****

**RAMNIRANJAN JHUNJHUNWALA COLLEGE**

**GHATKOPAR (W), MUMBAI - 400 086**

**DEPARTMENT OF INFORMATION TECHNOLOGY**

**2020 - 2021**

**M.Sc.( I.T.) SEM I**

**Image and Vision Processing**

**Name: KALPESH B GADGE**

**Roll No.: 25**



This is to certify that MR .**KALPESH B GADGE** with Seat No **25** has successfully completed the necessary course of experiments in the subject of **IMAGE AND VISION PROCESSING** during the academic year **2020-2021** complying with the requirements of **RAMNIRANJAN JHUNJHUNWALA COLLEGE OF ARTS, SCIENCE AND COMMERCE**, for the course of **M.Sc. (IT)** Semester - I.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Internal Examiner Date:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Head of Department College Seal External Examiner

**INDEX**

|  |  |  |
| --- | --- | --- |
| **Prac no.** | **Aim** | **Date** |
| 1 | Implement Basic Intensity transformation functions  A) Image Inverse  B) Log Transformation  C) Power-law Transformation |  |
| 2 | Piecewise Transformation  A) Contrast Stretching  B) Thresholding  C) Bit-Plane Slicing |  |
| 3 | Implement Histogram Equalization |  |
| 4 | Image filtering in Spatial Domain  A) Low-pass Filter/Smoothing Filters  (Average, Weighted Average, Median and Gaussian)  B) High-pass Filter / Sharpening Filter  (Laplacian Filter, Sobel, Robert and Prewitt Filter to detect edge) |  |
| 5 | Analyze image in Frequency Domain  A) Low Pass/Smoothing filter  B) High Pass/Sharpening filter |  |
| 6 | Color Image Processing  A) Pseudocoloring  B) Separating the RGB Channels  C) Color Slicing |  |
| 7 | Image Compression Techniques and watermarking  A) Implement Huffman Coding  B) Add a watermark to the image |  |
| 8 | Basic Morphological Transformations  A) Boundary Extraction  B) Thinning and Thickening  C) Hole filling and Skeletons |  |

**Practical 1: Implement Basic Intensity transformation functions**

**1. Image Inverse:**

The negative of an image is achieved by replacing the intensity ‘i’ in the original image by ‘i-1’, i.e. the darkest pixels will become the brightest and the brightest pixels will become the darkest. Image negative is produced by subtracting each pixel from the maximum intensity value. For example in an 8-bit grayscale image, the max intensity value is 255, thus each pixel is subtracted from 255 to produce the output image.  
The negative or inverse of an image with intensity levels in the range [0, L-1] is obtained by using the negative transformation, which is given by the expression,

S = L – 1 – r

Where L – 1 (Maximum intensity pixel value)

S(Output pixel value)

r (Input pixel of an image)

imread():

The *imread()* function reads images from the graphics files.

**[A](https://in.mathworks.com/help/matlab/ref/imread.html" \l "btnczv9-1-A) = imread([filename](https://in.mathworks.com/help/matlab/ref/imread.html" \l "mw_abb56802-ede7-4bfd-a64e-0743144c1db9))** reads the image from the file specified by filename, inferring the format of the file from its contents. If filename is a multi-image file, then imread reads the first image in the file.

**[A](https://in.mathworks.com/help/matlab/ref/imread.html" \l "btnczv9-1-A) = imread([filename](https://in.mathworks.com/help/matlab/ref/imread.html" \l "mw_abb56802-ede7-4bfd-a64e-0743144c1db9),[fmt](https://in.mathworks.com/help/matlab/ref/imread.html" \l "btnczv9-1-fmt))** additionally specifies the format of the file with the standard file extension indicated by fmt. If imread cannot find a file with the name specified by filename, it looks for a file named *filename.fmt*.

im2double():

[I2 = im2double(I)](https://in.mathworks.com/help/matlab/ref/im2double.html" \l "d122e624739)

[I2 = im2double(I,'indexed')](https://in.mathworks.com/help/matlab/ref/im2double.html" \l "d122e624760)

[I2](https://in.mathworks.com/help/matlab/ref/im2double.html" \l "mw_2f2def76-c130-4671-87d4-457d8bfbcd11) = im2double([I](https://in.mathworks.com/help/matlab/ref/im2double.html" \l "buhz5vt-1-I)) converts the image I to double precision. I can be a grayscale intensity image, a truecolor image, or a binary image. im2double rescales the output from integer data types to the range [0, 1].

[I2](https://in.mathworks.com/help/matlab/ref/im2double.html" \l "mw_2f2def76-c130-4671-87d4-457d8bfbcd11) = im2double([I](https://in.mathworks.com/help/matlab/ref/im2double.html" \l "buhz5vt-1-I),'indexed') converts the indexed image I to double precision. im2double adds an offset of 1 to the output from integer data types.

subplot():

**subplot([m](https://in.mathworks.com/help/matlab/ref/subplot.html" \l "btw1t4b-1-m),[n](https://in.mathworks.com/help/matlab/ref/subplot.html" \l "btw1t4b-1-n),[p](https://in.mathworks.com/help/matlab/ref/subplot.html" \l "btw1t4b-1-p))** divides the current figure into an m-by-n grid and creates axes in the position specified by p. MATLAB® numbers subplot positions by row. The first subplot is the first column of the first row, the second subplot is the second column of the first row, and so on. If axes exist in the specified position, then this command makes the axes the current axes

**imshow():**

**imshow([I](https://in.mathworks.com/help/images/ref/imshow.html" \l "bvmnrxi-1-I))** displays the grayscale image I in a figure. imshow uses the default display range for the image data type and optimizes figure, axes, and image object properties for image display.

**Code:**

x = imread('input.png'); #Reading the image

x = rgb2gray(x); #Converting RGB image to gray-level

x = im2double(x); #Converting image into double data type

[row col] = size(x); #taking image size into matrix form

for i = 1:row #reading row value from starting to end and storing in (i) variable

for j = 1:col #reading row value from starting to end and storing in (j) variable

N(i,j)=1-x(i,j); #subtracting input matrix values from 255 and storing in new Matrix(N)

endfor

endfor

figure;

imshow(x);

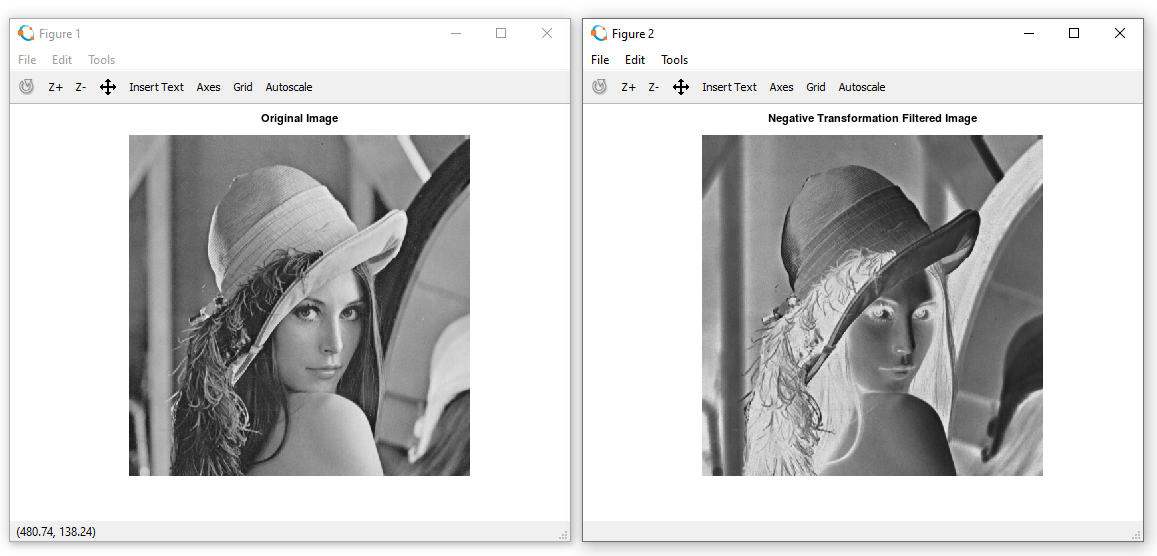
title('Original Image');

figure;

imshow(N);

title('Negative Transformation Filtered Image');

**Output:**



**2. Log Transformation:**

The log transformation maps a narrow range of low intensity values in the input into a wider range of output levels. We use the transformation if this type to extend the values of dark pixel in an image while compress the higher-level values.

The general form of the log transformation is:

s = c log (r + 1)

Where c is a constant, and r ≥ 0.

Where s and r are the pixel values of the output and the input image and c is a constant. The value 1 is added to each of the pixel value of the input image because if there is a pixel intensity of 0 in the image, then log (0) is equal to infinity. So 1 is added, to make the minimum value at least 1.

During log transformation, the dark pixels in an image are expanded as compare to the higher pixel values. The higher pixel values are kind of compressed in log transformation. This result in following image enhancement.

rgb2gray():

**[I](https://in.mathworks.com/help/matlab/ref/rgb2gray.html" \l "buiz8mj-1-I) =rgb2gray([RGB](https://in.mathworks.com/help/matlab/ref/rgb2gray.html" \l "buiz8mj-1-RGB))** converts the truecolor image RGB to the grayscale image I. The rgb2gray function converts RGB images to grayscale by eliminating the hue and saturation information while retaining the luminance.

imwrite():

**imwrite([A](https://in.mathworks.com/help/matlab/ref/imwrite.html" \l "btv3cny-1-A),[filename](https://in.mathworks.com/help/matlab/ref/imwrite.html" \l "btv3cny-1-filename))** writes image data A to the file specified by filename, inferring the file format from the extension. imwrite creates the new file in your current folder. The bit depth of the output image depends on the data type of A and the file format.

**Code:**

x = imread('input.png'); #Reading the image

x = rgb2gray(x); #Converting RGB image to gray-level image

x = im2double(x); #Converting image into double data type

[row col] = size(x); #taking image size into matrix form

c=2; #here we are taking constant value into c variable

for i = 1:row #reading row value from starting to end and storing in (i) variable

for j = 1:col #reading row value from starting to end and storing in (j) variable

N(i,j)=c\*log(1+x(i,j)); #Here we are doing log calculation and storing the value into N

endfor

endfor

figure

imshow(x);

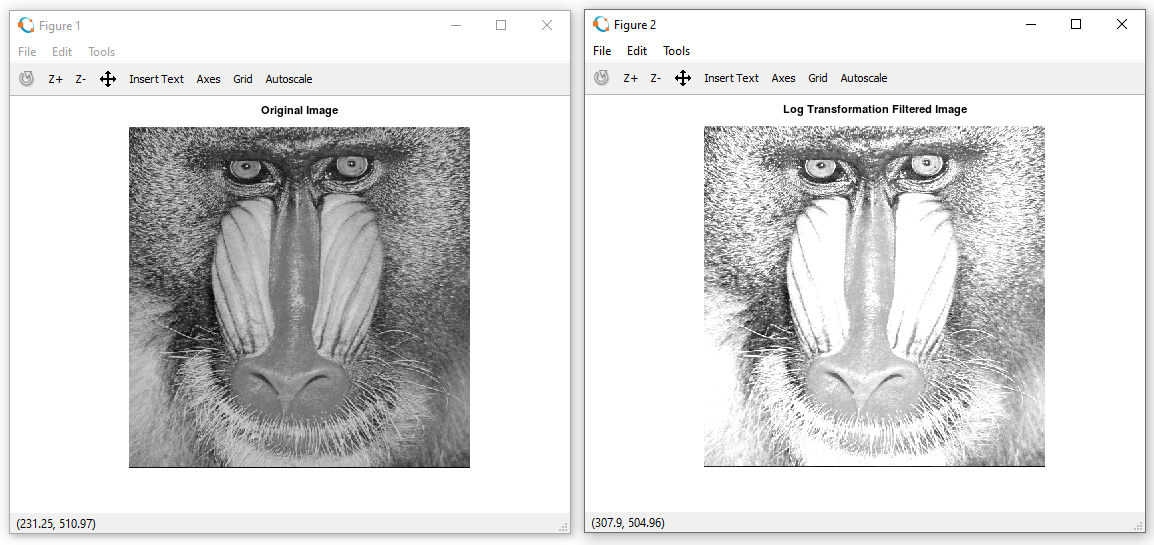
title('Original Image');

figure

imshow(N);

title('Log Transformation Filtered Image');

