Lab Report 1

Aim

To go through the basics of image processing tools like reading, displaying, converting colour and doing some arithmetic operations on the medical images.

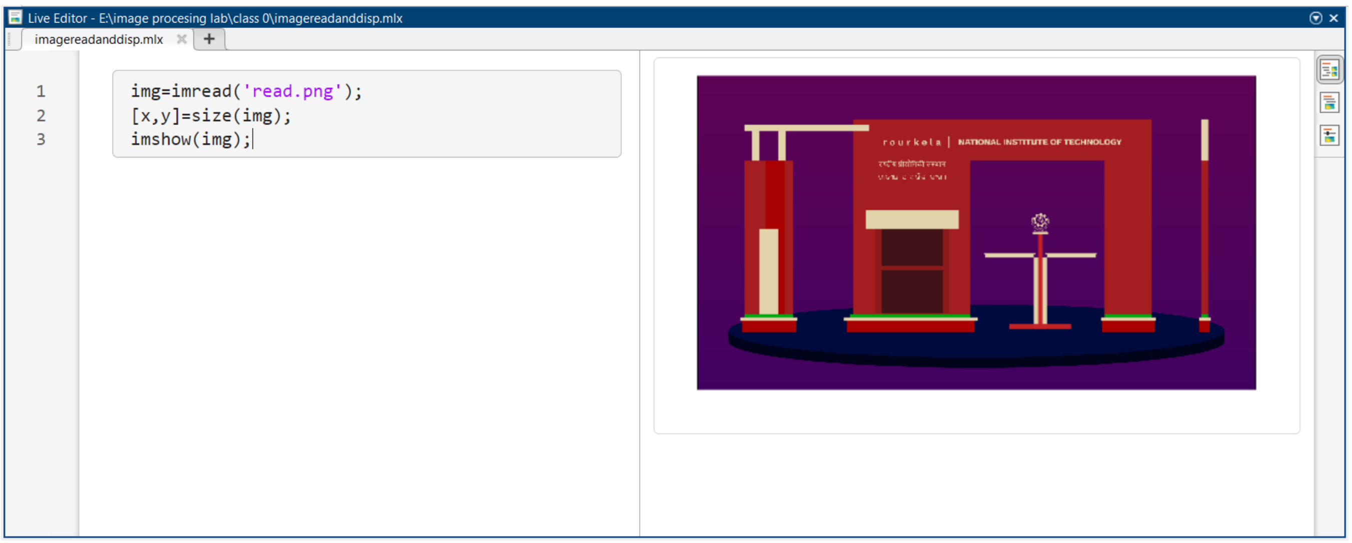
Theory

Reading and Displaying an Image

- Clear the MATLAB workspace of any variables and clear the command window using the commands `clear; close all;` and `clc;`
- Use *`imread()`* to read image files into a matrix in MATLAB. Once you imread an image, it is stored as an ND-array in memory.
- Use *`size()`* to see the property of image matrix like dimensions and colour bit in MATLAB.
- Use *`imshow()`* to show the image.

CODE

```
img=imread('read.png');
[x,y]=size(img);
imshow(img);
```


Live Editor - E:\image procesing lab\class 0\imagereadanddisp.mlx
imagereadanddisp.mlx

```
img=imread('read.png');
[x,y]=size(img);
imshow(img);
```

Figure 1: use of imread( ) and imshow( ) function in MATLAB

Image Arithmetic

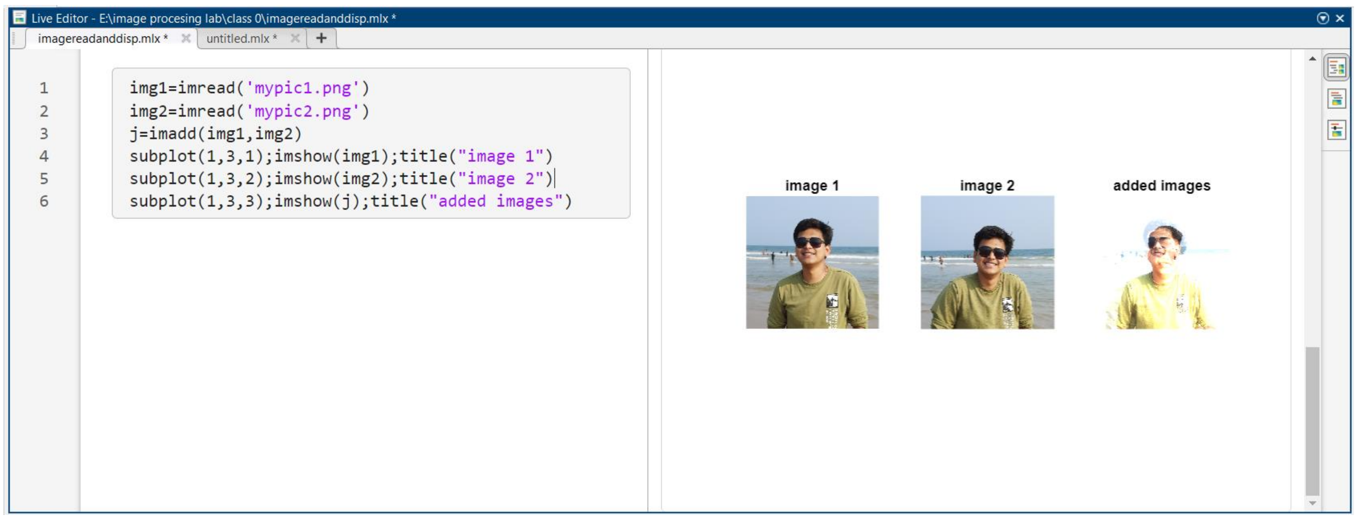
- As images are represented in a matrix format to perform image arithmetic the size of images should be same. Operation on two images leads to a new image
- Use *`imadd()`* to add two image files. The corresponding value of the matrix of two images are added and in return we get a new image. We can also use a constant value instead of image, this will add the

image pixels with a constant. The two images should be of same dimension to do the addition as it involves the matrix addition.

- Use *imsubtract()* to subtract two image files.
- Use *immultiply()* to multiply two files. The corresponding value of the matrix of two images are multiplied and the resultant image is a new image. The two images should be of same dimension to do the multiply as it involves the matrix. We can also use a constant value instead of image, this will multiply the image pixels with a constant. If the pixel values are fractional then it will round it off to nearest value
- Use *imdivide()* to divide two or more image files.

CODE

```
img1=imread('mypic1.png')
img2=imread('mypic2.png')
j=imadd(img1,img2)
subplot(1,3,1);imshow(img1);title("image 1")
subplot(1,3,2);imshow(img2);title("image 2")
subplot(1,3,3);imshow(j);title("added images")
```



