LAB - 01

Aim (1_1): List of Commands Learned For Image Processing:

1). clear

- This command will clear the all the declared variables from the memory.
- It also clears or removes all the saved variables from the workplace.

```
>> clear
>>
```

2). clc

- This command will clear whole previously executed command history from the command window.
- Although this command not removes the history of the command executed.

```
>> clc

>>
```

3). %

- This is used to do line comment.

```
>> %
>> disp("<this is digital image processing>")
<this is digital image processing>
```

5). help <Command Name>

- After writing the command name after the help you can retrieve all the information about the given command.
- With all the different version of command with different types of arguments.

```
>> help plot
 plot - 2-D line plot
    This MATLAB function creates a 2-D line plot of the data in Y versus the
    corresponding values in X.

    Vector and Matrix Data
      plot(X,Y)
      plot(X,Y,LineSpec)
      plot(X1,Y1,...,Xn,Yn)
      plot(X1,Y1,LineSpec1,...,Xn,Yn,LineSpecn)
      plot(Y)
      plot(Y,LineSpec)

    Table Data
      plot(tbl,xvar,yvar)
      plot(tbl,yvar)

    Additional Options
      plot(ax,___)
      plot(___,Name,Value)
      p = plot(___)

     Examples
       Create Line Plot
       Plot Multiple Lines
       Create Line Plot From Matrix
       Specify Line Style
       Specify Line Style, Color, and Marker
       Display Markers at Specific Data Points
       Specify Line Width, Marker Size, and Marker Color
       Add Title and Axis Labels
       Plot Durations and Specify Tick Format
       Plot Coordinates from a Table
       Plot Multiple Table Variables on One Axis
       Specify Axes for Line Plot
       Modify Lines After Creation
       Plot Circle

     See also title, xlabel, ylabel, xlim, ylim, legend, hold, gca, yyaxis,
       plot3, loglog, Line

     Introduced in MATLAB before R2006a
     Documentation for plot
     Other uses of plot
```

6). log(<number>)

- This will return the logarithmic value of the given number.

```
>> log(94)

ans =

    4.5433
```

7). ones(<Different Types of arguments>)

- This command is used to define unit matrix for the given dimensions.
- Here different type of argument defines the size of the unit matrix.
- 1 argument – This will define square matrix of given argument.
- More than one number of arguments defines unit matrix.

```
>> ones(2,3)

ans =

     1     1     1
     1     1     1
```

8). zeros(<Different Types of arguments>)

- This command is used to define zero matrix for the given dimensions.
- Here different type of argument defines the size of the zero matrix.
- 1 argument – This will define square matrix of given argument.
- More than one number of arguments defines zero matrix.

```
>> zeros(2,2)

ans =

     0     0
     0     0
```

9). eye(<Different Types of arguments>)

- This command is used to define identity matrix for the given dimensions.
- Here different type of argument defines the size of the identity matrix.
- 1 argument – This will define square matrix of given argument.
- More than one number of arguments defines identity matrix.

```
>> eye(4,4)

ans =

     1     0     0     0
     0     1     0     0
     0     0     1     0
     0     0     0     1
```

10). imread(<Absolute Path of the Image/Relative Path From Current Folder/URL of the image>)

- This function is used to read the image from the local storage or using the URL of the image.
- This function can be used without semi-colon and with semi-colon;
- If we use semi-colon at the it will read the image but not show the output matrix for the image immediately.
- If we do not use semi-colon at the it will read the image but show the output.
- matrix for the image immediately. This can slower the speed of the command execution if the image is larger.

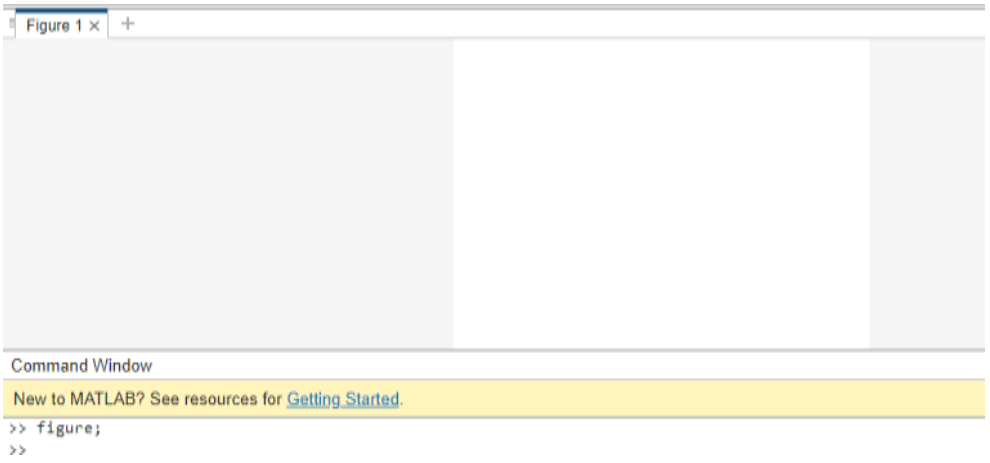
11). Imshow(<image variable returned using any image method or imread function>)

- We can show the image which is passed as a variable of the image.

- Performed in lab1_2

12). figure

- Using this figure command, we can open new image on new tab.
- If we do not use figure and apply more than one imshow() this is overwriting on same tab instead of opening a new image.



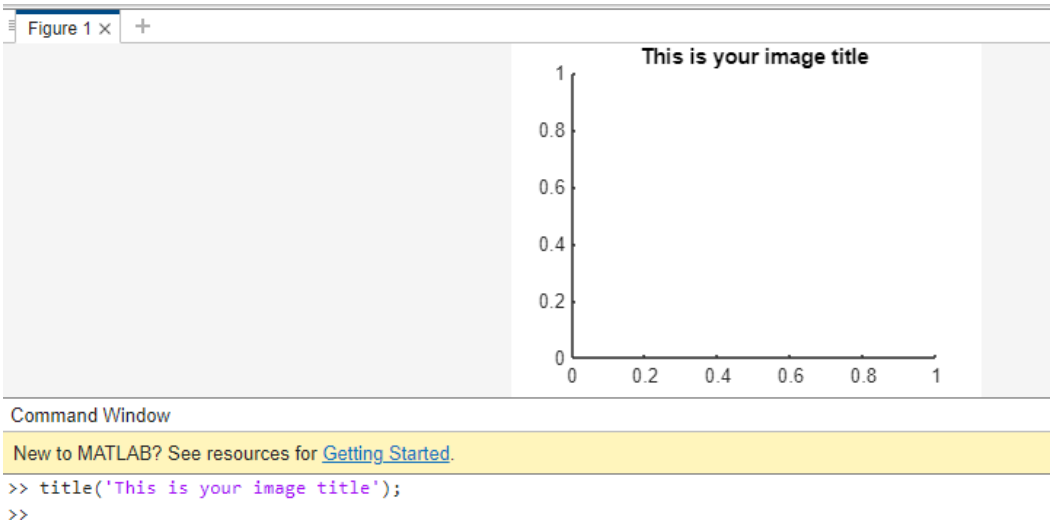
13). close all

- This command can close all the current tabs which are opened before executing this command.

```
>> figure;

>> close all;
>>
```

14). title(<Your Image Title String>)

- This can give title string to the tab while using the imshow()function.



15). im2bw(<image variable returned using any image method or imread function>,<threshold value>)

- This function can convert the image variable to black & white image.
- With the use of threshold value, you can control brightness level of the black & white image.
- Threshold value must be between 0 to 1.

16). imwrite(<Variable of the modified image>,<Path And The name of the file>)

- With the use of this command we can save our images on local computer.
- Here 1 st variable is modified image variable.
- 2 nd variable is defining the path where we want to store our image.

17). imresize(<Variable of the image>,<New Required Size of the modified image>)

- Here our 1 st variable states the image we want to resize.
- Here we need to pass the new size of image required.

18). size(<variable of the image>)

- This function return the size of the image passed as the first argument.

19). subplot(x,y,i)

- Here our one tab is divided in x*y part.
- Here i shows the number of part we need to access from the x*y division.

20). imfinfo(<File Name With Absolute Path of the File/File name with the relative path of the file from current folder>)

- This function returns the image file info with all the parameters.

21). rgb2gray (<image variable returned using any image method or imread function>)

- This function can convert the rgb image to grey scale image.

The above commands are used in lab1_2.

22). Convert One Variable type to Other Variable Type

- The default data type is double
- The default data type of image is uint8
- To change data type (i.e. p = uint8(5))

```
>> p = uint8(5);
>> p_double = double(p);
>> p_int16 = int16(100);
```

Name	Value	Size	Class
p	5	1×1	uint8
p_double	5	1×1	double
p_int16	100	1×1	int16

23). Define 1-D Array

- Declare Array -> a = [1 2 3 4]
- Array Index Start from 1.
- We can use, or [space] to separate two elements in the array.
- Element can be altered using command window and workspace both.
- You can do basic operation on arrays directly (Condition - Size of array should be same).
- For element wise basic operation use period (.) before your operator -> a(1) .* b(1)

Name	Value	Size	Class
a	[1,5,3]	1×3	double
b	[4,5,6]	1×3	double
c	[5,10,9]	1×3	double
element...	[4,25,18]	1×3	double
first_ele...	1	1×1	double

```
>> a = [1 2 3];
>> a = [1, 2, 3];
>> first_element = a(1);
>> a(2) = 5;
>> b = [4 5 6];
c = a + b;
>> elementwise_mult = a .* b;
>>
```

24). Define 2-D Array

- Defining the two-dimensional Array
- b = [1 2 3
  4 5 6
  7 8 9]

  Or

  b = [1 2 3;4 5 6;7 8 9]

Workspace

Name	Value	Size	Class
a	[1,5,3]	1×3	double
b	[1,2,3;4,5,6;7,10,9]	3×3	double
c	[5,10,9]	1×3	double
col3	[3;6;9]	3×1	double
element	6	1×1	double
elementwise_mult	[4,25,18]	1×3	double
submatrix	[2,3;5,6]	2×2	double

```
>> b = [1 2 3;
     4 5 6;
     7 8 9];
>> element = b(2,3);
>> row2 = b(2,:);
>> col3 = b(:,3);
>> submatrix = b(1:2, 2:3);
>> b(3,2) = 10;
>>
```

25). max(<variable name>)

- This command will give us the maximum from the given array.
- Here if we are taking multidimensional array this function will return max and give output of less one dimension as a output.

max_A	[7,9,5]	1×3	double	`>> A = [1 3 5; 7 9 2];` `max_A = max(A);`

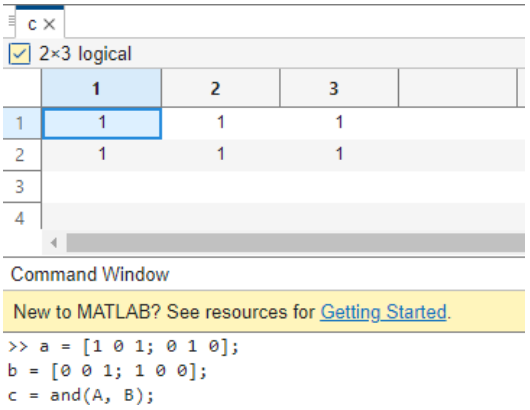
26). min(<variable name>)

- This command will give us the minimum from the given array.
- Here if we are taking multidimensional array this function will return min and give output of less one dimension as a output.

min_B	[1,3,2]	1×3	double	`>> B = [1 3 5; 7 9 2];` `min_B = min(B);`

27). and (<First Variable>,<Second Varible>)

- This function is used to do bitwise and operation of the first image with the Second image.

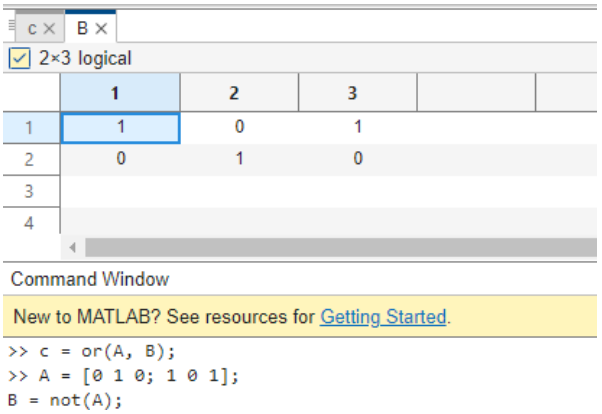


28). or (<First Variable>,<Second Varible>)

- This function is used to do bitwise or operation of the first image with the Second image.

29). not (<Image Variable>)

- This function is used to do inverse value of the pixel.
- This function converts the 0 in 1 and 1 in 0.



30). im2double(<Variable>)

- This function is used to convert your given variable to double data type.

31). Writing External Function In Octave:

- Using this we can write function externally which can be called from the any other program.
- This helps us to reduce the rewriting of our code.
- Systex:
  function <return type> = <function name>(<Arguments>);
  <Function Code>
  endfunction
- Here some constraints while writing the user generated function and also while using them.
- Function name and file name should be same.
- If user generated function is not in the same directory than we need to import that file before use.
- We can also write function in same file and use it multiple time as and when we need it.

32). sin(<angle>):

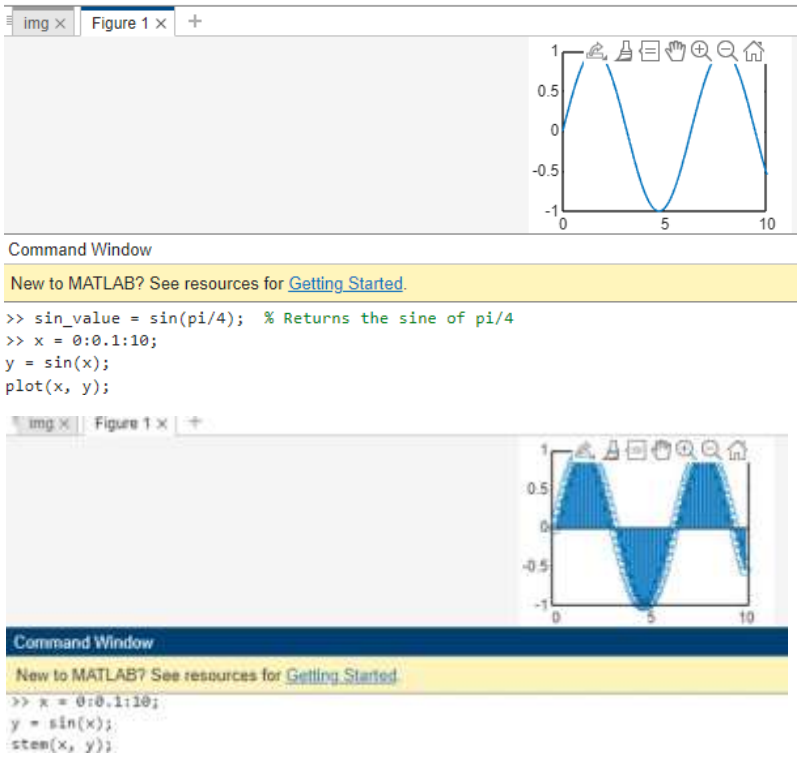
- This function is used to return the sin function of the given angle.
- This function will return the sin value for the given angle.

33). plot(<x axis value or funtion>,<y axis value or funtion>):

- This function is used to plot the graph.
- Here first and second parameters shows the value and function of the x-axis and y-axis.
- This is the linear graph function.

34). stem(<x axis value or funtion>,<y axis value or funtion>):

- This function is used to plot the graph.



- Here first and second parameters shows the value and function of the x-axis and y-axis.
- This is the point wise graph function. This graph shows the intencity of the each pixel intensity in the given image or variable or function.

35). imhist(<variable or function>) / hist(<variable or function>) :

- This function is used to plot the graph.
- Here we only need to pass the function or the variable of the image to this function. Inbuilt function will automatically plot the graph for the given function or image variable.
- This function will plot linear graph.

36). strel(shape, parameters):

- Create a strel(structuring element) object for morphology operations.
- The structuring element can have any type of shape.

37). imdilate(im, SE) | imdilate(im, SE, shape) & imerode(im, SE) | imerode(im, SE, shape):

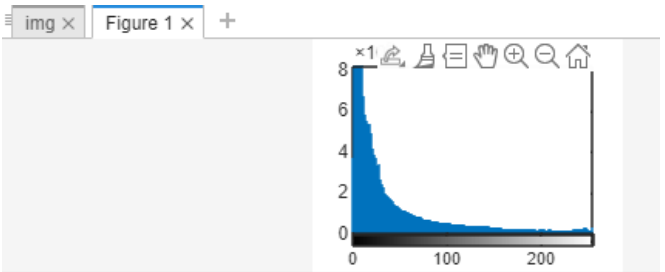
- Perform morphological dilation.
- The dilation is performed with the structuring element se which can be a:
  - Strel object
  - Array of strel objects as returned by @strel/getsequence
  - Matrix of 0's and 1's
- To perform a non-flat dilation, SE must be a strel object.

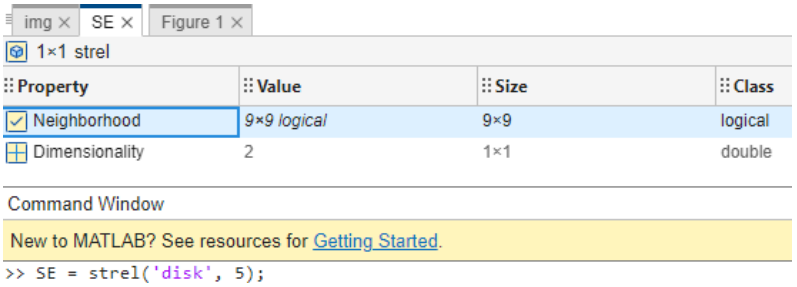
38). imopen(img, SE) | imdilate(imerode(img,se),se):

- Perform morphological opening.
- The matrix img must be numeric while SE can be a:
  - Strel object
  - Array of strel objects as returned by @strel/getsequence
  - Matrix of 0's and 1's

39). imclose(img, SE) | imerode(imdilate(img,se),se):

- Perform morphological closing.
- The matrix img must be numeric while SE can be a:





B	2×3 logical	2×3	logical
c	2×3 logical	2×3	logical
closed_img	3968×2976×3 u...	3968×2976×3	uint8
img	3968×2976×3 u...	3968×2976×3	uint8
img_double	3968×2976×3 d...	3968×2976×3	double
SE	1×1 strel	1×1	strel
sin_value	0.7071	1×1	double

```
>> img = imread('img.jpg');
SE = strel('disk', 5);
closed_img = imclose(img, SE);  % Perform morphological closing
>> img = imread('image.jpg');
SE = strel('disk', 5);
dilated_img = imdilate(img, SE);
eroded_img = imerode(img, SE);
```

Aim (1_2): Getting familiar with MATLAB and performing basic operations on image.

