differenceImage);
title(&#39;Difference&#39;)
imwrite(unionImage, &#39;union_image.jpg&#39;);
imwrite(intersectionImage, &#39;intersection_image.jpg&#39;);
imwrite(complementImageA, &#39;complement_imageA.jpg&#39;);
imwrite(differenceImage, &#39;difference_image.jpg&#39;);
disp(&#39;Set operation images saved.&#39;);
```

Output :



Result:

Thus, the set operations of an image have been implemented using MATLAB.

Ex.No:3b   Date:

IMPLEMENTATION OF LOCAL AVERAGING USING NEIGHBORHOOD PROCESSING

Aim:

To implement Local averaging operations of an image using Matlab.

Software Used:

MATLAB

Theory:

Local averaging using neighborhood processing is a fundamental technique in image processing. It involves smoothing or blurring an image by computing the average value of pixels in a local neighborhood around each pixel. The goal is to reduce noise and fine details in the image while preserving its overall structure. Here's the theory behind the process.

Neighborhood Selection:

In this technique, a fixed-size neighborhood (also known as a kernel or filter) is defined around each pixel in the image. This neighborhood is typically square or rectangular and can vary in size. Common neighborhood sizes are 3x3, 5x5, or 7x7, but the choice depends on the specific application and desired level of smoothing.

Kernel Creation:

A kernel is created with values that represent the weights assigned to each pixel within the neighborhood. For local averaging, all values in the kernel are typically set to 1, and the sum of the kernel values is often normalized to 1 by dividing each value by the total number of values in the kernel. This ensures that the operation doesn't change the overall brightness of the image.

Convolution Operation:

To perform local averaging, a convolution operation is applied to the image. Convolution is a mathematical operation that combines two functions to produce a third function. In image processing, the convolution operation combines the pixel values in the neighborhood with the corresponding values in the kernel. The result is a weighted sum of pixel values, which effectively represents the average value of the pixels in the neighborhood.

Pixel Replacement:

The new value for the pixel at the center of the neighborhood is computed based on the weighted sum, and it replaces the original pixel value. This process is repeated for every pixel in the image.

Smoothing Effect:

The convolution operation effectively smooths the image by averaging pixel values in local regions.Pixels with strong noise or high-frequency details are averaged with their neighbors, leading to ablurring effect that reduces the impact of noise and enhances the visibility of larger-scale features in the image.

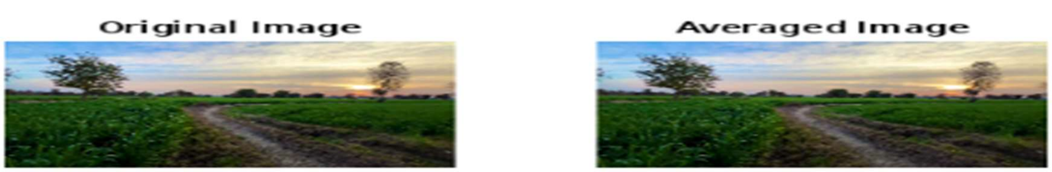
Adjustable Smoothing:

The degree of smoothing can be controlled by adjusting the size of the neighborhood and the values in the kernel. Larger neighborhoods or kernels with larger values will produce more significant smoothing, while smaller neighborhoods or kernels with smaller values will result in less smoothing. Local averaging using neighborhood processing is a simple yet powerful technique with a wide range of applications in image processing, such as noise reduction, edge-preserving smoothing, and feature extraction. It's a building block for more advanced filtering and processing techniques used in computer vision, image enhancement, and computer graphics.

Program:

```
inputImage = imread(&#39;image.jpg&#39;);

neighborhoodSize = 3;

filter = fspecial(&#39;average&#39;, neighborhoodSize);

averagedImage = imfilter(inputImage, filter);

subplot(1, 2, 1);

imshow(inputImage);

title(&#39;Original Image&#39;);

subplot(1, 2, 2);

imshow(averagedImage);

title(&#39;Averaged Image&#39;);

imwrite(averagedImage, &#39;averaged_image.jpg&#39;);

