DEPARTMENT OF COMPUTER SCIENCE AND DESIGN

CD19P02 – FUNDAMENTALS OF IMAGE PROCESSING

LABORATORY RECORD

SUBMITED BY : 221701026

Junide Chris (csd 3rd year)

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, discrete time, 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.
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 first-order 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
fft2	2-D fast Fourier transform
ifftshift	Inverse FFT shift
imcomplement	Complement image

immultiply	Multiply two images or multiply image by constant
imsubtract	Subtract one image from another or subtract constant from image
imdivide	Divide one image into another or divide image by constant
imadd	Add two images or add constant to image

Ex.No:1   **IMPLEMENTATION OF IMAGE PROCESSING COMMANDS**

Date:

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.
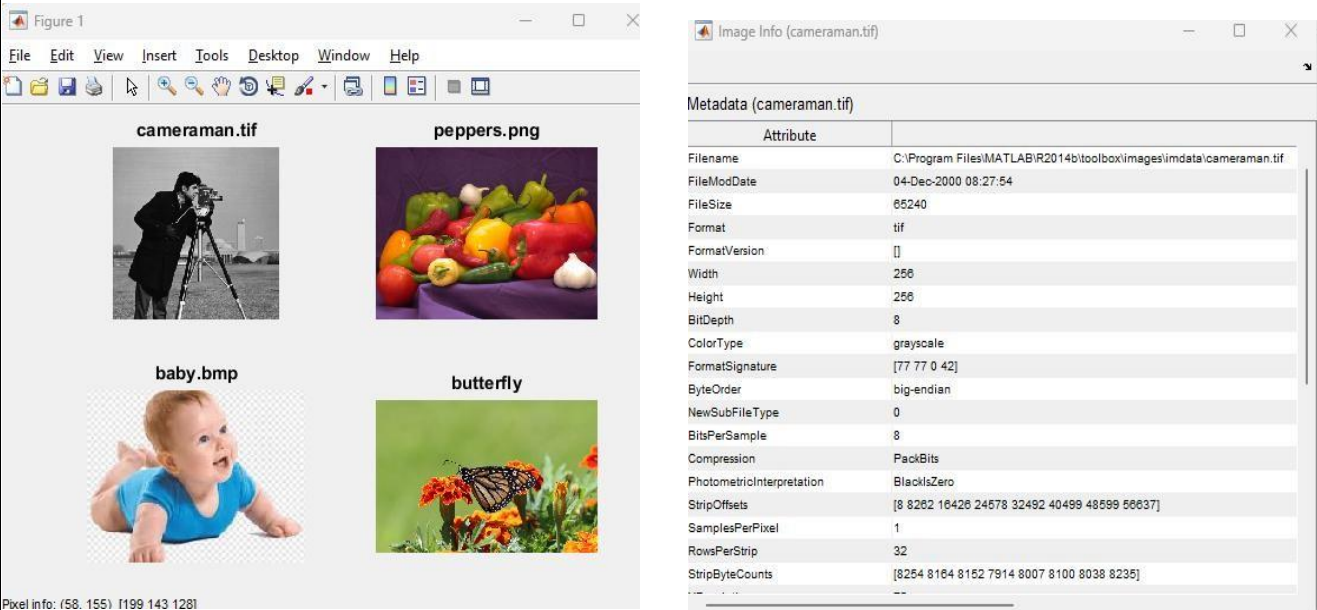
impixelinfo

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.

Output:

Figure 1 window displaying: cameraman.tif, peppers.png, baby.bmp, butterfly

Pixel info: (58, 155) [199 143 128]

Image Info (cameraman.tif)

Metadata (cameraman.tif)

Attribute	
Filename	C:\Program Files\MATLAB\R2014b\toolbox\images\imdata\cameraman.tif
FileModDate	04-Dec-2000 08:27:54
FileSize	65240
Format	tif
FormatVersion	[]
Width	256
Height	256
BitDepth	8
ColorType	grayscale
FormatSignature	[77 77 0 42]
ByteOrder	big-endian
NewSubFileType	0
BitsPerSample	8
Compression	PackBits
PhotometricInterpretation	BlackIsZero
StripOffsets	[8 8262 16426 24578 32492 40499 48599 56637]
SamplesPerPixel	1
RowsPerStrip	32
StripByteCounts	[8254 8164 8152 7914 8007 8100 8038 8235]

1.1) Program

```
clear
close all
clc
I=imread('golden.jpeg');
imshow(I);
```

Output:

1.1) Program

```
clear
close all
clc
I=imread('golden.jpeg');
imshow(I);
```

1.2) Program

```
clc;
clear all;
close all;
subplot(2,2,1), imshow('coffee.jpeg'),title('Coffee');
subplot(2,2,2), imshow('flower.jpg'),title('Flower');
subplot(2,2,3), imshow('Lightning (1).jpg'),title('Lightning');
subplot(2,2,4), imshow('giraffee.png'),title('Giraffee');
impixelinfo;
imageinfo('coffee.jpeg');
imageinfo('flower.jpg');
imageinfo('Lightning (1).jpg');
imageinfo('giraffee.png');
```

Output:

golden

tt   beetle

Pixel info: (250, 205) [131 138 146]

Image Info (q8.jpg)

Metadata (q8.jpg)

Attribute	Value
Filename	/MATLAB Drive/q8.jpg
FileModDate	28-Oct-2024 14:04:13
FileSize	189984
Format	jpg
FormatVersion	"
Width	1024
Height	692
BitDepth	24
ColorType	truecolor
FormatSignature	"
NumberOfSamples	3
CodingMethod	Huffman
CodingProcess	Sequential
Comment	0
AutoOrientedWidth	1024
AutoOrientedHeight	692
YCbCrPositioning	Centered
AutoOrientedWidth	1000
AutoOrientedHeight	1000

20:56

Metadata (golden.jpg)

Attribute	Value
Filename	/MATLAB Drive/golden.jpg
FileModDate	31-Jul-2024 03:18:21
FileSize	219035
Format	jpg
FormatVersion	"
Width	1000
Height	1000
BitDepth	24
ColorType	truecolor
FormatSignature	"
NumberOfSamples	3
CodingMethod	Huffman
CodingProcess	Sequential
Comment	{}
XMPData	[1x1 struct]
XResolution	300
YResolution	300
ResolutionUnit	Inch
YCbCrPositioning	Centered
AutoOrientedWidth	1000

1.3) Program

```
clc;
clear all;
close all;
A = rand(150);
imwrite(A,'myGray.png');
imshow('mygray.png')
```

Output:
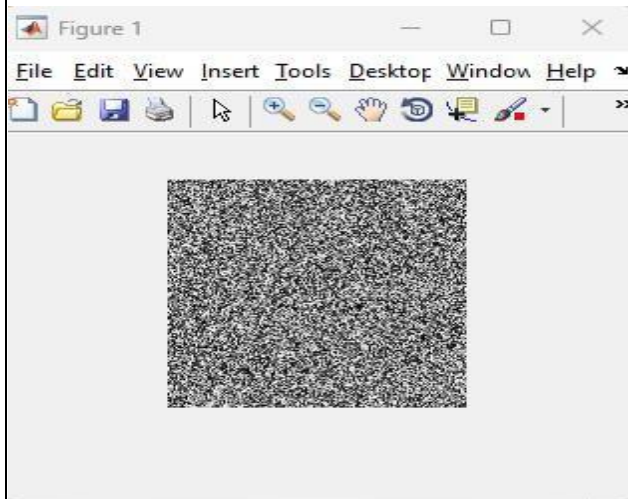
1.4) Program

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

Output:
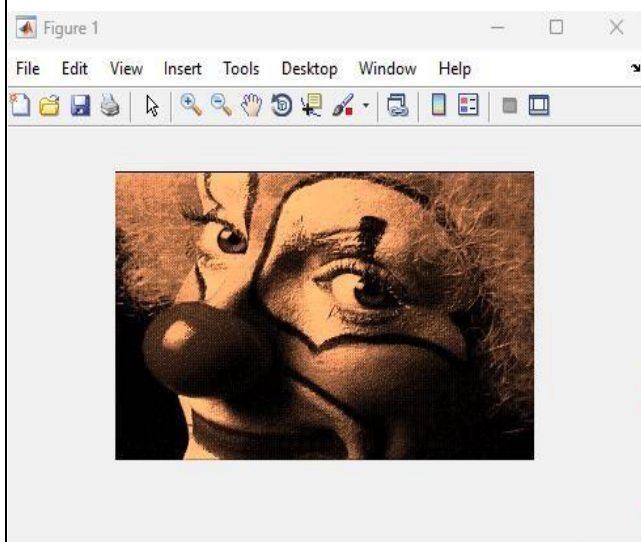
Result:

The important image commands have been displayed and studied.