1. Create the following matrix A: A = [43 21 22 11
−5 6 34 −21
12 17 −18 42]

```
>> A = [43 21 22, 11; -5 6 34 -21; 12 17 -18 42];
>> A

A =

    43    21    22    11
    -5     6    34   -21
    12    17   -18    42
```

a) Create a four element row vector named va that contains the elements of the second row of A.

```
>> va = A(2,:)

va =

    -5     6    34   -21
```

b) Create a three element row vector named vb that contains the elements of the third column of A.

```
>> vb = A(:, 3)

vb =

    22
    34
   -18
```

c) Create an eight element row vector named vc that contains the elements of the first and third rows of A.

```
vc =

    43    21    22    11    12    17   -18    42
```

d) Create a six element row vector named vd that contains the elements of the second and fourth columns of A.

```
vd =

    21    22
     6    34
    17   -18
```

2. Create the following three matrices:

A = [5 2 4
2 −5 8
1 −3 −7]

B = [10 7 3
−11 5 8
4 −3 −7]

C = [6 9 −4
10 5 8
2 −3 7]

```
>> A = [5 2 4; 2 -5 8; 1 -3 -7];
>> A

A =

     5     2     4
     2    -5     8
     1    -3    -7

>> B = [10 7 3; -11 5 8; 4 -3 -7];
>> B

B =

    10     7     3
   -11     5     8
     4    -3    -7

>> C = [6 9 -4; 10 5 8; 2 -3 7];
>> C

C =

     6     9    -4
    10     5     8
     2    -3     7
```

Calculate A + B and B + A to show that addition of matrices is commutative.

```
>> A + B

ans =

    15     9     7
    -9     0    16
     5    -6   -14

>> B + A

ans =

    15     9     7
    -9     0    16
     5    -6   -14
```

Calculate A + (B + C) and (A + B) + C to show that addition of matrices is Associative.

```
>> (A + B) + C

ans =

    21    18     3
     1     5    24
     7    -9    -7

>> A + (B + C)

ans =

    21    18     3
     1     5    24
     7    -9    -7
```

Calculate 3(A + C) and 3A + 3C to show that, when matrices are multiplied by a scalar, the multiplication is associative.

```
>> 3*(A + C)

ans =

    33    33     0
    36     0    48
     9   -18     0

>> 3*A + 3*C

ans =

    33    33     0
    36     0    48
     9   -18     0
```

Calculate A * (B + C) and A * B + A * C to show that matrix multiplication is distributive.

```
>> A * (B + C)

ans =

   102    76    27
    85   -66   -82
   -23    28   -49

>> A * B + A * C

ans =

   102    76    27
    85   -66   -82
   -23    28   -49
```

3. Create an array A = [1 2 3 4 5 6] and using built in functions for array find

```
>> A = [1 2 3 4 5 6];
>> A

A =

     1     2     3     4     5     6
```

Length of A.

```
>> length(A)

ans =

     6
```

Average of the elements of A.

```
>> mean(A)

ans =

    3.5000
```

Maximum element of A.

```
>> max(A)

ans =

     6
```

Minimum element of A.

```
>> min(A)

ans =

     1
```

Sum of all the elements of A.

```
>> sum(A)

ans =

    21
```

4. Calculate:

$$\frac{3^7 \log 76}{7^3 + 546} + \sqrt[3]{910}$$

```
>> [(power(3, 7)* log(76)) / (power(7, 3) + 546)] + nthroot(910, 3)

ans =

   20.3444
```

Using the ones and zeros commands, create a 4 x 6 matrix in which the first two rows are 0's and the next two rows are 1's.

```
>> zero_one = [zeros(2, 6); ones(2, 6)]

zero_one =

     0     0     0     0     0     0
     0     0     0     0     0     0
     1     1     1     1     1     1
     1     1     1     1     1     1
```

IMAGE PROCESSING TOOLBOX IN MATLAB:-

Source code:

```
img = imread('img.jpg');
subplot(2,3,1);
imshow(img);
imwrite(img,'img.jpg');
subplot(2,3,2);
imshow('img.jpg');
resize_img = imresize(img,[210,220]);
subplot(2,3,3);
crop_img = imcrop(img);
imshow(crop_img);
img_info = imfinfo('img.jpg');
```

2. Take your own photo (RGB image) and create the following images and save them for future use

Source code:

```
img = imread("paris.jpg");
imshow(img);
rgb_to_gray = rgb2gray(img);
imshow(rgb_to_gray);
bw_img = im2bw(img);
imshow(bw_img);
over_exposed_img = img + 50;
imshow(over_exposed_img);
under_exposed_img = img - 50;
imshow(under_exposed_img);
resize_img = imresize(img, [256, 256]);
imshow(resize_img);
```

a) Gray scale image

Black and white image

Over exposed and Under exposed

Resize the image to 256 x 256 pixels

bw_img	626x626 logical
img	626x626x3 uint8
over_exposed...	626x626x3 uint8
resize_img	256x256x3 uint8
rgb_to_gray	626x626 uint8
under_expose...	626x626x3 uint8

3. Take you own photo and process them for following results
a. Flip your image vertically
b. Create the mirror image
c. Rotate the image by 90 degrees.
d. Rotate the image by 270 degrees.

Source code:

```
img = imread("tulip.jpg");
imshow(img);
flip_img_ver = flip(img, 1);
imshow(flip_img_ver);
mirror_img = flip(img, 2);
imshow(mirror_img);
im90 = imrotate(img, 90);
imshow(im90);
im270 = imrotate(img, 270);
imshow(im270);
crop = imcrop(img);
imshow(crop);
```

Summarised learning:

In this exercise, I explored MATLAB commands for image processing, such as converting an image to grayscale, and manipulating exposure to create overexposed, underexposed, mirror, rotated images. These techniques provided foundational knowledge of preprocessing and analyzing images in various applications.

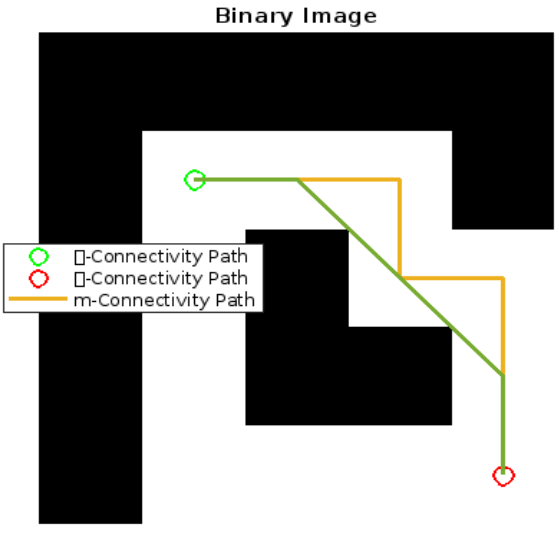
LAB - 02

AIM: Perform the following tasks.

1. Find a path between any two given points in image using 4, 8, and m - path process.

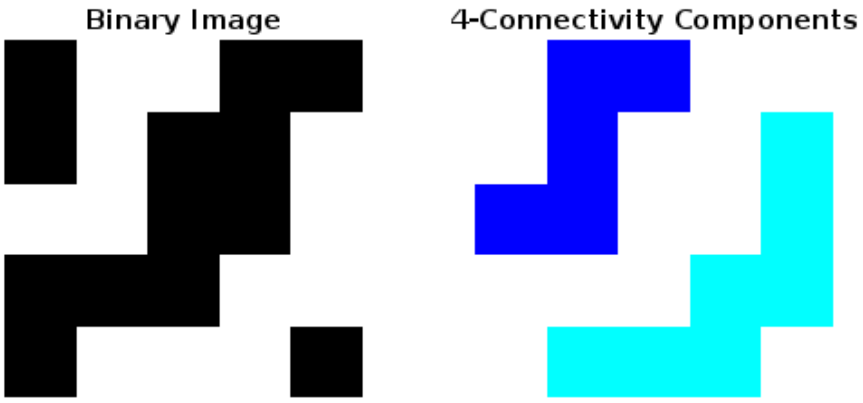
```
>> % Pathfinding in a Binary Image
clear; clc; close all;
% Binary Image (1 = path, 0 = obstacle)
img = [0 0 0 0 0;
      0 1 1 1 0;
      0 1 0 1 1;
      0 1 0 0 1;
      0 1 1 1 1];
% Display the image
figure;
imshow(img, 'InitialMagnification', 'fit');
title('Binary Image');
hold on;
start_point = [2, 2]; % Row, Column (y, x)
end_point = [5, 5];   % Row, Column (y, x)
plot(start_point(2), start_point(1), 'go', 'MarkerSize', 10, 'LineWidth', 2);
plot(end_point(2), end_point(1), 'ro', 'MarkerSize', 10, 'LineWidth', 2);
% Connectivity types to evaluate
connectivities = [4, 8, 'm'];
% Loop through each connectivity and find the path
for i = 1:length(connectivities)
   connectivity = connectivities(i);
   path = find_path(img, start_point, end_point, connectivity);
   if ~isempty(path)
       plot(path(:, 2), path(:, 1), 'LineWidth', 2); % Plot path
       legend_info{i} = [num2str(connectivity), '-Connectivity Path'];
   else
       legend_info{i} = [num2str(connectivity), '-No Path Found'];
   end
end
legend(legend_info, 'Location', 'best');
hold off;
%% Function to Find Path
function path = find_path(img, start_point, end_point, connectivity)
   [rows, cols] = size(img);
   visited = zeros(size(img));
   queue = [start_point];
   visited(start_point(1), start_point(2)) = 1;
   parent = cell(rows, cols);
   if connectivity == 4
```

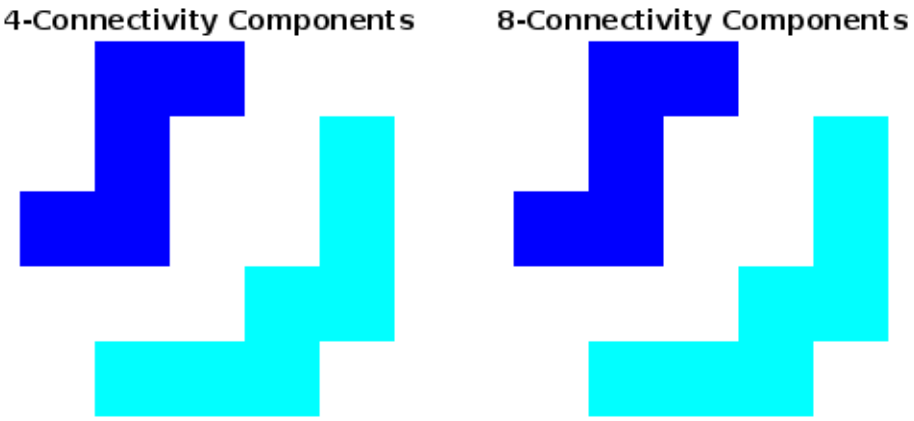
```
        directions = [0, 1; 1, 0; 0, -1; -1, 0]; % Right, Down, Left, Up
    elseif connectivity == 8 || connectivity == 'm'
        directions = [0, 1; 1, 1; 1, 0; 1, -1; 0, -1; -1, -1; -1, 0; -1, 1]; %
Diagonals included
    else
        error('Invalid connectivity');
    end
    while ~isempty(queue)
        current = queue(1, :);
        queue(1, :) = [];
        if isequal(current, end_point)
            break;
        end
        for d = 1:size(directions, 1)
            neighbor = current + directions(d, :);
            if neighbor(1) >= 1 && neighbor(1) <= rows && ...
               neighbor(2) >= 1 && neighbor(2) <= cols && ...
               img(neighbor(1), neighbor(2)) == 1 && ...
               visited(neighbor(1), neighbor(2)) == 0
                queue = [queue; neighbor];
                visited(neighbor(1), neighbor(2)) = 1;
                parent{neighbor(1), neighbor(2)} = current;
            end
        end
    end
    path = [];
    current = end_point;
    while ~isempty(current)
        path = [current; path];
        current = parent{current(1), current(2)};
    end
    if isempty(path) || ~isequal(path(1, :), start_point)
        path = [];
    end
end
```



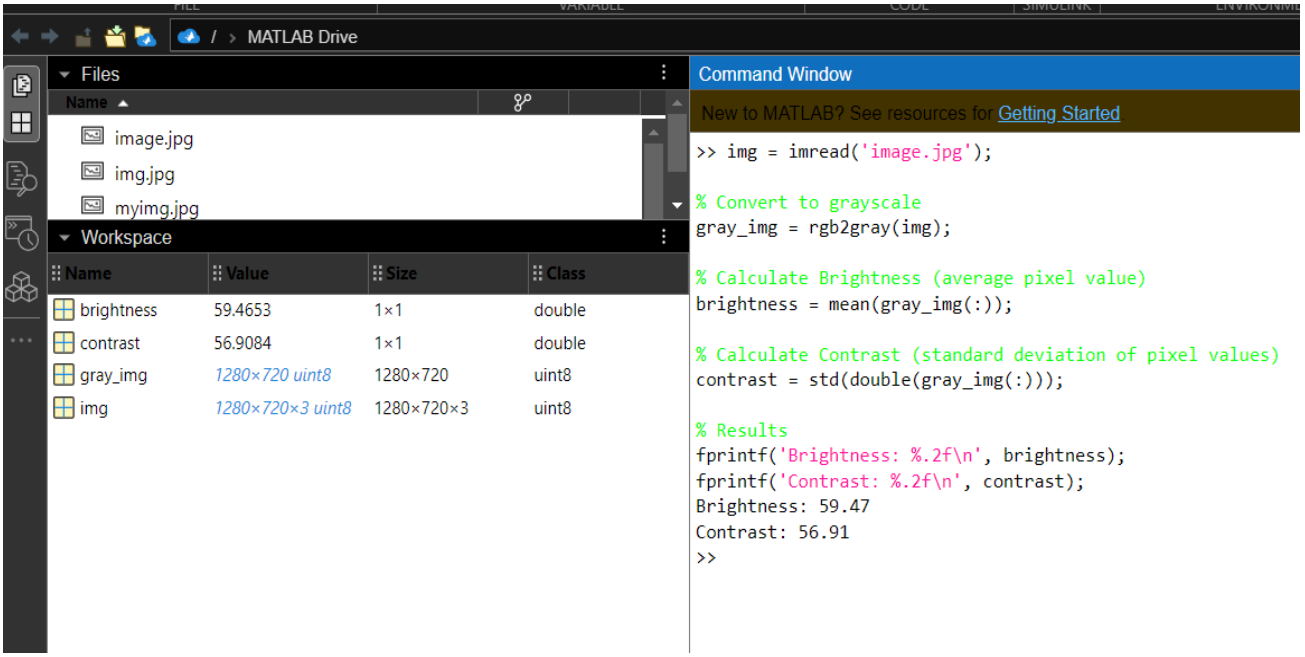
2. Find connected components using labelling process of 4 and 8 Connectivity.

```
>> % Connected Components using 4 and 8 Connectivity
clear; clc; close all;
% Binary Image
img = [0 1 1 0 0;
       0 1 0 0 1;
       1 1 0 0 1;
       0 0 0 1 1;
       0 1 1 1 0];
figure;
subplot(1, 2, 1);
imshow(img, 'InitialMagnification', 'fit');
title('Binary Image');
% Label connected components using 4-connectivity
cc4 = bwlabel(img, 4);
subplot(1, 2, 2);
imshow(label2rgb(cc4));
title('4-Connectivity Components');
% Display results
figure;
subplot(1, 2, 1);
imshow(label2rgb(cc4));
title('4-Connectivity Components');
% Label connected components using 8-connectivity
cc8 = bwlabel(img, 8);
subplot(1, 2, 2);
imshow(label2rgb(cc8));
title('8-Connectivity Components');
```





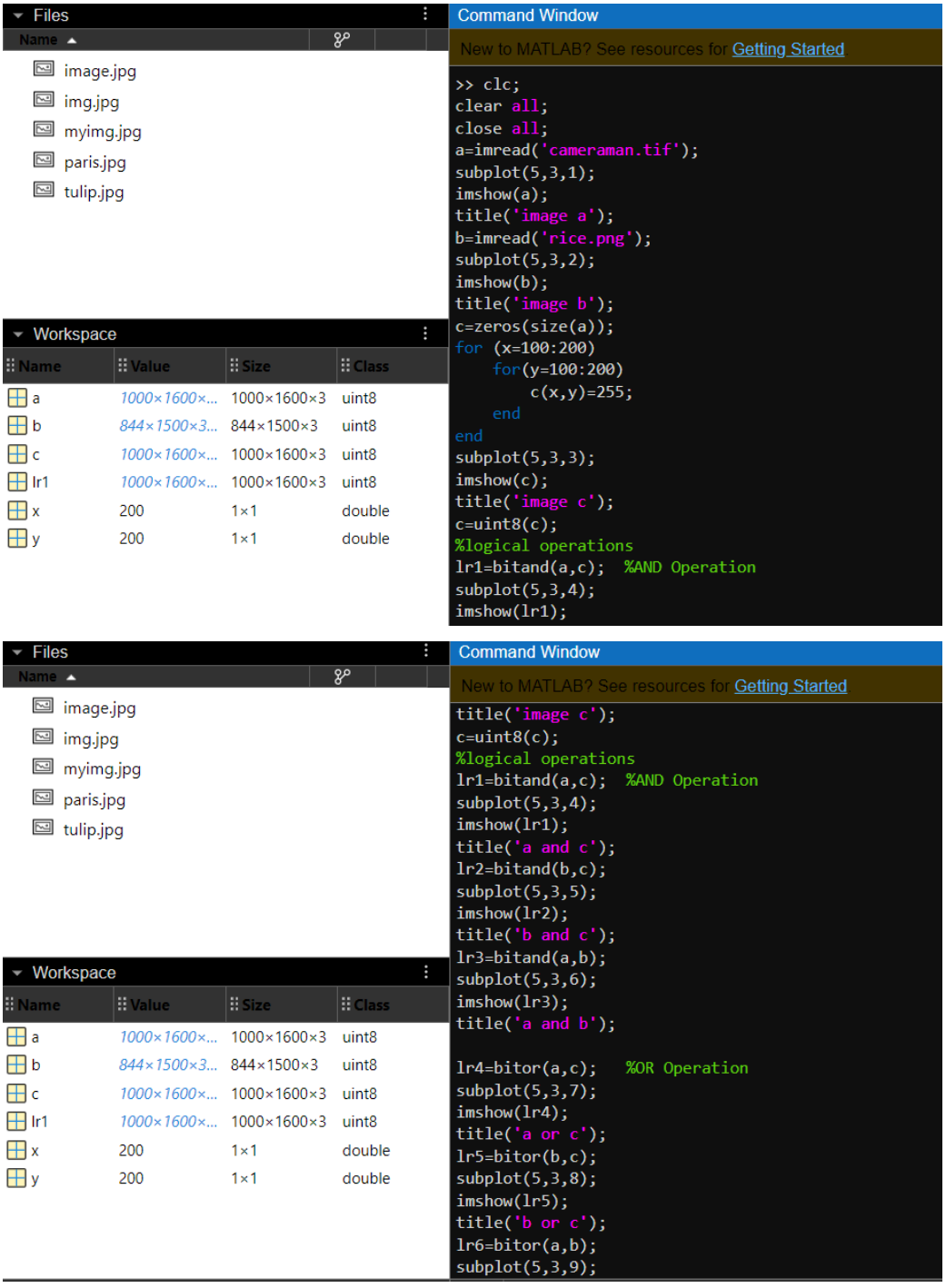
3. Calculate the brightness and contrast of images.