disp(&#39;Averaged image saved as &quot;averaged_image.jpg&quot;&#39;);
```

Output :



Result:

Thus, the local averaging using neighborhood processing of an image has been implemented using MATLAB.

Ex.No:4 Date:

IMPLEMENTATION OF CONVOLUTION OPERATIONS

Aim:

To implement Convolution operations of an image using Matlab.

Software Used:

MATLAB

Theory:

Convolution and correlation are the two fundamental mathematical operations involved in linear filters based on neighbourhood-oriented image processing algorithms.

Convolution :

Convolution processes an image by computing, for each pixel, a weighted sum of the values of that pixel and its neighbours. Depending on the choice of weights, a wide variety of image processing operations can be implemented. Different convolution masks produce different results when applied to the same input image. These operations are referred to as filtering operations and the masks as spatial filters. Spatial filters are often named based on their behaviour in the spatial frequency. Low-pass filters (LPFs) are those spatial filters whose effect on the output image is equivalent to attenuating the high-frequency components (fine details in the image) and preserving the low-frequency components (coarser details and homogeneous areas in the image). These filters are typically used to either blur an image or reduce the amount of noise present in the image. Linear low-pass filters can be implemented using 2D convolution masks with non-negative coefficients.

High-pass filters (HPFs) work in a complementary way to LPFs, that is, these preserve or enhance high-frequency components with the possible side-effect of enhancing noisy pixels as well. High-frequency components include fine details, points, lines and edges. In other words, these highlight transitions in intensity within the image. There are two in-built functions in MATLAB's Image Processing Toolbox (IPT) that can be used to implement 2D convolution: conv2 and filter2.

1. conv2 computes 2D convolution between two matrices. For example, C=conv2(A,B) computes the two-dimensional convolution of matrices A and B. If one of these matrices describes a two-dimensional finite impulse response (FIR) filter, the other matrix is filtered in two dimensions.

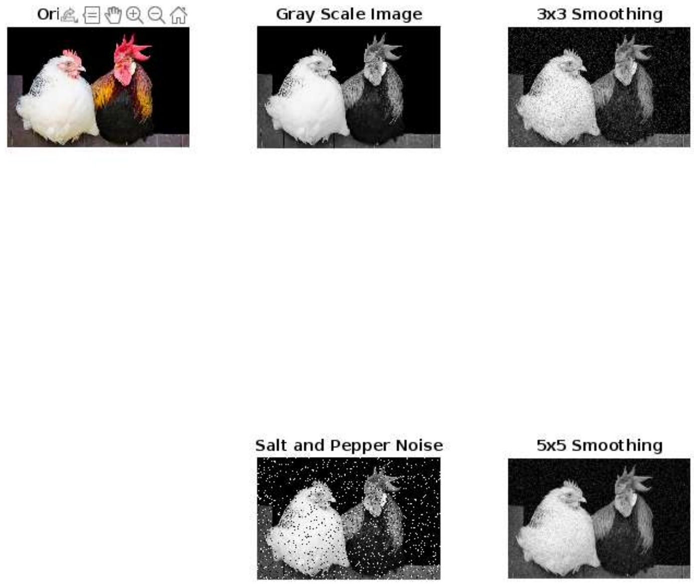
2. filter2 function rotates the convolution mask, that is, 2D FIR filter, by 180° in each direction to create a convolution kernel and then calls conv2 to perform the convolution operation.

Program :

```
clc;
clear all;
```

```
close all;
a=imread(&#39;pic1.jpeg&#39;);
subplot(2,4,1);
imshow(a);
title(&#39;Original Image&#39;);
b=rgb2gray(a);
subplot(2,4,2);
imshow(b);
title(&#39;Gray Scale Image&#39;);
c=imnoise(b,&#39;salt &amp; pepper&#39;,,0.1);
subplot(2,4,6);
imshow(c);
title(&#39;Salt and Pepper Noise&#39;);
h1=1/9*ones(3,3);
c1=conv2(c,h1,&#39;same&#39;);
subplot(2,4,3);
imshow(uint8(c1));
title(&#39;3x3 Smoothing&#39;);
h2=1/25*ones(5,5);
c2=conv2(c,h2,&#39;same&#39;);
subplot(2,4,7);
imshow(uint8(c2));
title(&#39;5x5 Smoothing&#39;);
```

Output :



Result:

Thus, the convolution operations of an image have been implemented using MATLAB.

Ex.No:5 Date:

IMPLEMENTATION OF HISTOGRAM EQUALIZATION

Aim:

To implement Histogram Equalization operations of an image using Matlab.

Software Used:

MATLAB

Theory:

Histogram of an image is a plot of number of occurrences of gray level in the image against the gray level value. For dark image, histogram is concentrated in the lower (dark) side of the gray scale. For bright image, histogram is concentrated on higher side of the gray scale. Equalization is a process that attempts to spread out the gray levels in an image so that they are evenly distributed across the range.

Histogram Processing:

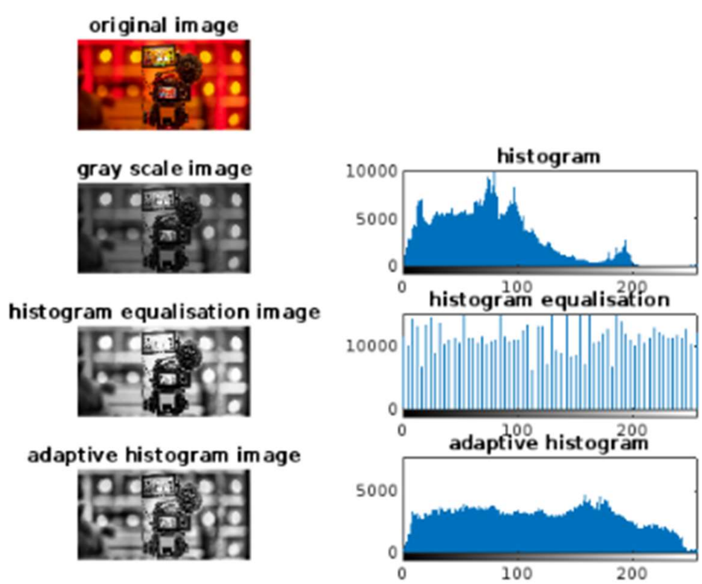
The contrast of an image can be modified by manipulating its histogram. A popular method is via Histogram equalization. Here, the given histogram is manipulated such that the distribution of pixel values is evenly spread over the entire range 0 to K-1. Histogram equalization can be done at a global or local level. In the global level the histogram of the entire image is processed whereas at the local level, the given image is subdivided and the histograms of the subdivisions (or sub images) are manipulated individually. When histogram equalization is applied locally, the procedure is called Adaptive Histogram Equalization.

Program :