**Output:**



**3. Power-Law Transformation:**

Power-law curves with fractional values of γ map a narrow range of dark input values into a wider range of output values, with the opposite being true for higher values of input levels. The nth power and nth root curves shown in below figure can be given by the expression as s = c r γ, This transformation function is also called as gamma correction. For various values of γ different levels of enhancements can be obtained. It is used to correct power law response phenomena. The different display monitors display images at different intensities and clarity. That means, every monitor has built-in gamma correction in it with certain gamma ranges and so a good monitor automatically corrects all the images displayed on it for the best contrast to give user the best experience. The gamma variation changes ratio of red green & blue along with intensity in color images. The difference between the log-transformation function and the power-law functions is that using the power-law function a family of possible transformation curves can be obtained just by varying the λ. This process is also called a gamma correction. The Power Low Transformations can be given by the expression:

s = c \* r ^ γ where, s is the output pixels value r is the input pixel value c and γ are the real numbers.

size():

A = size(Y), this function will return the size of each dimension of the array passed as input.

[a, b] = size(Y), this function will return the size of input matrix in 2 separate variables ‘a’ and ‘b’

A = size(Y,dim), this function will return the size of Y’s dimension, specified by the input scalar dim.

[dim1,dim2,dim3,…,dimN] = size(Y), this function will return the size of ‘n’ dimensions of input array X in separate variables.

imhist():

[[counts](https://in.mathworks.com/help/images/ref/imhist.html" \l "buo3qek-1-counts),[binLocations](https://in.mathworks.com/help/images/ref/imhist.html" \l "buo3qek-1-binLocations)] = imhist([I](https://in.mathworks.com/help/images/ref/imhist.html" \l "buo3qek-1-I)) calculates the histogram for the grayscale image I. The imhist function returns the histogram counts in counts and the bin locations in binLocations. The number of bins in the histogram is determined by the image type.

**Code:**

x = imread('baby.jpg'); #Reading the image

x = rgb2gray(x); #Converting RGB image to gray-level image

x = im2double(x); #Converting image into double data type

[row col] = size(x); #taking image size into matrix form

gamma=2; #here we are taking constant value into “gamma” variable

c=1; #here we are taking constant value into “c” variable

for i = 1:row #reading row value from starting to end and storing in (i) variable

for j = 1:col #reading row value from starting to end and storing in (j) variable

N(i,j)=c\*(x(i,j)^gamma); #Here we are doing power-law calculation and storing there values into N matrix

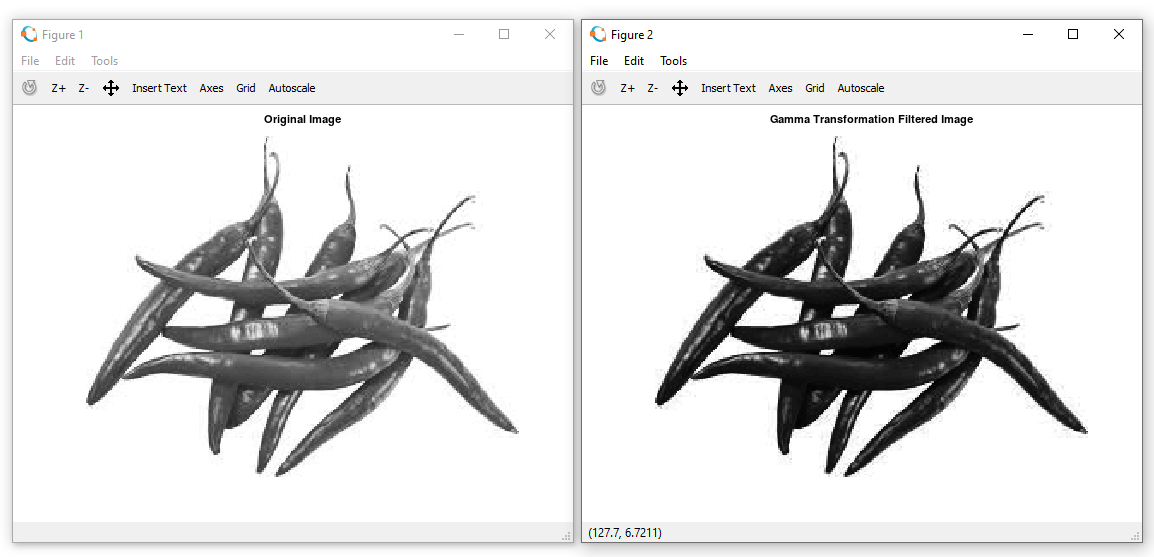
endfor

endfor

figure, imshow(x); title('Original Image');

figure, imshow(N); title('Gamma Transformation Filtered Image');

**Output:**



**Practical 2: Implement Piecewise Transformation**

**1. a. Contrast Stretching without user inputs:**

Contrast stretching is also known as normalization. It is a simple image enhancement technique. The quality of image is enhanced by stretching the range of intensity values.

**imadjust():**

Adjust image or colormap intensity (values).

Returns an image of equal dimensions to I, cmap, or RGB, with its intensity values adjusted, usually for the purpose of increasing the image contrast.

The values are rescaled according to the input and output limits, low\_in and high\_in, and low\_out and high\_out respectively. The first pair sets the lower and upper limits on the input image, values above and below them being clipped. The second pair sets the lower and upper limits for the output image, the interval to which the image will be scaled after clipping the input limits.

For example:

**imadjust (img, [0.2; 0.9], [0; 1])**

will clip all values in img outside the range [0.2 0.9], and then rescale them linearly into the range [0 1].

**Code:**

pkg load image;

a=imread('cat.png');

#agray=rgb2gray(a);

ad=im2double(a);

subplot(2,2,1); imshow(ad); title("before");

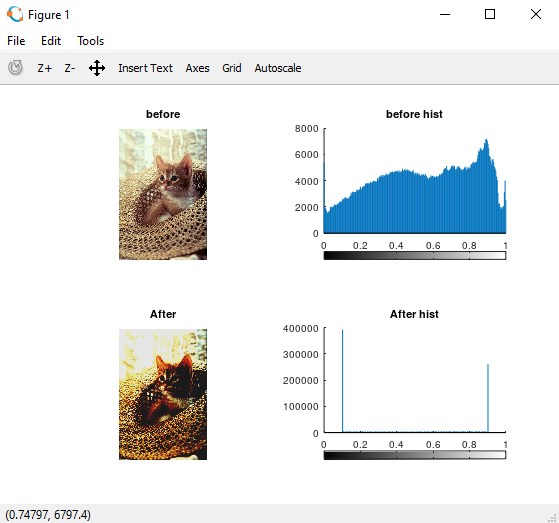
subplot(2,2,2); imhist(ad); title("before hist");

imad=imadjust(ad,[0.44 0.8],[0.1 0.9]);

subplot(2,2,3); imshow(imad); title("After");

subplot(2,2,4); imhist(imad); title("After hist");

**Output:**



**1. b. Contrast Stretching Using inputs from user r1,r2,s1,s2**

**Code:**

pkg load image;

clear all;

close all;

r = imread('fruits.png');

#r=rgb2gray(r);

r = im2double(r);

[m n] = size(r); % Getting the dimensions of the image.

#here we are taking 4 input from user

r1=input("Enter R1: ");

r2=input("Enter R2: ");

s1=input("Enter S1: ");

s2=input("Enter S2: ");

#Calculation of contrast stretching

a = s1/r1;

b = (s2-s1)/(r2-r1);

c = (255-s2)/(255-r2);

for i=1:m

for j=1:n

if r(i,j) < r1

s(i,j) = a\*r(i,j);

elseif r(i,j) < r2

s(i,j) = b\*(r(i,j)-r1)+s1;

else

s(i,j) = c\*(r(i,j)-r2)+s2;

endif

endfor

endfor

#Displaying the Original and Contrast Images

figure(3);

subplot(1,2,1)

imshow(r);

title("Original Image");

subplot(1,2,2)

imhist(r);

title('Histogtram Of Original Image');

figure(4);

subplot(1,2,1)

imshow(s);

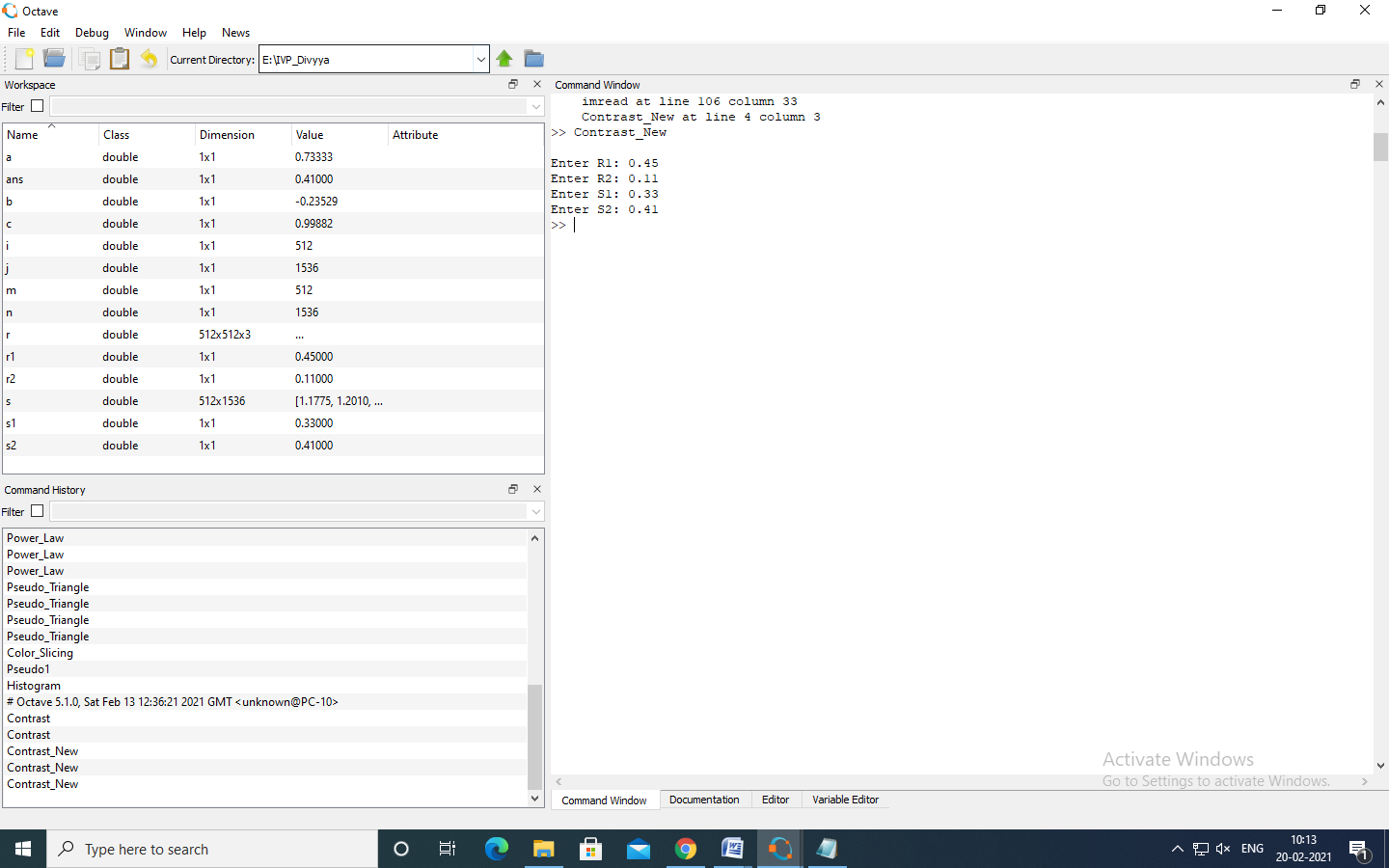
title("Contrast Streched Image");

