221701503

THEJESH NS

RAJALAKSHMI ENGINEERING COLLEGE

An AUTONOMOUS Institution

Affiliated to ANNA UNIVERSITY, Chennai

DEPARTMENT OF COMPUTER SCIENCE AND DESIGN

CD19P02 – FUNDAMENTALS OF IMAGE PROCESSING

LABORATORY MANUAL

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.

RAJALAKSHMI ENGINEERING COLLEGE (AUTONOMOUS), CHENNAI.

INDEX

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 axe**s.**

Program:

To read and show the image

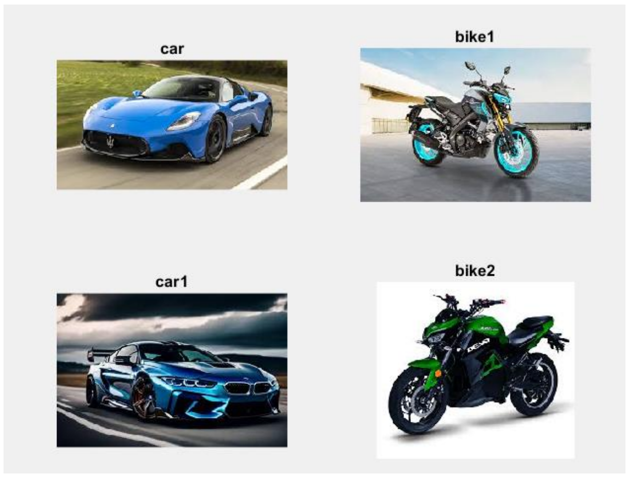
```
%Clear all workspace
clear
%Clear all open images
close all
%Clear Command windows
clc
%Read Image
I = imread('onion.png');
%Show Image
imshow(I);
```

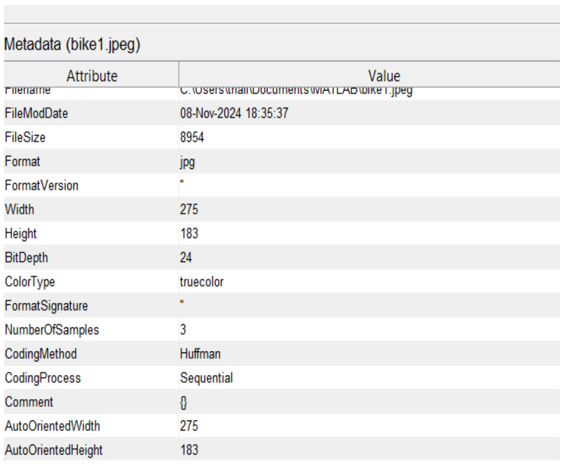
Output:

Program:

```
clc;
clear all;
close all;
subplot(2,2,1), imshow('cameraman.tif'),title('cameraman.tif');
subplot(2,2,2), imshow('peppers.png'),title('peppers.png');
subplot(2,2,3), imshow('baby.bmp'),title('baby.bmp');
subplot(2,2,4), imshow('butterfly.jpg'),title('butterfly');
impixelinfo;
imageinfo('cameraman.tif');
imageinfo('peppers.png');
imageinfo('baby.bmp');
imageinfo('butterfly.jpg');
```

Output:



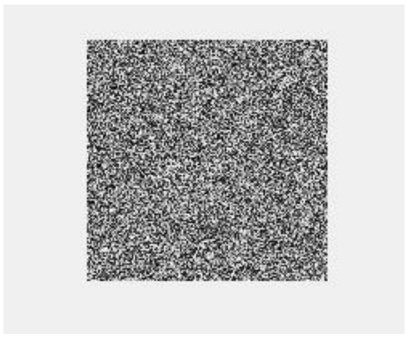

Metadata (bike1.jpeg)

Attribute	Value
Filename	C:\Users\thai\Documents\MATLAB\bike1.jpeg
FileModDate	08-Nov-2024 18:35:37
FileSize	8954
Format	jpg
FormatVersion	''
Width	275
Height	183
BitDepth	24
ColorType	truecolor
FormatSignature	''
NumberOfSamples	3
CodingMethod	Huffman
CodingProcess	Sequential
Comment	{}
AutoOrientedWidth	275
AutoOrientedHeight	183

Program:

```
% Write Gray scale Image to PNG
% Write a 150-by-150 array of grayscale values to a PNG file in the current folder.
clc;
clear all;
close all;
A = rand(150);
imwrite(A,'myGray.png');
imshow('mygray.png');
```

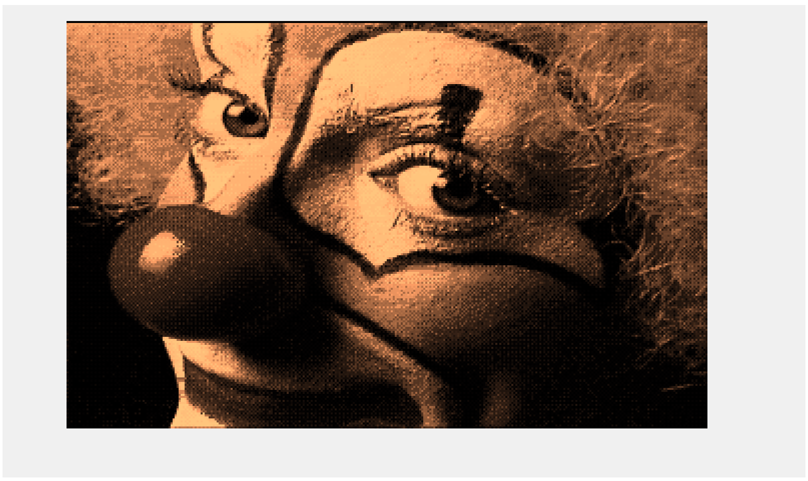
Output:



Program:

```
%write Indexed Image with MATLAB colormap

% Write image data to a new PNG file with the built-in MATLAB colormap, copper.

% Load sample image data from the file clown.mat.

clc;

clear all;

close all;

load clown.mat

% The image array X and its associated colormap, map, are loaded into the MATLAB workspace. map is a
matrix of 81 RGB vectors.

% Define a copper-tone colormap with 81 RGB vectors. Then, write the image data to a PNG file using the new
colormap.

newmap = copper(81);

imwrite(X,newmap,'copperclown.png');

% imwrite creates the file, copperclown.png, in your current folder.

% View the new file by opening it outside of MATLAB.

imshow('copperclown.png');
```

Output:



Result:
The important image commands have been displayed and studied

Ex.No:2a **IMPLEMENTATION OF AIRTHMETIC 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 =[2 05    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,[])
```

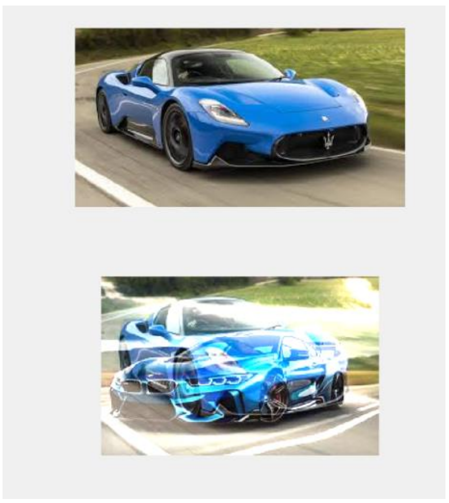
Program:

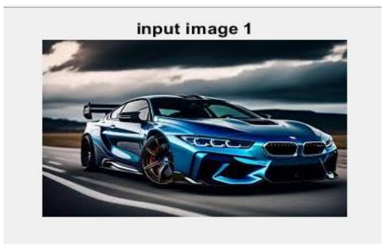
To perform arithmetic operation in an image

ADD

```
clc;
close all;
clear all;
I = imread('pic1.jpeg');
c=imread('pic3.jpeg');
J=imresize(I,[375 600]);
K = imadd(c, J);
figure;imshow(c);title('input image 1');
figure;imshow(J);title('input image 2');
figure;imshow(K);title('Output image');
subplot(2,2,1);imshow(I);
subplot(2,2,2);imshow(J);
subplot(2,2,3);imshow(K);
```

Output:



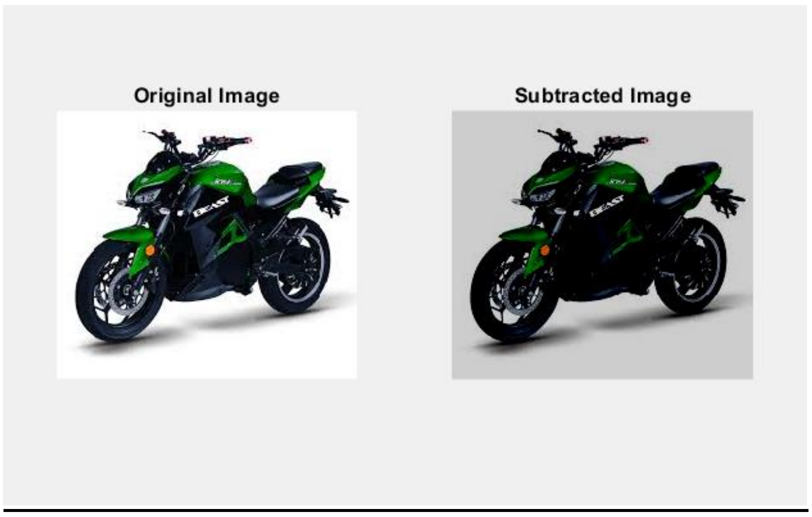


\

program

```
close all;
clear;
I = imread('rice.png'); %read a gray scale image into workspace
background = imopen(I,strel('disk',15)); %Eastimate the Background
Ip = imsubtract(I,background);
imshow(Ip,[]), title('Difference Image');
Iq = imsubtract(I,50); %subtract a constant value from the image
figure
subplot(1,2,1), imshow(I), title('Original Image');
subplot(1,2,2), imshow(Iq), title('Subtracted Image');
```

output

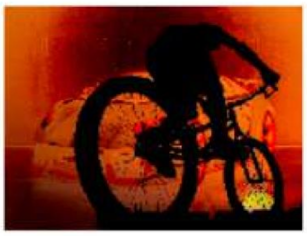


Program

Multiply

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

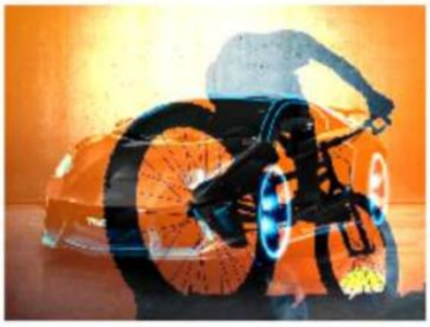
Output



Program:

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

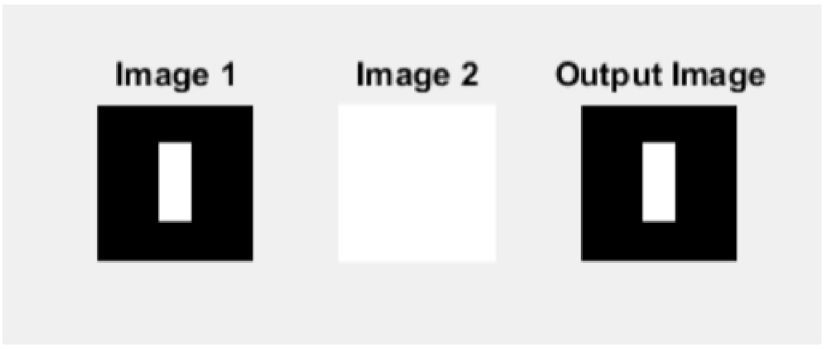
Logical X OR:
Syntax:
C = xor (a,b) ;

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:

```
 %Create an image of zeros
imageSize = [200, 200]; % Specify image size
i = zeros(imageSize); % Initialize image with zeros
% Define the region where you want to set ones
rowStart = 150;
rowEnd = 150;
colStart = 80;
colEnd = 120;
% Set ones in the specified region
i(rowStart:rowEnd, colStart:colEnd) = 1;
% Create an image of zeros
imageSize = [200, 200]; % Specify image size
j = ones(imageSize); % Initialize image with ones
% Perform logical AND operation between image 'i' and image 'j'
resultImage = i & j;
% Display the image1, image 2, and output image
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:

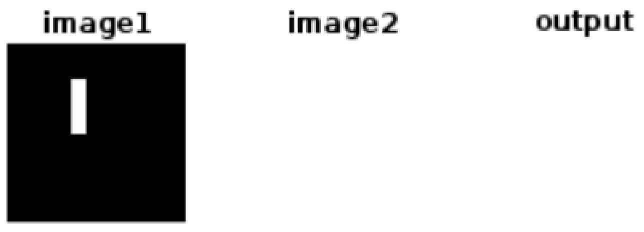


Program:

```
% Create an image of zeros
imageSize = [200, 200]; % Specify image size
i = zeros(imageSize); % Initialize image with zeros
% Define the region where you want to set ones
rowStart = 50;
rowEnd = 50;
colStart =180;
colEnd = 120;
```

```
% Set ones in the specified region
i(rowStart:rowEnd, colStart:colEnd) = 1;
% Create an image of zeros
imageSize = [200, 200]; % Specify image size
j = ones(imageSize); % Initialize image with ones
% Perform logical OR operation between image 'i' and image 'j'
resultImage = i | j;
% Display the image1, image 2, and output image
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');
```