Ex.No:2a	IMPLEMENTATION OF ARITHMETIC OPERATIONS

Date:

Aim:

To implement arithmetic operations of an image 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
```

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('rice.png');
background = imopen(I,strel('disk',15));
Ip = imdivide(I,background);
imshow(Ip,[])

2.1) Program

```
close all;
clear;
I = imread('flower (2) (1).jpg');
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:

2.2) Program:

```
clc;
close all;
clear all;
I = imread('flower (2) (1).jpg');
I16 = uint16(I);
J = immultiply(I16,I16);
subplot(1,2,1), imshow(I), title('Original Image');
subplot(1,2,2), imshow(J), title('Multiplied Image');
```

Output:

Result:

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

Ex.No:2b	IMPLEMENTATION OF LOGICAL OPERATIONS

Date:

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 0 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 binary mask 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

```
img1 = imread('giraffee.png');
img2 = imread('sunset.png');
grayImg1 = im2gray(img1);
grayImg2 = im2gray(img2);
binaryImg1 = imbinarize(grayImg1);
binaryImg2 = imbinarize(grayImg2);
binaryImg2 = imresize(binaryImg2, size(binaryImg1));
resultImage = binaryImg1 | binaryImg2;
subplot(2, 3, 1), imshow("giraffee.png"), title('Image 1');
subplot(2, 3, 2), imshow("sunset.png"), title('Image 2');
subplot(2, 3, 3), imshow(binaryImg1), title('Binary Image 1');
subplot(2, 3, 4), imshow(binaryImg2), title('Binary Image 2');
subplot(2, 3, 5), imshow(resultImage), title('Output Image');
```

Output:
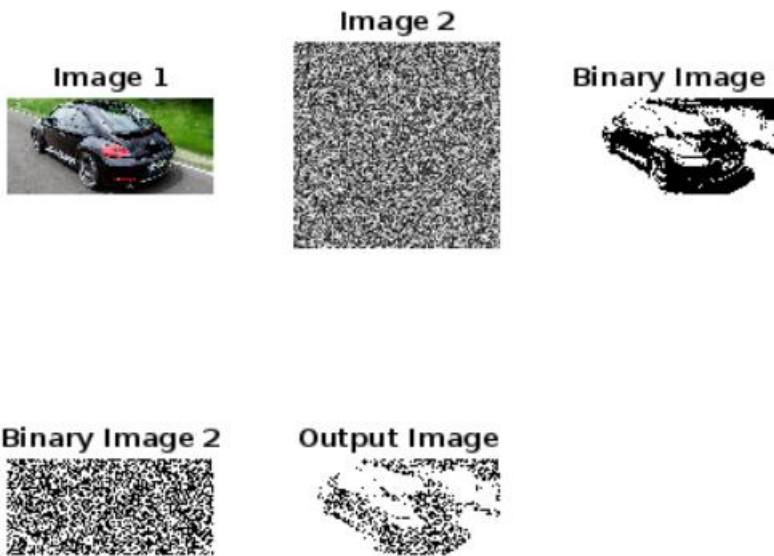
Program:- To perform NOT operation in an image

```
inputImage = imread('giraffee.png');
binaryImage = imbinarize(im2gray(inputImage));
resultImage = ~binaryImage;
subplot(1, 3, 1), imshow("giraffee.png"), title('Input Image');
subplot(1, 3, 2), imshow(binaryImage),title('Binary Image');
subplot(1, 3, 3), imshow(resultImage), title('Output Image');
```

Output:

Program:- To perform XOR operation in an image

```
img1 = imread('giraffee.png');
img2 = imread('Lightning (1).jpg');
binaryImg1 = imbinarize(im2gray(img1));
binaryImg2 = imbinarize(im2gray(img2));
binaryImg2 = imresize(binaryImg2, size(binaryImg1));
resultImage = xor(binaryImg1, binaryImg2);
subplot(2, 3, 1), imshow("giraffee.png"), title('Image 1');
subplot(2, 3, 2), imshow("Lightning (1).jpg"), title('Image 2');
subplot(2, 3, 3), imshow(binaryImg1), title('Binary Image 1');
subplot(2, 3, 4), imshow(binaryImg2), title('Binary Image 2');
subplot(2, 3, 5), imshow(resultImage), title('XOR Result Image');
```

Output:
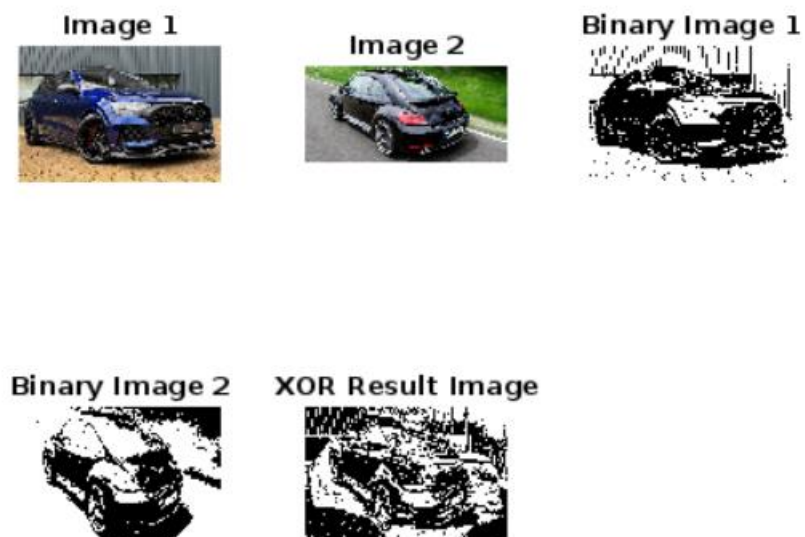
Result:

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

Ex.No:3a	IMPLEMENTATION OF SET OPERATIONS

Date:

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

```matlab
imageA = imread('image1-2 (1).jpg');
imageB = imread('Lightning (1).jpg');
if ~isequal(size(imageA), size(imageB))
    error('Input images must have the same dimensions.');
end
unionImage = max(imageA, imageB);
intersectionImage = min(imageA, imageB);
complementImageA = 255 - imageA;
differenceImage = abs(imageA - imageB);
subplot(2, 3, 1);imshow(uint8(imageA)); title('Image A');
subplot(2, 3, 2);imshow(uint8(imageB)); title('Image B');
subplot(2, 3, 3);imshow(unionImage);title('Union (Max)');
subplot(2, 3, 4);imshow(intersectionImage);title('Intersection (Min)');
subplot(2, 3, 5);imshow(complementImageA);title('Complement of A');
subplot(2, 3, 6);imshow(differenceImage);title('Difference');
imwrite(unionImage, 'union_image.jpg');
imwrite(intersectionImage, 'intersection_image.jpg');
imwrite(complementImageA, 'complement_imageA.jpg');
imwrite(differenceImage, 'difference_image.jpg');
disp('Set operation images saved.');
```

Output:
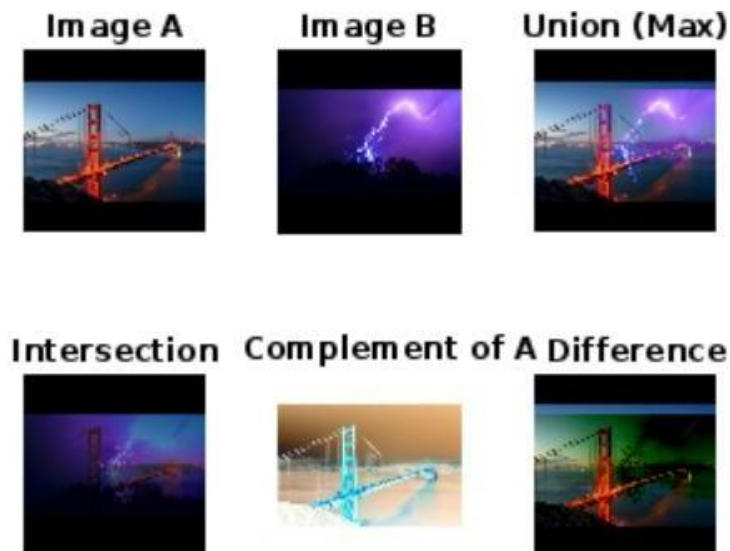
Result:

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

Ex.No:3b	IMPLEMENTATION OF LOCAL AVERAGING
	USING NEIGHBORHOOD PROCESSING

Date:

Aim:

To implement local averaging using neighborhood processing in 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 a blurring 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:- To perform local averaging using neighborhood processing in an image

```matlab
inputImage = imread('bridge.jpg');
neighborhoodSize = 3;
filter = fspecial('average', neighborhoodSize);
averagedImage = imfilter(inputImage, filter);
subplot(1, 2, 1);
imshow(inputImage);
title('Original Image');
subplot(1, 2, 2);
imshow(averagedImage);
title('Averaged Image');
imwrite(averagedImage, 'averaged_image.jpg');
disp('Averaged image saved as "averaged_image.jpg"');
```

Output:

Original Image   Averaged Image

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

.

Ex.No:4   **IMPLEMENTATION OF CONVOLUTION OPERATION**

Date:

Aim:

To implement Convolution operation 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:- To perform Convolution operation in an image

```matlab
clc;
clear all;
close all;
a = imread('Lightning (1).jpg');
subplot(2,4,1);
imshow(a);
title('Original Image');
b = rgb2gray(a);
subplot(2,4,2);
imshow(b);
title('Gray Scale Image');
c = imnoise(b, 'salt & pepper', 0.1);
subplot(2,4,3);
imshow(c);
title('Salt and Pepper Noise');
h1 = (1/9) * ones(3, 3);
c1 = conv2(double(c), h1, 'same');
subplot(2,4,4);
imshow(uint8(c1));
title('3x3 Smoothing');
h2 = (1/25) * ones(5, 5);
c2 = conv2(double(c), h2, 'same');
subplot(2,4,5);
imshow(uint8(c2));
title('5x5 Smoothing');
```

Output:

Result:

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

| Ex.No:5 | IMPLEMENTATION OF HISTOGRAM EQUALIZATION |

Date:

Aim:

To implement Histogram equalization 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 AdaptiveHistogramEqualization.