```
clc;
clear all;
close all;
a= imread(&#39;peppers.png&#39;);
subplot(4,2,1);
imshow(a);
title(&#39;original image&#39;);
b=rgb2gray(a);
subplot(4,2,3);
```

```
imshow(b);
title(&#39;gray scale image&#39;);
subplot(4,2,4);
imhist(b);
title(&#39;histogram&#39;);
subplot(4,2,5);
c=histeq(b);
imshow(c);
title(&#39;histogram equalisation image&#39;);
subplot(4,2,6);
imhist(c);
title(&#39;histogram equalisation&#39;);
subplot(4,2,7);
f=adapthisteq(b);
imshow(f);
title(&#39;adaptive histogram image&#39;);
subplot(4,2,8);
imhist(f);
title(&#39;adaptive histogram&#39;);
```

Output :



Result :

Thus, the Histogram equalization of an image have been implemented using MATLAB.

Ex.No:6 Date:

IMPLEMENTATION OF MEAN FILTER

Aim:

To implement Mean filter operations of an image using Matlab.

Software Used:

MATLAB

Theory:

When an image is acquired by a web camera or other imaging system, normally the vision system for which it is intended is unable to use it directly. The image may be corrupted by random variations in intensity, variations in illumination, poor contrast or noise that must be handle with in the early stages of vision processing. Therefore, mean filter is one of the techniques which is used to reduce noise of the images. This is a local averaging operation and it is a one of the simplest linear filter. The value of each pixel is replaced by the average of all the values in the local neighborhood.

Let f(i,j) is a noisy image then the smoothed image g(x,y) can be obtained by,

$$g(x,y)= 1/n \sum_{(i,j) \in S} f(i,j)$$

Where S is a neighborhood of (x,y) and n is the number of pixels in S.

Porgram :

```
clc;
close all;
clear all;
inputImage = imread(&#39;cameraman.tif&#39;);
filterSize = 5; % Define the filter size (e.g., 3x3, 5x5, etc.)
paddedImage = padarray(inputImage, [filterSize, filterSize], &#39;replicate&#39;);
outputImage = zeros(size(inputImage));
for i = 1:size(inputImage, 1)
```

```
for j = 1:size(inputImage, 2)
neighborhood = paddedImage(i:i+filterSize-1, j:j+filterSize-1);
meanValue = mean(neighborhood(:));
outputImage(i, j) = meanValue;
end
subplot(1, 2, 1);
imshow(inputImage);
title(&#39;Original Image&#39;);
subplot(1, 2, 2);
imshow(uint8(outputImage));
title(&#39;Mean Filtered Image&#39;);
```

Output :

Result:

The noise in an image is reduced using a mean filter, and it has been implemented using MATLAB.

Ex.No:7 Date:

IMPLEMENTATION OF ORDER STATISTICS FILTERS

Aim:

To implement Order Statistics operations of an image using Matlab.

Software Used:

MATLAB

Theory:

Order statistic filters are non-linear spatial filters whose response is based on the ordering(ranking) of the pixels contained in the image area encompassed by the filter, and then replacing the value in the center pixel with the value determined by the ranking result. The different types of order statistics filters include Median Filtering, Max and Min filtering and Mid-point filtering.

Median Filtering:

The median filter selects the middle value when the neighborhood values are sorted, making it effective at noise reduction and preserving edges.

$$K = (N+1)/2$$

Replaces the value of a pixel by the median of the pixel values in the neighborhood of that pixel.

Maximum Filtering:

The maximum filter selects the maximum value from the neighborhood, which enhances bright features and suppresses dark features. (K=N) The maximum filtering is achieved using the following equation

$$f(x,y) = \max g(s,t)$$

Minimum Filtering:

This filter selects the minimum value from the neighborhood, effectively enhancing dark features and suppressing bright features. (K=1) The minimum filtering is achieved using the following equation

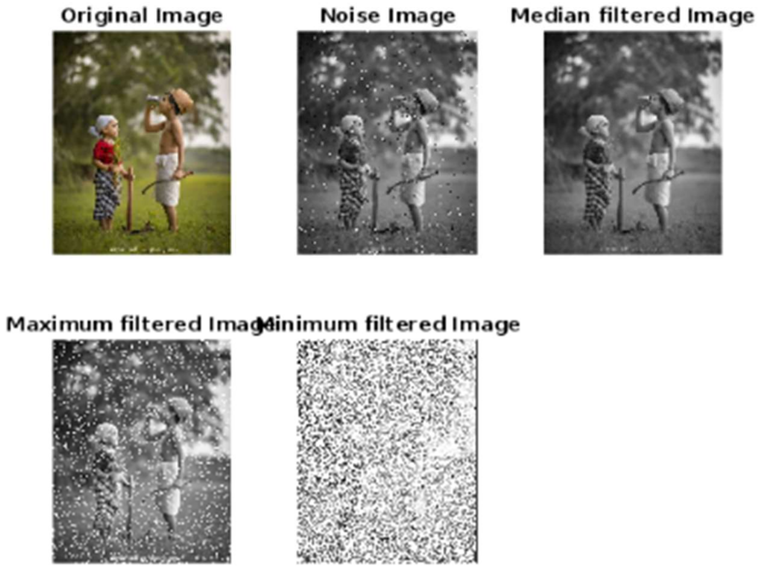
$$f(x,y) = \min g(s,t)$$

Program:

To perform order Statistics Filters in an image