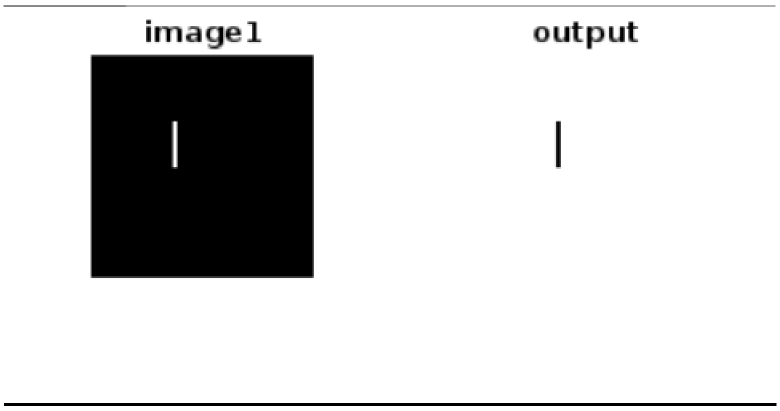
output image:



program:

```
imageSize = [500, 500];
i = zeros(imageSize);

rowStart = 150;
rowEnd = 250;
colStart = 180;
colEnd = 190;
i(rowStart:rowEnd, colStart:colEnd) = 1;
resultImage = ~i ;
subplot(2, 2, 1), imshow(i),title('image1');
subplot(2, 2, 2), imshow(resultImage),title('output');
```

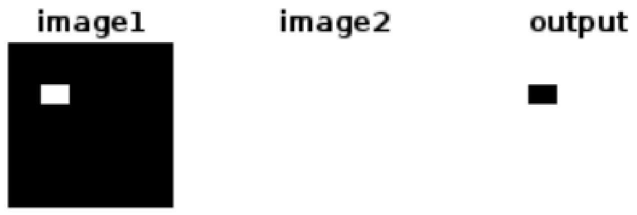
output:



Program:

```
imageSize = [400, 400];
i = zeros(imageSize);
rowStart = 100;
rowEnd = 150;
colStart = 80;
colEnd = 150;
i(rowStart:rowEnd, colStart:colEnd) = 1;
imageSize = [400, 400];
j = ones(imageSize);
resultImage = xor(i,j);
subplot(1, 3, 1), imshow(i), title('image1');
subplot(1, 3, 2), imshow(j), title('image2');
subplot(1, 3, 3), imshow(resultImage), title('output');
```

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**.**

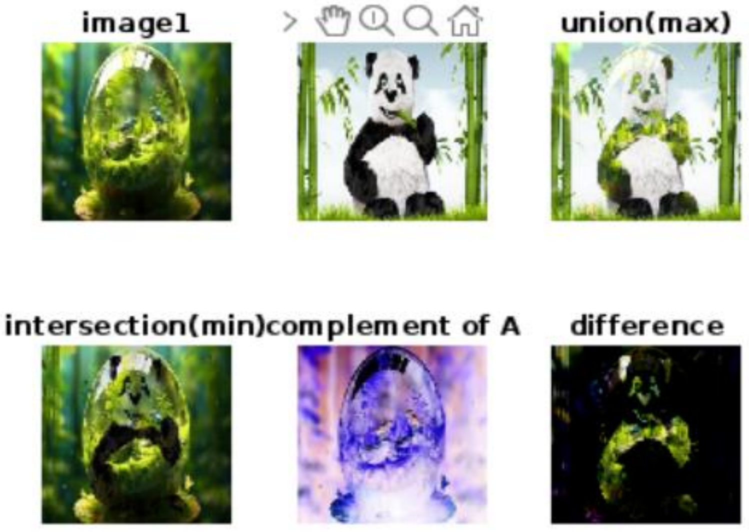
Difference:
Syntax:
differenceimage = abs (image A – image B) ;
The difference between two images is obtained by taking the absolute difference between their pixel values. This operation can be used for highlighting dissimilarities between images.

<u>Program:</u>

```
% Read the input images
A = imread('im1.jpeg'); % Replace with your image&#39;s path
B = imread('im2.jpeg'); % Replace with your image&#39;s path
imageA = imresize(A, [225,225]);
imageB = imresize(B, [225,225]);
% Ensure both images have the same size
if ~isequal(size(imageA), size(imageB))
error('Input images must have the same dimensions');
end
```

```
% Perform set operations
unionImage = max(imageA, imageB); % Union: Maximum pixel value
intersectionImage = min(imageA, imageB); % Intersection: Minimum pixel value
complementImageA = 255 - imageA; % Complement of image A
differenceImage = abs(imageA - imageB); % Difference: Absolute pixel difference
% Display the images
subplot(2, 3, 1);
imshow(imageA);
title('image1');
subplot(2, 3, 2);
imshow(imageB);
title('image2');
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');
% Save the resulting images
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.

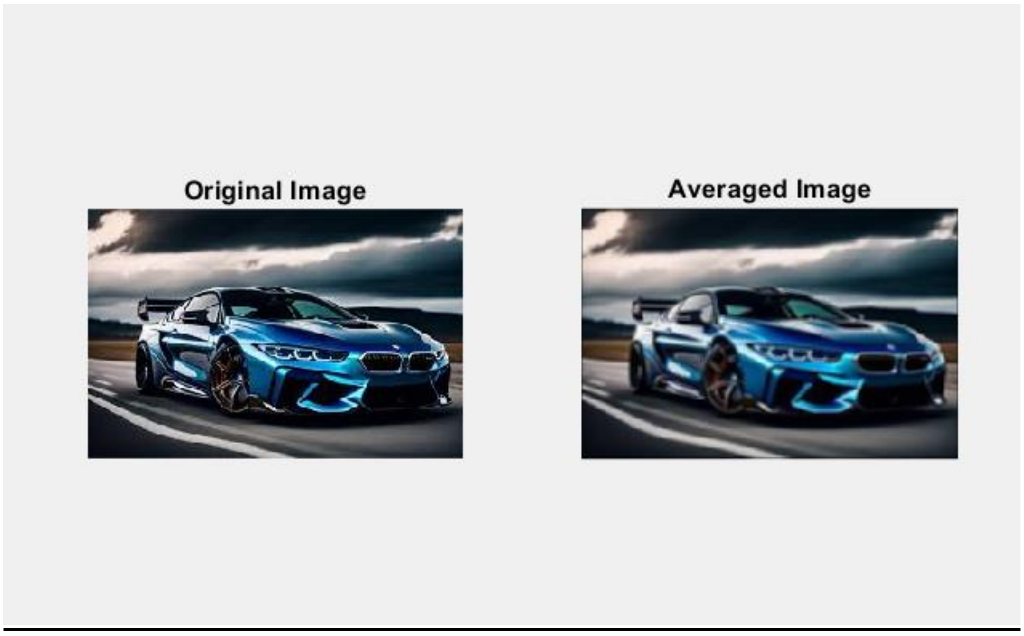
Adjustable Smoothing:
The degree of smoothing can be controlled by adjusting the size of the neighborhood and the values in the kernel. Larger neighborhoods or kernels with larger values will produce more significant smoothing, while smaller neighborhoods or kernels with smaller values will result in 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:

```
% Read the input image
inputImage = imread('butterfly1 (1).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:



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

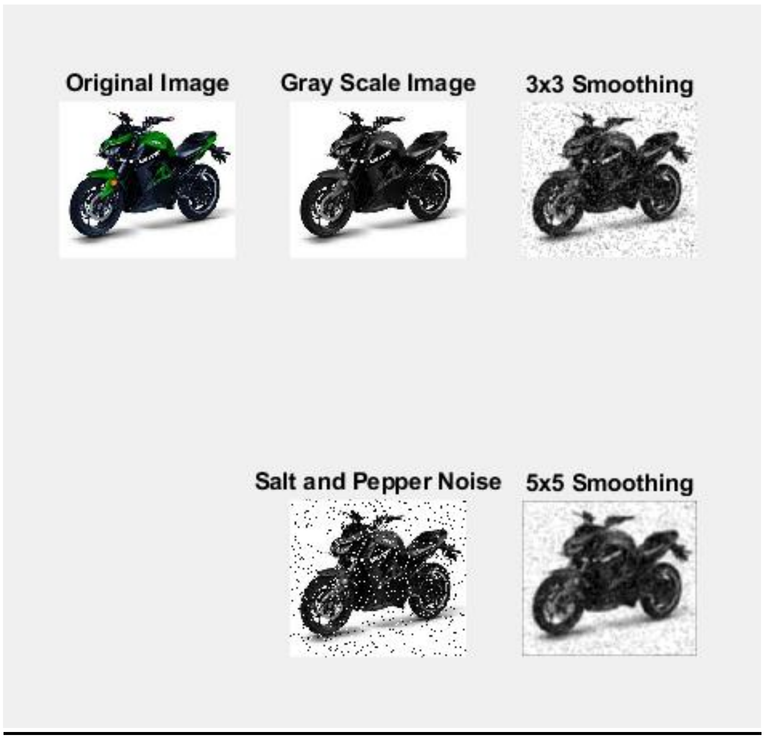
Theory:
Convolution and correlation are the two fundamental mathematical operations involved in linear filters based on neighbourhood-oriented image processing algorithms.
Convolution
Convolution processes an image by computing, for each pixel, a weighted sum of the values of that pixel and its neighbours. 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:
```
clc;
clear all;
close all;
a=imread('Moon.jpeg');
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(2,4,6);
imshow(c);
title('Salt and Pepper Noise');
h1=1/9*ones(3,3);
c1=conv2(c,h1,'same');
subplot(2,4,3);
imshow(uint8(c1));
title('3x3 Smoothing');
h2=1/25*ones(5,5);
c2=conv2(c,h2,'same');
subplot(2,4,7);
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.

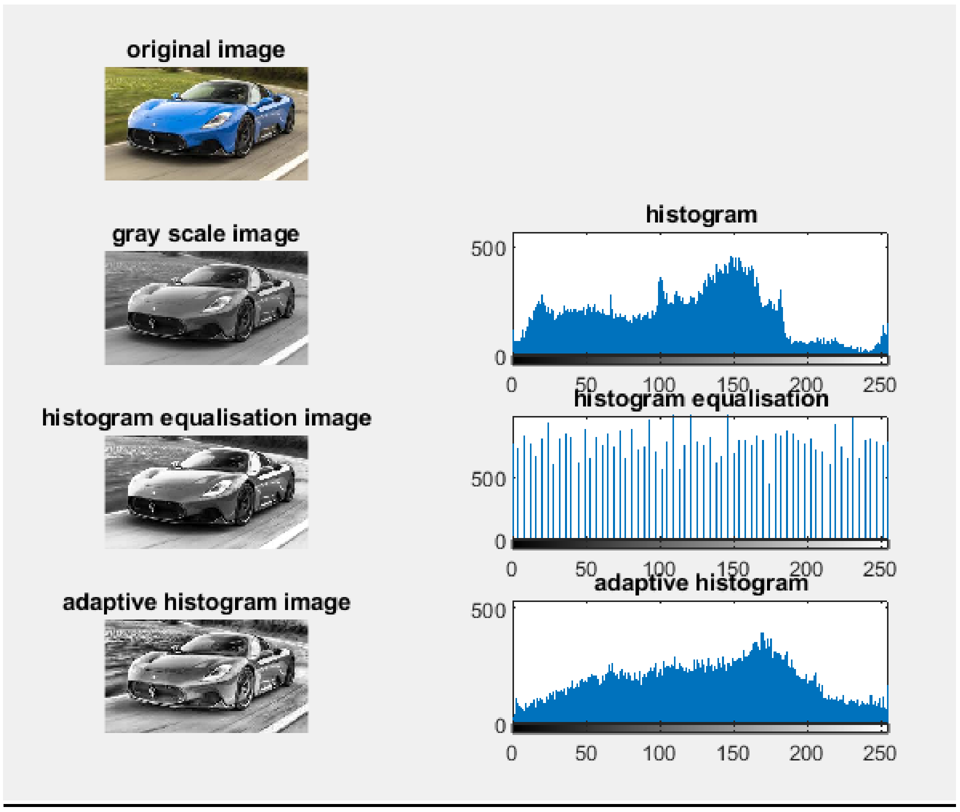
Histogram Processing:
The contrast of an image can be modified by manipulating its histogram. A popular method is via Histogram equalization. Here, the given histogram is manipulated such that the distribution of pixel values is evenly spread over the entire range 0 to K-1. Histogram equalization can 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 Adaptive HistogramEqualization.

<u>Program:</u>

```
clc;
clear all;
close all;
a= imread('peppers.png');
subplot(4,2,1);
imshow(a);
title('original image');
b=rgb2gray(a);
subplot(4,2,3);
imshow(b);
title('gray scale image');
subplot(4,2,4);
imhist(b);
title('histogram');
subplot(4,2,5);
c=histeq(b);
imshow(c);
title('histogram equalisation image');
subplot(4,2,6);
imhist(c);
title('histogram equalisation');
subplot(4,2,7);
f=adapthisteq(b);
imshow(f);
title('adaptive histogram image');
```