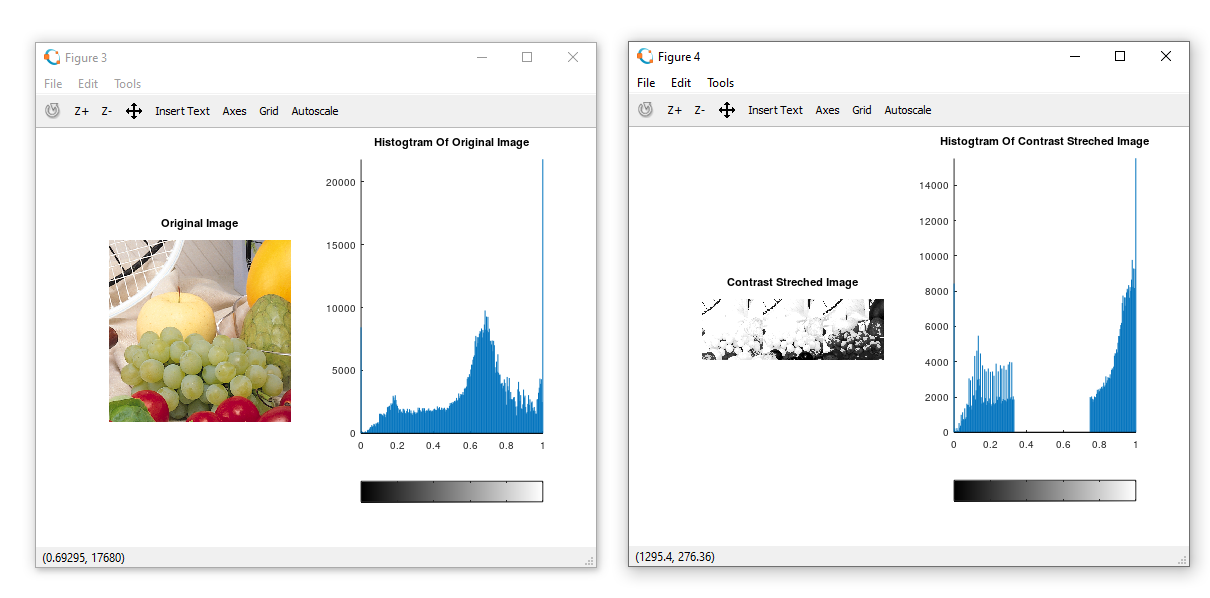
subplot(1,2,2)

imhist(s);

title('Histogtram Of Contrast Streched Image');

**Output:**





**2.** **Threshold:**

Image thresholding is a simple, yet effective, way of partitioning an image into a foreground and background. This [image analysis](https://in.mathworks.com/discovery/image-analysis.html) technique is a type of image segmentation that isolates objects by converting grayscale images into binary images. Image thresholding is most effective in images with high levels of contrast.

**zeros():**

**B = zeros(n)** returns an n-by-n matrix of zeros. An error message appears if n is not a scalar.

**B = zeros(m,n)** or B = zeros([m n]) returns an m-by-n matrix of zeros.

**Code:**

pkg load image;

clear all;

r=imread('monarch.png');

#r=rgb2gray(r);

#r=im2double(r);

imhist(r);

thr=150;

[m n]=size(r);

s=zeros(m,n);

for i=1:m

for j=1:n

if(r(i,j))>thr

s(i,j)=1;

else

s(i,j)=0;

endif

endfor

endfor

subplot(2,2,1); imshow(r); title("Original");

subplot(2,2,2); imhist(r); title("Original Histogram");

subplot(2,2,3); imshow(s); title("Thresholded Image");

subplot(2,2,4); imhist(s); title("Threshold Histogram");

**Output:**



**3. Bitplane Slicing:**

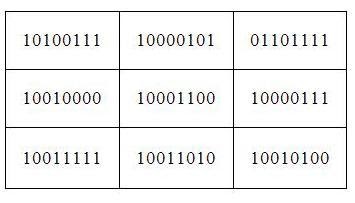
1. The nth plane in the pixels are multiplied by the constant 2^n-1

2. For instance, consider the matrix

A= A=[167 133 111

      144 140 135

      159 154 148] and the respective bit format



3. Combine the 8 bit plane and 7 bit plane.

For 10100111, multiply the 8 bit plane with 128 and 7 bit plane with 64.\

(1x128) + (0x64) + (1x0) + (0x0) + (0x0) + (1x0) + (1x0) + (1x0) = 128

4. Repeat this process for all the values in the matrix and the final result will be

[128 128 64

128 128 128

128 128 128]

**biget():**

Get bit at specified position

**Syntax**

C = bitget(A, *bit*)

Description

C = bitget(A, *bit*) returns the value of the bit at position *bit* in A. Operand A must be an unsigned integer or an array of unsigned integers, and *bit* must be a number between 1 and the number of bits in the unsigned integer class of A (e.g., 32 for the uint32 class).

**bitset():**

Set bit at specified position

Syntax

C = bitset(A, *bit*)

C = bitset(A, *bit*, v)

**Description**

C = bitset(A, *bit*) sets bit position *bit* in A to 1 (on). A must be an unsigned integer or an array of unsigned integers, and *bit* must be a number between 1 and the number of bits in the unsigned integer class of A (e.g., 32 for the uint32 class).

C = bitset(A,*bit*, v) sets the bit at position *bit* to the value v, which must be either 0 or 1.

**Code:**

%Image reconstruction by combining 8 bit plane and 7 bit plane

A=imread('coins.png');

B=zeros(size(A));

B=bitset(B,7,bitget(A,7));

B=bitset(B,8,bitget(A,8));

B=uint8(B);

figure,imshow(B);

%Image reconstruction by combining 8,7,6 and 5 bit planes

A=imread('coins.png');

B=zeros(size(A));

B=bitset(B,8,bitget(A,8));

B=bitset(B,7,bitget(A,7));

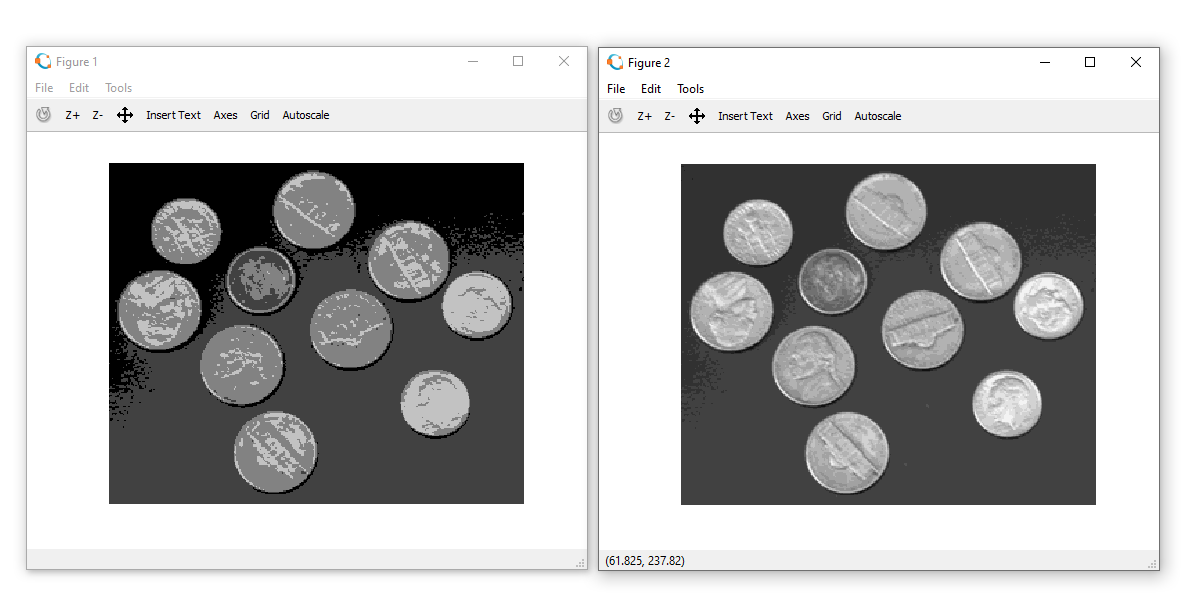
B=bitset(B,6,bitget(A,6));

B=bitset(B,5,bitget(A,5));

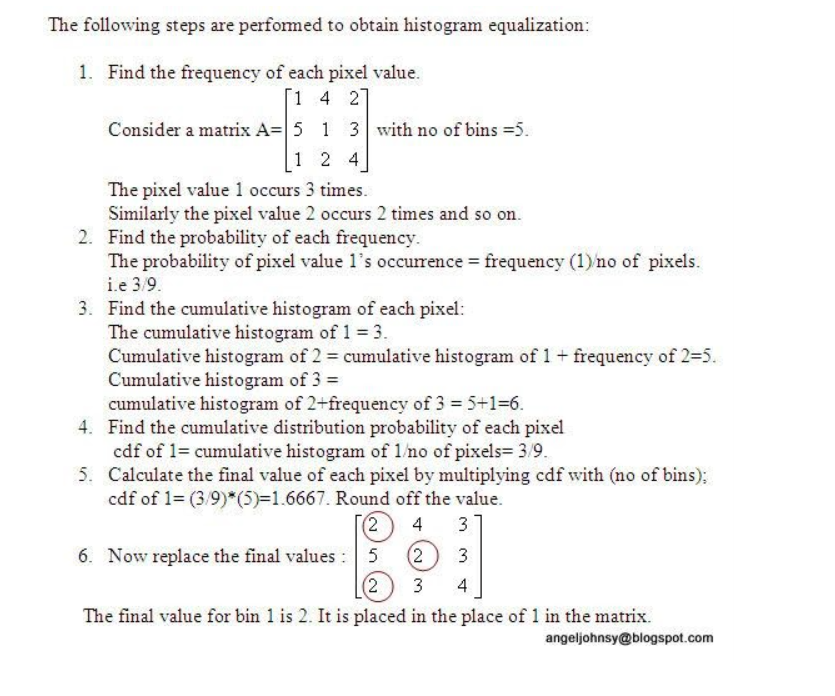
B=uint8(B);

figure,imshow(B);

**Output:**



**Practical 3: Implement Histogram Equalization**

****

**round():**

**Y = round([X](https://in.mathworks.com/help/matlab/ref/round.html" \l "buftmpz-X))** rounds each element of X to the nearest integer. In the case of a tie, where an element has a fractional part of exactly 0.5, the round function rounds away from zero to the integer with larger magnitude.

Y = round([X](https://in.mathworks.com/help/matlab/ref/round.html" \l "buftmpz-X),[N](https://in.mathworks.com/help/matlab/ref/round.html" \l "buftmpz-N)) rounds to N digits:

N > 0: round to N digits to the right of the decimal point.