Live Editor - E:\image procesing lab\class 0\imagereadanddisp.mlx *

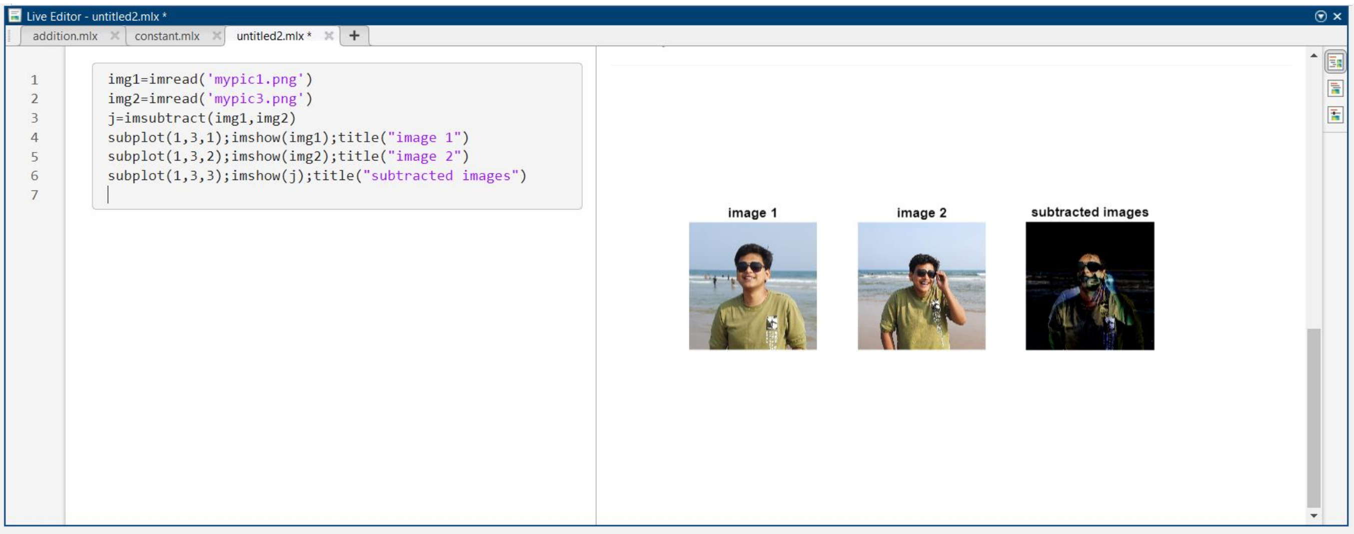
imagereadanddisp.mlx * | untitled.mlx *

```
img1=imread('mypic1.png')
img2=imread('mypic2.png')
j=imadd(img1,img2)
subplot(1,3,1);imshow(img1);title("image 1")
subplot(1,3,2);imshow(img2);title("image 2")
subplot(1,3,3);imshow(j);title("added images")
```

Figure 2: use of imadd( ) to add two images in in MATLAB

CODE

```
img1=imread('mypic1.png')
img2=imread('mypic3.png')
j=imsubtract(img1,img2)
subplot(1,3,1);imshow(img1);title("image 1")
subplot(1,3,2);imshow(img2);title("image 2")
subplot(1,3,3);imshow(j);title("subtracted images")
```



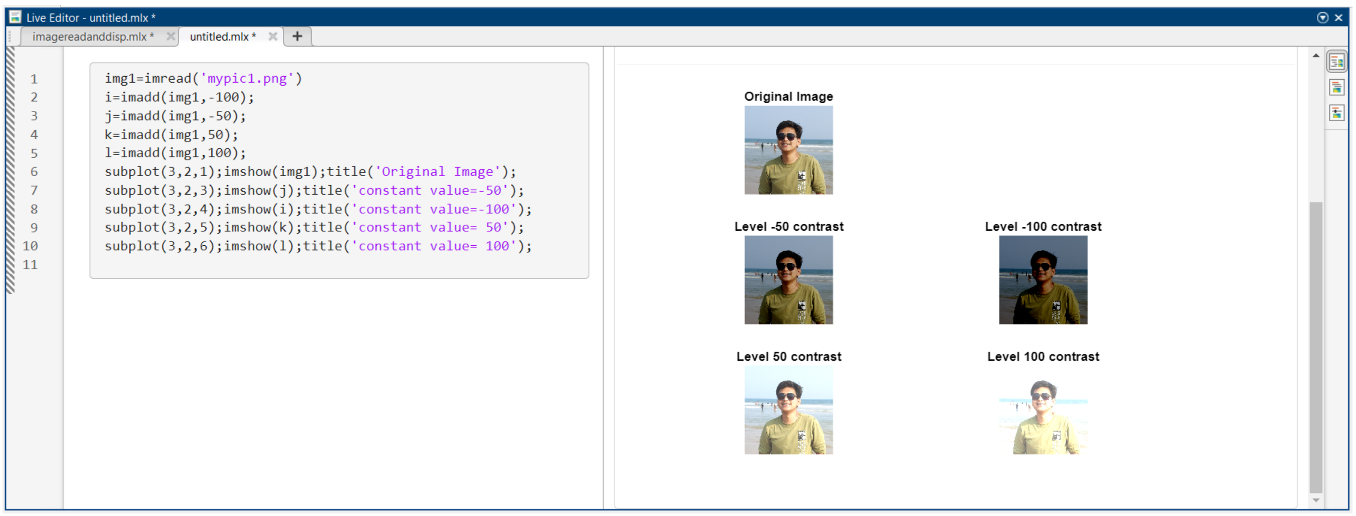
Live Editor - untitled2.mlx *

```
img1=imread('mypic1.png')
img2=imread('mypic3.png')
j=imsubtract(img1,img2)
subplot(1,3,1);imshow(img1);title("image 1")
subplot(1,3,2);imshow(img2);title("image 2")
subplot(1,3,3);imshow(j);title("subtracted images")
```

Figure 3: use of imsubtract( ) to subtract two images in in MATLAB

CODE

```
img1=imread('mypic1.png')
i=imadd(img1,-100);
j=imadd(img1,-50);
k=imadd(img1,50);
l=imadd(img1,100);
subplot(3,2,1);imshow(img1);title('Original Image');
subplot(3,2,3);imshow(j);title('constant value=-50');
subplot(3,2,4);imshow(i);title('constant value=-100');
subplot(3,2,5);imshow(k);title('constant value= 50');
subplot(3,2,6);imshow(l);title('constant value= 100');
```



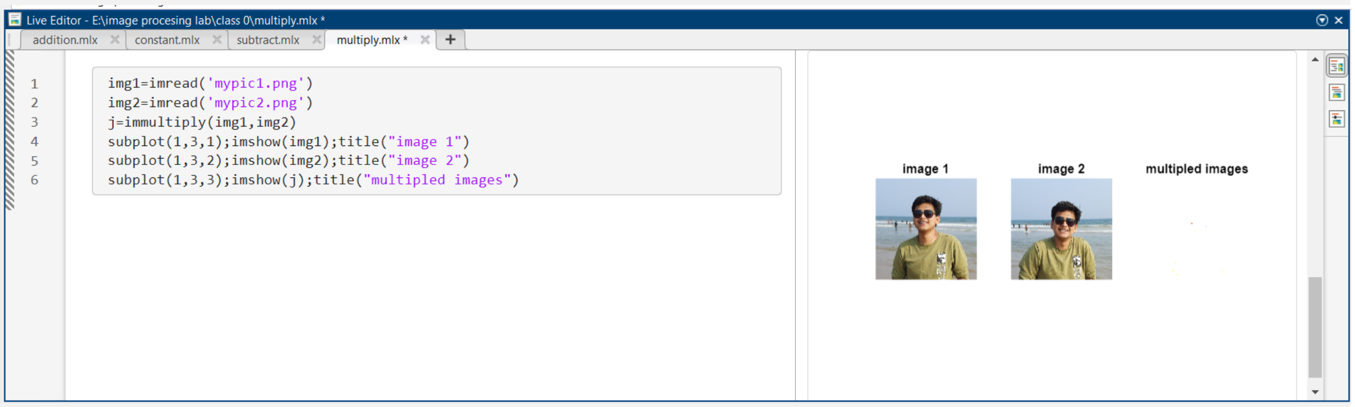
Live Editor - untitled.mlx *

```
img1=imread('mypic1.png')
i=imadd(img1,-100);
j=imadd(img1,-50);
k=imadd(img1,50);
l=imadd(img1,100);
subplot(3,2,1);imshow(img1);title('Original Image');
subplot(3,2,3);imshow(j);title('constant value=-50');
subplot(3,2,4);imshow(i);title('constant value=-100');
subplot(3,2,5);imshow(k);title('constant value= 50');
subplot(3,2,6);imshow(l);title('constant value= 100');
```