```
subplot(4,2,8);
imhist(f);
title('adaptive histogram');
```

Output:



Result:

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

EXP:5b
Date: **The correlation between the visual quality of an image with its histogram**

<u>Aim:</u>

To study the correlation between the visual quality of an image with its histogram

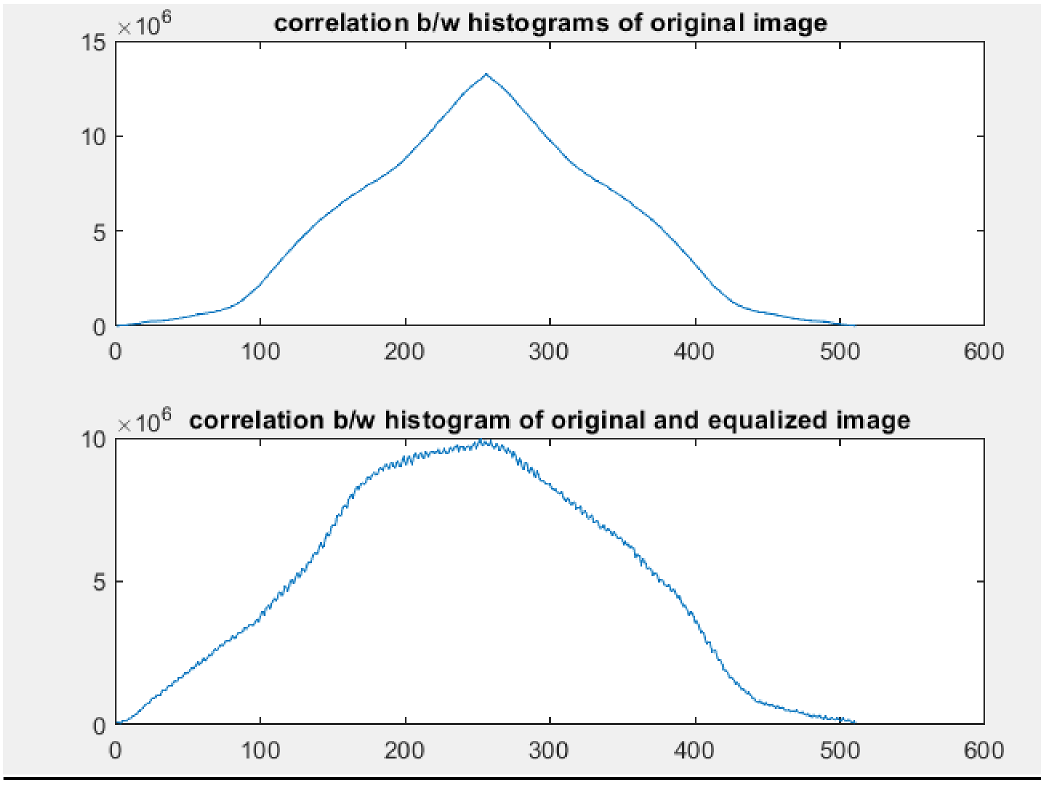
Software Used:

MATLAB

<u>Program:</u>

```
clc ;
clear;
close;
img= imread ('peppers.png');
img=rgb2gray(img);
% I = imresize (img ,[256 ,256]) ;
[ count , cells ]= imhist (img) ;
Iheq = histeq(img);
[count1,cells1 ]= imhist (Iheq);
%correlation between original image and Histogram equalized image
corrbsameimg = corr2(img,Iheq)
disp(corrbsameimg);
%correlation between the histograms of original image
x = xcorr ( count , count ) ;
%correlation between the histogram of original image and equalized image
x1 = xcorr ( count , count1 ) ;
subplot(2,1,1);
plot(x);
title('correlation b/w histograms of original image');
subplot(2,1,2);
plot(x1)
title('correlation b/w histogram of original and equalized image')
```

Output:



RESULT

Thus, the correlation between the visual quality of an image with its histogram has 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,

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

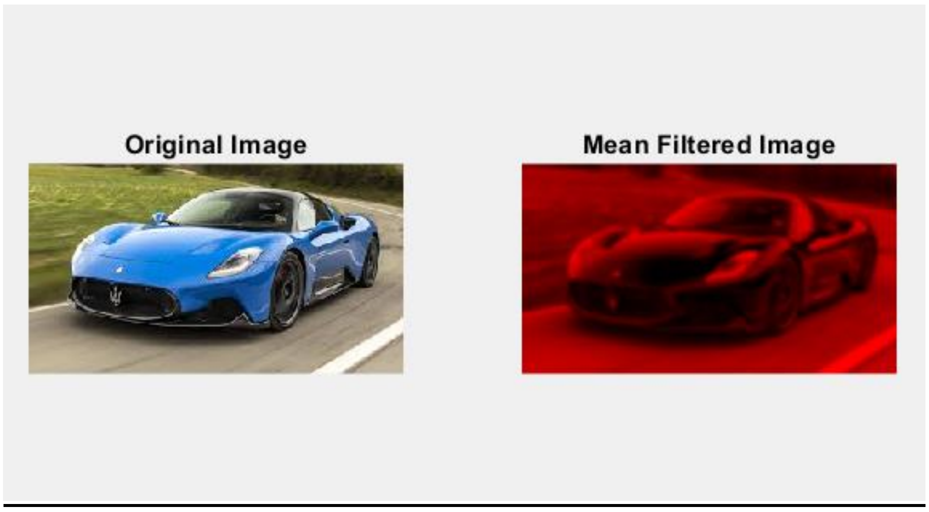
Where S is a neighborhood of (x,y) and n is the number of pixels in S.

Program:

```
clc;
close all;
clear all;
inputImage = imread('download.jpg'); % Load the input image
filterSize = 5; % Define the filter size (e.g., 3x3, 5x5, etc.)
% Pad the image to handle border pixels
paddedImage = padarray(inputImage, [filterSize, filterSize], 'replicate');
% Initialize the output image
outputImage = zeros(size(inputImage));
% Apply the mean filter
for i = 1:size(inputImage, 1)
    for j = 1:size(inputImage, 2)
        % Extract the neighborhood
        neighborhood = paddedImage(i:i+filterSize-1, j:j+filterSize-1);
        % Calculate the mean value of the neighborhood
        meanValue = mean(neighborhood(:));
        % Set the output pixel value
        outputImage(i, j) = meanValue;
    end
```

```
end
% Display the original and filtered images
subplot(1, 2, 1);
imshow(inputImage);
title('Original Image');
subplot(1, 2, 2);
imshow(uint8(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.