N = 0: round to the nearest integer.

N < 0: round to N digits to the left of the decimal point.

**Code:**

clear all;

close all;

pkg load image;

a=imread('monarch.png');

r=size(a,1);

c=size(a,2);

ah=uint8(zeros(r,c));

n=r\*c;

f=zeros(256,1);

pdf=zeros(256,1);

cdf=zeros(256,1);

cumm=zeros(256,1);

out=zeros(256,1);

for i=1:r

for j=1:c

values=a(i,j);

f(values+1)=f(values+1)+1;

pdf(values+1)=f(values+1)/n;

endfor

endfor

sum=0; L=255; size(pdf);

for i=1:size(pdf)

sum=sum+f(i);

cum(i)=sum;

cdf(i)=cum(i)/n;

out(i)=round(cdf(i)\*L);

endfor

for i=1:r

for j=1:c

ah(i,j)=out(a(i,j)+1);

endfor

endfor

figure;

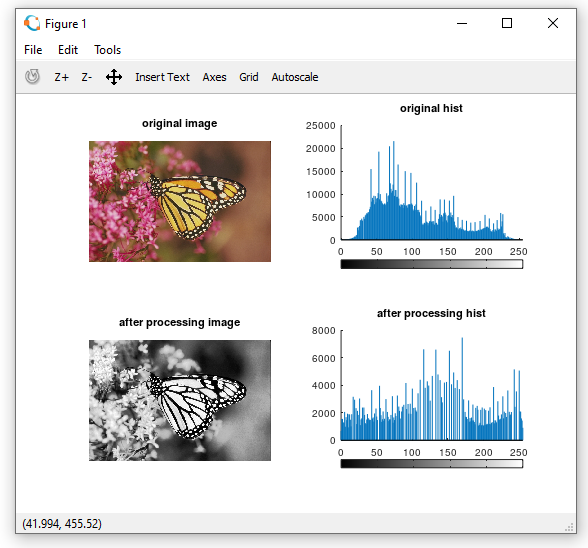
subplot(2,2,1), imshow(a); title('original image');

subplot(2,2,2), imhist(a); title('original hist');

subplot(2,2,3), imshow(ah); title('after processing image');

subplot(2,2,4), imhist(ah); title('after processing hist');

**Output:**



**Practical 4: Image filtering in Spatial Domain**

**1. a. Lowpass Average inbuilt**

**imnoise****():**

Add noise to an image

Syntax

J = imnoise(I,type)

J = imnoise(I,type,parameters)

**Description**

J = imnoise(I,type) adds noise of a given type to the intensity image I. type is a string that can have one of these values.

|  |  |
| --- | --- |
| Value | Description |
| 'gaussian' | Gaussian white noise with constant mean and variance |
| 'localvar' | Zero-mean Gaussian white noise with an intensity-dependent variance |
| 'poisson' | Poisson noise |
| 'salt & pepper' | On and off pixels |
| 'speckle' | Multiplicative noise |

**filter2():**

Perform two-dimensional linear filtering.

Y = filter2 (B, X)

Y = filter2 (B, X, SHAPE)

Apply the 2-D FIR filter B to X.

If the argument SHAPE is specified, return an array of the desired shape. Possible values are:

"full"

pad X with zeros on all sides before filtering.

"same"

unpadded X (default)

"valid"

trim X after filtering so edge effects are no included.

**Code:**

clear all;

close all;

pkg load image;

a=imread('hawk1.png');

#a=rgb2gray(a);

#imwrite(a,'hawk1.png');

a=im2double(a);

r=imnoise(a,'salt & pepper' );

f=ones(3,3)/9;

af=filter2(f,r);

figure

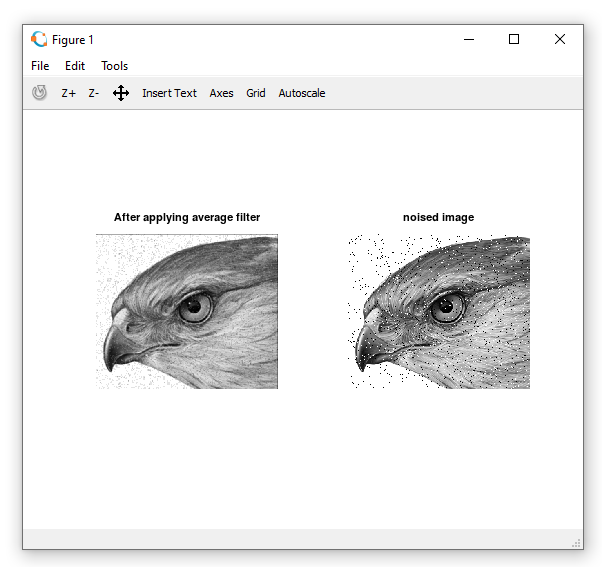
#imshow(a); title('original');

subplot(1,2,1);imshow(af); title('After applying average filter');

subplot(1,2,2)

imshow(r); title('Noised image');

**Output:**



**1. b. Lowpass Average not inbuilt**

floor():

**Y = floor([X](https://in.mathworks.com/help/matlab/ref/floor.html" \l "bug_5mn-1-X))** rounds each element of X to the nearest integer less than or equal to that element.

**Y = floor([t](https://in.mathworks.com/help/matlab/ref/floor.html" \l "bug_5mn-1-t))** rounds each element of the duration array t to the nearest number of seconds less than or equal to that element.

**Y = floor([t](https://in.mathworks.com/help/matlab/ref/floor.html" \l "bug_5mn-1-t),[unit](https://in.mathworks.com/help/matlab/ref/floor.html" \l "bug_5mn-1-unit))** rounds each element of t to the nearest number of the specified unit of time less than or equal to that element.

**Code:**

close all;

pkg load image;

im=imread('hawk1.png'); % To read image

#f=rgb2gray(CIm); % To convert RGB to Grayimage

Nim=imnoise(im,'salt & pepper'); % Adding salt & pepper noise to image

w=(1/16)\*[1 2 1;2 4 2;1 2 1]; % Defining the box filter mask

% get array sizes

[ma, na] = size(Nim)

[mb, nb] = size(w)

% To do convolution

c = zeros( ma+mb-1, na+nb-1 );

size\_c=size(c)

for i = 1:mb

for j = 1:nb

r1 = i

r2 = r1 + ma - 1

c1 = j

c2 = c1 + na - 1

c(r1:r2,c1:c2) = c(r1:r2,c1:c2) + w(i,j) \* (Nim);

end

end

% extract region of size(a) from c

r1 = floor(mb/2) + 1;

r2 = r1 + ma - 1;

c1 = floor(nb/2) + 1;

c2 = c1 + na - 1;

c = c(r1:r2, c1:c2);

figure

subplot(1,2,1)

imshow(Nim);

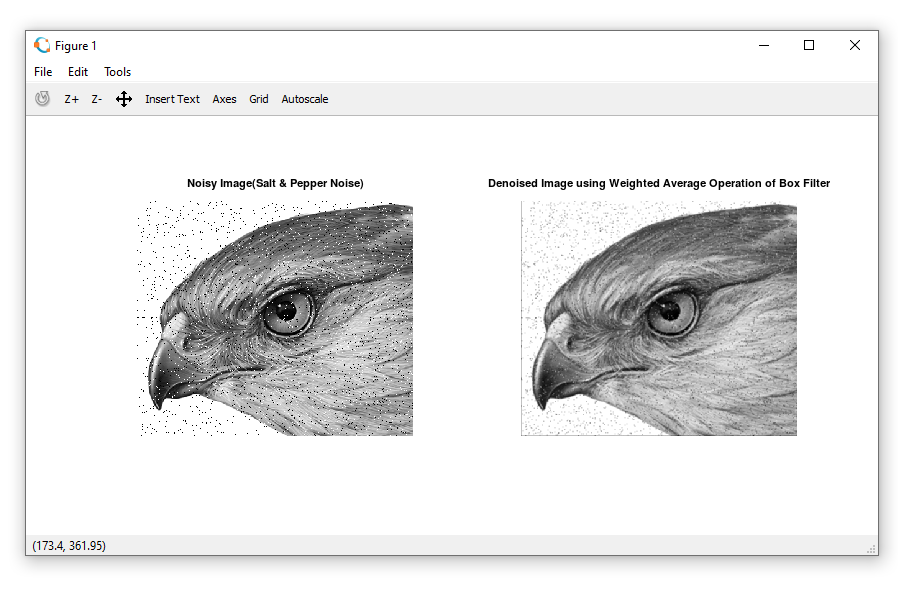
title('Noisy Image(Salt & Pepper Noise)');

subplot(1,2,2)

imshow(uint8(c));

title('Denoised Image using Weighted Average Operation of Box Filter');

**Output:**



**1. c. Low Pass Filter- (Median Filter)**

**sort****():**  
Sort array elements in ascending or descending order

Syntax

B = sort(A)

B = sort(A,dim)

B = sort(...,mode)

[B,IX] = sort(...)

**Description**

B = sort(A) sorts the elements along different dimensions of an array, and arranges those elements in ascending order.

|  |  |
| --- | --- |
| If A is a ... | sort(A) ... |
| Vector | Sorts the elements of A. |
| Matrix | Sorts each column of A. |
| Multidimensional array | Sorts A along the first non-singleton dimension, and returns an array of sorted vectors. |
| Cell array of strings | Sorts the strings in ASCII dictionary order. |

**Code:**

# Median Spatial Domain Filtering

pkg load image;

# Read the image

a=imread('hawk1.png');

img\_noisy1=imnoise(a,'salt & pepper' );

# Obtain the number of rows and columns of the image

[m, n] = size(img\_noisy1)

# Traverse the image. For every 3X3 area,

# find the median of the pixels and

# replace the center pixel by the median

img\_new1 = zeros(m, n);

for i=2: m-1

for j =2: n-1

temp = [img\_noisy1(i-1, j-1),

img\_noisy1(i-1, j),

img\_noisy1(i-1, j + 1),

img\_noisy1(i, j-1),

img\_noisy1(i, j),

img\_noisy1(i, j + 1),

img\_noisy1(i + 1, j-1),

img\_noisy1(i + 1, j),

img\_noisy1(i + 1, j + 1)] ;

temp = sort(temp);

img\_new1(i, j)= temp(4);