Figure 4: use of imadd( ) to add constant to image in MATLAB

CODE

```
img1=imread('mypic1.png')
img2=imread('mypic2.png')
j=immultiply(img1,img2)
subplot(1,3,1);imshow(img1);title("image 1")
subplot(1,3,2);imshow(img2);title("image 2")
subplot(1,3,3);imshow(j);title("multipled images")
```

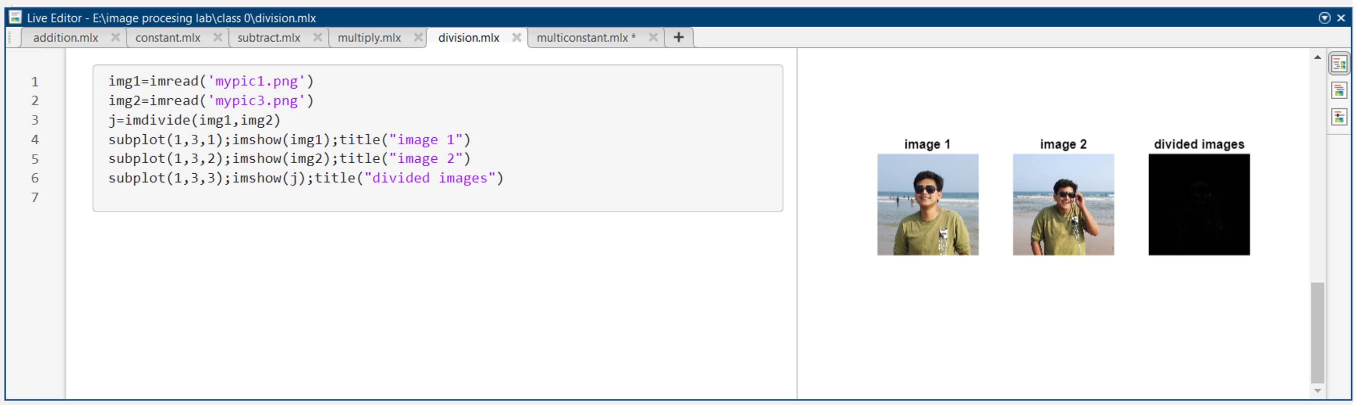


```
img1=imread('mypic1.png')
img2=imread('mypic2.png')
j=immultiply(img1,img2)
subplot(1,3,1);imshow(img1);title("image 1")
subplot(1,3,2);imshow(img2);title("image 2")
subplot(1,3,3);imshow(j);title("multipled images")
```

Figure 5: use of immultiply( ) to multiply two images in in MATLAB

CODE

```
img1=imread('mypic1.png')
img2=imread('mypic2.png')
j=imdivide(img1,img2)
subplot(1,3,1);imshow(img1);title("image 1")
subplot(1,3,2);imshow(img2);title("image 2")
subplot(1,3,3);imshow(j);title("divided images")
```



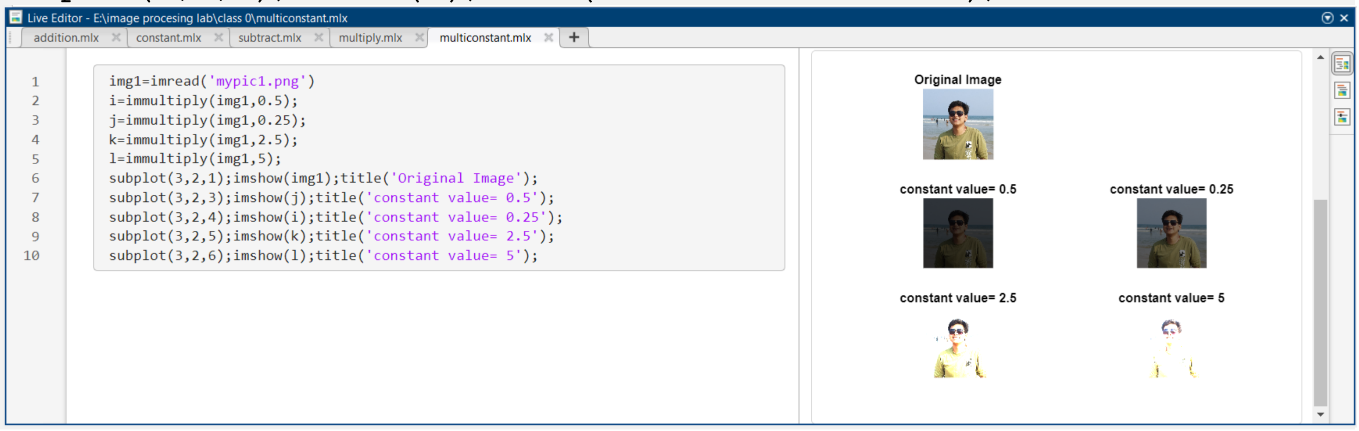
```
img1=imread('mypic1.png')
img2=imread('mypic3.png')
j=imdivide(img1,img2)
subplot(1,3,1);imshow(img1);title("image 1")
subplot(1,3,2);imshow(img2);title("image 2")
subplot(1,3,3);imshow(j);title("divided images")
```

Figure 6: use of imdivide( ) to divide two images in in MATLAB

CODE

```
img1=imread('mypic1.png')
i=immultiply(img1,0.5);
j=immultiply(img1,0.25);
k=immultiply(img1,2.5);
l=immultiply(img1,5);
```

```
subplot(3,2,1);imshow(img1);title('Original Image');
subplot(3,2,3);imshow(j);title('constant value= 0.5');
subplot(3,2,4);imshow(i);title('constant value= 0.25');
subplot(3,2,5);imshow(k);title('constant value= 2.5');
subplot(3,2,6);imshow(l);title('constant value= 5');
```



Live Editor - E:\image procesing lab\class 0\multiconstant.mlx

addition.mlx | constant.mlx | subtract.mlx | multiply.mlx | multiconstant.mlx

```
img1=imread('mypic1.png')
i=immultiply(img1,0.5);
j=immultiply(img1,0.25);
k=immultiply(img1,2.5);
l=immultiply(img1,5);
subplot(3,2,1);imshow(img1);title('Original Image');
subplot(3,2,3);imshow(j);title('constant value= 0.5');
subplot(3,2,4);imshow(i);title('constant value= 0.25');
subplot(3,2,5);imshow(k);title('constant value= 2.5');
subplot(3,2,6);imshow(l);title('constant value= 5');
```