```
clc;
clear all;
close all;
b = imread(&#39;C:\Users\indhu\Downloads\peppers.jpg&#39;);
subplot(2,3,1);
imshow(b);
title(&#39;Original Image&#39;);
a=rgb2gray(b);
a = im2double(a);
a = imnoise(a,&#39;salt &amp; pepper&#39;,0.02);
subplot(2,3,2);
imshow(a);
title(&#39;Noise Image&#39;);
I = medfilt2(a);
subplot(2,3,3);
imshow(I);
title(&#39;Median filtered Image&#39;);
x=rand(size(a));
a(x(:)&lt; 0.05)=0;
max_Img = ordfilt2(a,9,ones(3,3));
subplot(2,3,4);
imshow(max_Img);
title(&#39;Maximum filtered Image&#39;);
a(x(:)&lt; 0.95)=255;
min_Img = ordfilt2(a,1,ones(3,3));
subplot(2,3,5);
imshow(min_Img);
title(&#39;Minimum filtered Image&#39;);
```

Output:



Result :

The different Order Statistics filters in an image have been implemented using MATLAB

. Ex.No:8 Date:

REMOVE VARIOUS TYPES OF NOISE IN AN IMAGE

Aim:

To implement remove various Types of noise operations of an image using Matlab.

Software Used:

MATLAB

Theory:

Image noise is the random variation of brightness or color information in images produced by the sensor and circuitry of a scanner or digital camera. Image noise can also originate in film grain and in the unavoidable shot noise of an ideal photon detector .Image noise is generally regarded as an undesirable by-product of image capture. Although these unwanted fluctuations became known as "noise" by analogy with unwanted sound they are inaudible and such as dithering.

The types of Noise are following.

1. Salt and Pepper Noise
2. Gaussian Noise
3. Rayleigh Noise
4. Erlang Noise
5. Exponential Noise
6. Uniform Noise

Salt and Pepper Noise:

An image containing salt-and-pepper noise will have dark pixels in bright regions and bright pixels in dark regions. This type of noise can be caused by dead pixels, analog-to-digital converter errors, bite rrors in transmission, etc. This can be eliminated in large part by using dark frame subtraction and by interpolating around dark/bright pixels.

Gaussian Noise:

The standard model of amplifier noise is additive, Gaussian, independent at each pixel and independent of the signal intensity. In color cameras where more amplification is used in the blue color channel than in the green or red channel, there can be more noise in the blue channel. Amplifier noise is a major part of the "read noise & quot; of an image sensor, that is, of the constant noise level in dark areas of the image.

Rayleigh Noise:

Rayleigh noise is characterized by a Rayleigh probability distribution. This distribution is commonly used to model the amplitude of a signal that has passed through a random medium, resulting in attenuation and phase shifts. Rayleigh noise is characterized by an intensity distribution, similar to the Rayleigh distribution in signal processing. The distribution describes the probability of various pixel intensity values in the presence of noise.

Erlang Noise:

Erlang noise, also known as the Erlang distribution, is a statistical model used to describe the behavior of certain types of noise or random processes. In image processing, Erlang noise is not as commonly encountered as other noise models like Gaussian or Rayleigh noise. It is a continuous probability distribution that is often used to model the sum of independent exponential random variables. It is also known as the gamma distribution when the shape parameter is an integer. In image processing, Erlang noise can be used to model variations in pixel intensities, especially when the image acquisition process involves cumulative effects. This is different from many other noise models that assume each pixel is independently affected.

Program :

Rayleigh Noise:

```
clc;
close all;
clear all;
RGB = imread(&#39;saturn.png&#39;);
I = im2gray(RGB);
J = imnoise(I,&#39;gaussian&#39;,0,0.025);
K = wiener2(J,[5 5]);
subplot(2,3,1);
imshow(I)
title(&#39;Original Image&#39;);
subplot(2,3,2);
imshow(J)
title(&#39;Added Gaussian Noise&#39;);
subplot(2,3,3);
imshow(K);
title(&#39; Wiener Filtered Image&#39;);
```

Output:

Salt and Pepper Noise:

```
clc;
clear all;
close all;
I = imread(&#39;eight.tif&#39;);
J = imnoise(I,&#39;salt &amp; pepper&#39;,0.02);
subplot(2,3,1);
imshow(I)
title(&#39;Original Image&#39;);
subplot(2,3,2)
imshow(J)
title(&#39;Noisy Image&#39;);
Kmedian = medfilt2(J);
subplot(2,3,3);
imshow(Kmedian);
title(&#39;Noise removed Image&#39;);
```

Output:

Gaussian Noise:

```
clc;
close all;
clear all;
RGB = imread(&#39;saturn.png&#39;);
I = im2gray(RGB);
J = imnoise(I,&#39;gaussian&#39;,,0,0.025);
K = wiener2(J,[5 5]);
subplot(2,3,1);
imshow(I)
title(&#39;Original Image&#39;);
subplot(2,3,2);
imshow(J)
title(&#39;Added Gaussian Noise&#39;);
subplot(2,3,3);
imshow(K);
title(&#39; Wiener Filtered Image&#39;);
```

Output:

d.Erlang Noise:

```
clc;
close all;
clear all;
I = imread(&#39;eight.tif&#39;);
scale = 10;
```

```
shape= 5;
sizeSignal = size(I);
erlangNoise = scale*gamrnd(shape, 1, sizeSignal);
noisy = double(I) + erlangNoise;
noisy = min(max(noisy, 0), 255);
noisy = uint8(noisy);
denoised=medfilt2(noisy);
figure;
subplot(2, 3, 1);
imshow(I);
title(&#39;Input Image&#39;);
subplot(2, 3, 2);
imshow(noisy);
title(&#39;Noisy Image&#39;);
subplot(2, 3, 3);
imshow(denoised);
title(&#39;Denoised Image&#39;);
```

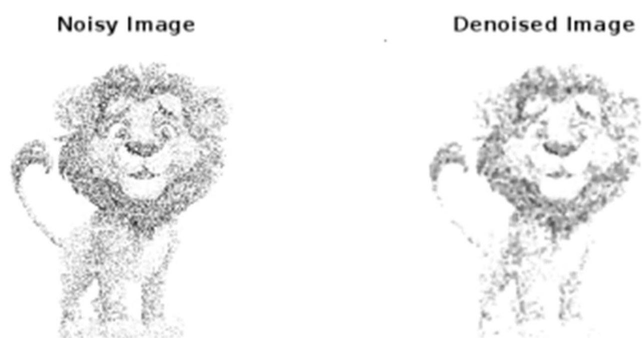
Output:

e.Uniform Noise:

```
I = imread(&#39;eight.tif&#39;);
minValue = 0;
maxValue = 255;
sizeImage = size(I);
```

```
uniformNoise = (maxValue - minValue) * rand(sizeImage) + minValue;
noisy = double(I) + uniformNoise;
noisy = min(max(noisy, 0), 255);
noisy = uint8(noisy);
denoised=medfilt2(noisy);
figure;
subplot(1, 2, 1);
imshow(noisy);
title(&#39;Noisy Image&#39;);
subplot(1, 2, 2);
imshow(denoised);
title(&#39;Denoised Image&#39;);
```

Output:



Result :

Thus, the various types of noise in an image have been removed and implemented using MATLAB.

Ex.No:9 Date:

IMPLEMENTATION OF SOBEL OPERATOR

Aim:

To implement Sobel operations of an image using Matlab.

Software Used:

MATLAB

Theory:

The Sobel operator is a fundamental tool in image processing for edge detection and gradient estimation. It is used to find edges or boundaries in images by measuring the rate of change of intensity at each pixel. The theory behind the Sobel operator involves convolution with a pair of kernels to compute the gradients in both the horizontal and vertical directions. Here is a detailed explanation of the theory behind the Sobel operator.

Gradient Calculation :

The Sobel operator is designed to compute the gradient of an image. The gradient represents the rate of change of pixel intensities, which is essential for identifying edges or abrupt changes in an image.

Convolution Operation :

The core operation of the Sobel operator involves convolution. Convolution is a mathematical operation that combines two functions to produce a third. In image processing, it is used to apply a kernel or filter to an image.

Sobel Kernels :

The Sobel operator uses two 3x3 convolution kernels, one for detecting changes in the horizontaldirection (Sobel-X) and the other for changes in the vertical direction (Sobel-Y).

Sobel-X Kernel:

-1 0 1 2 0 2 -1 0 1

Sobel-X Kernel:

-1 -2 -1 0 0 0 1 2 1

Gradient Computation :

To calculate the gradient at a given pixel, the Sobel operator convolves the image with both the Sobel- X and Sobel-Y kernels separately. The result of these two convolutions provides the horizontal gradient (Gx) and the vertical gradient (Gy) at each pixel.Edge Detection The Sobel operator highlights edges by emphasizing areas where the gradient magnitude (G) is high. A high gradient magnitude indicates a rapid change in pixel intensities, which is characteristic of edges or boundaries.

Thresholding :

To extract significant edges, a threshold can be applied to the gradient magnitude. Pixels with a gradient magnitude above a certain threshold are considered part of an edge, while pixels with lower magnitudes are often treated as non-edge pixels.

Noise Sensitivity :

The Sobel operator is sensitive to noise, as noise can create small variations that may be mistaken for edges. Preprocessing steps, such as Gaussian smoothing, are sometimes applied to reduce noise before applying the operator.

Applications :

The Sobel operator is widely used in image processing and computer vision tasks, including object detection, feature extraction, image segmentation.

Program :