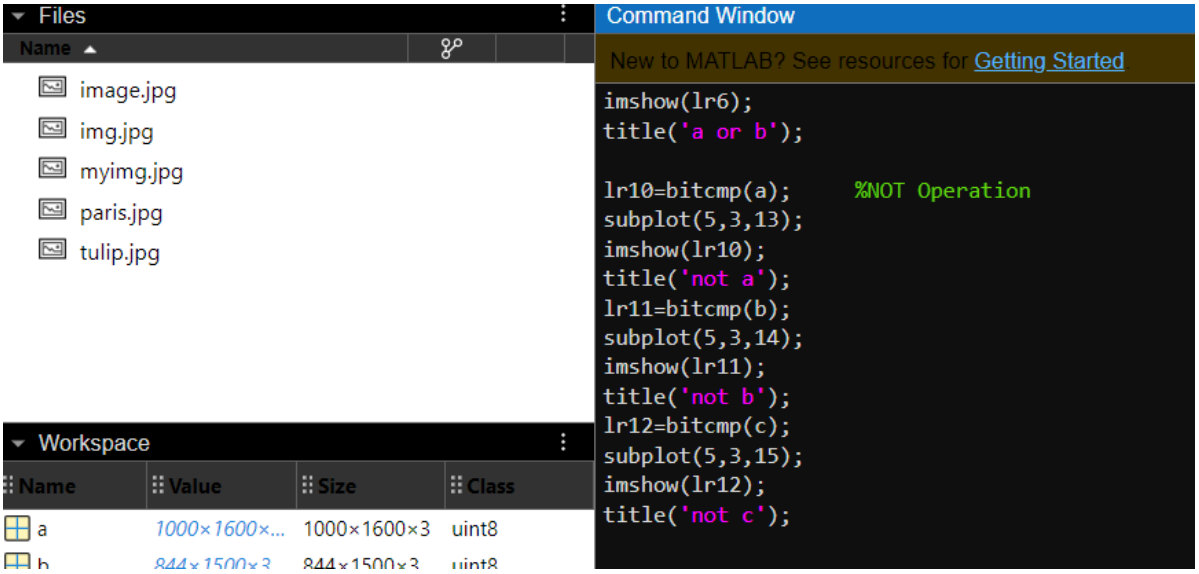


4. Perform AND, OR and NOT logical operations on the images.

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.




Files
Name
image.jpg
img.jpg
myimg.jpg
paris.jpg
tulip.jpg
Workspace
Name
Value
Size
Class
a
1000×1600×...
1000×1600×3
uint8
Command Window
New to MATLAB? See resources for Getting Started
imshow(lr6);
title('a or b');
lr10=bitcmp(a); %NOT Operation
subplot(5,3,13);
imshow(lr10);
title('not a');
lr11=bitcmp(b);
subplot(5,3,14);
imshow(lr11);
title('not b');
lr12=bitcmp(c);
subplot(5,3,15);
imshow(lr12);
title('not c');

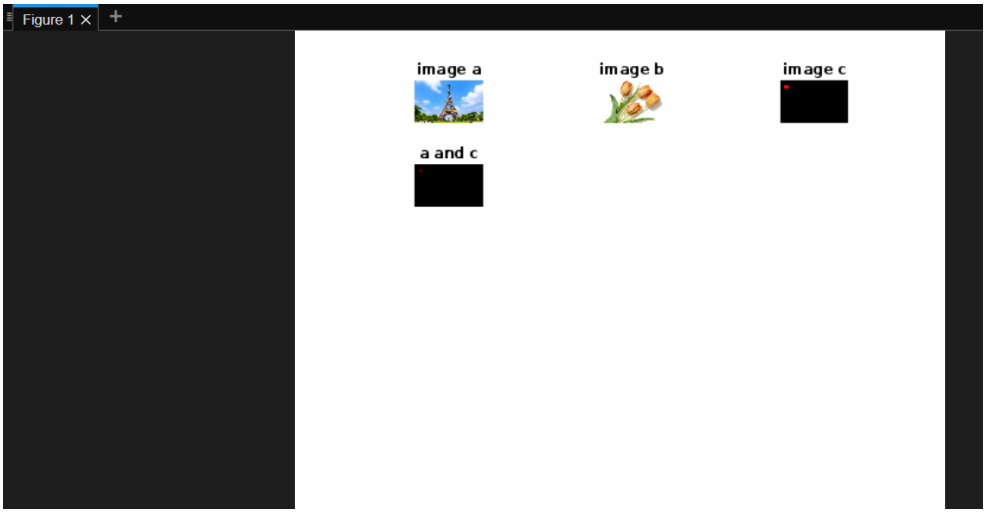


Figure 1
image a
image b
image c
a and c

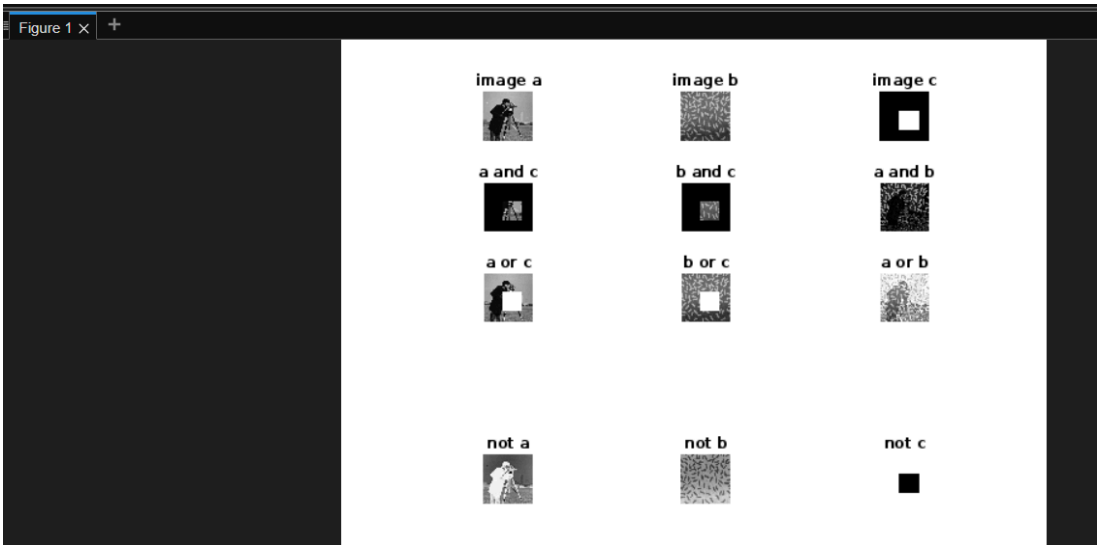


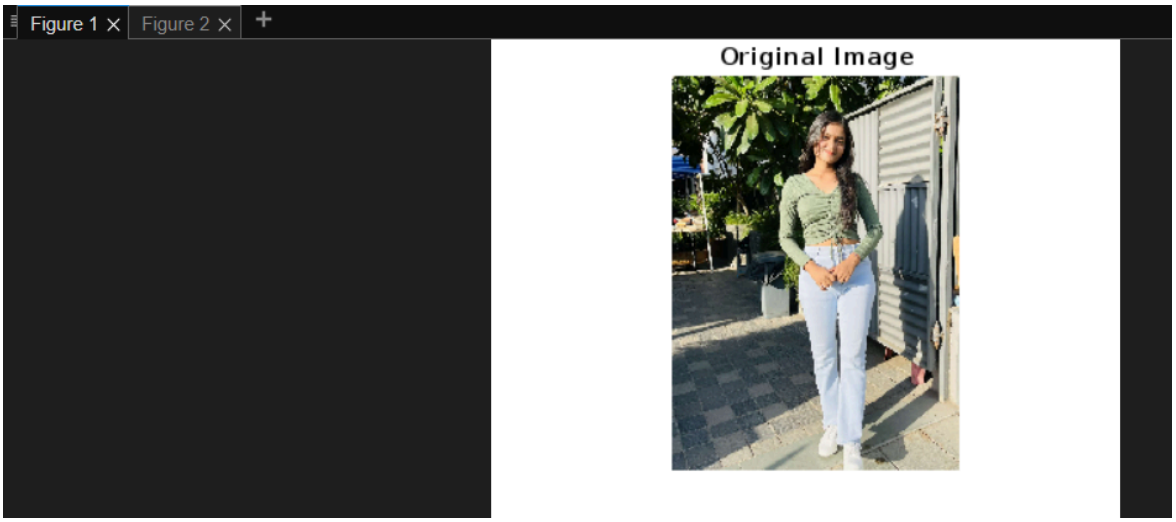
Figure 1
image a
image b
image c
a and c
b and c
a and b
a or c
b or c
a or b
not a
not b
not c

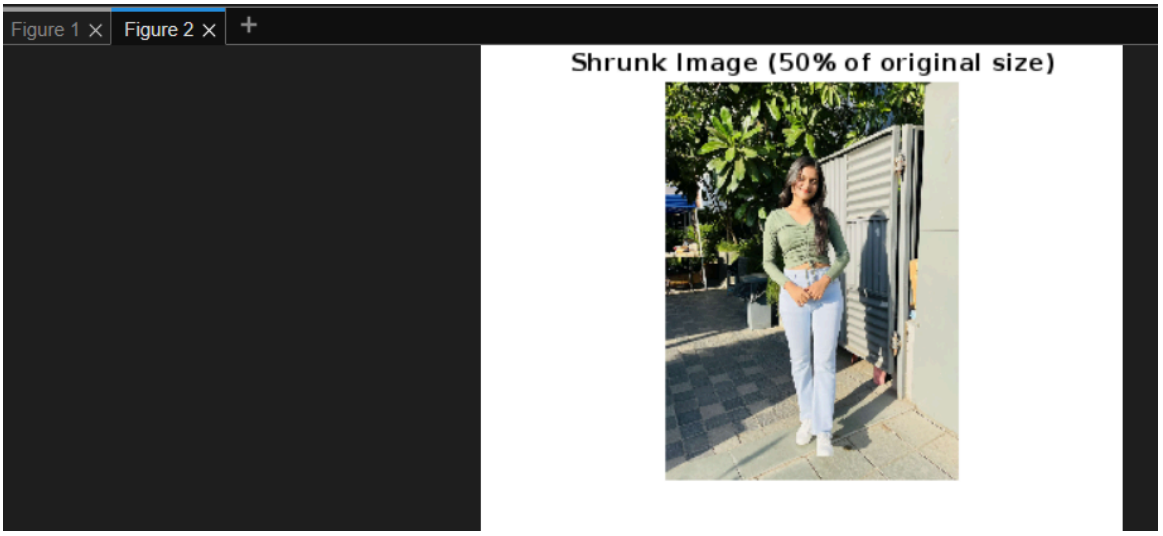
5. Perform Image Shrinking Operation on the image.

contrast	56.9084	1×1	double
gray_img	*1280×720 uint8*	1280×720	uint8
img	*2208×1656×3 uint8*	2208×1656×3	uint8
shrink_img	*1104×828×3 uint8*	1104×828×3	uint8

New to MATLAB? See resources for Getting Started

```
>> % Read the original image
img = imread('myimg.jpg');

% Shrinking operation: resize the image by 50%
shrink_img = imresize(img, 0.5);  % Shrink the image to 50% of its original size

% Displaying both original and shrunk images
figure;
imshow(img);
title('Original Image');

figure;
imshow(shrink_img);
title('Shrunk Image (50% of original size)');
>>
```

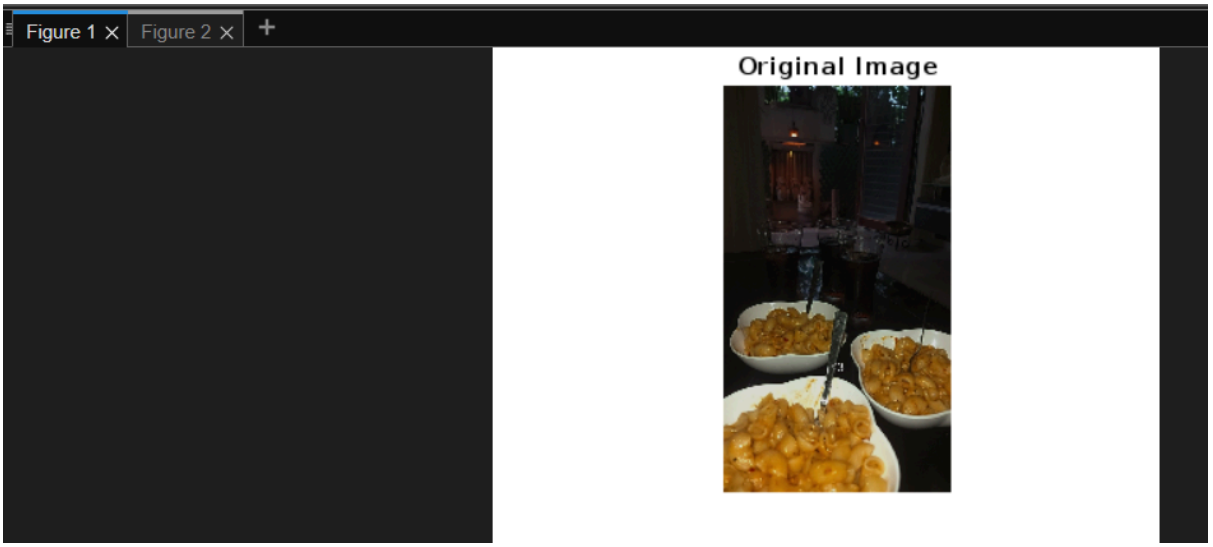




6. Perform Image Transformation (Rotation).

gray_img	1280×720 uint8	1280×720	uint8
img	1280×720×3 uint8	1280×720×3	uint8
rotated_img	1415×1415×3 uint8	1415×1415×3	uint8
shrink_img	1104×828×3 uint8	1104×828×3	uint8

```
>> % Read the original image
img = imread('image.jpg');

% Performing image rotation by 45 degrees
rotated_img = imrotate(img, 45);

% Displaying both original and rotated images
figure;
imshow(img);
title('Original Image');

figure;
imshow(rotated_img);
title('Rotated Image (45 degrees)');
>>
```



Summarised learning:

This exercise encompasses foundational image processing techniques, including pathfinding, connectivity analysis, brightness/contrast calculations, logical operations, shrinking, transformations, and advanced tasks, providing a comprehensive understanding of image manipulation.

LAB - 03

AIM: Implement basic intensity transformation functions :-

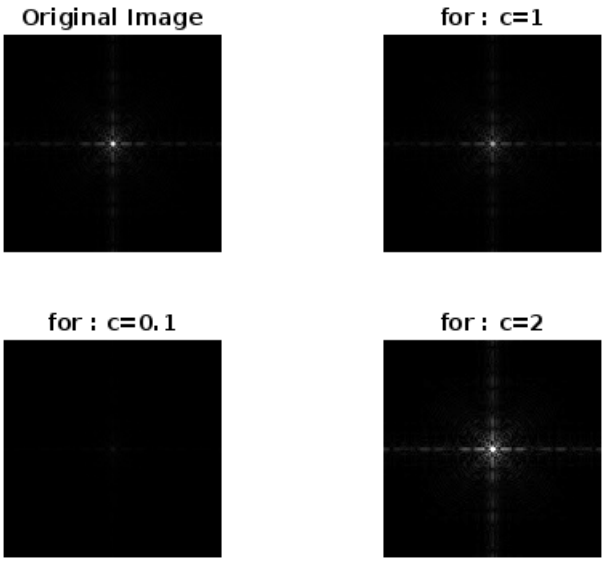
- ➔ Image Negatives
- ➔ Log Transformations
- ➔ Power-Law (Gamma) Transformations
- ➔ Contrast Stretching (Piecewise Linear transformation)

1. Take your own grayscale photo and apply negative transformation.

```
clc;
clear;
img = imread('myimg2.jpeg');
[h, w, ~] = size(img);
gray_img_manual = 0.2989 * img(:,:,1) + 0.5870 * img(:,:,2) + 0.1140 *
img(:,:,3);
negative_img_manual = 255 - gray_img_manual;
gray_img_builtin = rgb2gray(img);
negative_img_builtin = imcomplement(gray_img_builtin);
figure;
subplot(2, 3, 1);
imshow(img);
title("Original RGB Image");
subplot(2, 3, 2);
imshow(gray_img_manual, []);
title("Grayscale Image (Manual)");
subplot(2, 3, 3);
imshow(negative_img_manual, []);
title("Negative Image (Manual)");
subplot(2, 3, 4);
imshow(gray_img_builtin);
title("Grayscale Image (Built-in)");
subplot(2, 3, 5);
imshow(negative_img_builtin);
title("Negative Image (Built-in)");
set(gcf,'Position',[100, 100, 1000, 600]);
```

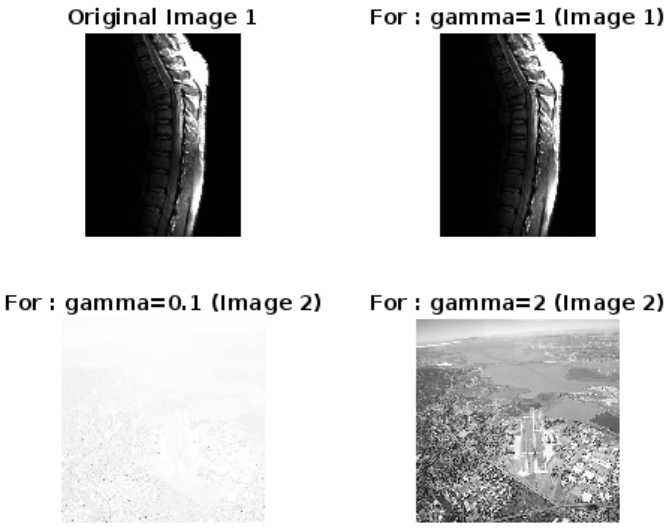
2. Consider image ex_log.tif. Enhance the image by applying log transformation.

```
clc
clear
img = imread('20211225_124959.jpg');
img = im2double(img);
%Applying log transformation
c = 1;
new1 = c * log(1 + img);
c = 0.1;
new2 = c * log(1 + img);
c = 2;
new3 = c * log(1 + img);
subplot(2,2,1)
imshow(img)
title("Original Image")
subplot(2,2,2)
imshow(new1)
title("for : c=1")
subplot(2,2,3)
imshow(new2)
title("for : c=0.1")
subplot(2,2,4)
imshow(new3)
title("for : c=2")
```



3. Consider images ex_power1.tif and ex_power2.tif and enhance them with power law transformation.

```
clc;
clear;
img1 = imread('ex_power1.jpeg'); % Image for gamma = 1
img2 = imread('ex_power2.jpeg'); % Image for gamma = 0.1 and 2
img1 = im2double(img1);
img2 = im2double(img2);
gamma1 = 1;
new1 = img1 .^ gamma1;
gamma2 = 0.1;
new2 = img2 .^ gamma2;
gamma3 = 2;
new3 = img2 .^ gamma3;
subplot(2,2,1);
imshow(img1);
title("Original Image 1");
subplot(2,2,2);
imshow(new1);
title("For : gamma=1 (Image 1)");
subplot(2,2,3);
imshow(new2);
title("For : gamma=0.1 (Image 2)");
subplot(2,2,4);
imshow(new3);
title("For : gamma=2 (Image 2)");
```



4. Consider your over exposed photo (that you generated for assignment 1) and enhance it by power law transformation. Specify the value of gamma which is suitable for this enhancement.

```
clc
clear
img = imread('overexposed_img.jpg');
img = im2double(img);
%Applying power transformation
gamma =1.7;
new1 = img .^ gamma;
subplot(2,2,1)
imshow(img)
title("Original Image")
subplot(2,2,2)
imshow(new1)
title("for : gamma=1.7")
```

Thus, when the gamma value is greater than 1, the transformation tends to darken the bright regions, which helps in correcting over exposure.