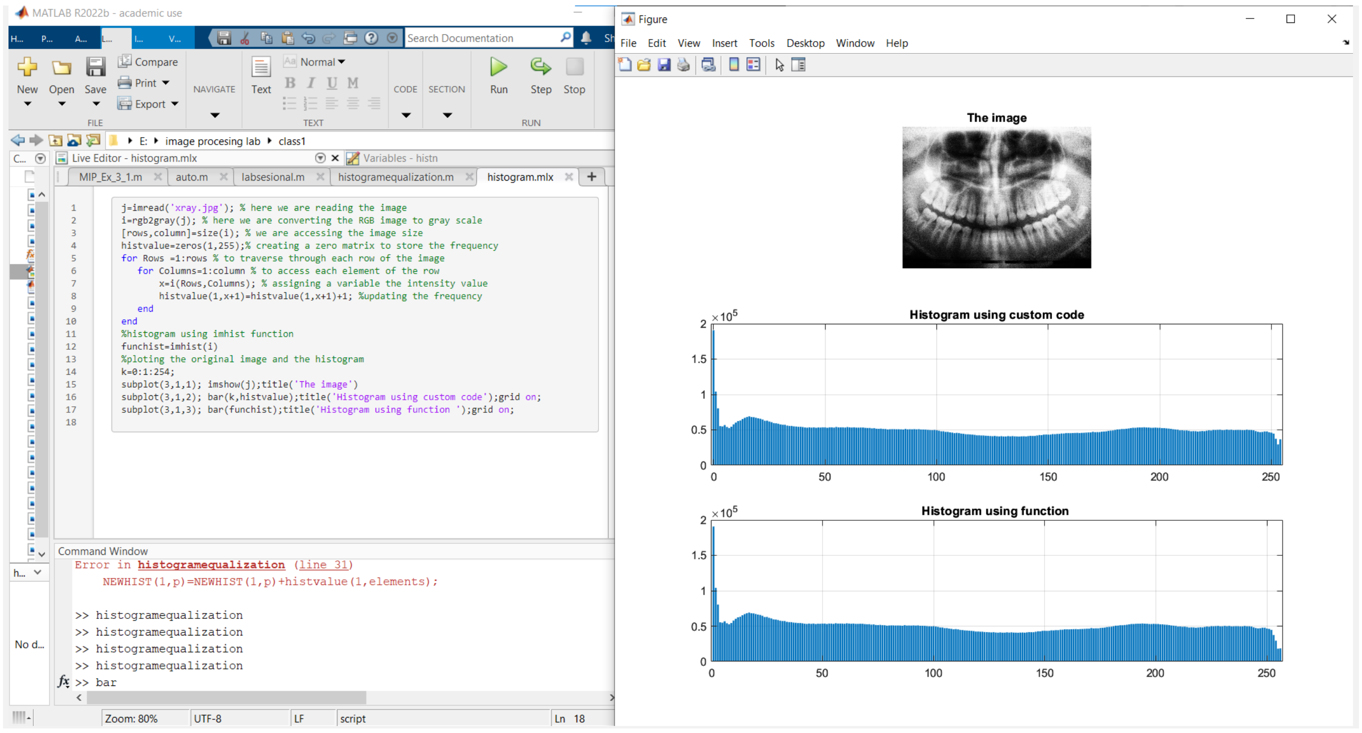
Figure 7: use of immultiply( ) to multiply a constant to images in in MATLAB

Image Histogram

- An image histogram is a gray-scale value distribution showing the frequency of occurrence of each gray-level value.
- *imhist()* displays a plot of the histogram. If the input image is an indexed image, then the histogram shows the distribution of pixel values above a color bar of the colormap cmap.
- The following code describes how to plot a histogram of an image without the inbuilt function.

CODE

```
j=imread('xray.jpg'); % here we are reading the image
i=rgb2gray(j); % here we are converting the RGB image to gray scale
[rows,column]=size(i); % we are accessing the image size
histvalue=zeros(1,255);% creating a zero matrix to store the frequency
for Rows =1:rows % to traverse through each row of the image
   for Columns=1:column % to access each element of the row
       x=i(Rows,Columns); % assigning a variable the intensity value
       histvalue(1,x+1)=histvalue(1,x+1)+1; %updating the frequency
   end
end
%plotting the original image and the histogram
k=0:1:254;
subplot(2,1,1); imshow(j);title('The image')
subplot(2,1,2); plot(k,histvalue);title('Histogram of image');grid on;
```



```matlab
j=imread('xray.jpg'); % here we are reading the image
i=rgb2gray(j); % here we are converting the RGB image to gray scale
[rows,column]=size(i); % we are accessing the image size
histvalue=zeros(1,255);% creating a zero matrix to store the frequency
for Rows =1:rows % to traverse through each row of the image
   for Columns=1:column % to access each element of the row
       x=i(Rows,Columns); % assigning a variable the intensity value
       histvalue(1,x+1)=histvalue(1,x+1)+1; %updating the frequency
   end
end
%histogram using imhist function
funchist=imhist(i)
%ploting the original image and the histogram
k=0:1:254;
subplot(3,1,1); imshow(j);title('The image')
subplot(3,1,2); bar(k,histvalue);title('Histogram using custom code');grid on;
subplot(3,1,3); bar(funchist);title('Histogram using function ');grid on;
```

```
Error in histogramequalization (line 31)
    NEWHIST(1,p)=NEWHIST(1,p)+histvalue(1,elements);

>> histogramequalization
>> histogramequalization
>> histogramequalization
>> histogramequalization
>> bar
```

Figure 8: code for plotting the histogram of an image in MATLAB

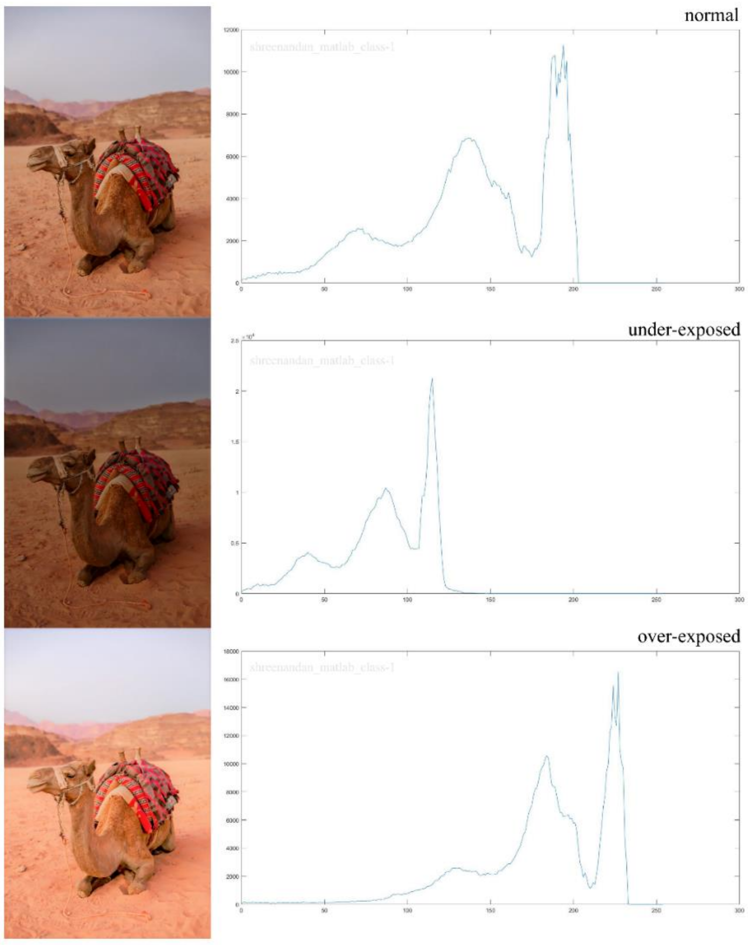


Figure 9: Variation in the histograms of normal, underexposed and over exposed images.