Program:- To perform Histogram Equalization in an image

```
clc;
clear;
close all;
a = imread('messi.jpg');
subplot(4,2,1);imshow(a);title('Original Image');
b = rgb2gray(a);
subplot(4,2,3);imshow(b);title('Grayscale Image');
subplot(4,2,4);imhist(b);title('Histogram');
c = histeq(b);
subplot(4,2,5);imshow(c);title('Histogram Equalization Image');
subplot(4,2,6);imhist(c);title('Histogram Equalization');
f = adapthisteq(b);
subplot(4,2,7);imshow(f);title('Adaptive Histogram Equalization Image');
subplot(4,2,8);imhist(f);title('Adaptive Histogram Equalization');
```

Output:
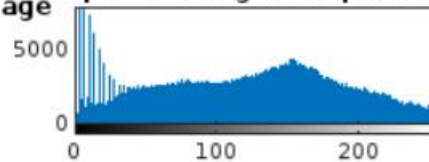
Program:- Correlation between the visual quality of an image with its histogram.

```
clc;
clear;
close all;
img = imread('messi.jpg');
img = rgb2gray(img);
[count, ~] = imhist(img);
Iheq = histeq(img);
[count1, ~] = imhist(Iheq);
corrbsameimg = corr2(img, Iheq);
disp(corrbsameimg);
x = xcorr(count,count);
x1 = xcorr(count, count1);
figure;
subplot(2, 1, 1);
plot(x);
title('Correlation Between Histograms of Original Image');
subplot(2, 1, 2);
plot(x1);
title('Correlation Between Histogram of Original and Equalized Image');
```

Output:
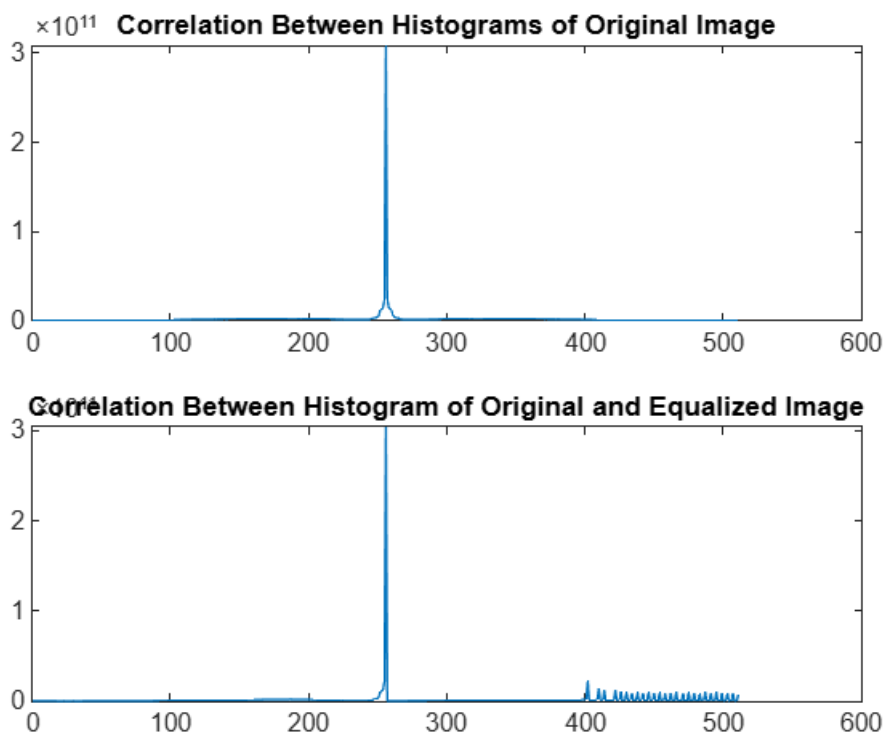
Result:

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

Ex.No:6	IMPLEMENTATION OF MEAN FILTER

Date:

Aim:

To implement mean filter in an image reduce noise in digital images 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,

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

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

Program:- To perform Mean Filter in an image

```
clc;
close all;
clear all;
inputImage = imread('messi.jpg');
inputImage = double(inputImage);
filterSize = 5;
[rows, cols, channels] = size(inputImage);
paddedImage = padarray(inputImage, [filterSize, filterSize], 'replicate');
outputImage = zeros(size(inputImage));
for c = 1:channels
    for i = 1:rows
        for j = 1:cols
            neighborhood = paddedImage(i:i+filterSize-1, j:j+filterSize-1,c);
            meanValue = mean(neighborhood(:));
            outputImage(i, j, c) = meanValue;
        end
    end
end

outputImage = uint8(outputImage);
subplot(1, 2, 1);
imshow(uint8(inputImage));
title('Original Image');
subplot(1, 2, 2);
imshow(outputImage);
title('Mean Filtered Image');
```