5. Consider your under exposed photo (that you generated for assignment 1) and enhance it by power law transformation. Specify the value of gamma which is suitable for this enhancement.

```
clc
clear
img = imread('underexposed_img.jpg');
img = im2double(img);
%Applying power transformation
gamma =0.4;
new1 = img .^ gamma;
subplot(2,2,1)
imshow(img)
title("Original Image")
subplot(2,2,2)
imshow(new1)
title("for : gamma=0.4")
```

Thus, when the gamma value is less than 1, the transformation tends to brighten the dark regions, which helps in correcting under exposure.

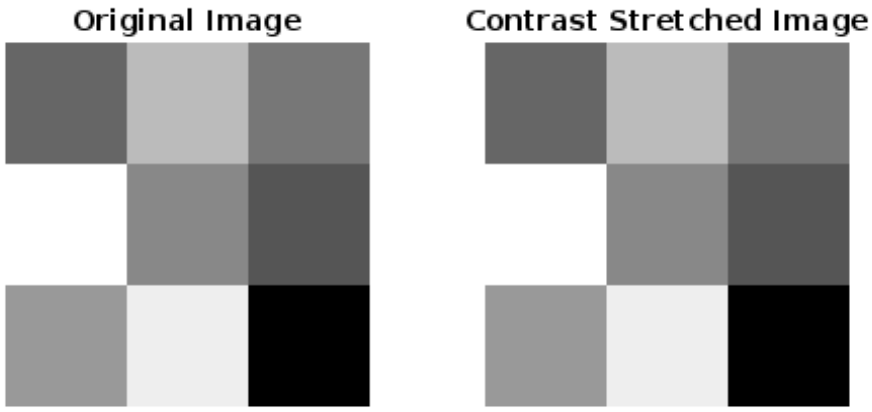
6. Contrast Stretching (Example) : A 3 x 3 8 bits/pixel image is given by

7	**12**	**8**
16	**9**	**6**
10	**15**	**1**

Apply contrast stretch to the image so that the new image has a dynamic range of [0, 255]. Also show the output image. Sketch the transformation you used for contrast stretching.

```
clc;
clear;
original_img = [7, 12, 8;
                16, 9, 6;
                10, 15, 1];
subplot(1, 2, 1);
imshow(original_img, []);
title('Original Image');
r1 = min(original_img(:));
r2 = max(original_img(:));
[x, y] = size(original_img);
new_img = zeros(x, y);
for i = 1:x
   for j = 1:y
       new_img(i, j) = (original_img(i, j) - r1) * (255 / (r2 - r1));
   end
end
new_img = uint8(new_img);
subplot(1, 2, 2);
imshow(new_img);
title('Contrast Stretched Image');
Figure;
x_values = [0, r1, r2, 255];
```

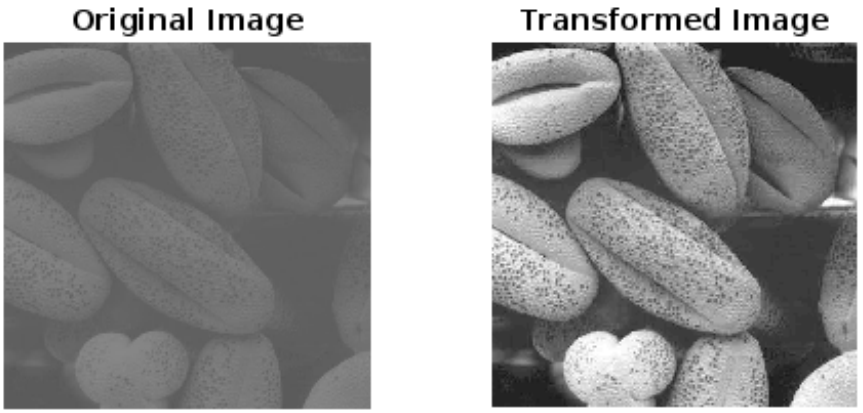
```
y_values = [0, 0, 255, 255];
plot(x_values, y_values, 'b-o', 'LineWidth', 2);
xlabel('Original Pixel Value');
ylabel('Stretched Pixel Value');
title('Contrast Stretching Transformation');
grid on;
axis([0 255 0 255]);
```



7. Do contrast stretching for the image ex_contrast.tif. Obtain contrast stretched image from low contrast image and apply thresholding.

```
clc;
clear;
img = imread('ex_contrast.jpeg');
figure;
subplot(1,2,1);
imshow(img);
title('Original Image');
[x, y, z] = size(img);
```

```
if z == 3
   img = rgb2gray(img);
end
r2 = max(max(img));
r1 = min(min(img));
new1 = img;
temp1 = r2 - r1;
for i = 1:x
   for j = 1:y
       new1(i, j) = (255/temp1)*(img(i, j) - r1);
   end
end
subplot(1,2,2);
imshow(new1);
title('Transformed Image');
```



Summarised learning:

This exercise applies various intensity transformations to enhance image quality, including negatives, log, power-law, and contrast stretching, demonstrating their effectiveness in improving visibility and dynamic range for diverse image conditions.

LAB - 04

AIM: Implement following Image Enhancement Techniques :–

- **Intensity Level Slicing**
- **Bit Plane Slicing & Reconstruction**
- **Histogram Equalization**

1. Take the image 'dollar.tif' and separate out its bit planes.

a. Reconstruct your image using higher order 2 bit planes.

b. Reconstruct your image using higher order 4 bit planes.

c. Experiment with the bit planes and derive your conclusions.

```
clc
clear
img = imread('dollar.tif');
imshow(img)
[x, y] = size(img);
out = img;
bit_plane5 = img;
bit_plane6 = img;
for plane = 1:8
for i = 1:x
for j = 1:y
out(i,j) = bitget(img(i,j),plane);
end
end
out = double(out);
subplot(2,4,plane);
imshow(out);
title("Bit- Plane : " + plane);
%if(plane==7)
% bit_plane7 = out;
%end
%if(plane==8)
% bit_plane8 = out;
%end
end
%answer = bit_plane7 + bit_plane8;
%imshow(answer)
```

Here we've extracted all the planes for the image as image is 8 bit Image. There will be 8 planes.

We can even separate multiple planes and sum them up for better resolution. For given code we are extracting 7th and 8th plane and summing them up.

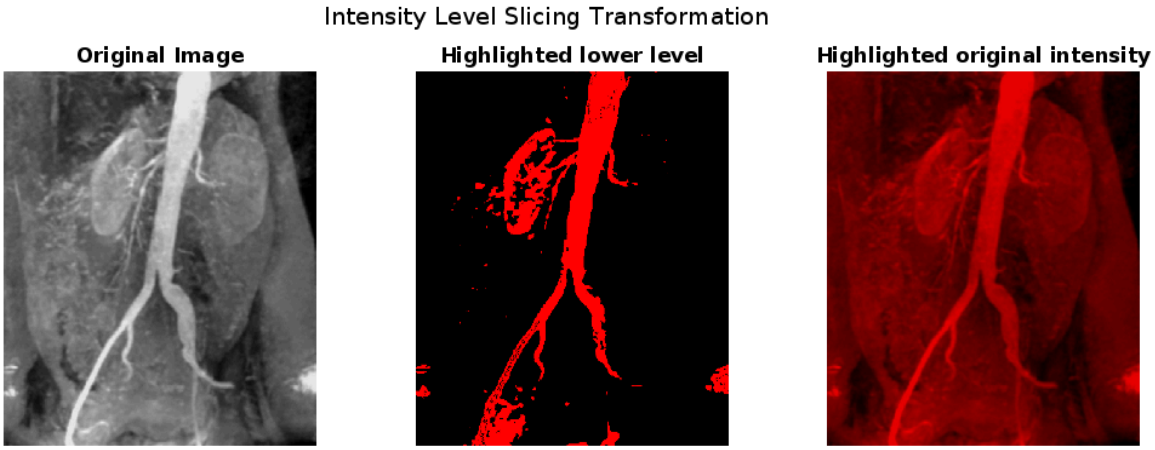
2. Consider the image kidney.tif and perform intensity level slicing transformation within the range (150 – 230)

a. Highlight the given intensity range and keep all other intensities to a lower level.

b. Highlight the given intensity range and keep all other intensities as it is.

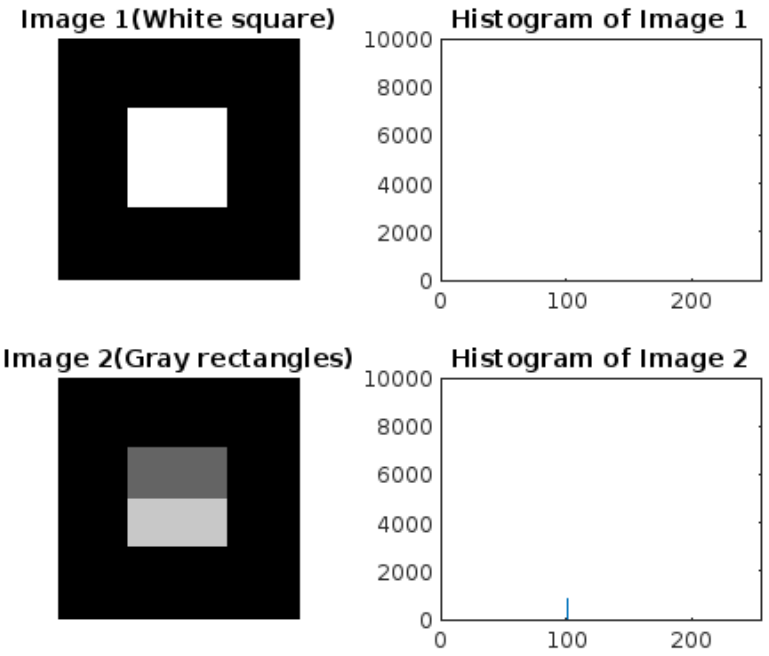
```
clc;
clear;
img = imread('kidney.jpeg');
img = double(img);
highlighted_lower = zeros(size(img));  % part (a)
highlighted_original = zeros(size(img));  % part (b)
lower_bound = 150;
upper_bound = 230;
% Performing intensity level slicing
for i = 1:size(img, 1)
   for j = 1:size(img, 2)
       if img(i,j) >= lower_bound && img(i,j) <= upper_bound
           highlighted_lower(i,j) = 255;
           highlighted_original(i,j) = img(i,j);
       else
           highlighted_lower(i,j) = 0;
           highlighted_original(i,j) = img(i,j);
       end
   end
end
figure;
subplot(1, 3, 1);
imshow(uint8(img));
title('Original Image', 'FontSize', 12);
subplot(1, 3, 2);
imshow(uint8(highlighted_lower));
title('Highlighted lower level', 'FontSize', 12);
subplot(1, 3, 3);
imshow(uint8(highlighted_original));
```

```
title('Highlighted original intensity', 'FontSize', 12);
sgtitle('Intensity Level Slicing Transformation', 'FontSize', 14);
set(gcf, 'Position', [100, 100, 1200, 400]);
```



3. Can two visually different image have same histogram? If yes synthesize two grayscale images which are visually different but having the same histogram and also show the histogram. If no, justify your answer.
imhist(image): gives the histogram of the image.

```
clc;
clear;
image1 = zeros(100, 100);
image1(30:70, 30:70) = 255;
image2 = zeros(100, 100);
image2(30:50, 30:70) = 100;
image2(51:70, 30:70) = 200;
% verifying histograms
hist_image1 = imhist(uint8(image1));
hist_image2 = imhist(uint8(image2));
figure;
subplot(2, 2, 1);
imshow(uint8(image1));
title('Image 1(White square)');
subplot(2, 2, 2);
bar(hist_image1);
title('Histogram of Image 1');
xlim([0 255]);
subplot(2, 2, 3);
imshow(uint8(image2));
title('Image 2(Gray rectangles)');
subplot(2, 2, 4);
bar(hist_image2);
title('Histogram of Image 2');
xlim([0 255]);
disp('Histograms are equal:');
disp(isequal(hist_image1, hist_image2));
```



Therefore the answer is YES, two visually different images can have the same histogram.

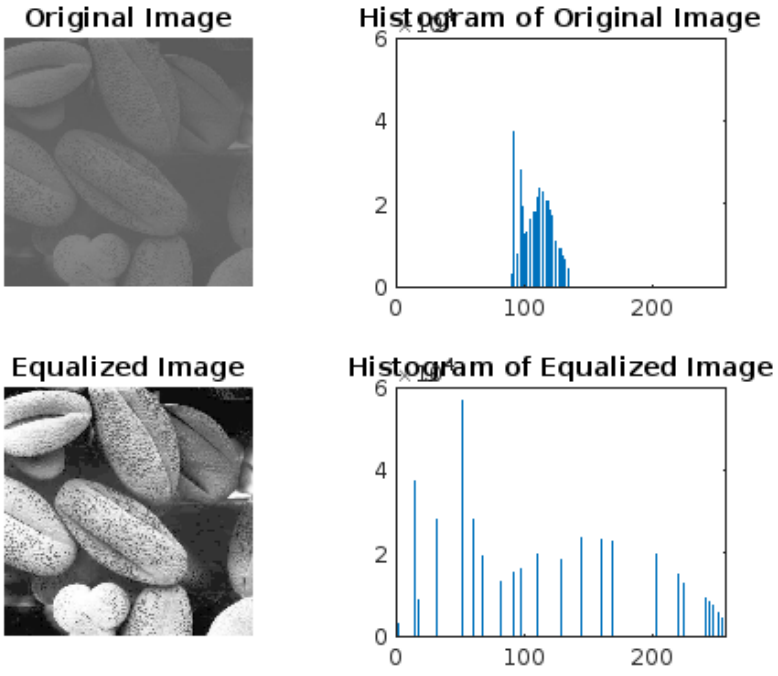
This is because a histogram represents the frequency of pixel intensity values in an image, so it does not contain any information about the spatial arrangement of those pixel values.

4. Histogram Equalization:

a. Use the function histeq(image) on the image ex_contrast.tif.

```
clc;
clear;
img = imread('ex_contrast.jpeg');
hist_orig = imhist(img);
cdf = cumsum(hist_orig) / sum(hist_orig);
img_eq = 255 * cdf(double(img) + 1);
img_eq = uint8(img_eq);
hist_eq = imhist(img_eq);
subplot(2,2,1);
imshow(img);
title('Original Image');
subplot(2,2,2);
bar(hist_orig);
title('Histogram of Original Image');
subplot(2,2,3);
imshow(img_eq);
title('Equalized Image');
subplot(2,2,4);
```

```
bar(hist_eq);
title('Histogram of Equalized Image');
```



Summarised learning:

This exercise demonstrates advanced image enhancement techniques, including bit-plane slicing, intensity-level slicing, and histogram equalization, showcasing their ability to manipulate intensity ranges, reconstruct details, and improve image contrast effectively.

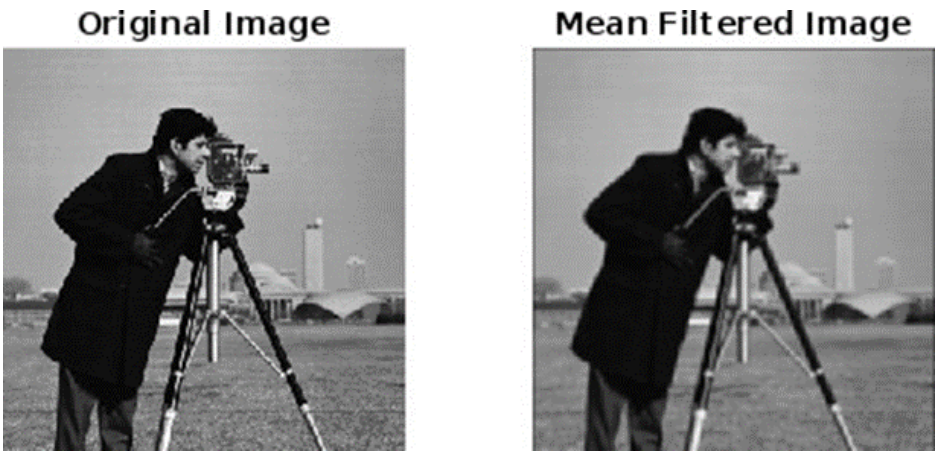
LAB - 05

1. Perform all spatial filters of chapter 03 for image smoothing and contrasting and combination of them.
2. Do some significant research base changes in above all experiments and observe and conclude about changes and results for each and explain briefly.

Mean filter:

Smoothens the image by averaging neighbourhood pixel values, reducing noise but potentially blurring edges.