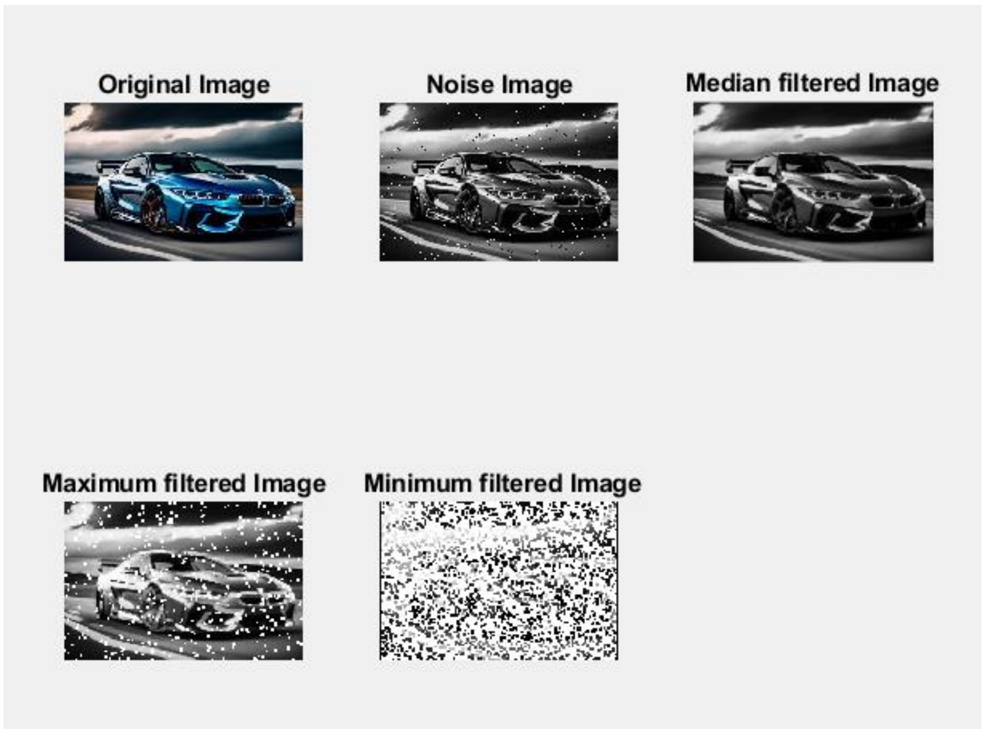
Maximum Filtering:
The maximum filter selects the maximum value from the neighborhood, which enhances bright features and suppresses dark features. (K=N)
The maximum filtering is achieved using the following equation
f(x,y) = max g(s,t)

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:
```
clc;
clear all;
close all;
b = imread('moon.jpg');
subplot(2,3,1);
imshow(b);
title('Original Image');
a=rgb2gray(b);
a = im2double(a);
a = imnoise(a,'salt & pepper',0.02);
subplot(2,3,2);
imshow(a);
title('Noise Image');
I = medfilt2(a);
subplot(2,3,3);
```

```
imshow(I);
title('Median filtered Image');
x=rand(size(a));
a(x(:)< 0.05)=0;
max_Img = ordfilt2(a,9,ones(3,3));
subplot(2,3,4);
imshow(max_Img);
title('Maximum filtered Image');
a(x(:)< 0.95)=255;
min_Img = ordfilt2(a,1,ones(3,3));
subplot(2,3,5);
imshow(min_Img);
title(' Minimum filtered Image');
```

Output:



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.

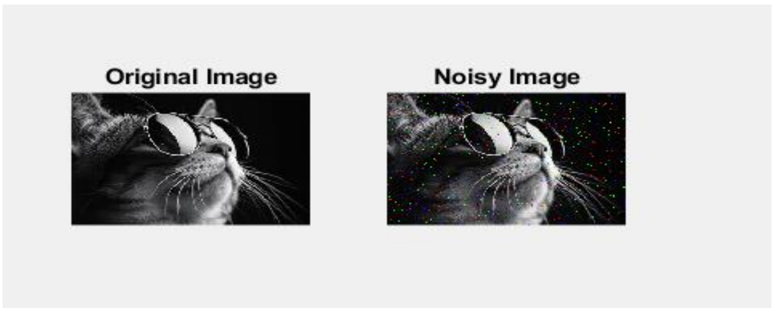
Program:

To remove various types of noise in an image

Salt and Pepper Noise:

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

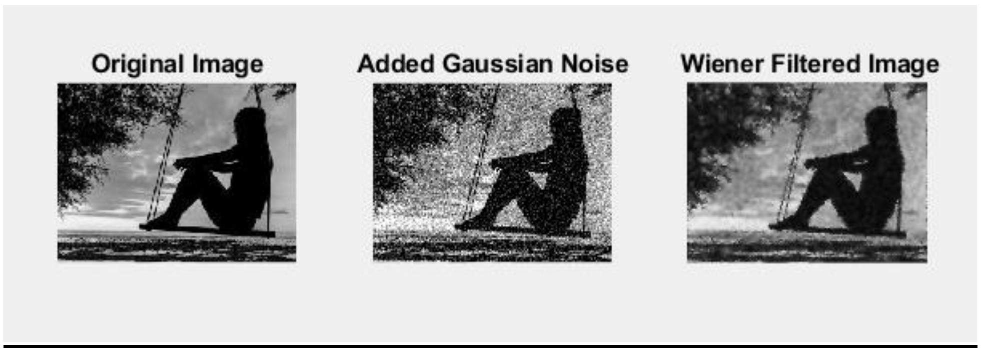
<u>Output:</u>



Gaussian Noise:
Program:

```
clc;
close all;
clear all;
RGB = imread('saturn.png');
I = im2gray(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

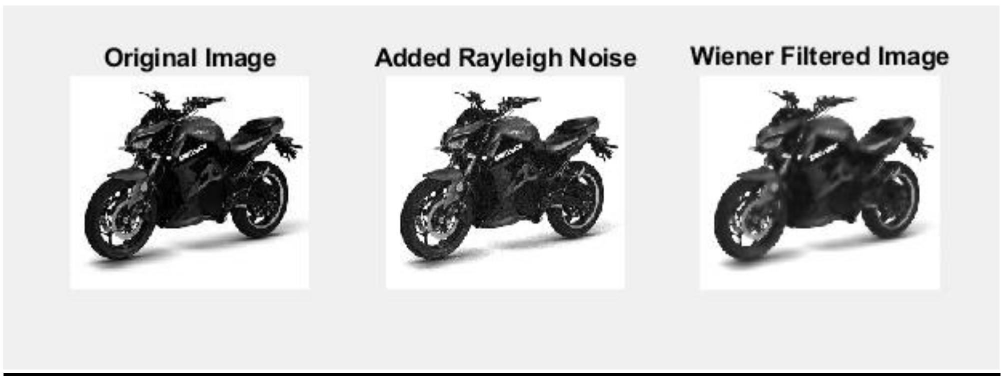


Rayleigh Noise
Program:

```
clc;
close all;
clear all;

% Read the image
RGB = imread('saturn.png');
I = im2gray(RGB);

% Add Rayleigh noise to the image
rayleighNoise = raylrnd(0.05, size(I)); % Generate Rayleigh noise
J = im2double(I) + rayleighNoise; % Add the noise to the image
```

```
% Apply Wiener filter to remove the noise
K = wiener2(J, [5 5]);

% Display results
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');
```

<u>Output</u>



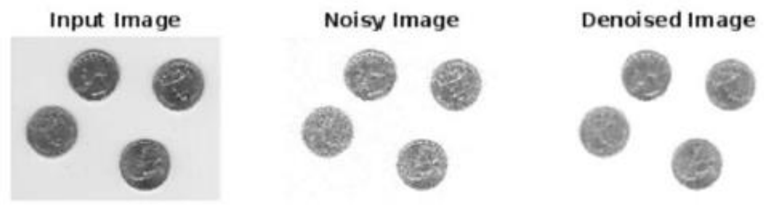
<u>d.Erlang Noise:</u>

<u>program:</u>

```
clc;
close all;
clear all;
I = imread('eight.tif');
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:

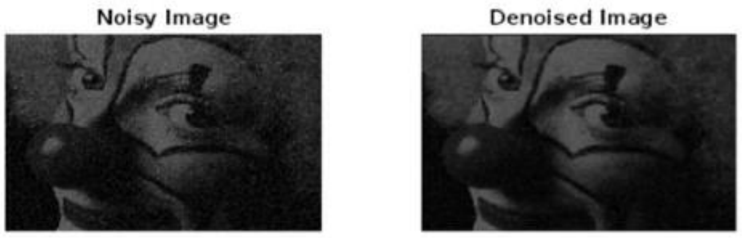


d.Erlang Noise:

program:

```
clc;
close all;
clear all;
I = imread('copperclown.png');
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, 2, 1);
imshow(noisy);
title('Noisy Image');
subplot(1, 2, 2);
imshow(denoised);
title('Denoised Image');
```

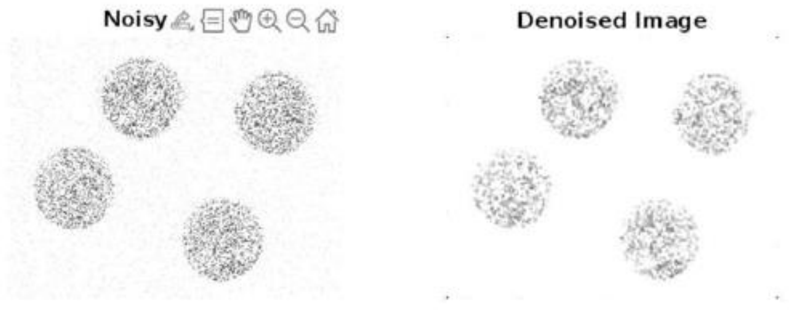
output:



d.Uniform Noise:
program:

```
I = imread('eight.tif');
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('Noisy Image');
subplot(1, 2, 2);
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

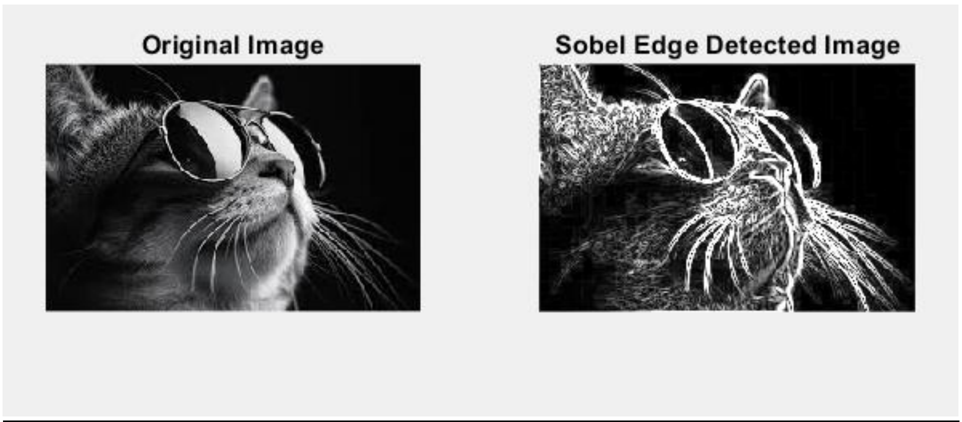
Theory:
The Sobel operator is a fundamental tool in image processing for edge detection and gradient estimation. It is used to find edges or boundaries in images by measuring the rate of change of intensity at each pixel. The theory behind the Sobel operator involves convolution with a pair of kernels 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:

```
% MATLAB code to detect edges using Sobel operator
% Load the input image
a = imread('peppers.png');

% Convert the input image to grayscale
b = rgb2gray(a);

% Convert the grayscale image to double precision
gray_img = double(b);

% Specify the Sobel operator kernels
h_kernel = [-1, 0, 1; -2, 0, 2; -1, 0, 1];  % Horizontal kernel
v_kernel = [-1, -2, -1; 0, 0, 0; 1, 2, 1];  % Vertical kernel

% Use the Sobel operator kernels to calculate gradients
c = imfilter(gray_img, h_kernel);
d = imfilter(gray_img, v_kernel);

% Calculate the gradient magnitude
gradient_magnitude = sqrt(c.^2 + d.^2);

% Display the original image and the edge detected image
figure;
subplot(2, 2, 1);
imshow(a);
title('Original Image');