```
a = imread(&#39;peppers.png&#39;);
b = rgb2gray(a);
gray_img = double(b);
h_kernel = [-1, 0, 1; -2, 0, 2; -1, 0, 1];
v_kernel = [-1, -2, -1; 0, 0, 0; 1, 2, 1];
c = imfilter(gray_img, h_kernel);
d = imfilter(gray_img, v_kernel);
gradient_magnitude = sqrt(c.^2 + d.^2);
figure;
subplot(2, 2, 1);
imshow(a);
title(&#39;Original Image&#39;);
subplot(2, 2, 2);
imshow(uint8(gradient_magnitude));
title(&#39;Sobel Edge Detected Image&#39;);
```

Output :

Result :

The SOBEL operator in digital images for edge detection has been implemented using MATLAB.

PROJECT

IMPLEMENTATION OF EDGE DETECTION IN IMAGES

Aim:

To implement Edge Detection of an image using Matlab.

Software Used:

VS CODE
language : **Python**

Theory:

Edge detection is an image processing technique that identifies the edges in an image by looking for rapid changes in intensity. Edges are often associated with object boundaries in a scene. Edge detection is a fundamental image processing technique for identifying and locating the boundaries or edges of objects in an image. It is used to identify and detect the discontinuities in the image intensity and extract the outlines of objects present in an image.

How it works

Edge detection algorithms use a high-pass filter to measure the rate of change at each pixel. The filter output is then thresholded to determine which pixels represent edges.

Why it's important

Edge detection is a fundamental tool in image processing, computer vision, and machine vision. It's used for image segmentation and data extraction.

How to detect edges

Points on an edge can be detected by finding the local maxima or minima of the first derivative, or by finding the zero-crossing of the second derivative.

.

There are various types of edge detection techniques, which include the following:

- Sobel Edge Detection
- Canny Edge Detection
- Laplacian Edge Detection
- Prewitt Edge Detection
- Roberts Cross Edge Detection

- Scharr edge detection

The goal of edge detection algorithms is to identify the most significant edges within an image or scene. These detected edges should then be connected to form meaningful lines and boundaries, resulting in a segmented image that contains two or more distinct regions. The segmented results are subsequently used in various stages of a machine vision system for tasks such as object counting, measuring, feature extraction, and classification.

Edge Detection Approaches

There are several approaches to edge detection. Let's talk about the most common approaches one by one.

Sobel Edge Detection

Sobel edge detection is a popular technique used in image processing and computer vision for detecting edges in an image. It is a gradient-based method that uses convolution operations with specific kernels to calculate the gradient magnitude and direction at each pixel in the image. Here's a detailed explanation of Sobel edge detection.

The Sobel operator uses two 3x3 convolution kernels (filters), one for detecting changes in the x-direction (horizontal edges) and one for detecting changes in the y-direction (vertical edges). These kernels are used to compute the gradient of the image intensity at each point, which helps in detecting the edges. Here are the Sobel kernels:

Horizontal Kernel (Gx):

This kernel is used to detect horizontal edges by emphasizing the gradient in the x-direction.

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

The Gx kernel emphasizes changes in intensity in the horizontal direction. The positive values (+1 and +2) on the right side will highlight bright areas, while the negative values (-1 and -2) on the left side will highlight dark areas, effectively detecting horizontal edges.

Vertical Kernel (Gy):

This kernel is used to detect vertical edges by emphasizing the gradient in the y-direction.

$$G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

The Gy kernel emphasizes changes in intensity in the vertical direction. Similarly, the positive values (+1 and +2) at the bottom will highlight bright areas, while the negative values (-1 and -2) at the top will highlight dark areas, effectively detecting vertical edges.

Let's walk through an example of Sobel edge detection using Python and the OpenCV library. Here's the Step-by-Step Example:

1. *Load and Display the Image*: First, we need to load a sample image and display it to understand what we're working with.
2. *Convert to Grayscale*: Convert the image to grayscale as the Sobel operator works on single-channel images.
3. *Apply Gaussian Smoothing (Optional)*: Apply a Gaussian blur to reduce noise and make edge detection more robust.
4. *Apply Sobel Operator*: Use the Sobel operator to calculate the gradients in the x and y directions.
5. *Calculate Gradient Magnitude*: Compute the gradient magnitude from the gradients in the x and y directions. A threshold is applied to the gradient magnitude image to classify pixels as edges or non-edges. Pixels with gradient magnitude above the threshold are considered edges.
6. *Normalization*: The gradient magnitude and individual gradients are normalized to the range 0-255 for better visualization.
7. *Display the Resulting Edge Image*: Normalize and display the edge-detected image.

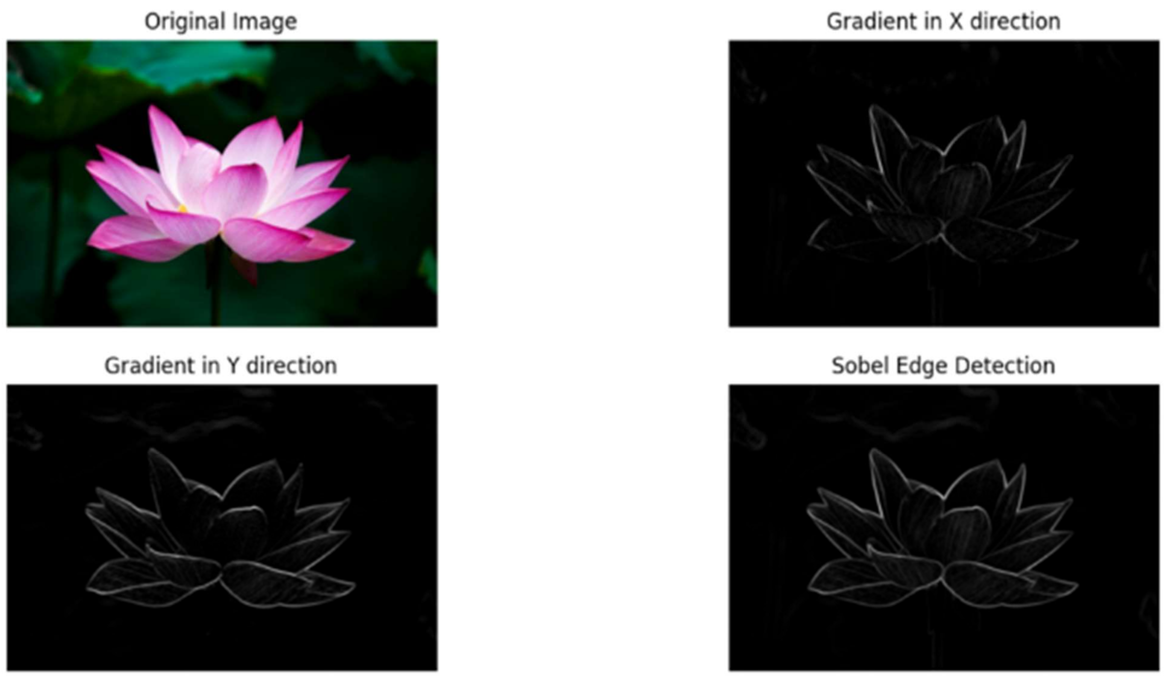
Here, in the following code for sobel operator cv2.CV_64F specifies the desired depth of the output image. Using a higher depth helps in capturing precise gradient values, especially when dealing with small or fine details. For Gx the values (1, 0) means taking the first derivative in the x-direction and zero derivative in the y-direction. For Gy the values (0, 1) means taking the first derivative in the y-direction and zero derivative in the x-direction. ksize=3 specifies the size of the extended 3x3 Sobel kernel

PROGRAM :