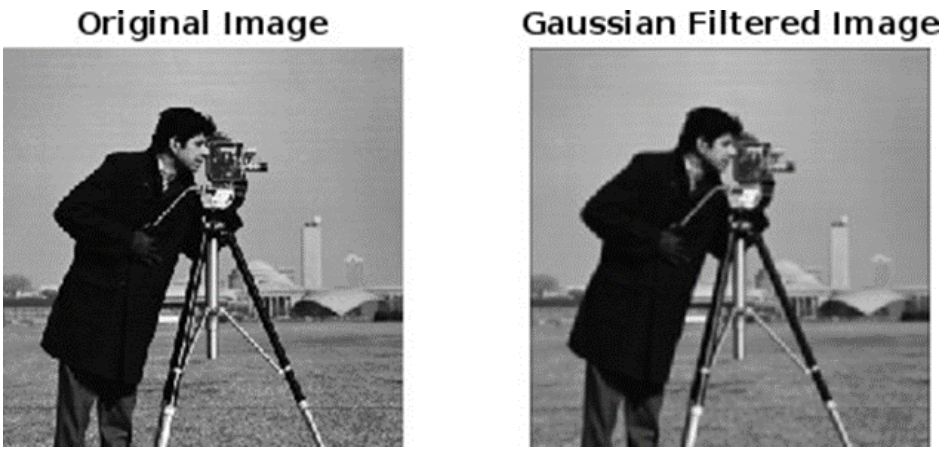
```
img =
imread('cameraman.tif'); img
= double(img);

% Mean filter

ksize = 3; % Kernel size
h = fspecial('average', ksize);
mean_filtered = imfilter(img, h);

% Display

subplot(1, 2, 1); imshow(uint8(img)); title('Original Image');
subplot(1, 2, 2); imshow(uint8(mean_filtered)); title('Mean Filtered Image');
```



Gaussian filter:

Applies a Gaussian function for smoother blending, reducing noise while preserving some edge sharpness.

```
% Gaussian filter

sigma = 1; % Standard deviation for Gaussian
filter h = fspecial('gaussian', [ksize ksize],
sigma); gaussian_filtered = imfilter(img, h);
%
Display
figure;

subplot(1, 2, 1); imshow(uint8(img)); title('Original Image');
subplot(1, 2, 2); imshow(uint8(gaussian_filtered)); title('Gaussian Filtered
Image');
```

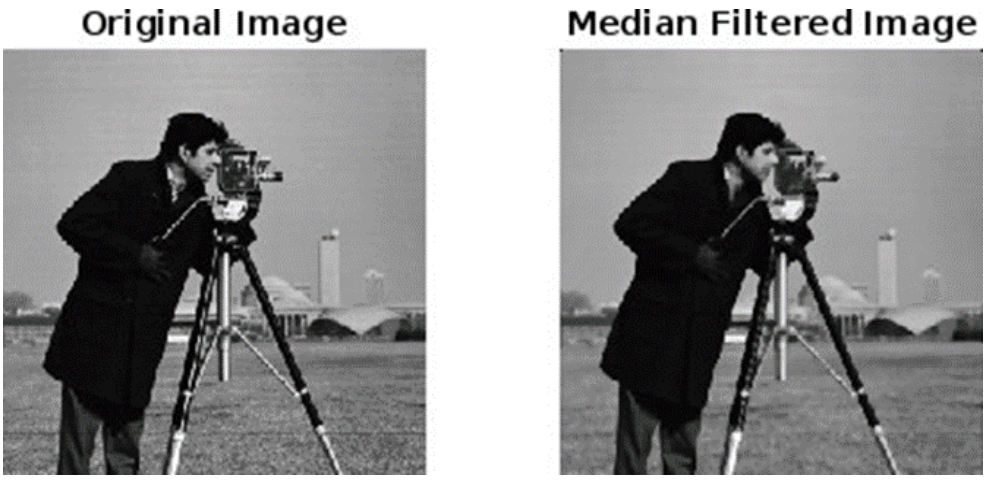


Median filter:

Replaces each pixel with the median of its neighbourhood, effectively reducing salt-and-pepper noise without blurring edges significantly.

```
% Median filter

median_filtered = medfilt2(uint8(img), [ksize ksize]);
%
Display
figure;

subplot(1, 2, 1); imshow(uint8(img)); title('Original Image');
```

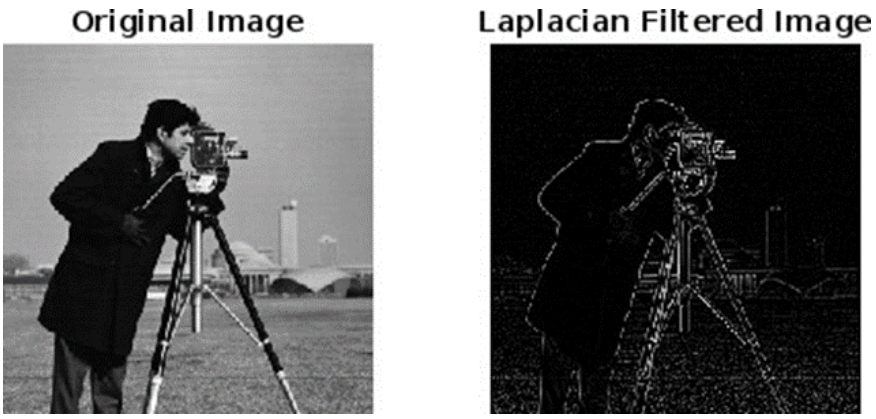
```
subplot(1, 2, 2); imshow(median_filtered); title('Median Filtered Image');
```



Laplacian Filter:

Highlights edges by computing the second derivative, useful for edge enhancement but sensitive to noise.

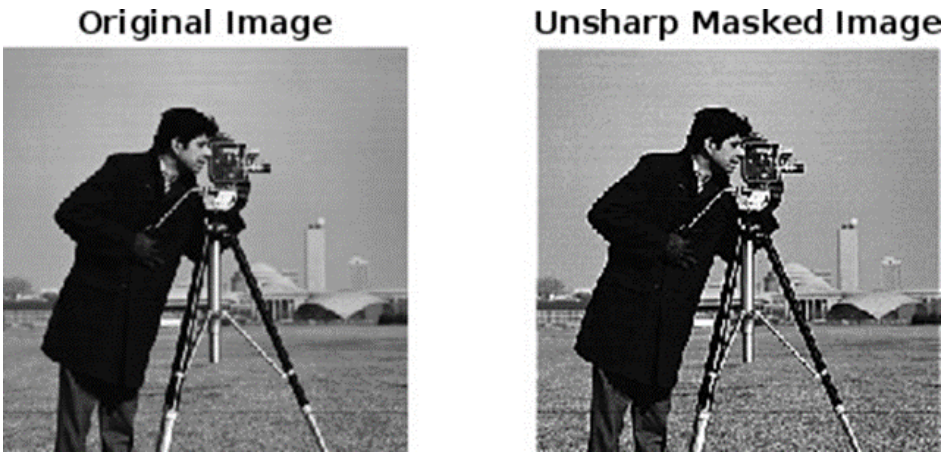
```
% Laplacian filter for edge
enhancement
h = fspecial('laplacian', 0.2);
laplacian_filtered = imfilter(img,
h);
%
Display
figure;

subplot(1, 2, 1); imshow(uint8(img)); title('Original Image');

subplot(1, 2, 2); imshow(uint8(laplacian_filtered)); title('Laplacian Filtered
Image');
```



Unsharp masking:

Enhances contrast by subtracting a blurred version from the original, boosting sharpness.

```
% Unsharp masking for contrast enhancement

blurred = imfilter(img, fspecial('gaussian', [ksize ksize], sigma));
unsharp_masked = img + 1.5 * (img - blurred); % Adjust weight as
needed
%
Display
figure;

subplot(1, 2, 1); imshow(uint8(img)); title('Original Image');

subplot(1, 2, 2); imshow(uint8(unsharp_masked)); title('Unsharp Masked Image');
```

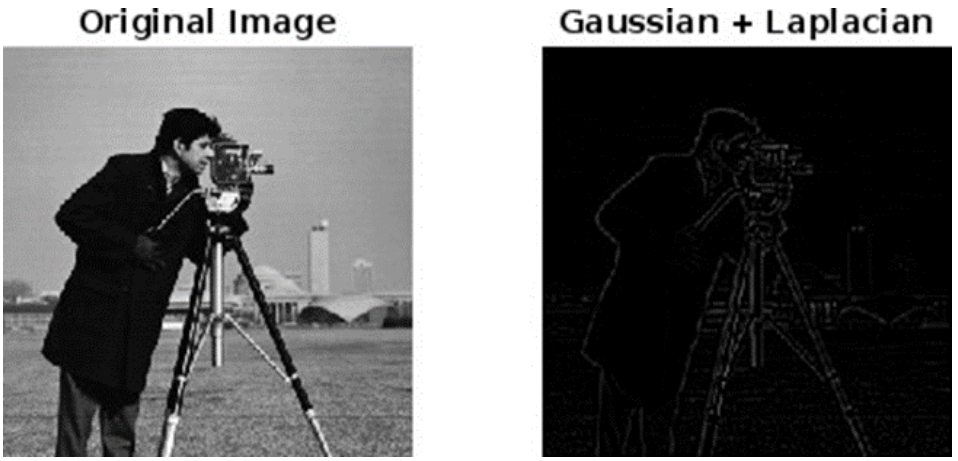


Gaussian + Laplacian:

First smooth with Gaussian, then enhance edges with Laplacian.

```
% Gaussian smoothing

gaussian_smoothed = imfilter(img, fspecial('gaussian', [ksize ksize], sigma));
% Laplacian edge enhancement
combined = imfilter(gaussian_smoothed, fspecial('laplacian', 0.2));
%
Display
figure;
```

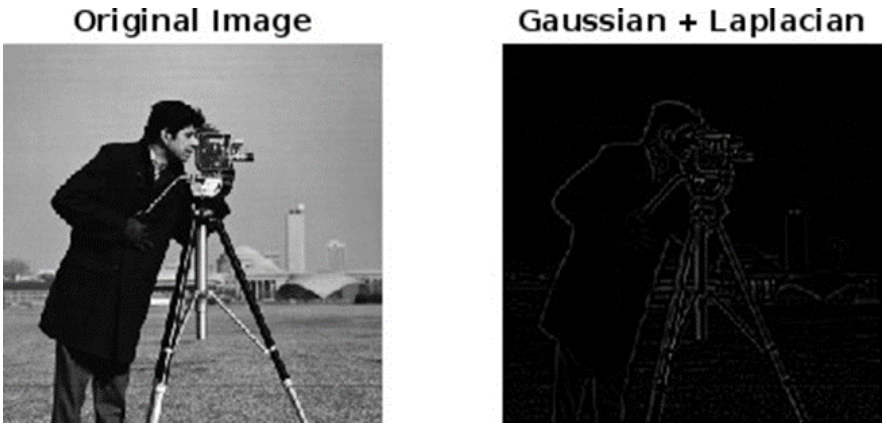
```
subplot(1, 2, 1); imshow(uint8(img)); title('Original Image');
subplot(1, 2, 2); imshow(uint8(combined)); title('Gaussian +
Laplacian');
```



Median + Unsharp Masking:

First reducing noise with Median, then apply Unsharp Masking for contrast.

```
% Gaussian smoothing

gaussian_smoothed = imfilter(img, fspecial('gaussian', [ksize ksize], sigma));
% Laplacian edge enhancement
combined = imfilter(gaussian_smoothed, fspecial('laplacian', 0.2));
%
Display
figure;
subplot(1, 2, 1); imshow(uint8(img)); title('Original Image');
subplot(1, 2, 2); imshow(uint8(combined)); title('Gaussian +
Laplacian');
```



Observations from above analysis:

Parameters that are noticed while performing the above filters:

Gaussian Sigma Value: Higher sigma value provides stronger smoothing but can blur edges. Whereas lower sigma value retains more detail.

Kernel Size: Larger kernel sizes in mean and median filters increase the smoothing but lack details.

Contrast Filter Strength: In unsharp masking, increasing the weight of the original image amplifies sharpness but risks introducing halo effects.

Smoothing Filters: Median filters are ideal for reducing salt-and-pepper noise, while Gaussian filters are effective for general smoothing with adjustable edge retention.

Contrasting Filters: The Laplacian filter significantly improves edge visibility but it can also increase noise if applied without prior smoothing.

Combinations: Sequential combinations like Gaussian + Laplacian balance noise reduction with edge enhancement, providing a sharper and cleaner image.

LAB - 06

1. Perform all filters of chapter 05 and combination of them.
2. Do some significant research base changes in above all experiments and observe and conclude about changes and results for each and explain in briefly.

Basic Filters are:
- Mean
- Median
- Min
- Max

Mean Filter: It replaces the value of each pixel in an image with the average of the values in its neighbouring pixels. This technique is often used for **noise reduction** by blurring the image.

```
img = imread('cameraman.tif');
img = double(img);
subplot(1, 3, 1);
imshow(uint8(img));
title('Original Image');
% Define the filter size (3x3)
ksize = 3;
[M, N] = size(img);
img1 = zeros(M, N);
% Step 3: Add Gaussian noise
mean = 0;
SD = 15;
gaussianNoise = SD * randn(size(img)) + mean;
noisyImg = double(img) + gaussianNoise;
% Clip the values to stay within the valid pixel range [0, 255]
noisyImg = max(min(noisyImg, 255), 0);
noisyImg = uint8(noisyImg);
subplot(1, 3, 2);
imshow(noisyImg);
title('Noisy Image');
% Apply mean filter
img2 = zeros(M, N);
for i = 1:M
for j = 1:N
% Define the neighbourhood boundaries with boundary checks
rMin = max(1, i - floor(ksize / 2));
rMax = min(M, i + floor(ksize / 2));
cMin = max(1, j - floor(ksize / 2));
cMax = min(N, j + floor(ksize / 2));
% Extract neighbourhood and compute the mean
```

```
neighbour = noisyImg(rMin:rMax, cMin:cMax);
img2(i, j) = sum(neighbour(:)) / (ksize * ksize);
end
end
img2 = uint8(img2);
subplot(1, 3, 3);
imshow(img2);
title('Mean Image');
```

Observation:

Noise Reduction, Blurred Image, Effect on edges of an image.

```
img = imread('cameraman.tif');
img = double(img);
subplot(1, 4, 1);
imshow(uint8(img));
title('Original Image');
% Define the filter size
ksize = 3;
[M, N] = size(img);
img1 = zeros(M, N); % For mean image
img2 = zeros(M, N); % For geometric mean image
% Step 3: Add Gaussian noise
mean = 0;
SD = 20;
gaussianNoise = SD * randn(size(img)) + mean;
noisyImg = double(img) + gaussianNoise;
% Clip the values to stay within the valid pixel range [0, 255]
noisyImg = max(min(noisyImg, 255), 0);
noisyImg = uint8(noisyImg);
subplot(1, 4, 2);
imshow(noisyImg);
title('Noisy Image');
% Apply mean filter
for i = 1:M
for j = 1:N
% Define the neighbourhood boundaries with boundary checks
rMin = max(1, i - floor(ksize / 2));
rMax = min(M, i + floor(ksize / 2));
cMin = max(1, j - floor(ksize / 2));
```

```
cMax = min(N, j + floor(ksize / 2));
% Extract neighbourhood and compute the mean
neighbour = noisyImg(rMin:rMax, cMin:cMax);
img1(i, j) = sum(neighbour(:)) / (ksize * ksize);
end
end
img1 = uint8(img1);
subplot(1, 4, 3);
imshow(img1);
title('AM Image');
% Apply geometric mean filter
for i = 1:M
for j = 1:N
% Define the neighbourhood boundaries with boundary checks
rMin = max(1, i - floor(ksize / 2));
rMax = min(M, i + floor(ksize / 2));
cMin = max(1, j - floor(ksize / 2));
cMax = min(N, j + floor(ksize / 2));
% Extract the neighbourhood
neighbourhood = noise Img(rMin:rMax, cMin:cMax);
% Compute the geometric mean of the neighbourhood
product = prod(neighbourhood(:)); % Product of all pixels in the
neighbourhood
img2(i, j) = product^(1 / numel(neighbourhood)); % Geometric mean
end
end
img2 = uint8(img2);
subplot(1, 4, 4);
imshow(img2);
title('GM Image');
```

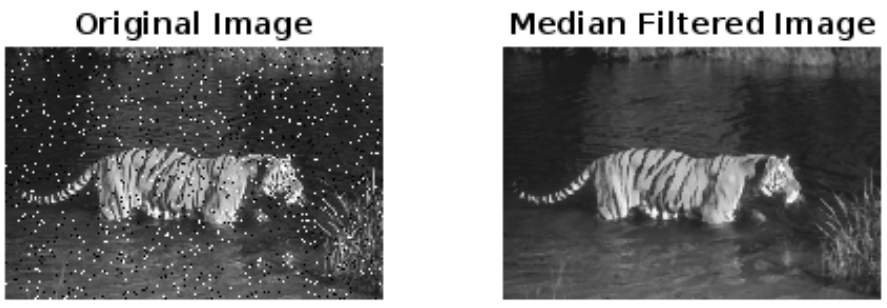
Observation:

Geometric Mean has better visualisation than arithmetic mean.

Median Filter:

It replaces each pixel with the median value of the pixels in its neighbourhood. This is particularly effective in removing salt-and-pepper noise, as it preserves edges better than the mean filter and is less affected by extreme values in the neighbourhood.

```
img = imread('peppernoise.png');
img = double(img);
figure;
subplot(1, 2, 1);
imshow(uint8(img));
title('Original Image');
% Define the kernel size
ksize = 3;
[M, N] = size(img);
img1 = zeros(M, N);
% Apply median filter manually
for i = 1:M
   for j = 1:N
       % Define the boundaries of the neighbourhood
       rMin = max(1, i - floor(ksize / 2));
       rMax = min(M, i + floor(ksize / 2));
       cMin = max(1, j - floor(ksize / 2));
       cMax = min(N, j + floor(ksize / 2));

       % Get the neighbourhood and calculate the median
       neighbour = img(rMin:rMax, cMin:cMax);
       img1(i, j) = median(neighbour(:));
   end
end
% Convert back to uint8 and display the filtered image
img1 = uint8(img1);
subplot(1, 2, 2);
imshow(img1);
title('Median Filtered Image');
```



Observation:

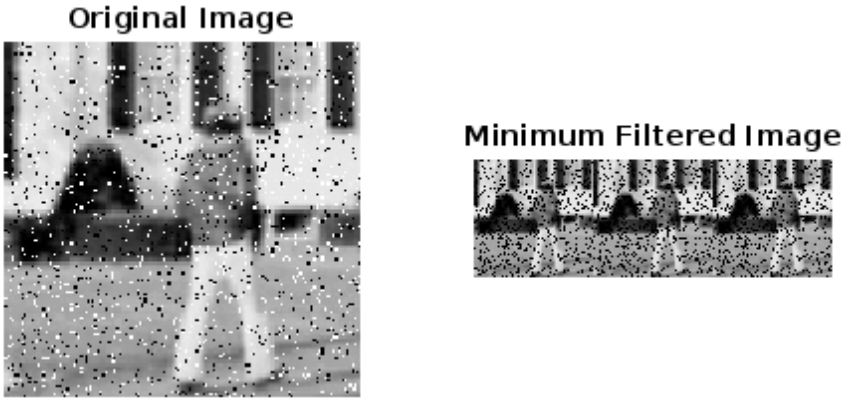
The median filter is highly effective in reducing pepper noise by replacing each pixel with the median value of its neighborhood. This removes dark spots caused by pepper

noise without excessively blurring the image, preserving edges and details better than other smoothing filters

Min Filter:

The min filter works by replacing each pixel value in the image with the minimum value of its surrounding neighbourhood. It is often used for noise reduction, specifically for removing salt noise (i.e., 255 values), but may also help in removing pepper noise (0 values).

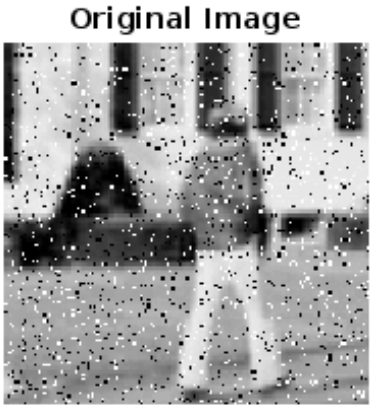
```
img = imread('saltnoise.png');
img = double(img);
figure; % Open a new figure window
subplot(1, 2, 1);
imshow(uint8(img));
title('Original Image');
% Set kernel size to 3 for better performance
ksize = 3;
[M, N] = size(img);
img1 = zeros(M, N);
% Apply minimum filter manually
for i = 1:M
   for j = 1:N
       % Define the neighbourhood boundaries
       rMin = max(1, i - floor(ksize / 2));
       rMax = min(M, i + floor(ksize / 2));
       cMin = max(1, j - floor(ksize / 2));
       cMax = min(N, j + floor(ksize / 2));

       % Get the neighbourhood and find the minimum value
       neighbour = img(rMin:rMax, cMin:cMax);
       img1(i, j) = min(neighbour(:));
   end
end
% Convert the filtered image back to uint8 and display
img1 = uint8(img1);
subplot(1, 2, 2);
imshow(img1);
title('Minimum Filtered Image');
```

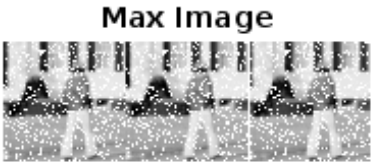


Observation:
Removal of salt noise. Mean, geometric mean, and median filters can also help reduce the pepper noise, but the min filter might be the most effective.