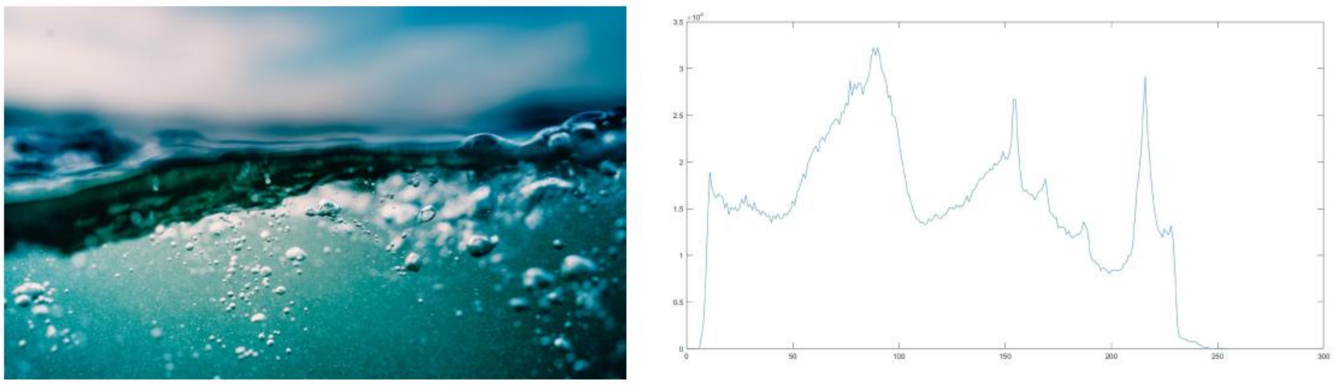


Figure 10: Histograms of underwater images.

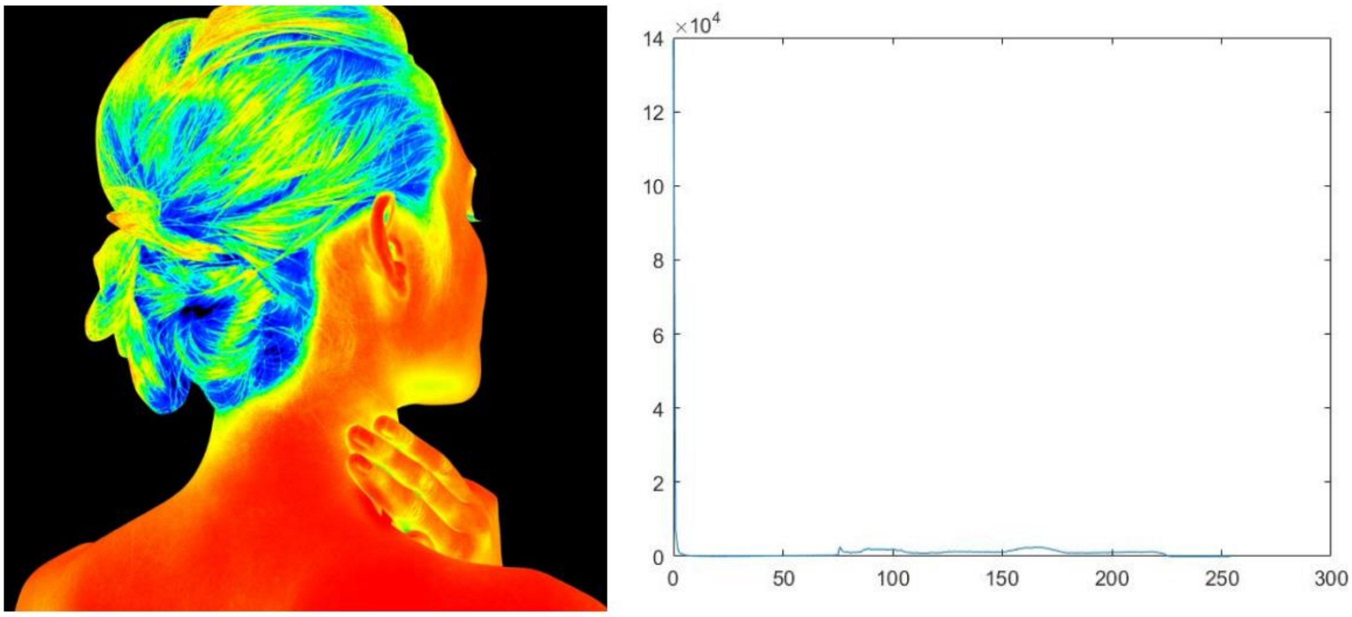


Figure 11: Histogram of thermal images.

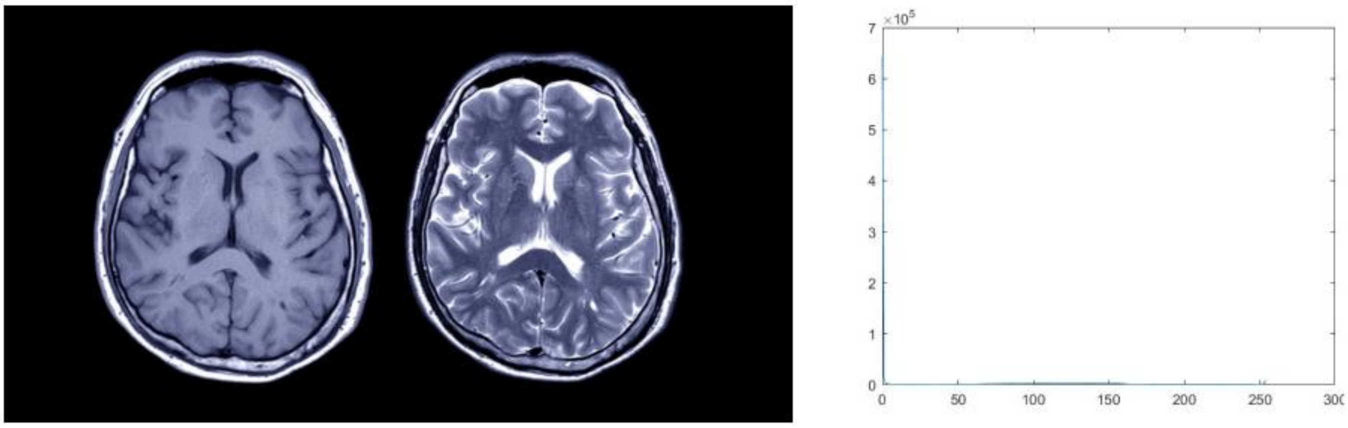


Figure 12: Histogram of medical image (MRI both T1 and T2 waited images).

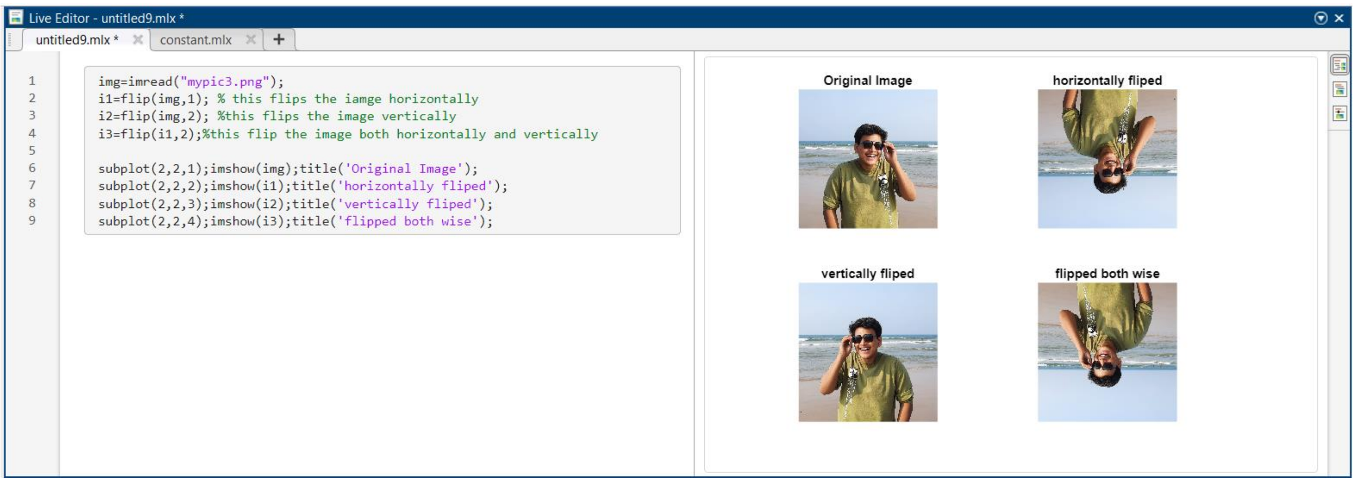
Exercise(Assignment)

Exrcise1.