endfor

endfor

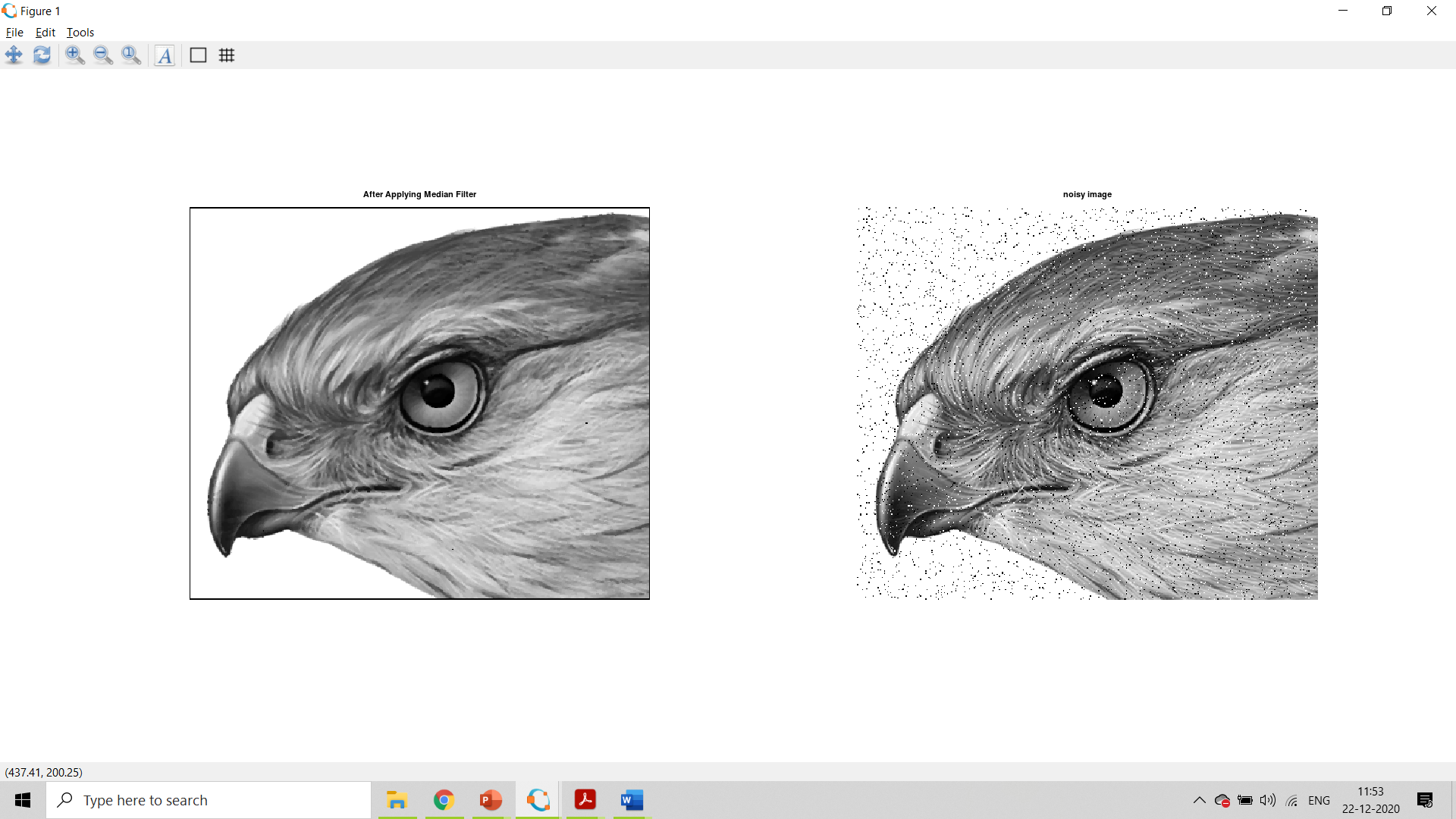
img\_new1 = uint8(img\_new1);

figure

subplot(1,2,1); imshow(img\_new1); title('After Applying Median Filter');

subplot(1,2,2); imshow(img\_noisy1);title('noisy image');

**Output:**



**2. a. Second order derivative – The Laplacian Filter**

padarray():

**[B](https://in.mathworks.com/help/images/ref/padarray.html" \l "d122e210839) = padarray([A](https://in.mathworks.com/help/images/ref/padarray.html" \l "d122e210575),[padsize](https://in.mathworks.com/help/images/ref/padarray.html" \l "d122e210625))** pads array A with an amount of padding in each dimension specified by padsize. The padarray function pads numeric or logical images with the value 0 and categorical images with the category <undefined>. By default, paddarray adds padding before the first element and after the last element of each dimension.

uint8():

8-bit unsigned integer arrays

y = uint8(10);

**Code:**

%Input Image

clear all;

A=imread('coins.png');

size(A);

figure,

subplot(2,2,1);imshow(A); title('original Image');

%Preallocate the matrices with zeros

I1=A;

I=zeros(size(A));

I2=zeros(size(A));

%Filter Masks

F1=[0 2 0;2 -8 2; 0 2 0];

#F2=[1 1 1;1 -8 1; 1 1 1];

%Padarray with zeros

A=padarray(A,[1,1]);

A=double(A);

size(A);

%Implementation of the equation in Fig.D

for i=1:size(A,1)-2

for j=1:size(A,2)-2

I(i,j)=sum(sum(F1.\*A(i:i+2,j:j+2)));

end

end

I=uint8(I);

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

%Sharpenend Image

B=I1-I;

subplot(2,2,4); imshow(B);title('Sharpened Image');

**Output:**



**2. b. First Order Derivative -Sobel Operator for edge detection without using edge function**

**abs():**  
Absolute value and complex magnitude

Syntax

Y = abs(X)

**Description**

abs(X) returns an array Y such that each element of Y is the absolute value of the corresponding element of X.

sqrt():

Square root

**Syntax**

B = sqrt(X)

Description

B = sqrt(X) returns the square root of each element of the array X. For the elements of X that are negative or complex, sqrt(X) produces complex results.

**Code:**

clear all;

A=imread(‘coins.png');

figure,

subplot(1,2,1); imshow(A); title('Original');

C=double(A);

size(C)

for i=1:size(C,1)-2

for j=1:size(C,2)-2

%Sobel mask for x-direction:

Gx=((C(i+2,j)+2\*C(i+2,j+1)+C(i+2,j+2))-(C(i,j)+2\*C(i,j+1)+C(i,j+2)));

%Sobel mask for y-direction:

Gy=((C(i,j+2)+2\*C(i+1,j+2)+C(i+2,j+2))-(C(i,j)+2\*C(i+1,j)+C(i+2,j)));

%The gradient of the image

# B(i,j)=abs(Gx)+abs(Gy);

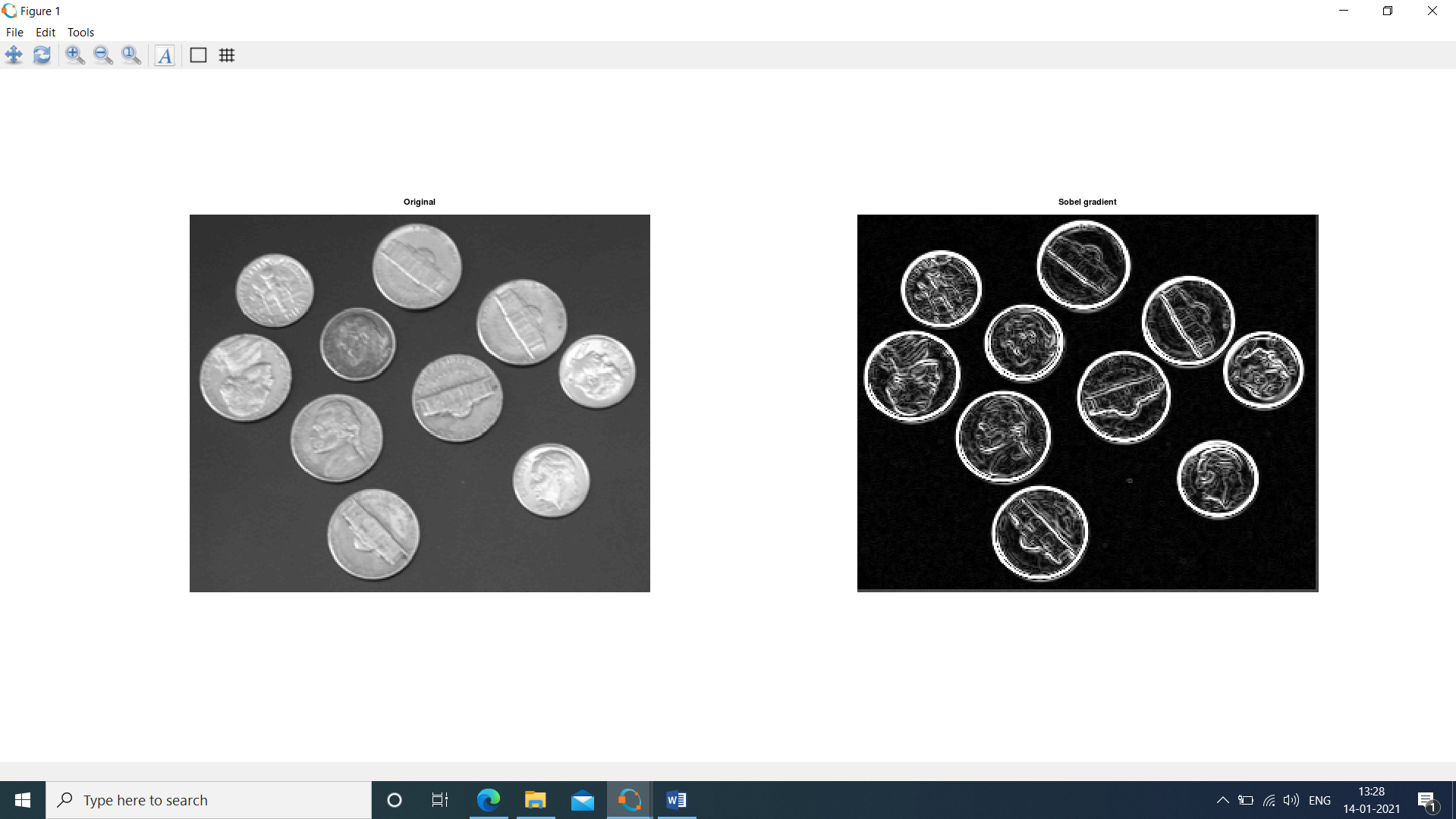
A(i,j)=sqrt(Gx.^2+Gy.^2);

end

end

subplot(1,2,2); imshow(A); title('Sobel gradient');

**Output:**

****

**2. c. First Order Derivative -Sobel Operator for edge detection using edge function**

**Code:**

#load package of image

pkg load image;

#Take input image

img=imread("peppers.png");

#function to find edge using sobel filter

sobel = edge(img,'Sobel');

figure 1,

subplot(1,2,1)

imshow(img);

title('Original Image');

subplot(1,2,2)

imshow(sobel);

title("Edge detection using sobel filter");

#function to find edge using sobel filter

robert = edge(img,'Roberts');

prewitt = edge(img,'Prewitt');

figure 2,

subplot(1,2,1)

imshow(robert);

title('Edge detection using robert filter');