Output:

Result:

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

| Ex.No:7 | **IMPLEMENTATION OF ORDER STATISTICS FILTERS** |

Date:

Aim:

　　　　To implement Order Statistics filters in 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

Output:

```
clc;

clear all;

close all;

b = imread('messi.jpg');

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

a = rgb2gray(b);

a = im2double(a);

a_noisy = imnoise(a, 'salt & pepper', 0.02);

subplot(2,3,2);imshow(a_noisy);title('Noisy Image');

I = medfilt2(a_noisy);

subplot(2,3,3);imshow(I);title('Median Filtered Image');

max_Img = ordfilt2(a_noisy, 9, ones(3,3));

subplot(2,3,4);imshow(max_Img);title('Maximum Filtered Image');

min_Img = ordfilt2(a_noisy, 1, ones(3,3));

subplot(2,3,5);imshow(min_Img);title('Minimum Filtered Image');
```

Original Image

Mean Filtered Image

Result:

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

Ex.No:8	REMOVE VARIOUS TYPES OF NOISE IN AN IMAGE

Date:

Aim:

To Remove Various types of Noise in an Image 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.

● Salt and Pepper Noise
● Gaussian Noise
● Rayleigh Noise
● Erlang Noise
● Exponential Noise
● 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, bit errors 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" 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.

Exponential Noise:

Exponential noise, also known as exponential distribution, is a statistical model that describes random variations in pixel intensities in digital images. This type of noise can be encountered in image processing due to various factors, and it is important to understand and address it. Exponential noise is characterized by the exponential probability distribution. This distribution is often used to model the time between events in a Poisson process, but it can also describe random variations in Image intensities.

Uniform Noise:

Uniform noise, also known as uniform distribution, is a statistical model used to describe variations in pixel intensities in digital images. It is one of the simpler noise models and is often encountered in image processing due to various sources of noise. Uniform noise follows the uniform probability distribution, which is characterized by a constant probability density over a specified range of Values. In image processing, uniform noise can be used to model variations in pixel intensities that result from various factors, such as sensor noise, quantization errors, or other sources of interference during image acquisition.

Rayleigh Noise:

```
clc;
close all;
clear all;
RGB = imread('messi.jpg');
I = im2gray(RGB);
rayleighNoise = raylrnd(0.05, size(I));
J = im2double(I) + rayleighNoise;
K = wiener2(J, [5 5]);
subplot(2,3,1);
imshow(I)
title('Original Image');
subplot(2,3,2);
imshow(J)
title('Added Rayleigh Noise');
subplot(2,3,3);
imshow(K);
title('Wiener Filtered Image');
```

Output:

Salt and Pepper Noise:

```
clc;
clear all;
close all;
I = imread('messi.jpg');
I = rgb2gray(I);
J = imnoise(I, 'salt & pepper', 0.02);
subplot(2, 3, 1);
imshow(I);
title('Original Grayscale Image');
subplot(2, 3, 2);
imshow(J);
title('Noisy Image');
Kmedian = medfilt2(J, [3 3]);
subplot(2, 3, 3);
imshow(Kmedian);
title('Noise Removed Image');

Output:
```

d.Erlang noise:

```
clc;
close all;
clear;
I = imread('messi.jpg');
I = rgb2gray(I);
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('Input Image');
subplot(2, 3, 2);
imshow(noisy);
title('Noisy Image');
subplot(2, 3, 3);
imshow(denoised);
title('Denoised Image');
```

Output:
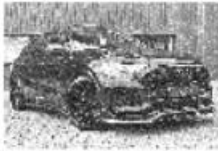
Gaussian noise:

```
clc;
clear all;
close all;
RGB = imread('messi.jpg');
I = rgb2gray(RGB);
J = imnoise(I, 'gaussian', 0, 0.025);
K = wiener2(J, [5 5]);
subplot(2, 3, 1);
imshow(I);
title('Original Image');
subplot(2, 3, 2);
imshow(J);
title('Added Gaussian Noise');
subplot(2, 3, 3);
imshow(K);
title('Wiener Filtered Image');
```
Output:

Exponential noise:

```
clc;
clear all;
close all;
I = imread('messi.jpg');
I = rgb2gray(I);
lambda = 0.1;
sizeSignal = size(I);
exponentialNoise = -log(1 - rand(sizeSignal)) / lambda;
noisy = double(I) + exponentialNoise;
noisy = min(max(noisy, 0), 255);
noisy = uint8(noisy);
denoised = medfilt2(noisy);
figure;
subplot(1, 3, 1);
imshow(I);
title('Original Image');
subplot(1, 3, 2);
imshow(noisy);
title('Noisy Image');
subplot(1, 3, 3);
imshow(denoised);
title('Denoised Image');
```