- B = flip(A,dim) reverses the order of the elements in A along dimension dim. For example, if A is a matrix, then flip(A,1) reverses the elements in each column, and flip(A,2) reverses the elements in each row.

CODE

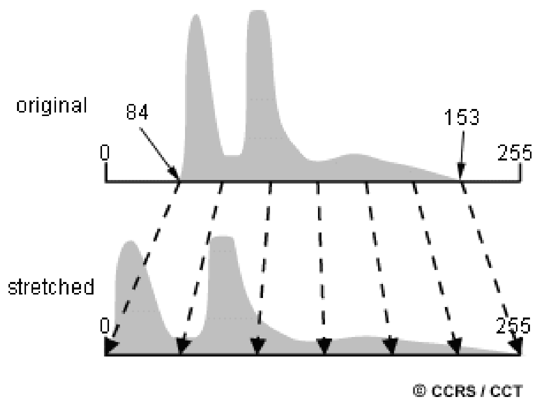
```
img=imread("mypic3.png");
i1=flip(img,1); % this flips the iamge horizontally
i2=flip(img,2); %this flips the image vertically
i3=flip(i1,2);%this flip the image both horizontally and vertically

subplot(2,2,1);imshow(img);title('Original Image');
subplot(2,2,2);imshow(i1);title('horizontally fliped');
subplot(2,2,3);imshow(i2);title('vertically fliped');
subplot(2,2,4);imshow(i3);title('flipped both 
wise');subplot(3,2,6);imshow(l);title('constant value= 5');
```



```
img=imread("mypic3.png");
i1=flip(img,1); % this flips the iamge horizontally
i2=flip(img,2); %this flips the image vertically
i3=flip(i1,2);%this flip the image both horizontally and vertically

subplot(2,2,1);imshow(img);title('Original Image');
subplot(2,2,2);imshow(i1);title('horizontally fliped');
subplot(2,2,3);imshow(i2);title('vertically fliped');
subplot(2,2,4);imshow(i3);title('flipped both wise');
```

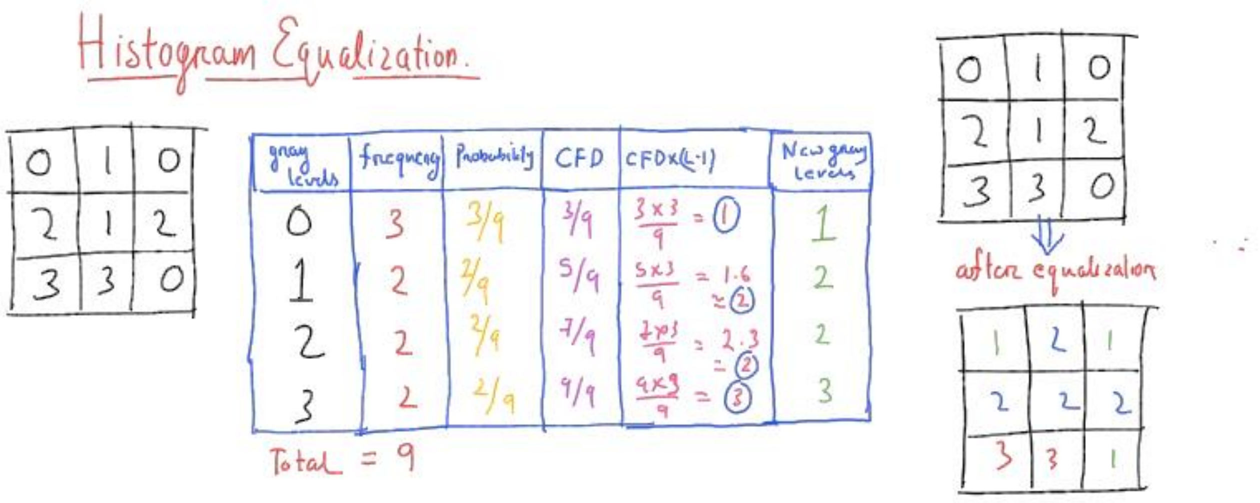
Figure 13: use of flip( ) to flip images in MATLAB

Histogram Equalization



- Histogram Equalization is a computer image processing technique used to improve contrast in images. It accomplishes this by effectively spreading out the most frequent intensity values, i.e. stretching out the intensity range of the image. This method usually increases the global contrast of images when its usable data is represented by close contrast values. This allows for areas of lower local contrast to gain a higher contrast.

Figure 14: graphical depiction of histogram equalization.


Histogram Equalization.

0	1	0
2	1	2
3	3	0

gray levels	frequency	Probability	CFD	CFD×(L-1)	New gray levels
0	3	3/9	3/9	$\frac{3\times3}{9} = \boxed{1}$	1
1	2	2/9	5/9	$\frac{5\times3}{9} = 1.6 \approx \boxed{2}$	2
2	2	2/9	7/9	$\frac{7\times3}{9} = 2.3 \approx \boxed{2}$	2
3	2	2/9	9/9	$\frac{9\times3}{9} = \boxed{3}$	3