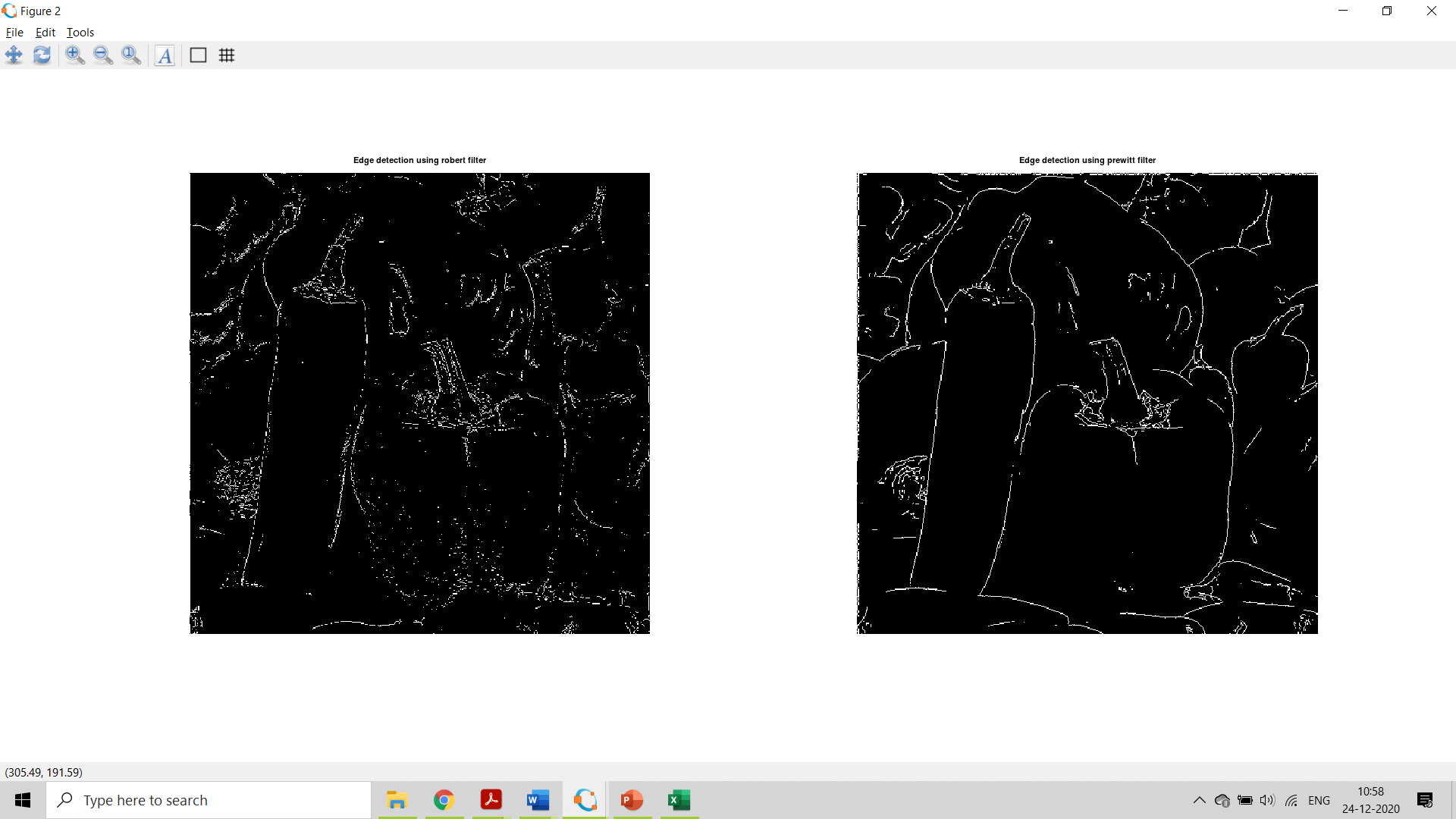
subplot(1,2,2)

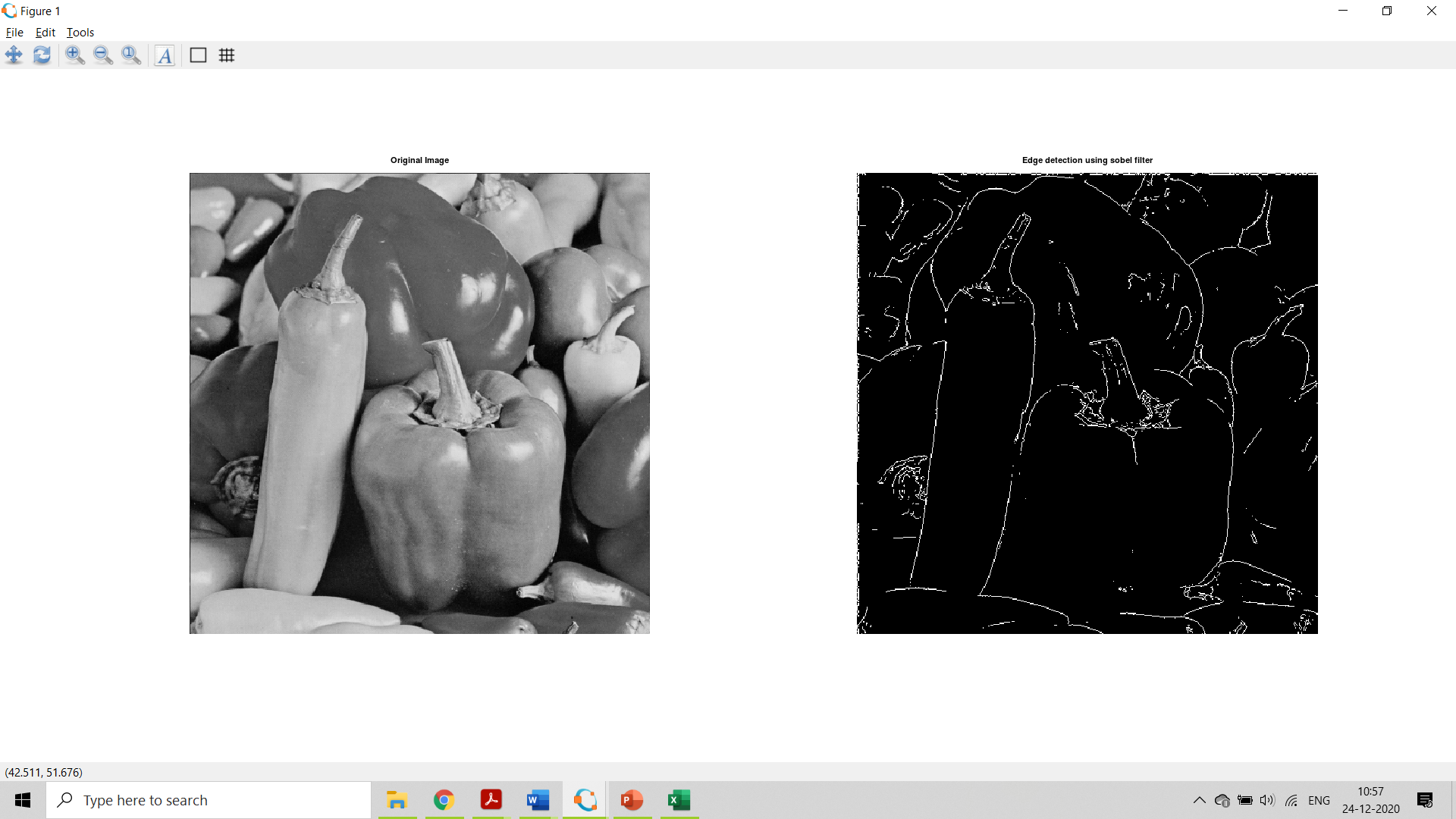
imshow(prewitt);

title("Edge detection using prewitt filter");

**Output:**







**Practical 6: Image Color Image Processing**

**1. a. Pseudocoloring**

**colormap():**

the purpose of colormap is to define the colors of the graphics objects like image, surface and patch objects. A colormap is basically a matrix with values between 0 & 1. Colormaps can have any length, but width-wise they must have 3 columns. Each row of the matrix defines one color by using an RGB triplet.

**Code:**

pkg load image; close all; clear all;

%READ INPUT IMAGE

A = imread('coins.png');

%RESIZE IMAGE

A = imresize(A,[256 256]);

%PRE-ALLOCATE THE OUTPUT MATRIX

Output = ones([size(A,1) size(A,2)]);

%COLORMAPS

#maps={'jet(256)';'hsv(256)';'cool(256)';'spring(256)';'summer(256)';'parula(256)';'hot(256)'};

%COLORMAP 1

map = colormap(jet(256));

Red = map(:,1);

Green = map(:,2);

Blue = map(:,3);

R1 = Red(A);

G1 = Green(A);

B1 = Blue(A);

%COLORMAP 2

map = colormap(cool(256));

Red = map(:,1);

Green = map(:,2);

Blue = map(:,3);

%RETRIEVE POSITION OF UPPER TRIANGLE

[x,y]=find(triu(Output)==1);

Output(:,:,1) = Red(A);

Output(:,:,2) = Green(A);

Output(:,:,3) = Blue(A);

for i=1:numel(x)

Output(x(i),y(i),1)=R1(x(i),y(i));

Output(x(i),y(i),2)=G1(x(i),y(i));

Output(x(i),y(i),3)=B1(x(i),y(i));

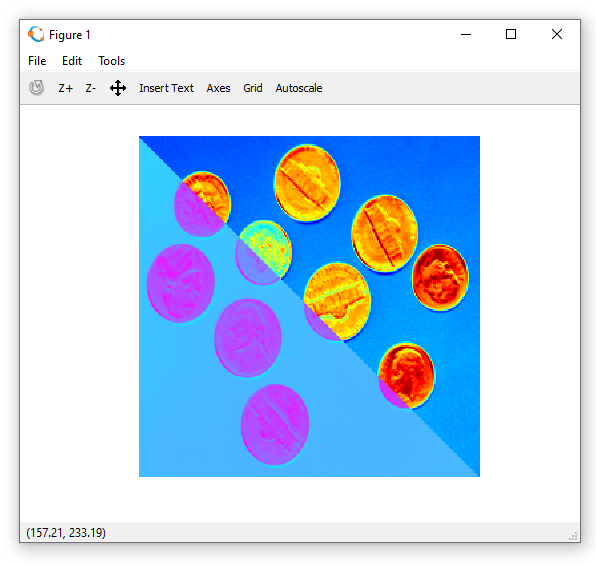
end

Output = im2uint8(Output);

%FINAL IMAGE

imshow(Output);

**Output:**



**1. b. Pseudocoloring**

**Code:**

pkg load image;

clear all;

img = imread('hawk1.png'); % Read image

figure, imshow(img);title("original Image");

red = 0.66\*img;

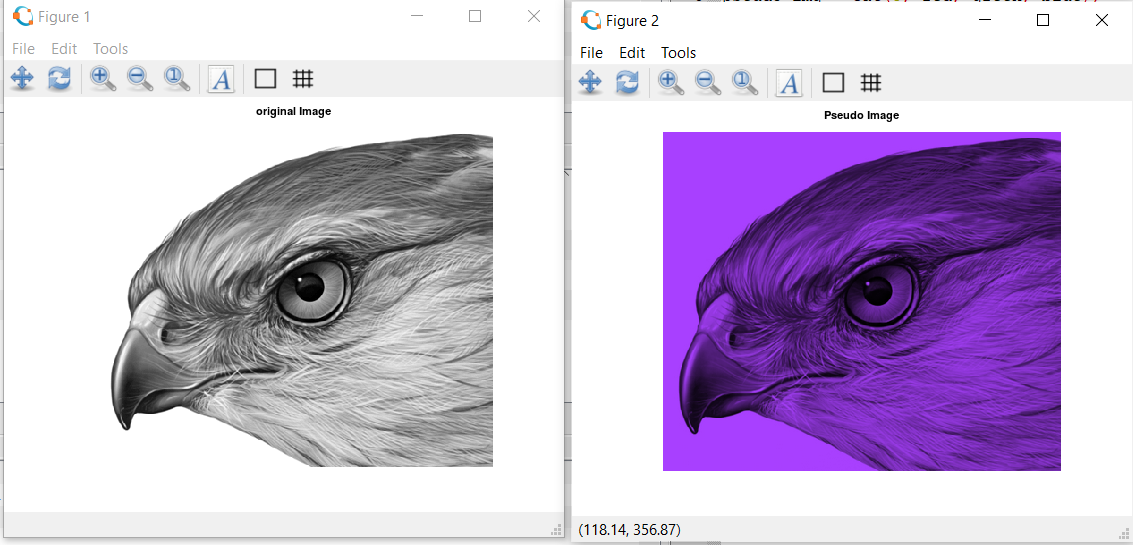
green=0.25\*img;

blue = img;