```
import cv2

import numpy as np

import matplotlib.pyplot as plt

image_path = 'flower.jpg'  # Replace with your image path

image = cv2.imread(image_path)

gray_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
```

```
# Apply Gaussian smoothing (optional)
blurred_image = cv2.GaussianBlur(gray_image, (3, 3), 0)
Gx = cv2.Sobel(blurred_image, cv2.CV_64F, 1, 0, ksize=3)
Gy = cv2.Sobel(blurred_image, cv2.CV_64F, 0, 1, ksize=3)
G = np.sqrt(Gx**2 + Gy**2)
Gx = np.uint8(255 * np.abs(Gx) / np.max(Gx))
Gy = np.uint8(255 * np.abs(Gy) / np.max(Gy))
G = np.uint8(255 * G / np.max(G))
plt.figure(figsize=(15, 10))
plt.subplot(2, 2, 1)
plt.imshow(cv2.cvtColor(image, cv2.COLOR_BGR2RGB))
plt.title('Original Image')
plt.axis('off')
plt.subplot(2, 2, 2)
plt.imshow(Gx, cmap='gray')
plt.title('Gradient in X direction')
plt.axis('off')
plt.subplot(2, 2, 3)
plt.imshow(Gy, cmap='gray')
plt.title('Gradient in Y direction')
plt.axis('off')
plt.subplot(2, 2, 4)
plt.imshow(G, cmap='gray')
plt.title('Sobel Edge Detection')
plt.axis('off')
plt.show()
```

OUTPUT :



Canny Edge Detection

Canny Edge Detection is a multi-stage algorithm to detect a wide range of edges in images. It was developed by John F. Canny in 1986 and is known for its optimal edge detection capabilities. The algorithm follows a series of steps to reduce noise, detect edges, and improve the accuracy of edge detection.

Following are the steps of steps of Canny Edge Detection:

1. ***Noise Reduction using Gaussian Blurring*:**

The first step in the Canny edge detection algorithm is to smooth the image using a Gaussian filter. This helps in reducing noise and unwanted details in the image. The Gaussian filter is applied to the image to convolve it with a Gaussian kernel. The Gaussian kernel (or Gaussian function) is defined as:

$$G(x, y) = \frac{1}{2\pi\sigma^2} \exp\left(-\frac{x^2 + y^2}{2\sigma^2}\right)$$

This step helps to remove high-frequency noise, which can cause spurious edge detection.

2. ***Gradient Calculation*:**

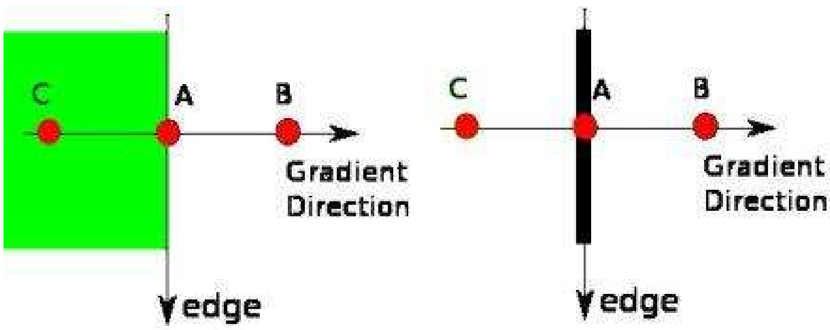
After noise reduction, the Sobel operator is used to calculate the gradient intensity and direction of the image. This involves calculating the intensity gradients in the x and y directions (Gx and Gy). The gradient magnitude and direction are then computed using these gradients.

$$Edge\_Gradient\ (G) = \sqrt{G_x^2 + G_y^2}$$

$$Angle\ (\theta) = \tan^{-1}\left(\frac{G_y}{G_x}\right)$$

3. ***Non-Maximum Suppression*:**

To thin out the edges and get rid of spurious responses to edge detection, non-maximum suppression is applied. This step retains only the local maxima in the gradient direction. The idea is to traverse the gradient image and suppress any pixel value that is not considered to be an edge, i.e., any pixel that is not a local maximum along the gradient direction.