Output:

Uniform noise:

```
clc;
clear all;
close all;
I = imread('messi.jpg');
I = rgb2gray(I);
minValue = -50;
maxValue = 50;
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, 3, 1);
imshow(I);
title('Original Image');
subplot(1, 3, 2);
imshow(noisy);
title('Noisy Image');
subplot(1, 3, 3);
imshow(denoised);
title('Denoised Image');
```

Output:
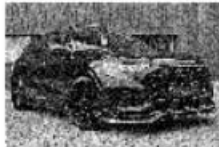
Result:

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

Ex. No:9	IMPLEMENTATION OF SOBEL OPERATOR

Date:

Aim:

To implement SOBEL operator in digital images for edge detection 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 horizontal direction (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:- To perform Sobel operator in an image**

```matlab
a = imread('messi.jpg');

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('Original Image');

subplot(2, 2, 2);

imshow(uint8(gradient_magnitude));

title('Sobel Edge Detected Image');
```

Output:

Original Image   Sobel Edge Detected Image
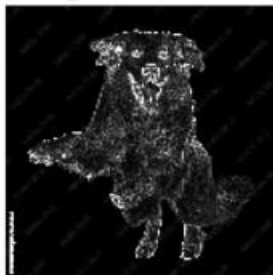
Result:

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

PROJECT

PANORAMA IMAGE STITCHING

Aim:

To implement Number Plate Detection of an image using Matlab.

Software Used:

MATLAB

Theory:

The idea is to design a Computer Vision algorithm that takes in any number of images(even at different rotations or scales) and then stitches all of them to form a panorama. The algorithm uses SIFT keypoint detection, matches the key points, and stitches a pair of images using RANSAC and Homography matrices.

NOTE –

The algorithm works for images with a resolution of approximately 600*650 pixels.

All input images must be provided in order

The images were loaded from the data folder into the algorithm and then converted to single, greyscale images. Each of the images is then passed through a SIFT detector to find the keypoint indices of each image.

In order to stitch the images together, each of the image's key points needed to be matched using their SIFT descriptors. RANSAC is then used to find each of the transformations between images. It is used because it finds the most likely transformation by removing outliers. This transformation is then applied to the next image, and the two are stitched together.

Program:

function x = image_stitching()

```
%{
Code influenced (and restructured) from:
% http://www.vlfeat.org/applications/sift-mosaic-code.html
%}

% read images
% ADD 600*650 resolution images IN ORDER!

im1=imread('../data/1.JPG');
im2=imread('../data/2.JPG');
im3=imread('../data/3.JPG');
im4=imread('../data/4.JPG');
im5=imread('../data/5.JPG');
imagesOriginal = {im1, im2, im3, im4, im5};
figure(1) ; clf ;
montage({im1,im2,im3,im4,im5});
numImages=5;

% make single and grayscale
img1 = rgb2gray(im2single(im1)) ;
img2 = rgb2gray(im2single(im2)) ;
img3 = rgb2gray(im2single(im3)) ;
img4 = rgb2gray(im2single(im4)) ;
img5 = rgb2gray(im2single(im5)) ;
imagesGrayScale = {img1, img2, img3, img4, img5} ;

% Get keypoints for each image and store it.
```

```
for j=1:numImages
    [fG,dG]=vl_sift(imagesGrayScale{j});
    f{j}=fG; d{j}=dG;
end

%{
start with middle image. Stitch images to its left and right
So for 5 images - start with image3. Stitch img4 to it. then image2. then image1
followed by
image5.
if 6 images -> stitch 3&2. Then 4 followed by 1,5 and 6


Following is the algorithm to do this
%}

midImageIndex=ceil(numImages/2);
rotationIndices=floor(numImages/2);

%{
match first 2 images.
iF numImages is odd then match midImage with midImage+1
if numImages is even then match midImage with midImage-1
%}
if mod(numImages,2)==0
    imgIndex1=midImageIndex-1; imgIndex2=midImageIndex;
    [X1,X2, matches] =
matchImages(f{imgIndex1},d{imgIndex1},f{imgIndex2},d{imgIndex2});
    H=ransac(X1,X2,matches);
```

```matlab
    mosaic =
drawMosaic(H,imagesOriginal{imgIndex1},imagesOriginal{imgIndex2});

    mosaicSingle = rgb2gray(im2single(mosaic)) ;

    indicesToIgnore = -1; %don't stitch  midImageIndex-1 again, since already done it.
else

    imgIndex1 = midImageIndex; imgIndex2 = midImageIndex+1;

    [X1,X2, matches] =
matchImages(f{imgIndex1},d{imgIndex1},f{imgIndex2},d{imgIndex2});

    H=ransac(X1,X2,matches);

    mosaic =
drawMosaic(H,imagesOriginal{imgIndex1},imagesOriginal{imgIndex2});

    mosaicSingle = rgb2gray(im2single(mosaic)) ;

    indicesToIgnore = 1;
end


% now match the remianing images

for j=1:rotationIndices

  for k = [-1,1] %to simulate alternatively stitching right and left.

     imageIndex = midImageIndex + (j*k) ;

     %3-1 then 3+1 then 3-2 then 3+2.

     if (j*k) ~=indicesToIgnore && (imageIndex~=0)

        %ignore already matched image, or if we reach index 0.

        [fG,dG] = vl_sift(mosaicSingle);

        [X1,X2, matches] = matchImages(fG,dG,f{imageIndex},d{imageIndex});

        H=ransac(X1,X2,matches);

        mosaic = drawMosaic(H,mosaic,imagesOriginal{imageIndex});

        mosaicSingle = rgb2gray(im2single(mosaic)) ;

      end
```

```matlab
    end
end


figure(2) ; clf ;

imagesc(mosaic) ; axis image off ;

title('Mosaic') ;

% write image

imwrite(mosaic,'../stitched.png');


% HELPER METHOD - match keypoints.

function [X1,X2,matches] = matchImages(f1,d1,f2,d2)

    [matches, scores] = vl_ubcmatch(d1,d2) ;

    % matches returns the indices of both images where keypoints matched.

    % get (x,y) coordinates of all pixels that matched. add 3d coordinate.

    X1 = f1(1:2,matches(1,:)) ; X1(3,:) = 1 ;

    X2 = f2(1:2,matches(2,:)) ; X2(3,:) = 1 ;

end


% HELPER METHOD - Do ransac.

function H = ransac(X1,X2,matches)

    clear H score ok ; % set variables to 0.

    numMatches = size(matches,2) ;

    for t = 1:100

        % get subset of 4 points

        subset = vl_colsubset(1:numMatches, 4) ;

        A = [] ;

        % create homography matrix.
```

```matlab
    for i = subset
        A = cat(1, A, kron(X1(:,i)', vl_hat(X2(:,i)))) ;
    end
    [U,S,V] = svd(A) ;
    %H = last column after doing SVD.
    H{t} = reshape(V(:,9),3,3) ;


    % step 3 score homography
    X2_ = H{t} * X1 ; % PROJECTION of X1 with homography
    du = X2_(1,:)./X2_(3,:) - X2(1,:)./X2(3,:) ;
    dv = X2_(2,:)./X2_(3,:) - X2(2,:)./X2(3,:) ;
    ok{t} = (du.*du + dv.*dv) < 6*6 ; % if distance < threshold.
    score(t) = sum(ok{t}) ;
    end
    %step 4 - find the best one.
    [score, best] = max(score) ;
    H = H{best} ;
end


% HELPER METHOD - join 2 images based on homography model.
function mosaic = drawMosaic(H,im1,im2)
    box2 = [1  size(im2,2) size(im2,2)  1 ;
        1 1       size(im2,1)  size(im2,1) ;
        1 1      1          1] ;
    box2_ = inv(H) * box2 ;
    box2_(1,:) = box2_(1,:) ./ box2_(3,:) ;
    box2_(2,:) = box2_(2,:) ./ box2_(3,:) ;
```

```
    ur = min([1 box2_(1,:)]):max([size(im1,2) box2_(1,:)]) ;
    vr = min([1 box2_(2,:)]):max([size(im1,1) box2_(2,:)]) ;


    [u,v] = meshgrid(ur,vr) ;
    im1_ = vl_imwbackward(im2double(im1),u,v) ;


    z_ = H(3,1) * u + H(3,2) * v + H(3,3) ;
    u_ = (H(1,1) * u + H(1,2) * v + H(1,3)) ./ z_ ;
    v_ = (H(2,1) * u + H(2,2) * v + H(2,3)) ./ z_ ;
    im2_ = vl_imwbackward(im2double(im2),u_,v_) ;


    mass = ~isnan(im1_) + ~isnan(im2_) ;
    im1_(isnan(im1_)) = 0 ;
    im2_(isnan(im2_)) = 0 ;
    mosaic = (im1_ + im2_) ./ mass ;
end
end
```

INPUT IMAGE:

OUTPUT IMAGE

Result:

Thus, the implemention of Number Plate Detection using MATLAB was done successfully.

BY: 221701012 - S DANUSH NARAYAN

221701024 – N JAYASABHAREESH

221701026 – JUNIDE CHRIS A