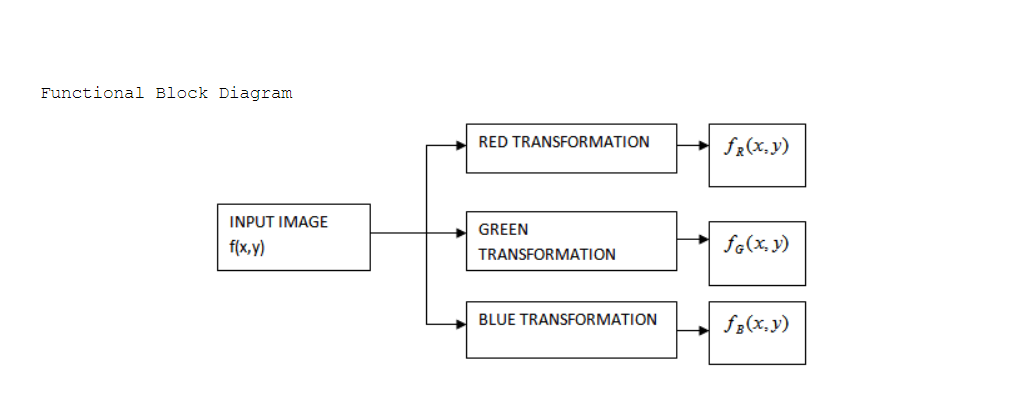
pseudo\_img = cat(3, red, green, blue);

figure, imshow(pseudo\_img), title('Pseudo Image');

**Output:**

****

2. **Separating the RGB Channels**

****

**Code:**

#Separate RGB Channels from the Image

clear all;

close all;

pkg load image;

img1 = imread('airplane.png');

imshow(img1)

img\_r=img1;

img\_r(:,:,2)=0;

img\_r(:,:,3)=0;

figure(1),

imshow(img\_r);

img\_g=img1;

img\_g(:,:,1)=0;

img\_g(:,:,3)=0;

figure(2),

imshow(img\_g);

img\_b=img1;

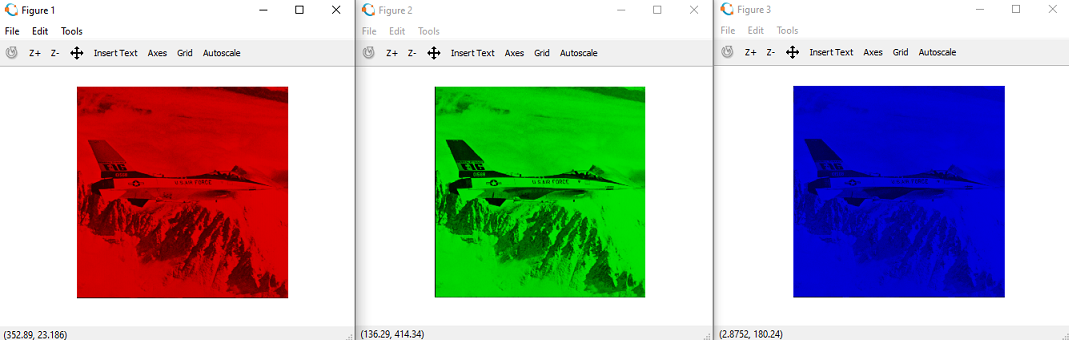
img\_b(:,:,1)=0;

img\_b(:,:,2)=0;

figure(3),

imshow(img\_b);

**Output:**



3. **Color Intensity Slicing**

**Code:**

pkg load image;

clear all;

close all;

#im=input('Enter the file name);

input\_image=imread('hawk.png');

k=rgb2gray(input\_image);

[x y z]=size(k);

% z should be one for the input image

k=double(k);

for i=1:x

for j=1:y

if k(i,j)>=0 && k(i,j)<50

m(i,j,1)=k(i,j,1)+25;

m(i,j,2)=k(i,j)+50;

m(i,j,3)=k(i,j)+60;

end

if k(i,j)>=50 && k(i,j)<100

m(i,j,1)=k(i,j)+55;

m(i,j,2)=k(i,j)+68;

m(i,j,3)=k(i,j)+70;

end

if k(i,j)>=100 && k(i,j)<150

m(i,j,1)=k(i,j)+52;

m(i,j,2)=k(i,j)+30;

m(i,j,3)=k(i,j)+15;

end

if k(i,j)>=150 && k(i,j)<200

m(i,j,1)=k(i,j)+50;

m(i,j,2)=k(i,j)+40;

m(i,j,3)=k(i,j)+25;

end

if k(i,j)>=200 && k(i,j)<=256

m(i,j,1)=k(i,j)+120;

m(i,j,2)=k(i,j)+60;

m(i,j,3)=k(i,j)+45;

end

end

end

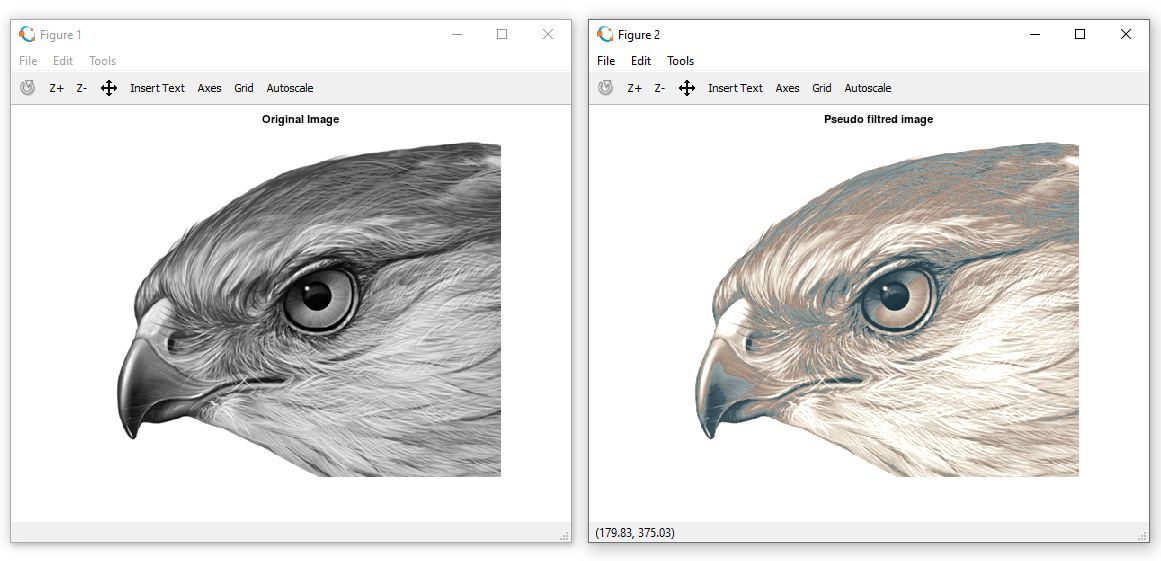
figure, imshow(uint8(k),[]);

title('Original Image');

figure, imshow(uint8(m),[]);

title("Pseudo filtred image");

**Output:**



**Practical 7: Image Compression Techniques and watermarking**

**1. Implement Huffman Coding**

**repmat():**

Repmat command repeats the elements of an array in output. Repetition is depended on parameter list, therefore every time we need to declare parameters within a bracket after repmat command.

**Syntax:**

Repmat(number,number of times)

**Huffmandict():**

Generate Huffman code dictionary for source with known probability model.

SYNTAX:

**[[dict,avglen] = huffmandict(symbols,prob)](https://in.mathworks.com/help/comm/ref/huffmandict.html" \l "d122e50782)**

huffmandict([symbols](https://in.mathworks.com/help/comm/ref/huffmandict.html" \l "d122e50944),[prob](https://in.mathworks.com/help/comm/ref/huffmandict.html" \l "mw_8454f6ac-0e06-4914-8780-b4a060166c01)) generates a binary Huffman code dictionary, dict, for the source symbols, symbols, by using the maximum variance algorithm. The input prob specifies the probability of occurrence for each of the input symbols. The length of prob must equal the length of symbols. The function also returns average codeword length avglen of the dictionary, weighted according to the probabilities in the input prob.

**Huffmanenco():**

Encode sequence of symbols by Huffman encoding

SYNTAX:

**[code = huffmanenco(sig,dict)](https://in.mathworks.com/help/comm/ref/huffmanenco.html" \l "d122e51176)**

[code](https://in.mathworks.com/help/comm/ref/huffmanenco.html" \l "mw_3101491a-0082-4ca6-bb8e-f41a3552c432) = huffmanenco([sig](https://in.mathworks.com/help/comm/ref/huffmanenco.html" \l "d122e51319),[dict](https://in.mathworks.com/help/comm/ref/huffmanenco.html" \l "mw_8375f27d-df67-434b-8b66-111d4bef63ce)) encodes input signal sig using the Huffman codes described by input code dictionary dict. sig can have the form of a vector, cell array, or alphanumeric cell array. If sig is a cell array, it must be either a row or a column. dict is an N-by-2 cell array, where N is the number of distinct possible symbols to encode. The first column of dict represents the distinct symbols and the second column represents the corresponding codewords. Each codeword is represented as a row vector, and no codeword in dict can be the prefix of any other codeword in dict. You can generate dict using the [huffmandict](https://in.mathworks.com/help/comm/ref/huffmandict.html) function.

Huffmandeco():

Decode binary code by Huffman decoding.

SYNTAX:

**[sig = huffmandeco(code,dict)](https://in.mathworks.com/help/comm/ref/huffmandeco.html" \l "d122e50478)**

[sig](https://in.mathworks.com/help/comm/ref/huffmandeco.html" \l "mw_f6897bf7-fbfb-47f4-a731-9d7df8e07a11) = huffmandeco([code](https://in.mathworks.com/help/comm/ref/huffmandeco.html" \l "d122e50635),[dict](https://in.mathworks.com/help/comm/ref/huffmandeco.html" \l "mw_7cf75e06-46a8-425c-85b9-c3dfc90fd379)) decodes the numeric Huffman code vector, code, by using the Huffman codes described by input code dictionary dict. Input dict is an N-by-2 cell array, where N is the number of distinct possible symbols in the original signal that encodes code. The first column of dict represents the distinct symbols, and the second column represents the corresponding codewords. Each codeword is represented as a numeric row vector, and no codeword in dict can be the prefix of any other codeword in dict. You can generate dict by using the [huffmandict](https://in.mathworks.com/help/comm/ref/huffmandict.html) function and code by using the [huffmanenco](https://in.mathworks.com/help/comm/ref/huffmanenco.html) function. If all symbols in dict are numeric, output sig is a vector. If any symbol in dict is alphabetic, sig is a one-dimensional cell array.

de2bi():

Convert decimal numbers to binary vectors.

SYNTAX:

**[b = de2bi(d)](https://in.mathworks.com/help/comm/ref/de2bi.html" \l "d122e26788)**

 converts a nonnegative decimal integer d to a binary row vector. If d is a vector, the output b is a matrix in which each row is the binary form of the corresponding element in d.

**numel():**

Number of elements in array or subscripted array expression

**Syntax**

n = numel(A)

n = numel(A,varargin)

**Description**

n = numel(A) returns the number of elements, n, in array A.

n = numel(A,varargin) returns the number of subscripted elements, n, in A(index1,index2,...,indexn), where varargin is a cell array whose elements are index1, index2, ..., indexn.

**Code:**

pkg load communications;

sig = repmat([3 3 1 3 3 3 3 3 2 3],1,50);

symbols = [1 2 3];

p = [0.1 0.1 0.8];

dict = huffmandict(symbols,p);

hcode = huffmanenco(sig,dict);

dhsig = huffmandeco(hcode,dict);

isequal(sig,dhsig);

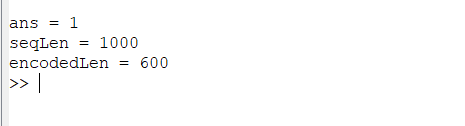
binarySig = de2bi(sig);

seqLen = numel(binarySig)

binaryhcode = de2bi(hcode);

encodedLen = numel(binaryhcode);

**Output:**



**2. Add a watermark to the image**

Digital image watermarking is the method in which data is embedded in a multimedia file such as an image or a video, so as to verify the credibility of the content or the identity of the owner.