In the above image, point A is located on the edge in the vertical direction. The gradient direction is perpendicular to the edge. Points B and C lie along the gradient direction. Therefore, Point A is compared with Points B and C to determine if it represents a local maximum. If it does, Point A proceeds to the next stage; otherwise, it is suppressed and set to zero.

4. ***Double Thresholding*:**

After non-maximum suppression, the edge pixels are marked using double thresholding. This step classifies the edges into strong, weak, and non-edges based on two thresholds: high and low. Strong edges are those pixels with gradient values above the high threshold, while weak edges are those with gradient values between the low and high thresholds.

Given the gradient magnitude M and two thresholds T_{high} and T_{low}, the classification can be mathematically expressed as:

Strong Edges:

$$E_{\text{strong}}(i,j) = \begin{cases} 1 & \text{if } M(i,j) \geq T_{\text{high}} \\ 0 & \text{otherwise} \end{cases}$$

Weak Edges:

$$E_{\text{weak}}(i,j) = \begin{cases} 1 & \text{if } T_{\text{low}} \le M(i,j) < T_{\text{high}} \\ 0 & \text{otherwise} \end{cases}$$

Non-Edges:

$$E_{\text{non-edge}}(i,j) = \begin{cases} 1 & \text{if } M(i,j) < T_{\text{low}} \\ 0 & \text{otherwise} \end{cases}$$

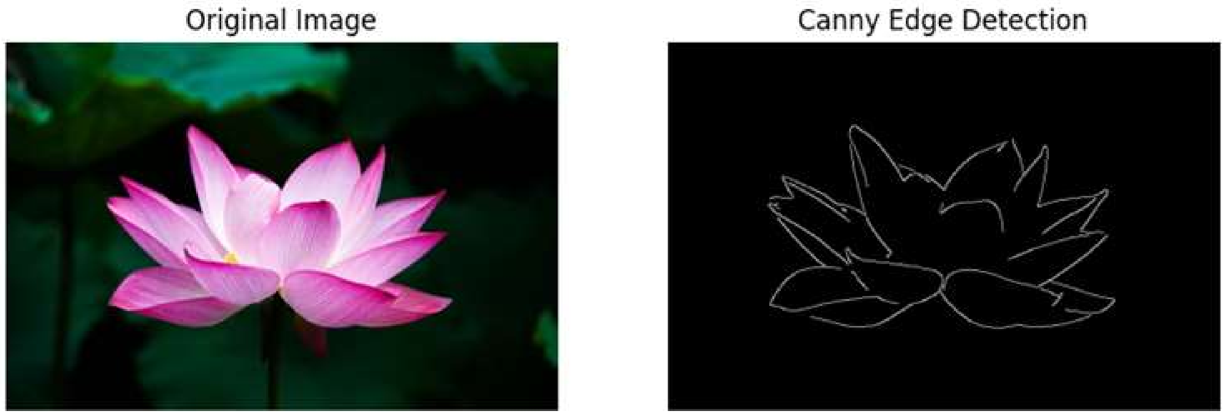
5. ***Edge Tracking by Hysteresis*:**

The final step is edge tracking by hysteresis, which involves traversing the image to determine which weak edges are connected to strong edges. Only the weak edges connected to strong edges are retained, as they are considered true edges. This step ensures that noise and small variations are ignored, resulting in cleaner edge detection.

PROGRAM :

```
import cv2

import numpy as np

import matplotlib.pyplot as plt

image_path = 'flower.jpg'  # Replace with your image path

image = cv2.imread(image_path, cv2.IMREAD_COLOR)

gray_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

blurred_image = cv2.GaussianBlur(gray_image, (5, 5), 1.4)

edges = cv2.Canny(blurred_image, 100, 200)

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

plt.subplot(1, 2, 1)

plt.imshow(cv2.cvtColor(image, cv2.COLOR_BGR2RGB))

plt.title('Original Image')

plt.axis('off')

plt.subplot(1, 2, 2)

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

plt.title('Canny Edge Detection')

plt.axis('off')
```

```
plt.show()
```



RESULT :

Thus, the implemention of Edge detection in images using Python was done successfully.