subplot(2, 2, 2);
imshow(uint8(gradient_magnitude));
title('Sobel Edge Detected Image');
```

OUTPUT:



Result:

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

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

CD19P02–FUNDAMENTALSOFIMAGEPROCESSING

MINIPROJECT

FACEDETECTIONUSINGMATLAB

submittedby

THEJESH N S	221701503
NAVEEN RAJ	221701501

DEPARTMENTOFCOMPUTERSCIENCEANDDESIGN

RAJALAKSHMI ENGINEERING COLLEGE
RajalakshmiNagar,Thandalam,
Tamil Nadu 602105

ABSTRACT

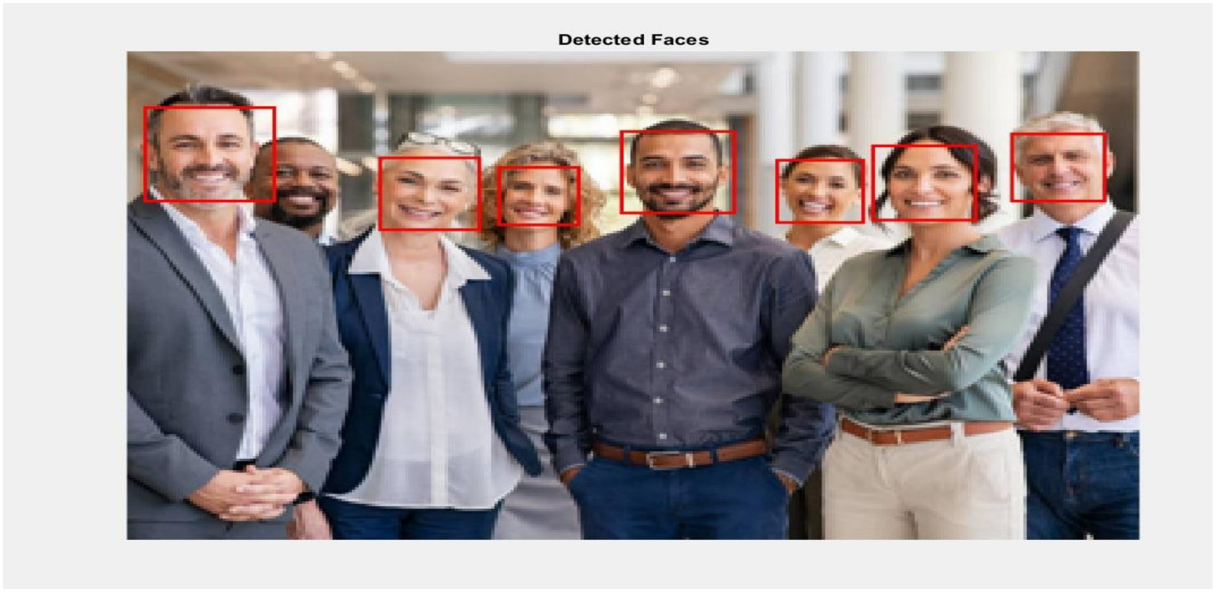
This project focuses on the development of a face detection system using MATLAB,utilizingcomputervisiontechniquestoautomaticallydetectandlocate human faces within images and video frames. The approach leverages the
Viola-Jonesalgorithm,awidelyusedmethodthatemploysHaar-likefeaturesand a cascade of classifiers for real-time face detection. The system processes input images by applying pre-processing steps, detecting facial features, and then identifying faces with high accuracy across various lighting conditions and orientations. MATLAB’s extensive libraries and built-in functions, such as the
vision.CascadeObjectDetector,areutilizedtoimplementthealgorithm, offering an efficient and scalable solution. The face detection system is capable of identifying multiple faces simultaneously, making it suitable for applications in areas such as security, surveillance, human-computer interaction, and multimedia processing. The project demonstrates the effectiveness of MATLAB for image processing and computer vision tasks, providing a solid foundation for further exploration and development in facial recognition and related fields.

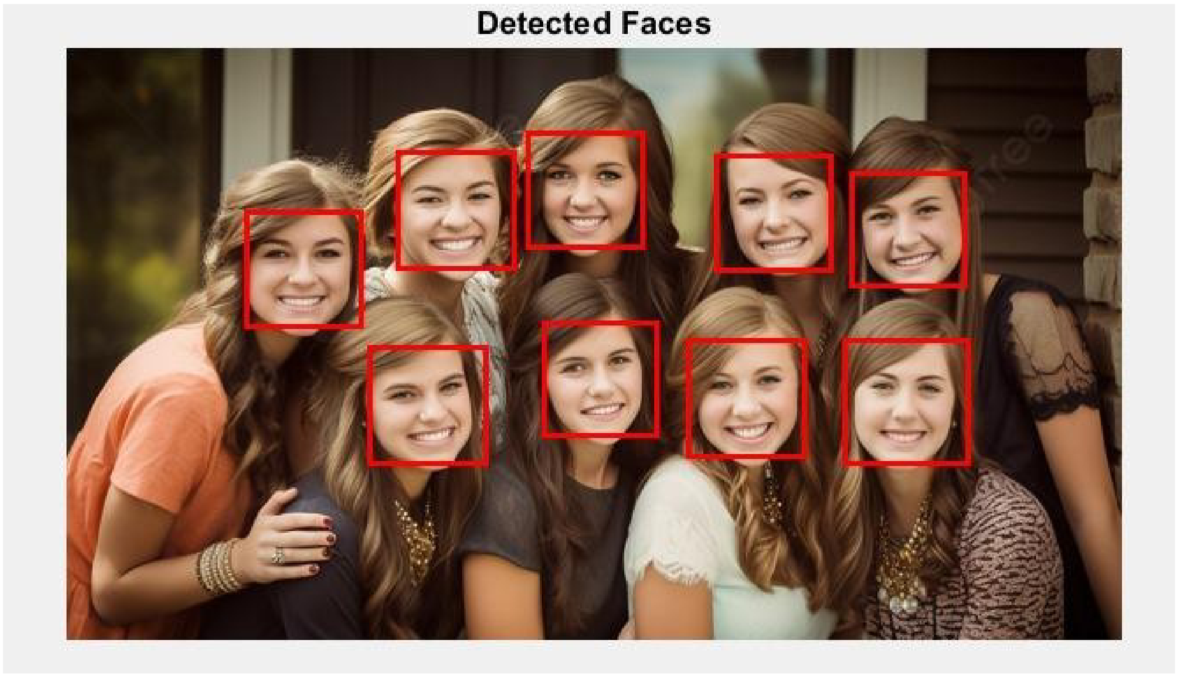
Matlab code for Face detection:

```
clear;
closeall;
fileName=['people.jpeg'];%Exampleimagefilename

%Checkifthefileexistsinthecurrentdirectory if
~exist(fileName, 'file')
   disp('Thespecifiedimagefiledoesnotexist.');
   return;
else
   % Read the selected image
   Image=imread(fileName);
end

%Resizetheimagetoasmallerscaleifit'stoolarge(e.g.,50%oforiginalsize)
scaleFactor = 0.5; % Adjust the scale factor as needed
Image=imresize(Image,scaleFactor);

%Createafacedetectorobject
faceDetector=vision.CascadeObjectDetector();

%Detectfacesintheimage
bboxes=step(faceDetector,Image);

%Displaytheresizedimagewithdetectedfaces
figure;
imshow(Image);
title('DetectedFaces
'); hold on;

%Drawrectanglesaroundeachdetectedface
for i = 1:size(bboxes, 1)
   rectangle('Position',bboxes(i,:),'LineWidth',2,'EdgeColor','r');
```

```
endhol
doff;

%DisplaythenumberoffacesdetectedintheCommandWindow
numFaces = size(bboxes, 1);
fprintf('Numberofdetectedfaces:%d\n',numFaces);
```

OUTPUT


Detected Faces


Detected Faces