Max Filter:

```
img = imread('saltnoise.png');
img = double(img);
subplot(1, 2, 1);
imshow(uint8(img));
title('Original Image');
ksize = 2;
[M, N] = size(img);
img1 = zeros(M, N);
for i = 1:M
for j = 1:N
rMin = max( 1,i-floor(ksize/2) );
rMax = min( M,i+floor(ksize/2) );
cMin = max( 1,j-floor(ksize/2) );
cMax = min( N,j+floor(ksize/2) );
neighbour = img(rMin:rMax, cMin:cMax);
img1(i, j) = max(neighbour(:));
end
end
img1 = uint8(img1);
subplot(1, 2, 2);
imshow(img1);
title('Max Image');
```





Observation:

When applied to salt noise, the max filter tends to increase the brightness of the already white or bright pixels in the noisy areas.

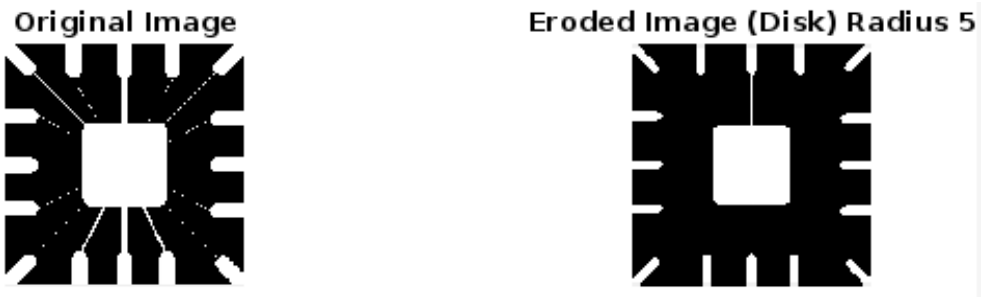
LAB - 7 & 8

1. Perform all morphological operations and combinations of them that you learned in chapter 09.
2. Do some significant research base changes in above all experiments and observe and conclude about changes and results for each and explain briefly.

Basic morphological operations in the Digital Image Processing are:
➔ Erosion
➔ Dilation
➔ Opening
➔ Closing
➔ Boundary Extraction

Erosion:

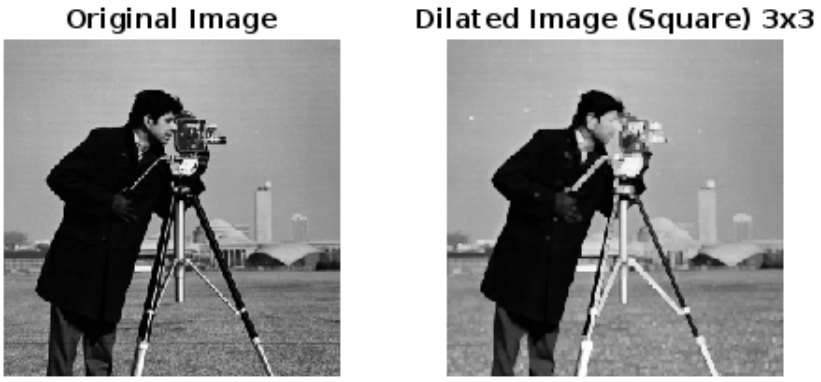
```
img = imread('wirebondmask.png');
subplot(2,3,1)
imshow(img, []);
title('Original Image');
% Perform Erosion with a Disk-Shaped Structuring Element of Radius 5
SE2 = strel('disk', 5);
erodedImageDisk = imerode(img, SE2);
subplot(2,3,3)
imshow(erodedImageDisk, []);
title('Eroded Image (Disk) Radius 5');
```



Observation:
Erosion on a wire bond mask helps refine edges and remove small noise by shrinking white regions. However, excessive erosion may thin out or disconnect important features.

Dilation:

```
img = imread('cameraman.tif');
subplot(1,2,1)
imshow(img, []);
title('Original Image');
SE3 = strel('square', 3);
dilatedImage7 = imdilate(img, SE3);
subplot(1,2,2)
imshow(dilatedImage7, []);
title('Dilated Image (Square) 3x3');
```



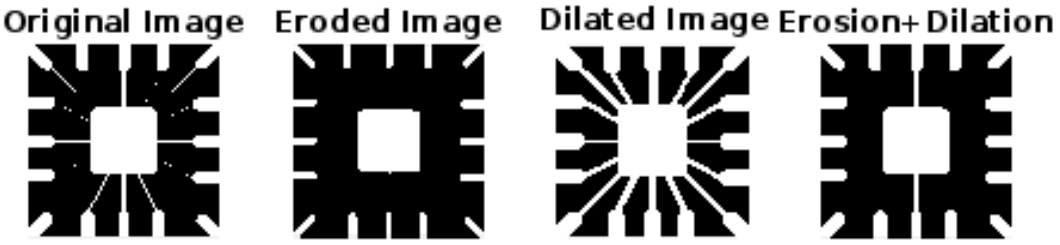
Observation:

Dilation on cameraman.tiff expands the bright regions, filling small gaps and connecting nearby features. This operation enhances details but may cause some loss of finer textures.

Erosion + Dilation:

```
img = imread('wirebondmask.png');
subplot(1,4,1)
imshow(img, []);
title('Original Image');
% Perform Erosion
SE2 = strel('disk', 5); % Use disk shape
erodedImageDisk = imerode(img, SE2);
subplot(1,4,2)
imshow(erodedImageDisk, []);
title('Eroded Image');
% Perform Dilation
SE3 = strel('square', 7);
dilatedImage7 = imdilate(img, SE3);
subplot(1,4,3)
imshow(dilatedImage7, []);
title('Dilated Image');
% Perform (Erosion + Dilation)
```

```
openedImage = imopen(img, SE3);
subplot(1,4,4)
imshow(openedImage, []);
title('Erosion+Dilation');
```



Observation:

Erosion followed by dilation on a wire bond mask removes small noise particles and smoothens edges, preserving essential structures while reducing unwanted artifacts.

Opening & Closing:

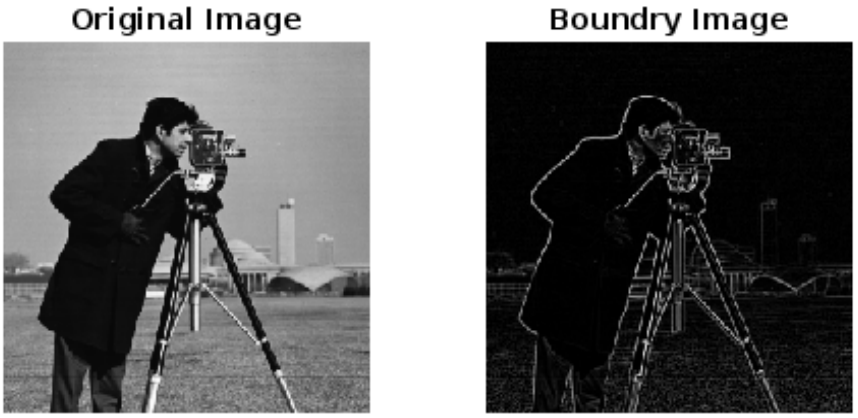
```
img = imread('fingerprint.jpg');
subplot(1,3,1)
imshow(img)
title('Original Image');
%Performing Closing
SEerode = strel('square', 2);
erode = imerode(img, SEerode);
subplot(1,3,2)
imshow(erode)
title('Closing Image');
%Performing Opening
SEdialate = strel('square', 2);
dialate = imdilate(img, SEdialate);
subplot(1,3,3)
imshow(dialate)
title('Opening Image');
```

Observation:

Opening on a fingerprint image removes small noise and smoothens ridge structures by eroding and then dilating the image.
Closing fills small gaps in the ridges and smoothens the fingerprint's valley areas by dilating and then eroding.

Boundary Extraction:

```
img = imread('cameraman.tif');
subplot(1,2,1)
imshow(img)
title('Original Image');
SE = strel('square', 3);
erode = imerode(img,SE);
boundryImage = img-erode;
subplot(1,2,2)
imshow(boundryImage);
title('Boundry Image');
```



Observation:
Boundary extraction highlights the edges of objects by detecting where the foreground and background meet. This can be achieved by subtracting the eroded image from the original, revealing the object boundaries.

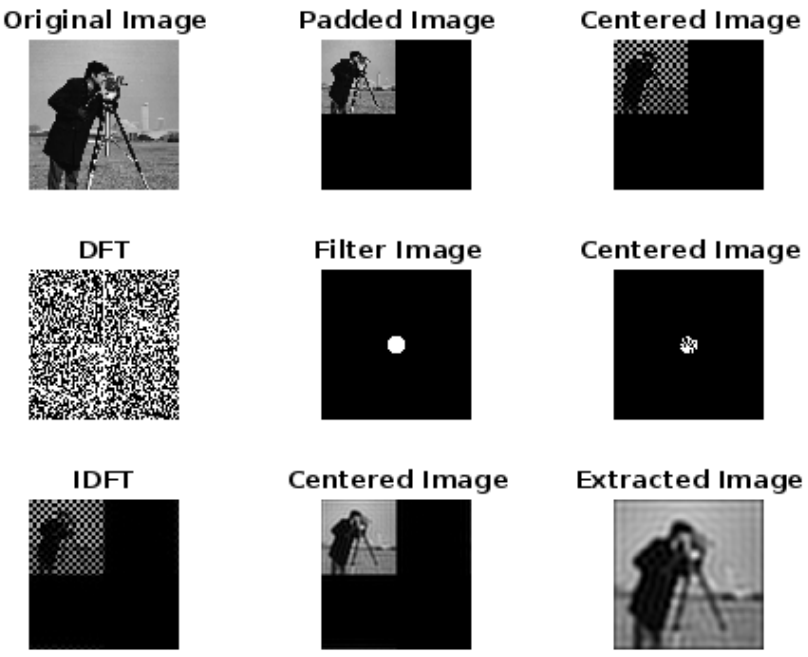
LAB - 09

AIM : Write a program to implement DFT And LOW PASS FILTER in matlab

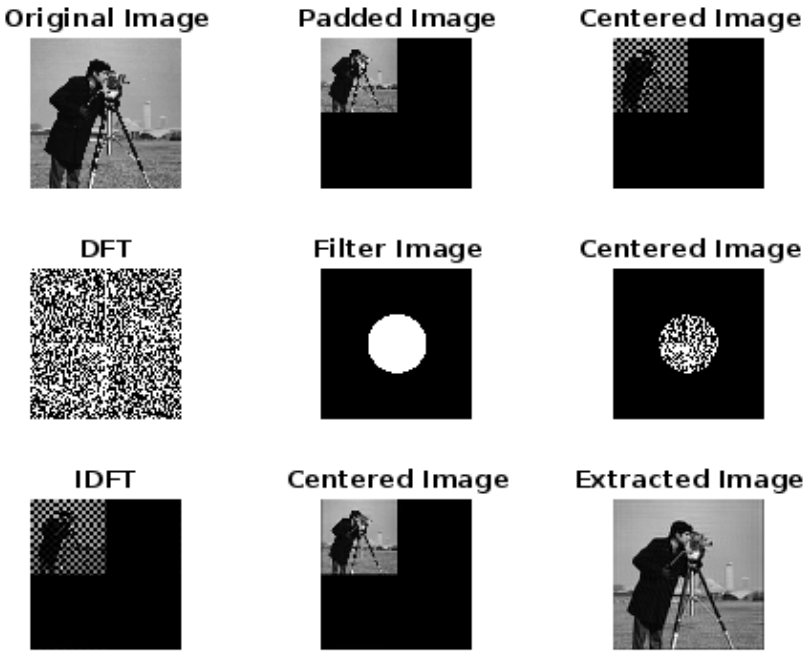
```
>> clc
clear
img = imread("cameraman.tif");
img = im2double(img);
[M, N] = size(img);
% STEP 01 : Decide padding size
P = 2*M;
Q = 2*N;
subplot(3, 3, 1);
imshow(img);
title("Original Image");
% STEP 02 : Create padded Image
ap = padarray(img,[M N],0,'post');
subplot(3, 3, 2);
imshow(ap);
title("Padded Image");
% STEP 03 : Center the spectrum
apc = ap;
for i = 0:P-1
for j = 0:Q-1
apc(i+1,j+1) = ap(i+1, j+1) * ((-1)^(i+j));
end
end
subplot(3, 3, 3);
imshow(apc)
title("Centered Image");
% STEP 04 : Taking fourier Transform
F = fft2(apc);
subplot(3, 3, 4);
imshow(F);
title("DFT");
% STEP 05 : Creating Ideal low pass filter
d0 = 30;
H = zeros([P, Q]);
for i = 1:P
for j = 1:Q
D = sqrt((i-P/2)^2 + (j-Q/2)^2);
if(D<d0)
H(i,j) = 1;
else
H(i,j) = 0;
end
end
end
subplot(3, 3, 5);
```

```
imshow(H);
title("Filter Image");
% STEP 06 : Applying filter
G = F .* H;
subplot(3, 3, 6);
imshow(G);
title("Centered Image");
% STEP 07 : Applying Inverse filter
g = ifft2(G);
subplot(3, 3, 7);
imshow(g);
title("IDFT");
% STEP 08 : Centering spectrum
og = ap;
for i = 0:P-1
for j = 0:Q-1
og(i+1,j+1) = g(i+1, j+1) * ((-1)^(i+j));
end
end
subplot(3, 3, 8);
imshow(og)
title("Centered Image");
% STEP 09 : Extrcating Image
og = real(og);
newImage = imcrop(og, [1, 1, M, N]);
subplot(3, 3, 9);
imshow(newImage)
title("Extracted Image");
```

This example is for d0 = 30.



This example is for d0 = 100.



Summarised learning:

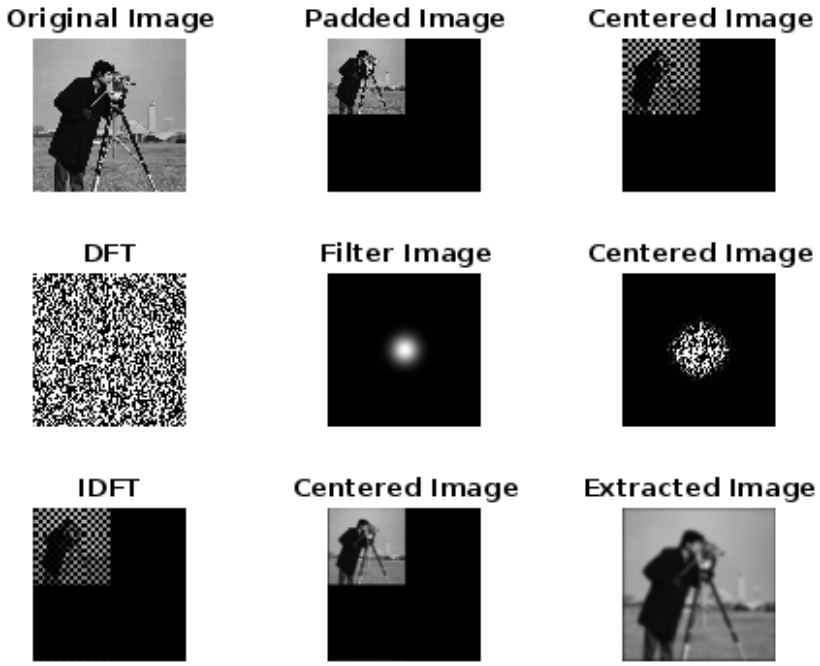
Here we have taken d0 = 30 which will allow more low level frequencies to pass through. So smoothing will be higher compared to the second example d0 = 100. As we increase d0 value it'll do less smoothing but decreasing it will result in ringing effect, so solution to this is Gaussian low pass filter.

Gaussian low pass smoothing filter on Image in frequency domain:

```
>> clear
img = imread("cameraman.tif");
img = im2double(img);
[M, N] = size(img);
% STEP 01 : Decide padding size
P = 2*M;
Q = 2*N;
subplot(3, 3, 1);
imshow(img);
title("Original Image");
% STEP 02 : Create padded Image
ap = padarray(img,[M N],0,'post');
subplot(3, 3, 2);
```

```
imshow(ap);
title("Padded Image");
% STEP 03 : Center the spectrum
apc = ap;
for i = 0:P-1
for j = 0:Q-1
apc(i+1,j+1) = ap(i+1, j+1) * ((-1)^(i+j));
end
end
subplot(3, 3, 3);
imshow(apc)
title("Centered Image");
% STEP 04 : Taking fourier Transform
F = fft2(apc);
subplot(3, 3, 4);
imshow(F);
title("DFT");
% STEP 05: Creating Gaussian low pass filter
H = zeros([P, Q]);
sigma = 30;
for i = 1:P
for j = 1:Q
D = sqrt((i-P/2)^2 + (j-Q/2)^2);
H(i,j) = exp(-(D^2) / (2*sigma^2));
end
end
subplot(3, 3, 5);
imshow(H);
title("Filter Image");
% STEP 06 : Applying filter
G = F .* H;
subplot(3, 3, 6);
imshow(G);
title("Centered Image");
% STEP 07 : Applying Inverse filter
g = ifft2(G);
subplot(3, 3, 7);
imshow(g);
title("IDFT");
% STEP 08 : Centering spectrum
og = ap;
for i = 0:P-1
for j = 0:Q-1
og(i+1,j+1) = g(i+1, j+1) * ((-1)^(i+j));
end
end
subplot(3, 3, 8);
imshow(og)
title("Centered Image");
% STEP 09 : Extrcating Image
```

```
og = real(og);
newImage = imcrop(og, [1, 1, M, N]);
subplot(3, 3, 9);
imshow(newImage)
title("Extracted Image");
```



Summarised learning:

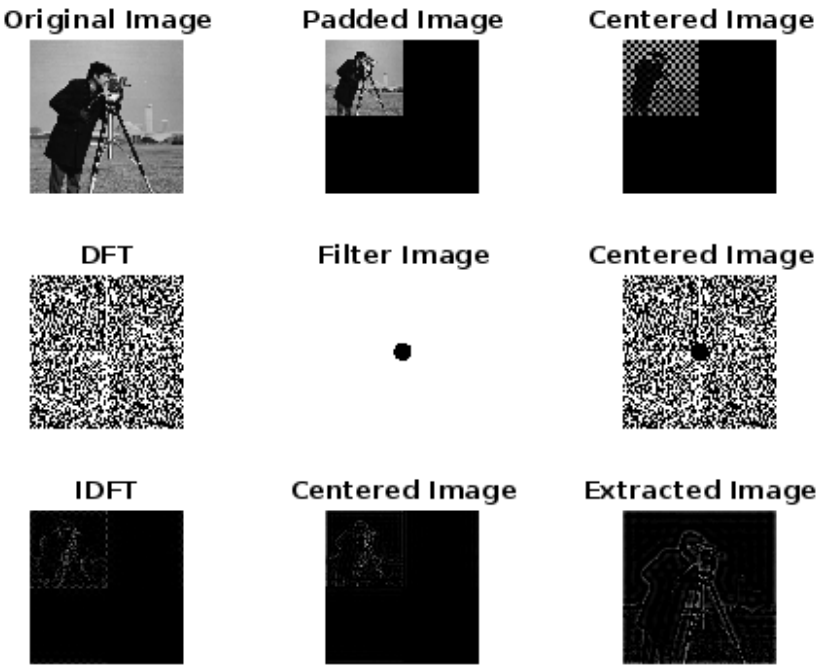
Gaussian filter filters out the limitations of the previous filter which was a ringing effect caused by cutting down the frequency suddenly. Here we can see how it changed the shape of the filter image and there's no ringing effect in the output image.