Total = 9

0	1	0
2	1	2
3	3	0

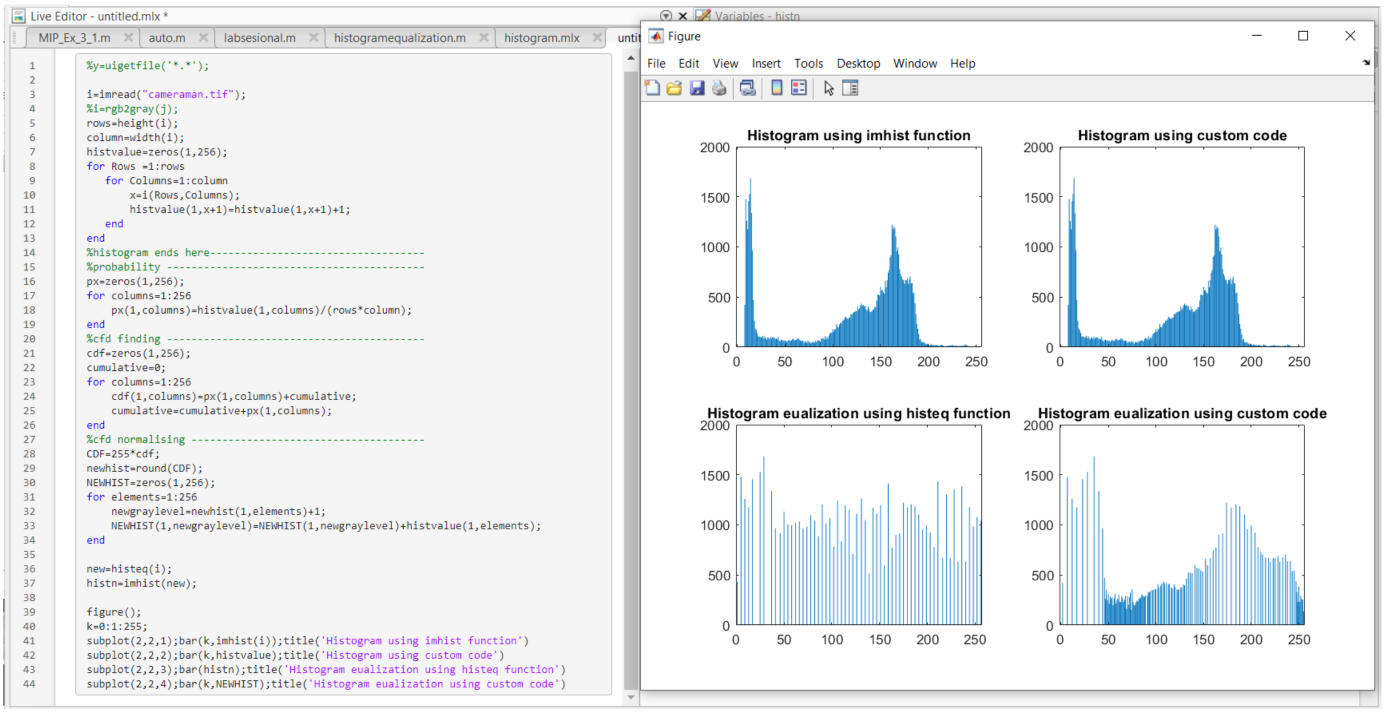
$\Downarrow$
after equalization

1	2	1
2	2	2
3	3	1

Figure 15: process of finding histogram equalization.

CODE

```
y=uigetfile('*.*');
j=imread(y);
i=rgb2gray(j);
rows=height(i);
column=width(i);
histvalue=zeros(1,256);
for Rows =1:rows
   for Columns=1:column
       x=i(Rows,Columns);
       histvalue(1,x+1)=histvalue(1,x+1)+1;
   end
end
%histogram ends here-----------------------------------
%probability -----------------------------------------
px=zeros(1,256);
for columns=1:256
    px(1,columns)=histvalue(1,columns)/(rows*column);
end
%cfd finding -----------------------------------------
cdf=zeros(1,256);
cumulative=0;
for columns=1:256
    cdf(1,columns)=px(1,columns)+cumulative;
    cumulative=cumulative+px(1,columns);
end
%cfd normalising ---------------------------------------
CDF=255*cdf;
newhist=round(CDF);
NEWHIST=zeros(1,256);
for elements=1:256
    newgraylevel=newhist(1,elements)+1;

NEWHIST(1,newgraylevel)=NEWHIST(1,newgraylevel)+histvalue(1,elements);
end


new=histeq(i);
histn=imhist(new);


figure();
k=0:1:255;
subplot(2,2,1);bar(k,imhist(i));title('Histogram using imhist
function')
subplot(2,2,2);bar(k,histvalue);title('Histogram using custom code')
subplot(2,2,3);bar(histn);title('Histogram eualization using histeq
function')
subplot(2,2,4);bar(k,NEWHIST);title('Histogram eualization using
custom code')
```



```
%y=uigetfile('*.*');

i=imread("cameraman.tif");
%i=rgb2gray(j);
rows=height(i);
column=width(i);
histvalue=zeros(1,256);
for Rows =1:rows
   for Columns=1:column
       x=i(Rows,Columns);
       histvalue(1,x+1)=histvalue(1,x+1)+1;
   end
end
%histogram ends here-----------------------------------
%probability ------------------------------------------
px=zeros(1,256);
for columns=1:256
    px(1,columns)=histvalue(1,columns)/(rows*column);
end
%cfd finding ------------------------------------------
cdf=zeros(1,256);
cumulative=0;
for columns=1:256
    cdf(1,columns)=px(1,columns)+cumulative;
    cumulative=cumulative+px(1,columns);
end
%cfd normalising --------------------------------------
CDF=255*cdf;
newhist=round(CDF);
NEWHIST=zeros(1,256);
for elements=1:256
    newgraylevel=newhist(1,elements)+1;
    NEWHIST(1,newgraylevel)=NEWHIST(1,newgraylevel)+histvalue(1,elements);
end

new=histeq(i);
histn=imhist(new);

figure();
k=0:1:255;
subplot(2,2,1);bar(k,imhist(i));title('Histogram using imhist function')
subplot(2,2,2);bar(k,histvalue);title('Histogram using custom code')
subplot(2,2,3);bar(histn);title('Histogram eualization using histeq function')
subplot(2,2,4);bar(k,NEWHIST);title('Histogram eualization using custom code')
```

Live Editor - untitled.mlx *
MIP_Ex_3_1.m
auto.m
labsesional.m
histogramequalization.m
histogram.mlx
Variables - histn
Figure
File Edit View Insert Tools Desktop Window Help
Histogram using imhist function
Histogram using custom code
Histogram eualization using histeq function
Histogram eualization using custom code
0 50 100 150 200 250
0 500 1000 1500 2000

Figure 16: plots showing histograms before and after equalization.

Auto Focus

- Autofocus is the process of improving the image quality on the basis of sharpness of the image. Image which is sharp will have frequency (change in intensity) more as compared to an image of same dimension and same content with less blur.
- By calculating the frequency over all the pixel, we can conclude which image is more focused.

CODE

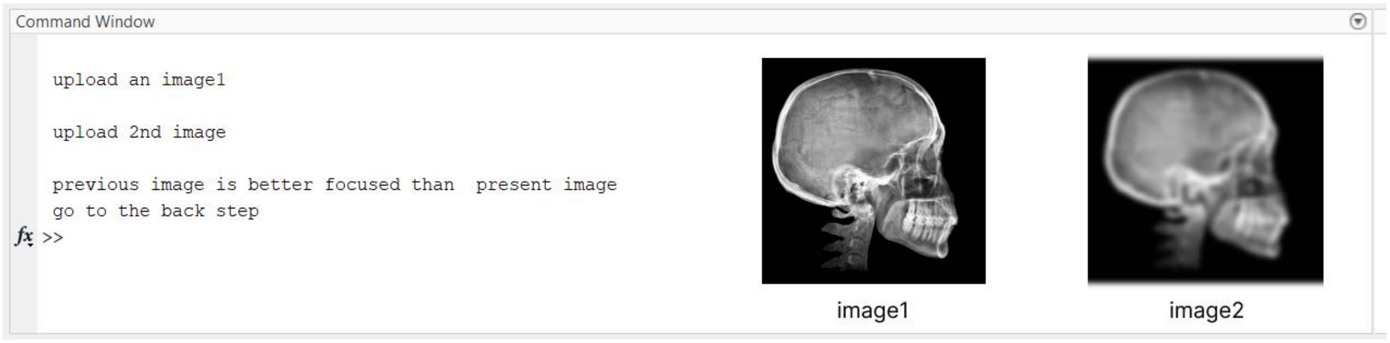
```
function [temp]=autofocus(b)
[m n]=size(b);
temp=double(0);

for i=1:m-1
    for j=1:n
        k(i,j)=b(i+1,j)-b(i,j); %substracting the adjacent row pixels%
        temp=temp+double(k(i,j));% adding all the differenced pixels%
    end
end
end
```

Function 1:function for calculating the cumulative frequencies.