**Imresize():**

Resize image

**[B](https://in.mathworks.com/help/matlab/ref/imresize.html" \l "d122e653113) = imresize([A](https://in.mathworks.com/help/matlab/ref/imresize.html" \l "d122e652513),[scale](https://in.mathworks.com/help/matlab/ref/imresize.html" \l "d122e652552))** returns image B that is scale times the size of A. The input image A can be a grayscale, RGB, or binary image. If A has more than two dimensions, imresize only resizes the first two dimensions. If scale is in the range [0, 1], B is smaller than A. If scale is greater than 1, B is larger than A. By default, imresize uses bicubic interpolation.

**Code:**

pkg load image;

clear all;

close all;

#Input Image where we want to apply watermark

f=imread('bandon.tif');

#For watermarking, size of inputimage and watermarking image should be same

#there for we changed the size of image using imresize and dispalyed

fr=imresize(f,[560 560]);

figure;imshow(fr);title('Original Image with resized value');

#Watermarking Image

w=imread('water.jpg');

#Again Resized the Watermarking Image

wr=imresize(w,[560 560]);

figure;imshow(wr);title('Watermark');

#Applied watermarking

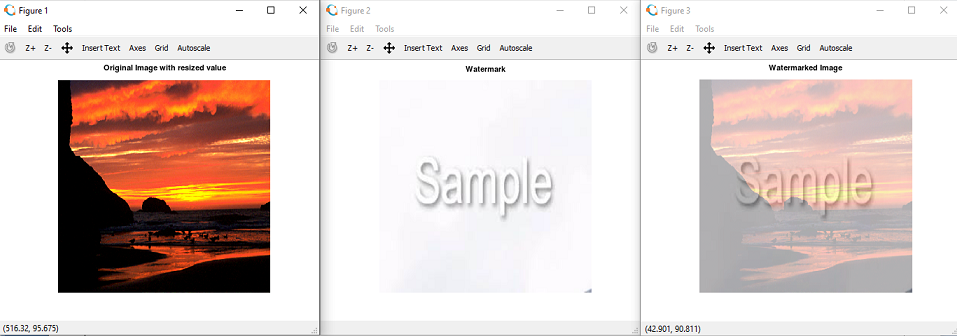
alpha=0.7;

fw=(1-alpha)\*fr + alpha.\*wr;

#Display the watermarked Image

figure;imshow(fw);title('Watermarked Image');

**Output:**



**Practical 8: Implement Basic Morphological Transformations**

**1. Boundary Extraction**

Extracting the boundary is the important process to gain the information and understand the feature of an image. Boundary extraction is the first process in preprocessing in order to present the features of the image. This process can help the researcher to gain the data from the image.

Boundary Extraction in Octave

Let A be an Image matrix and B be a structuring element.

**Formula for Boundary Extraction:**

[https://4.bp.blogspot.com/-g1tGnauY5y8/Twf_yHjEIXI/AAAAAAAAAjY/OXzENpKPf3Q/s320/boundaryextraction.JPG](https://4.bp.blogspot.com/-g1tGnauY5y8/Twf_yHjEIXI/AAAAAAAAAjY/OXzENpKPf3Q/s1600/boundaryextraction.JPG)

Steps to be followed:

* Convert the image into binary image.
* Perform Erosion:

Erode binary image A by structuring element B. (i.e) [https://4.bp.blogspot.com/-v4qgB6iEtpI/TwgAmv0pAJI/AAAAAAAAAjo/ZQXkn4BEmVk/s1600/boundaryextraction1.JPG](https://4.bp.blogspot.com/-v4qgB6iEtpI/TwgAmv0pAJI/AAAAAAAAAjo/ZQXkn4BEmVk/s1600/boundaryextraction1.JPG)

* Subtraction:  
                Subtract the binary image A from the Eroded image.(i.e) [https://2.bp.blogspot.com/-dZADA-5teOg/TwgBSPv_I2I/AAAAAAAAAjw/kaLeNc8bkFc/s1600/boundaryextraction2.JPG](https://2.bp.blogspot.com/-dZADA-5teOg/TwgBSPv_I2I/AAAAAAAAAjw/kaLeNc8bkFc/s1600/boundaryextraction2.JPG)

**imdialate():**

Dialates the image.

**[J](https://in.mathworks.com/help/images/ref/imdilate.html" \l "d122e123531) = imdilate([I](https://in.mathworks.com/help/images/ref/imdilate.html" \l "d122e123262),[SE](https://in.mathworks.com/help/images/ref/imdilate.html" \l "d122e123305))**

 dilates the grayscale, binary, or packed binary image I, returning the dilated image, J. SE is a structuring element object or array of structuring element objects, returned by the [strel](https://in.mathworks.com/help/images/ref/strel.html) or [offsetstrel](https://in.mathworks.com/help/images/ref/offsetstrel.html) functions.

**imerode():**

erodes the image.

**[J](https://in.mathworks.com/help/images/ref/imerode.html" \l "d122e125766) = imerode([I](https://in.mathworks.com/help/images/ref/imerode.html" \l "d122e125474),[SE](https://in.mathworks.com/help/images/ref/imerode.html" \l "d122e125517))** erodes the grayscale, binary, or packed binary image I, returning the eroded image, J. SE is a structuring element object or array of structuring element objects, returned by the [strel](https://in.mathworks.com/help/images/ref/strel.html) or [offsetstrel](https://in.mathworks.com/help/images/ref/offsetstrel.html) functions.

**Code:**

pkg load image;

clear all;

close all;

A=imread('pepper.jpg');

C=rgb2gray(A);

C(C<225)=0;

s=strel('disk',4,0);%Structuring element

D=~im2bw(C);%binary Image

F=imerode(D,s);%Erode the image by structuring element

figure,imshow(A);title('Original Image');

figure,imshow(D);title('Binary Image');

%Difference between binary image and Eroded image

figure,imshow(D-F);title('Boundary extracted Image');

**Output:**



**2. Thinning and Thickening**

**bwmorph():**

Morphological operations on binary images

SYNTAX:

**[BW2 = bwmorph(BW,operation)](https://in.mathworks.com/help/images/ref/bwmorph.html" \l "d122e24066)**

Operations can be ‘skel’ ,’thin’ ,’thicken’, ‘fill’, etc.

**Code:**

pkg load image;

clear all;

close all;

org\_im=imread("Hit\_Miss.png");

subplot(2,2,1),

imshow(org\_im);title("Original image");

binary=im2bw(org\_im);

subplot(2,2,2),

imshow(binary);title("Binary image");

thin=bwmorph(binary,'thin');

subplot(2,2,3),

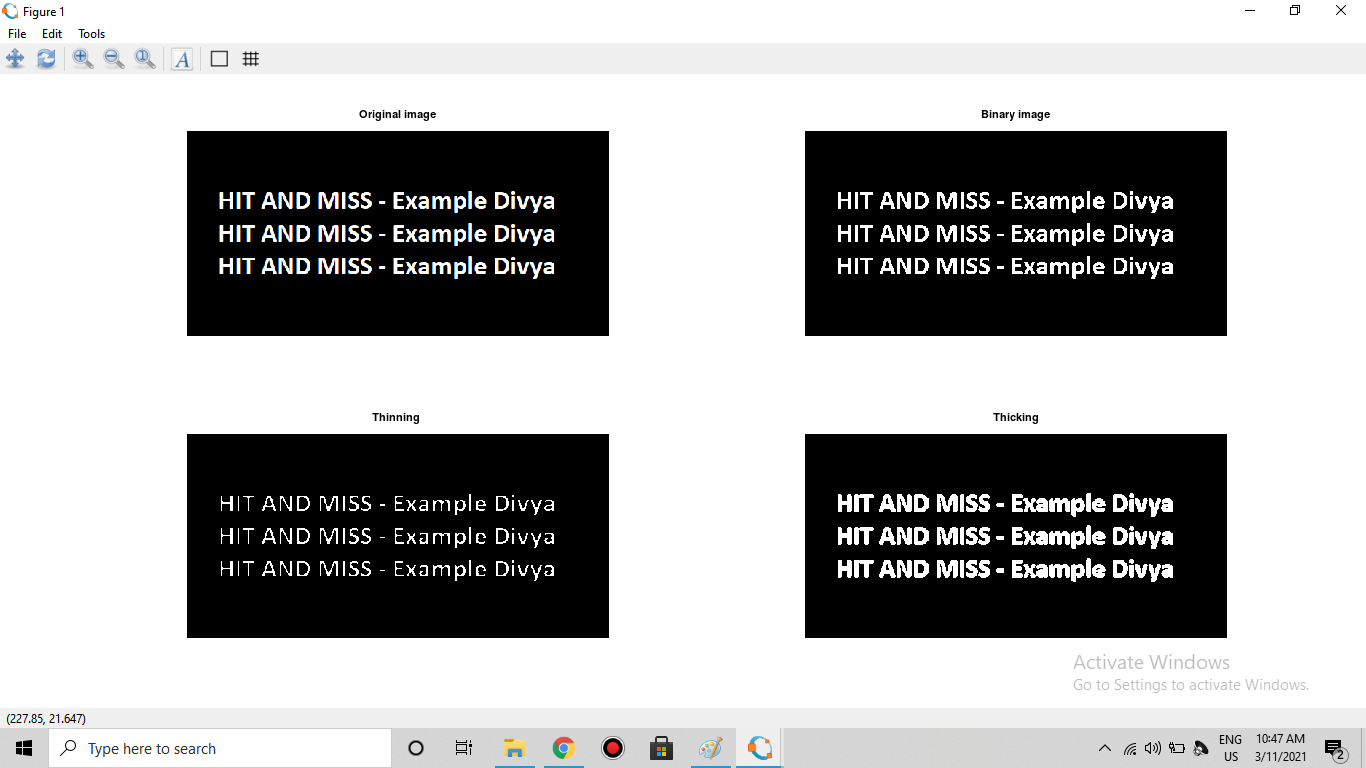
imshow(thin);title("Thinning ");

thick=bwmorph(binary,'thicken');

subplot(2,2,4),

imshow(thick);title("Thicking");

**Output:**



**3. Hole filling and Skeletons**

**Code:**

pkg load image;

clear all;

close all;

#read input image

org\_im=imread('coins.png');

#convert RGB image to binary image and display as input image

A=im2bw(org\_im);

subplot(2,2,1)

#figure 1;

imshow(A); title('Binary Image');

hole\_filled=bwfill(A,'holes');

subplot(2,2,2),

#figure 2;

imshow(hole\_filled); title('Hole Filled Image');

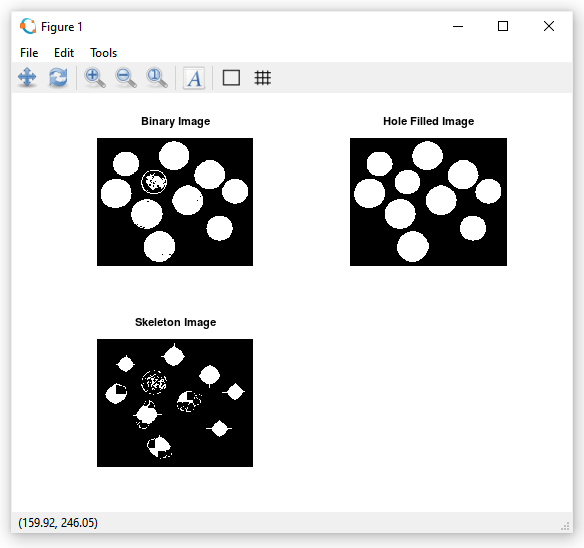
skel=bwmorph(A,'skel',8);

#figure 3;

subplot(2,2,3),

imshow(skel);title("Skeleton Image");

**Output:**

****