LAB - 10

AIM : Write a program to implement HIGH PASS FILTER And BAND * FILTER in matlab.

```
>> clear
img = imread("cameraman.tif");
img = im2double(img);
[M, N] = size(img);
% STEP 01 : Decide padding size
P = 2*M;
Q = 2*N;
subplot(3, 3, 1);
imshow(img);
title("Original Image");
% STEP 02 : Create padded Image
ap = padarray(img,[M N],0,'post');
subplot(3, 3, 2);
imshow(ap);
title("Padded Image");
% STEP 03 : Center the spectrum
apc = ap;
for i = 0:P-1
for j = 0:Q-1
apc(i+1,j+1) = ap(i+1, j+1) * ((-1)^(i+j));
end
end
subplot(3, 3, 3);
imshow(apc)
title("Centered Image");
% STEP 04 : Taking fourier Transform
F = fft2(apc);
subplot(3, 3, 4);
imshow(F);
title("DFT");
% STEP 05 : Creating Ideal high pass filter
d0 = 30;
H = zeros([P, Q]);
for i = 1:P
for j = 1:Q
D = sqrt((i-P/2)^2 + (j-Q/2)^2);
if(D<d0)
H(i,j) = 1;
else
H(i,j) = 0;
end
end
end
H = 1.-H;
```

```
subplot(3, 3, 5);
imshow(H);
title("Filter Image");
% STEP 06 : Applying filter
G = F .* H;
subplot(3, 3, 6);
imshow(G);
title("Centered Image");
% STEP 07 : Applying Inverse filter
g = ifft2(G);
subplot(3, 3, 7);
imshow(g);
title("IDFT");
% STEP 08 : Centering spectrum
og = ap;
for i = 0:P-1
for j = 0:Q-1
og(i+1,j+1) = g(i+1, j+1) * ((-1)^(i+j));
end
end
subplot(3, 3, 8);
imshow(og)
title("Centered Image");
% STEP 09 : Extrcating Image
og = real(og);
newImage = imcrop(og, [1, 1, M, N]);
subplot(3, 3, 9);
imshow(newImage)
title("Extracted Image");
```



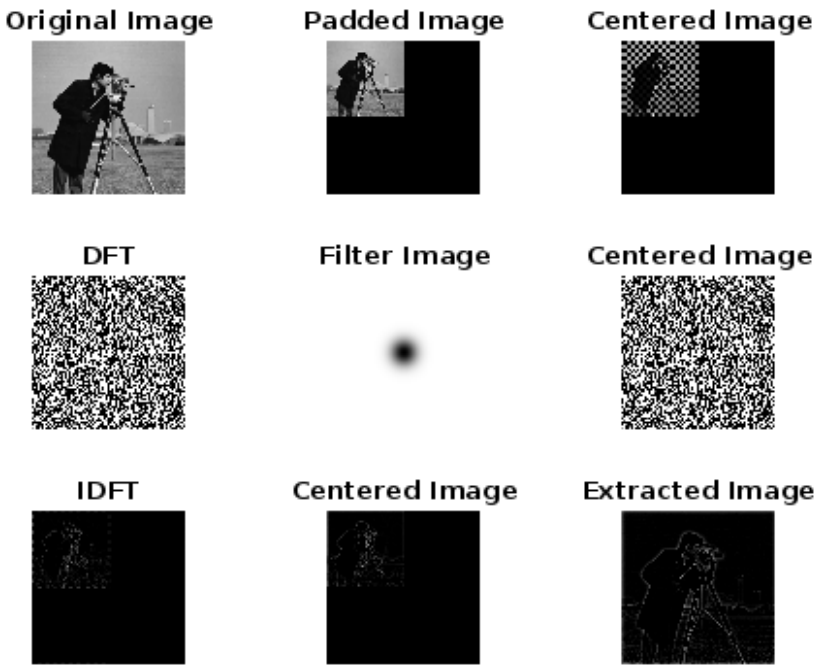
Summarised learning:

High pass filter is nothing but (1 - Low_pass_filter). We are allowing only high frequencies to pass through and cutting down the low frequencies.

Gaussian high pass filter on Image in frequency domain:

```
>> clear
img = imread("cameraman.tif");
img = im2double(img);
[M, N] = size(img);
% STEP 01 : Decide padding size
P = 2*M;
Q = 2*N;
subplot(3, 3, 1);
imshow(img);
title("Original Image");
% STEP 02 : Create padded Image
ap = padarray(img,[M N],0,'post');
subplot(3, 3, 2);
imshow(ap);
title("Padded Image");
% STEP 03 : Center the spectrum
apc = ap;
for i = 0:P-1
for j = 0:Q-1
apc(i+1,j+1) = ap(i+1, j+1) * ((-1)^(i+j));
end
end
subplot(3, 3, 3);
imshow(apc)
title("Centered Image");
% STEP 04 : Taking fourier Transform
F = fft2(apc);
subplot(3, 3, 4);
imshow(F);
title("DFT");
% STEP 05: Creating Gaussian high pass filter
H = zeros([P, Q]);
sigma = 30;
for i = 1:P
for j = 1:Q
D = sqrt((i-P/2)^2 + (j-Q/2)^2);
H(i,j) = exp(-(D^2) / (2*sigma^2));
end
end
H = 1.-H;
subplot(3, 3, 5);
imshow(H);
title("Filter Image");
```

```
% STEP 06 : Applying filter
G = F .* H;
subplot(3, 3, 6);
imshow(G);
title("Centered Image");
% STEP 07 : Applying Inverse filter
g = ifft2(G);
subplot(3, 3, 7);
imshow(g);
title("IDFT");
% STEP 08 : Centering spectrum
og = ap;
for i = 0:P-1
for j = 0:Q-1
og(i+1,j+1) = g(i+1, j+1) * ((-1)^(i+j));
end
end
subplot(3, 3, 8);
imshow(og)
title("Centered Image");
% STEP 09 : Extrcating Image
og = real(og);
newImage = imcrop(og, [1, 1, M, N]);
subplot(3, 3, 9);
imshow(newImage)
title("Extracted Image");
```



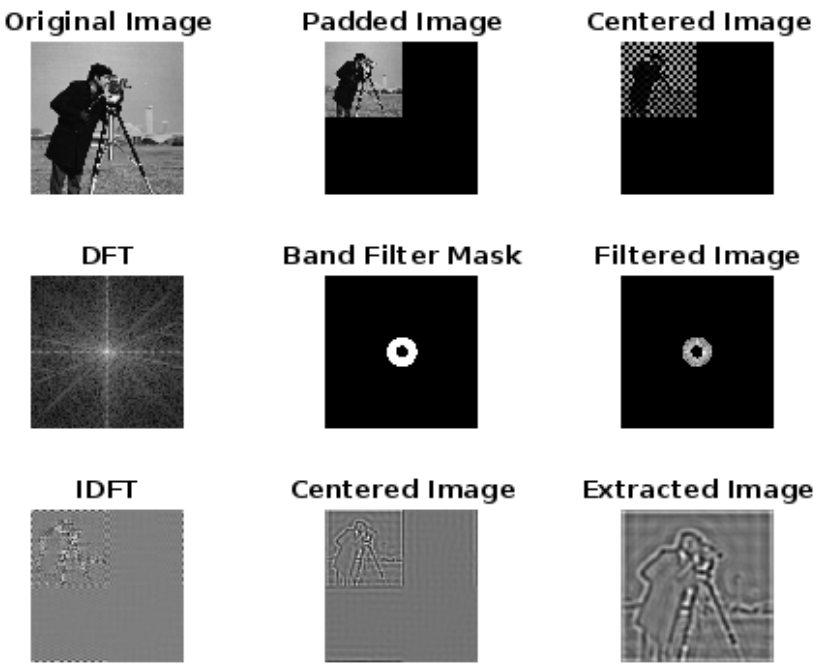
Summarised learning:

As we can see, there's also the ringing effect in the previous filtered image. So inorder to remove that we can use a Gaussian high pass filter and the output is as shown.

Band FILTER:

```
>> clear;
img = imread("cameraman.tif");
img = im2double(img);
[M, N] = size(img);
% STEP 01: Decide padding size
P = 2*M;
Q = 2*N;
subplot(3, 3, 1);
imshow(img);
title("Original Image");
% STEP 02: Create padded Image
ap = padarray(img, [M N], 0, 'post');
subplot(3, 3, 2);
imshow(ap);
title("Padded Image");
% STEP 03: Center the spectrum
apc = ap;
for i = 0:P-1
   for j = 0:Q-1
       apc(i+1, j+1) = ap(i+1, j+1) * ((-1)^(i+j));
   end
end
subplot(3, 3, 3);
imshow(apc);
title("Centered Image");
% STEP 04: Taking Fourier Transform
F = fft2(apc);
subplot(3, 3, 4);
imshow(log(abs(F)+1), []);
title("DFT");
% STEP 05: Creating Band-Pass Filter
H = zeros([P, Q]);
D_low = 20;  % Lower cutoff frequency
D_high = 50; % Upper cutoff frequency
for i = 1:P
   for j = 1:Q
       D = sqrt((i-P/2)^2 + (j-Q/2)^2); % Distance from the center
       if D > D_low && D < D_high
           H(i, j) = 1; % Allow frequencies within the band
       else
           H(i, j) = 0; % Block other frequencies
       end
   end
```

```
end
subplot(3, 3, 5);
imshow(H, []);
title("Band Filter Mask");
% STEP 06: Applying filter
G = F .* H;
subplot(3, 3, 6);
imshow(log(abs(G)+1), []);
title("Filtered Image");
% STEP 07: Applying Inverse Fourier Transform
g = ifft2(G);
subplot(3, 3, 7);
imshow(real(g), []);
title("IDFT");
% STEP 08: Centering spectrum
og = ap;
for i = 0:P-1
   for j = 0:Q-1
       og(i+1, j+1) = g(i+1, j+1) * ((-1)^(i+j));
   end
end
subplot(3, 3, 8);
imshow(real(og), []);
title("Centered Image");
% STEP 09: Extracting Image
og = real(og);
newImage = imcrop(og, [1, 1, N-1, M-1]);
subplot(3, 3, 9);
imshow(newImage, []);
title("Extracted Image");
```



Summarised learning:

This general band-pass filter selectively retains frequencies within a specified range, enabling targeted frequency manipulation in image processing.

LAB - 11

AIM : Write a program to implement huffman algorithm and arithmetic algorithms for compression and decompression processes on images and analyze your o/p.

Huffman algorithm:

```
>> clear
function [Table] = Huff_encode(Vector)
% find unique characters from vector
U = unique(Vector);
% find frequency & Probability
F = histc(Vector, U);
P = [U.'F/length(Vector).'];
% Encode
Q = sortrows(P, 2, 'descend');
Depth = 0;
for i=1:size(U,2) - 1
  code(i) = [repmat('1', [1 Depth]) '0'];
  Depth = Depth+1;
 end
 H(size(U,2),1) = string(repmat('1'), [1 Depth]));
% Q(:,3) = H(:,1);
 varNames = {'Pixels', 'Probability', 'Huffcode'};
 Table = sortrows(table(Q(:,1), Q(:,2), H(:,1), 'VariableNames', varNames), 1);
 Table = sortrows(Table,2);
 disp(Table)
end
```

```
>> function Decoded = Huff_decode(Encoded, HuffTable)
   % Convert Huffman table into a dictionary
   HuffDict = containers.Map(HuffTable.HuffCode, HuffTable.Pixels);
   % Initialize decoding variables
   tempCode = "";
   Decoded = [];
   % Decode each bit sequence
   for i = 1:length(Encoded)
       tempCode = tempCode + Encoded(i); % Build code
       if isKey(HuffDict, tempCode)
           Decoded = [Decoded; HuffDict(tempCode)]; % Append decoded symbol
           tempCode = ""; % Reset temporary code
       end
   end
end
```

Huff_encode.m

```
P = [U.' (F/length(Vector)).'];
% Encode
    Q = sortrows(P,2,'descend');
    Depth =0;
    for i=1:size(U,2)-1
        code = repmat('1',[1 Depth]);
        H(i,1) = string(code + string(0));
        Depth = Depth+1;
    end
    H(size(U,2),1) = string(repmat('1',[1 Depth]));
%     Q(:,3) = H(:,1);
    varNames = {'Pixels','Probability','Huffcode'};
    Table =sortrows (table( Q(:,1),Q(:,2),H(:,1),'VariableNames',varNames),1 );
    disp(Table)

end
```

Command Window

```
>> varNames = {'Pixels','Probability','Huffcode'};
    Table =sortrows (table( Q(:,1),Q(:,2),H(:,1),'VariableNames',varNames),1 );
    disp(Table)
```

Pixels	Probability	Huffcode
1	0.16667	"10"
4	0.16667	"110"
5	0.16667	"1110"
7	0.16667	"11110"
8	0.083333	"11111"
9	0.25	"0"

```
fx >>
```

Editor - Huff_encode.m | Variables - H

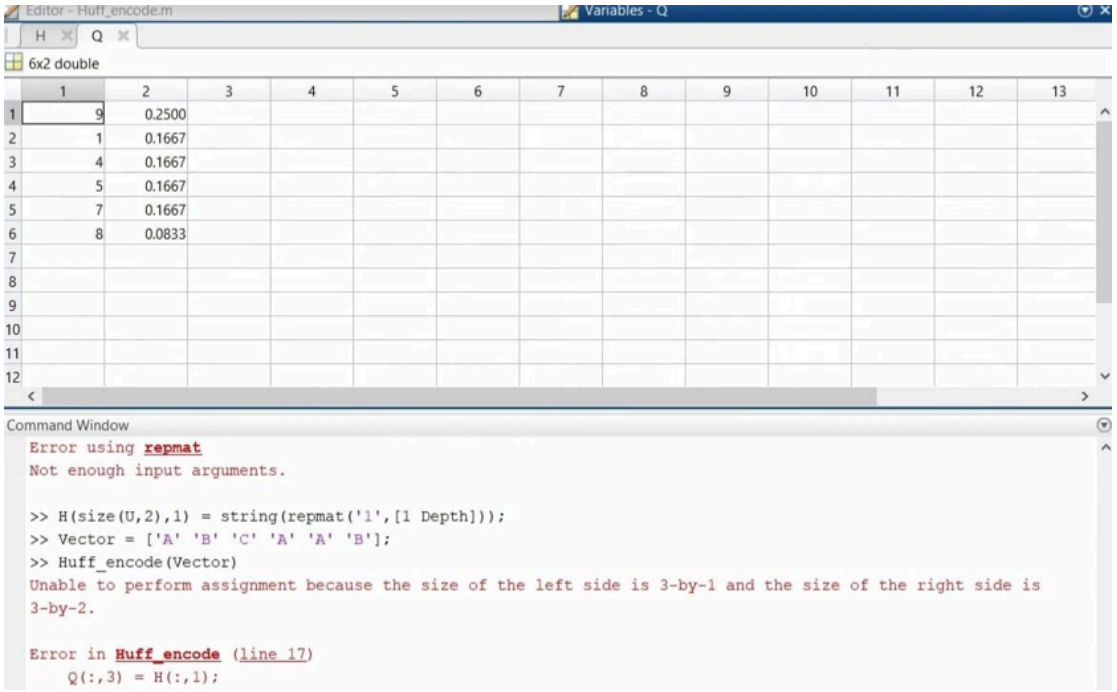
H

6x1 string

	1
1	0
2	10
3	110
4	1110
5	11110
6	11111
7	
8	
9	
10	
11	
12	

Command Window

```
Error using repmat
Not enough input arguments.

>> H(size(U,2),1) = string(repmat('1',[1 Depth]));
>> Vector = ['A' 'B' 'C' 'A' 'A' 'B'];
>> Huff_encode(Vector)
Unable to perform assignment because the size of the left side is 3-by-1 and the size of the right side is
3-by-2.

Error in Huff_encode (line 17)
    Q(:,3) = H(:,1);
```



Summarised learning:

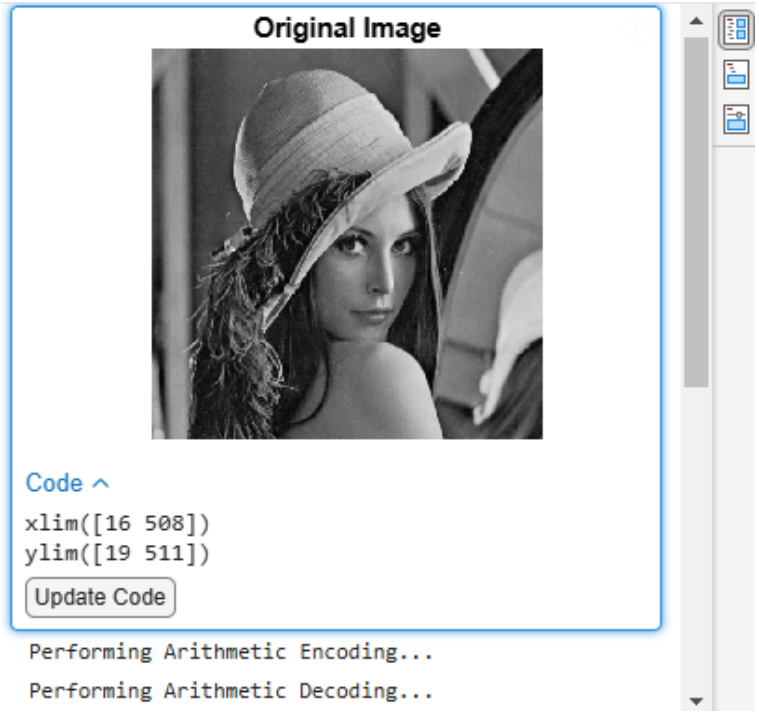
This code thus demonstrates Huffman encoding manually by calculating symbol frequencies, assigning variable-length codes based on probabilities, and organizing the results into a sorted table for efficient compression, whereas decompression involves traversing the Huffman tree using the encoded data to reconstruct the original sequence, demonstrating how variable-length codes can be decoded back into their corresponding symbols efficiently.

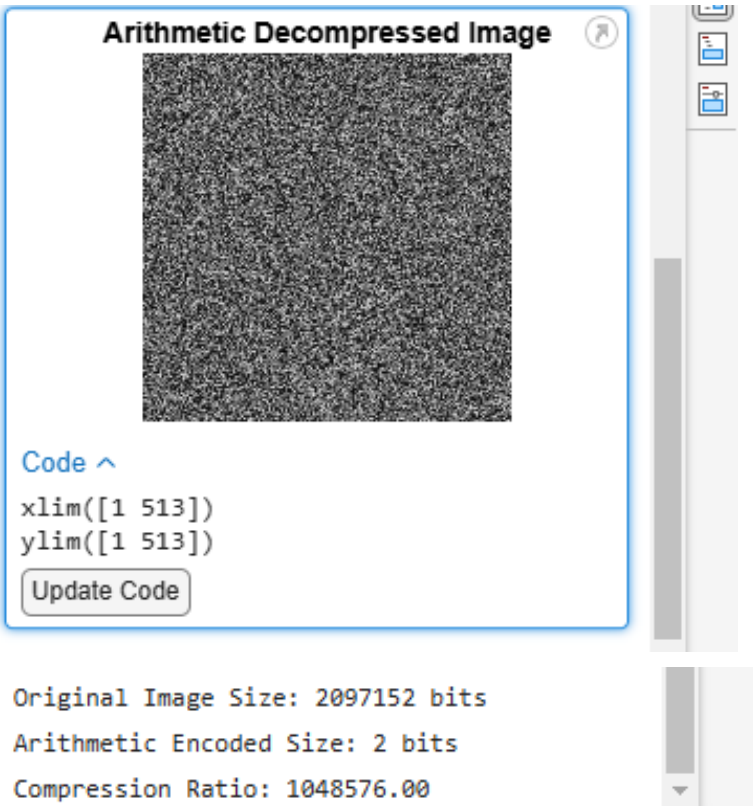
Arithmetic algorithm:

```
>> clc;
clear;
close all;
% Load the image
img = imread('woman.ppm'); % Example grayscale image
if size(img, 3) == 3
   img = rgb2gray(img); % Convert to grayscale if necessary
end
% Display the original image
figure;
imshow(img);
title('Original Image');
```

```
% Flatten the image into a 1D vector
img_vector = img(:);
% Arithmetic Encoding
fprintf('Performing Arithmetic Encoding...\n');
[encodedValue, symbolTable] = Arithmetic_encode(img_vector);
% Arithmetic Decoding
fprintf('Performing Arithmetic Decoding...\n');
decoded_vector = Arithmetic_decode(encodedValue, symbolTable, length(img_vector));
% Reshape the decoded vector into the original image dimensions
decoded_img = reshape(decoded_vector, size(img));
% Display the decompressed image
figure;
imshow(uint8(decoded_img));
title('Arithmetic Decompressed Image');
% Analyze Compression
original_size = numel(img_vector) * 8; % Original size in bits
encoded_size = ceil(log2(1 / abs(encodedValue))) + 1; % Encoded size in bits
compression_ratio = original_size / encoded_size;
fprintf('Original Image Size: %d bits\n', original_size);
fprintf('Arithmetic Encoded Size: %d bits\n', encoded_size);
fprintf('Compression Ratio: %.2f\n', compression_ratio);

function [encodedValue, symbolTable] = Arithmetic_encode(data)
   % Step 1: Calculate symbol probabilities
   symbols = unique(data);
   counts = histcounts(data, [symbols; max(symbols) + 1]);
   probabilities = counts / sum(counts);

   % Step 2: Build symbol table with ranges
   symbolTable = struct();
   low = 0;
   for i = 1:length(symbols)
       high = low + probabilities(i);
       symbolTable.(sprintf('s%d', symbols(i))) = [low, high];
       low = high;
   end
   % Step 3: Arithmetic encoding
   lower = 0;
   upper = 1;
   for i = 1:length(data)
       symbol = sprintf('s%d', data(i));
       range = upper - lower;
       lower = lower + range * symbolTable.(symbol)(1);
       upper = lower + range * (symbolTable.(symbol)(2) -
symbolTable.(symbol)(1));
   end

   % Final encoded value
   encodedValue = (lower + upper) / 2;
end
```

```
function decoded = Arithmetic_decode(encodedValue, symbolTable, dataLength)
   % Step 1: Invert the symbol table for lookup
   symbols = fieldnames(symbolTable);
   ranges = cell2mat(struct2cell(symbolTable));

   % Step 2: Decode symbols
   decoded = zeros(dataLength, 1);
   value = encodedValue;
   for i = 1:dataLength
       for j = 1:length(symbols)
           if value >= ranges(j, 1) && value < ranges(j, 2)
               decoded(i) = str2double(symbols{j}(2:end));
               range = ranges(j, 2) - ranges(j, 1);
               value = (value - ranges(j, 1)) / range;
               break;
           end
       end
   end
end
```





Summarised learning:

Arithmetic encoding efficiently compresses image data by representing it as a single value within a range based on symbol probabilities, while decoding accurately reconstructs the original data using the encoded value and probability ranges.

LAB - 12

AIM : Carry out the following algorithm and processes.

1. Write a program to implement LZW algorithms for compression and decompression processes on image and analyze your o/p.

```
>> % % %Program for LZW Encoding and Decoding
close all;
clear all;
clc;
% string to compress
str = 'DAFDADRDAFDA';
str=reshape(str.',1,[] );
display(str);
% pack it
[packed,table, alphabet]=LZW_encoder_string(uint8(str));
fprintf('\n');
fprintf('Output code is:\n'), packed
fprintf('\n');
fprintf('Output Table (New Elements):\n');
% print table
[m n] = size(table);
for i = numel(alphabet)+1:n
   fprintf('%d : %s\n', i, char(table{i}));
end
% return;
fprintf('\n');
fprintf('DECODING\n');
% unpack it
[unpacked,utable]=LZW_decoder_string(packed, alphabet);
% transfer it back to char array
fprintf('\nDecoded Stream is:')
 unpacked = char(alphabet(unpacked))
% test
isOK = strcmp(str,unpacked)
% show new table elements
fprintf('\n');
fprintf('Output Table (New Elements):\n');
% print table
[m n] = size(utable);
for i = numel(alphabet)+1:n
   fprintf('%d : %s\n', i, char(alphabet(utable{i})));
end
% % %Program for LZW Encoding and Decoding of Image
close all;
clear all;
```

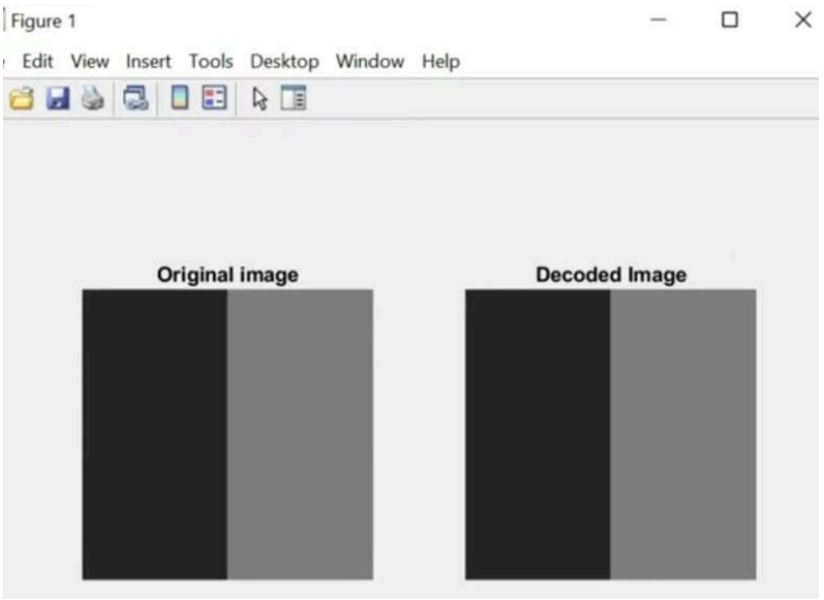
```
clc;
I = [39 39 126 126
    39 39 126 126
    39 39 126 126
    39 39 126 126];
disp('the pixel values are:');
disp(I);
subplot(1,2,1);
imshow(uint8(I));
title('Original image');
str=reshape(I.',1,[] );
display(str);
% pack it
[packed,table, alphabet]=LZW_encoder_image(uint8(str));
fprintf('\n');
fprintf('Output code is:\n'), packed
fprintf('\n');
fprintf('Output Table (New Elements):\n');
% print table
[m n] = size(table);
for i = numel(alphabet)+1:n
   fprintf('%d : %d\n', i, char(table{i}));
end
% return;
fprintf('\n');
fprintf('DECODING\n');
% unpack it
[unpacked,utable]=LZW_decoder_image(packed, alphabet);
% transfer it back to 1-D array
fprintf('\nDecoded Image is:')
 unpacked = alphabet(unpacked)
% show new table elements
fprintf('\n');
fprintf('Output Table (New Elements):\n');
% print table
[m n] = size(utable);
for i = numel(alphabet)+1:n
   fprintf('%d : %d\n', i, char(alphabet(utable{i})));
end
fprintf('\n');
DD=uint8(str);
Restore=reshape(DD,4,4);
subplot(1,2,2);
imshow(Restore.');
title('Decoded Image');
```

```
DECODING
Initialising Table
Table Entry  1: 39
Table Entry  2: 126
```

```
   w       k      output    index  symbol
   ------------------------------------
   NIL     39
```

```
Command Window
New to MATLAB? See resources for Getting Started.

    39      39       1       3       39
    39      39     126       1        4      39
   126     126     126       2        5     126
   126     126      39       2        6     126
    39      39      39
    39      39     126       3        7
    39      39     126     126      126
```

```
Command Window
New to MATLAB? See resources for Getting Started.

the pixel values are:
    39    39   126   126
    39    39   126   126
    39    39   126   126
    39    39   126   126
```



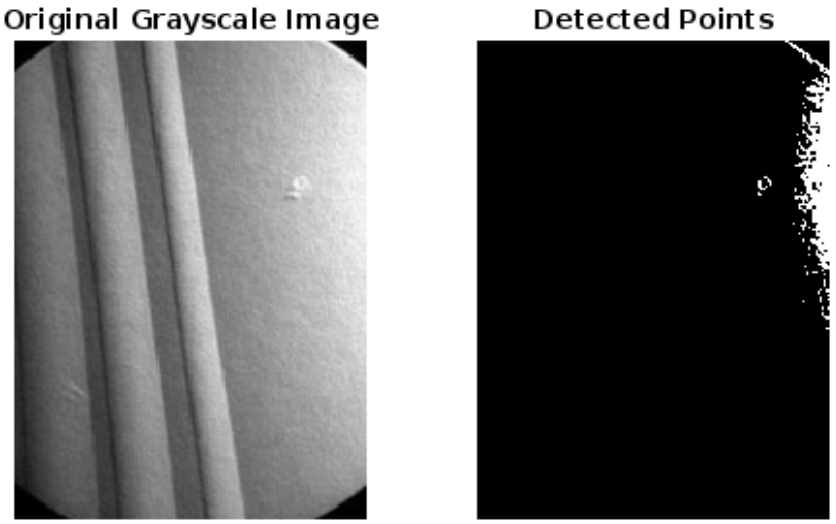
Summarised learning:

The program demonstrates LZW encoding and decoding for both strings and images, efficiently compressing data into codes and reconstructing the original input to verify lossless compression.

2. Implement any one image segmentation process on images to find out point, line and edges.

Point

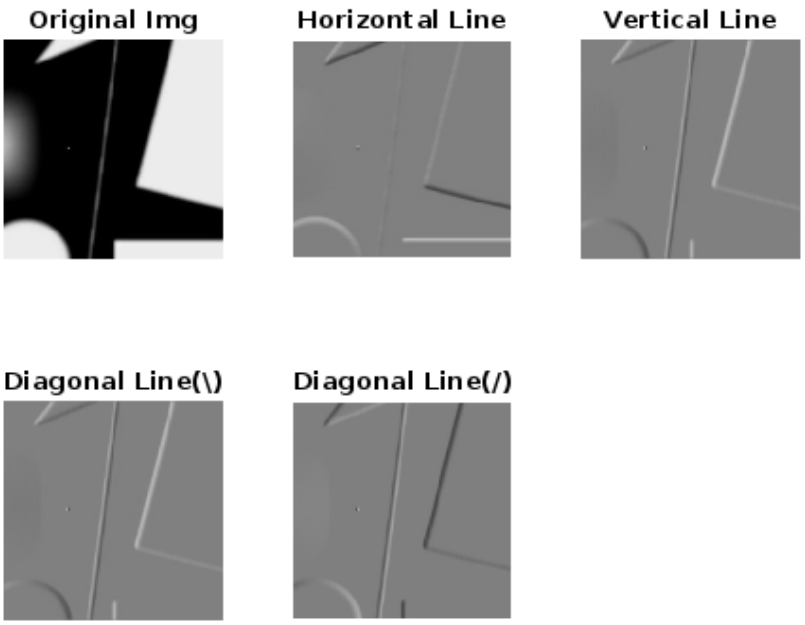
```
>> % Read the input image
img = imread('point.jpg');
% Convert the image to grayscale
gray_img = rgb2gray(img);
threshold = 230;
binary_img = gray_img > threshold;  % Create a binary image (points become white)
% Display the original and output images side by side
figure;
subplot(1, 2, 1);
imshow(gray_img);
title('Original Grayscale Image');
subplot(1, 2, 2);
imshow(binary_img);
title('Detected Points');
```



Line

```
>> % Read the input image
img = imread('line.jpg');
% Convert the image to grayscale
gray_img = rgb2gray(img);
% Define horizontal, vertical, and diagonal masks (kernels)
horizontal_mask = [-1 -1 -1; 0 0 0; 1 1 1];  % Horizontal line detection
vertical_mask = [-1 0 1; -1 0 1; -1 0 1];    % Vertical line detection
diagonal_mask_1 = [-1 0 1; -1 0 1; -1 0 1];  % Diagonal +45 degree
diagonal_mask_2 = [1 0 -1; 1 0 -1; 1 0 -1];  % Diagonal -45 degree
% Apply the filters to the grayscale image using convolution
```

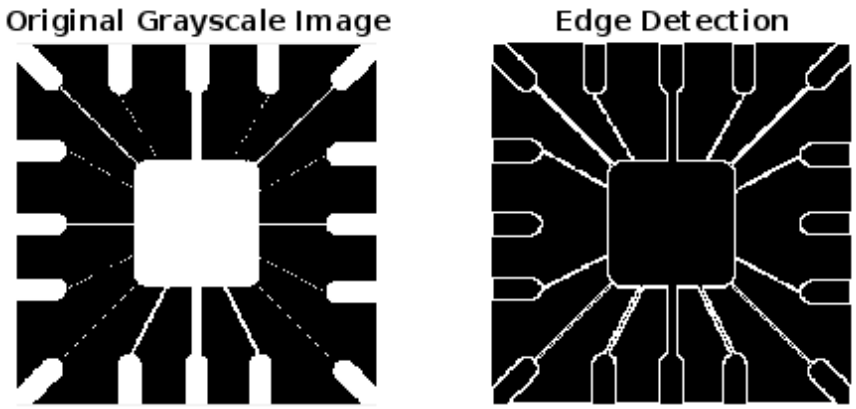
```
horizontal_lines = imfilter(double(gray_img), horizontal_mask);
vertical_lines = imfilter(double(gray_img), vertical_mask);
diagonal_lines_1 = imfilter(double(gray_img), diagonal_mask_1);
diagonal_lines_2 = imfilter(double(gray_img), diagonal_mask_2);
figure;
subplot(2, 3, 1);
imshow(gray_img);
title('Original Img');
subplot(2, 3, 2);
imshow(horizontal_lines, []);
title('Horizontal Line');
subplot(2, 3, 3);
imshow(vertical_lines, []);
title('Vertical Line');
subplot(2, 3, 4);
imshow(diagonal_lines_1, []);
title('Diagonal Line(\)');
subplot(2, 3, 5);
imshow(diagonal_lines_2, []);
title('Diagonal Line(/)');
```



Edge

```
% Read the input image
img = imread('wirebondmask.png');
% Convert the image to grayscale
gray_img = rgb2gray(img);
% Apply Sobel operator to detect edges
sobel_x = fspecial('sobel'); % Sobel filter for x direction
sobel_y = sobel_x';          % Sobel filter for y direction
% Convolve the image with Sobel filters in both directions
```

```
grad_x = imfilter(double(gray_img), sobel_x);
grad_y = imfilter(double(gray_img), sobel_y);
% Calculate the gradient magnitude
gradient_magnitude = sqrt(grad_x.^2 + grad_y.^2);
% Threshold the gradient magnitude to detect edges
edge_img = gradient_magnitude > 100; % Adjust threshold as needed
% Display the original grayscale image and the edge-detected image
subplot(1, 2, 1);
imshow(gray_img);
title('Original Grayscale Image');
subplot(1, 2, 2);
imshow(edge_img);
title('Edge Detection');
```



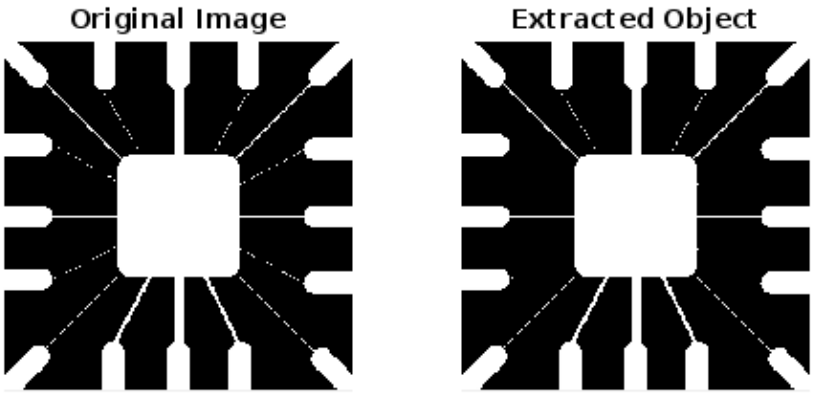
Summarised learning:

The code detects lines, edges, and points in an image by applying appropriate convolution filters (Laplacian for points, Sobel for edges, and custom masks for horizontal, vertical, and diagonal lines) to highlight significant features based on intensity variations.

3. Implement any one image segmentation process on image to extract particular object from image.

```
>> % Read the input image
img = imread('wirebondmask.png');
% Convert the image to grayscale (if it is not already)
gray_img = rgb2gray(img);
% Apply a threshold to create a binary image
threshold = 220;
binary_img = gray_img > threshold;
% Perform morphological operations to clean up the segmentation
binary_img = imfill(binary_img, 'holes');
```

```
binary_img = bwareaopen(binary_img, 50);    % Remove small objects
% Extract the segmented object from the original image using the binary mask
segmented_img = uint8(binary_img) .* img;  % Mask the object
% Display the original and segmented images
figure;
subplot(1, 2, 1);
imshow(img);
title('Original Image');
subplot(1, 2, 2);
imshow(segmented_img);
title('Extracted Object');
```



Summarised learning:

This code segments and extracts a specific object from an image using thresholding and morphological operations to separate it from the background.