CODE

```
%% reading the images%%%%%
fprintf('\n upload an image1 \n');
x=uigetfile('*.*');                   %to get access to the image
X=imread(x);                                   %read the image
fprintf('\n upload 2nd image\n');
y=uigetfile('*.*');                            % to get access to the
image
Y=imread(y);                                 % read the image
s1=size(X);                        % to find the size of the image
if length(s1)==3      % to find it is color image or gray scale image
X=rgb2gray(X);           %if color image convert it into gray scale
end
s2=size(Y);                     % to find the size of the image
if length(s2)==3    % to find it is color image or gray scale image
Y=rgb2gray(Y);       %if color image convert it into gray scale
end
%% auto focusing %%%%%%%%%%%
k1=autofocus(X);                                %call for auto focus
function
```

```
k2=autofocus(Y);
temp=0;
while(k2>k1)
    fprintf('\n present image is better focused than previous image');
    R=input('still u want to check then type 1 if not 0\n'); % giving
input 1 or 0
    if(R==1)
        fprintf('\n upload another image\n');
        x1=uigetfile('*.*');
        X1=imread(x1);             %if input is 1 read another image
        s3=size(X1);                       %to find size of the image
        if length(s3)==3
        X1=rgb2gray(X1);           % changing color image to gray
        end
        k2=autofocus(X1);                    % call for auto focus
function
     else
         fprintf('\n present image  is best focused\n');
         temp=1;
       break;
   end
end
 if(temp==0)
    fprintf('\n previous image is better focused than  present image\n
go to the back step\n');
 endend
```


Command Window

```
upload an image1

upload 2nd image

previous image is better focused than  present image
go to the back step
>>
```

Figure 17: command window showing which image is more focused.

Shreenandan Sahu |120BM0806

Lab Report 2

Aim

Using Discrete Fourier Transform (DFT) to analyse images and operate various filters on them.

Theory

- The two-dimensional discrete Fourier transform (DFT) of an image f(x,y) of size M x N is represented by:

$$F(u,v) = \sum_{x=0}^{M-1}\sum_{y=0}^{N-1} f(x,y)e^{-j2\pi(ux/M+vy/N)}$$

- The corresponding inverse of the above discrete Fourier transform is given by the following equation

$$f(x,y) = \frac{1}{MN}\sum_{u=0}^{M-1}\sum_{v=0}^{N-1} F(u,v)e^{j2\pi(ux/M+vy/N)}$$

- The magnitude and phase spectrum of an image f (x, y) is represented by

$$F(u,v) = |F(u,v)|\, e^{j\, arg\,\{F(u,v)\}}$$

$$|F(u,v)| = \left[R^2(u,v) + I^2(u,v)\right]^{1/2}$$

$$\phi(u,v) = \tan^{-1}\left[\frac{I(u,v)}{R(u,v)}\right]$$

- where R(u, v) and I(u, v) are the real and imaginary components of the spectrum F(u, v). Similarly, the power spectrum is represented by

$$\begin{aligned} P(u,v) &= |F(u,v)|^2 \\ &= R^2(u,v) + I^2(u,v) \end{aligned}$$

Translation Property:

$$f(x,y)(-1)^{x+y} \quad \xleftrightarrow{FT} \quad F(u\text{-}N/2, v\text{-}N/2)$$

Rotation Property:

$$x = r\cos\theta, \quad y = r\sin\theta, \quad u = \omega\cos\varphi, \quad v = \omega\sin\varphi$$

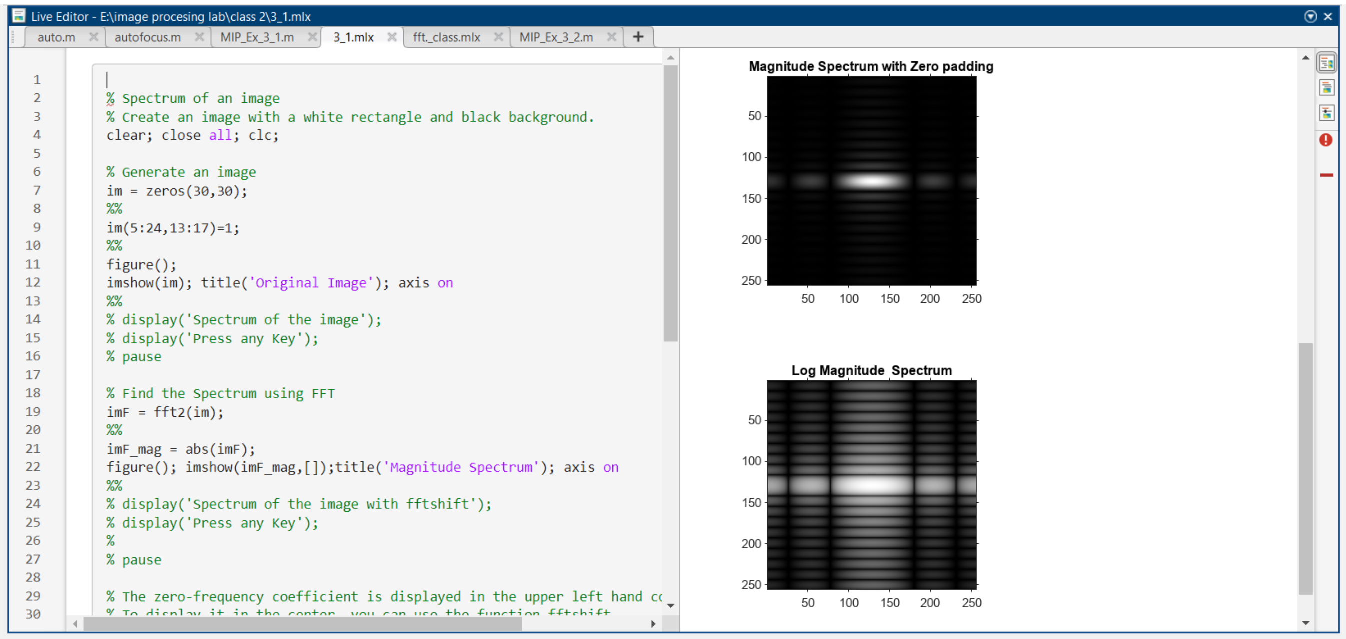
$$f(r, \theta+\theta_0) \quad \xleftrightarrow{FT} \quad F(\omega, \varphi+\varphi_0)$$

- We use *fft2()* and *fftshift()* for DFT and it’s translation property respectively

CODE

```
% Spectrum of an image
% Create an image with a white rectangle and black background.
clear; close all; clc;

% Generate an image
im = zeros(30,30);
%%
im(5:24,13:17)=1;
%%
figure();
imshow(im); title('Original Image'); axis on
%%
% display('Spectrum of the image');
% display('Press any Key');
% pause

% Find the Spectrum using FFT
imF = fft2(im);
%%
imF_mag = abs(imF);
figure(); imshow(imF_mag,[]);title('Magnitude Spectrum'); axis on
%%
% display('Spectrum of the image with fftshift');
% display('Press any Key');
%
% pause

% The zero-frequency coefficient is displayed in the upper left hand
corner.
% To display it in the center, you can use the function fftshift.
imF_mag = fftshift(imF);


imF_mag = abs(imF_mag);
figure(); imshow(imF_mag,[]);title('Magnitude Spectrum with
fftshift'); axis on
%%
```

```
% display('Spectrum of the image with zero padding');
% display('Press any Key');
% pause

% To create a finer sampling of the Fourier transform,
% you can add zero padding to im when computing its DFT.

imF=fft2(im, 256,256);
imF_mag = abs(fftshift(imF));
figure(); imshow(imF_mag,[]); title('Magnitude Spectrum with Zero
padding'); axis on
%%
% display('Spectrum of the image with log magnitude');
% display('Press any Key');
% pause

% To brighten the display, you can use a log function
imF_log_mag=log(1+imF_mag);
figure,imshow(imF_log_mag,[]);title('Log Magnitude  Spectrum'); axis
on

disp('End of the program');
```



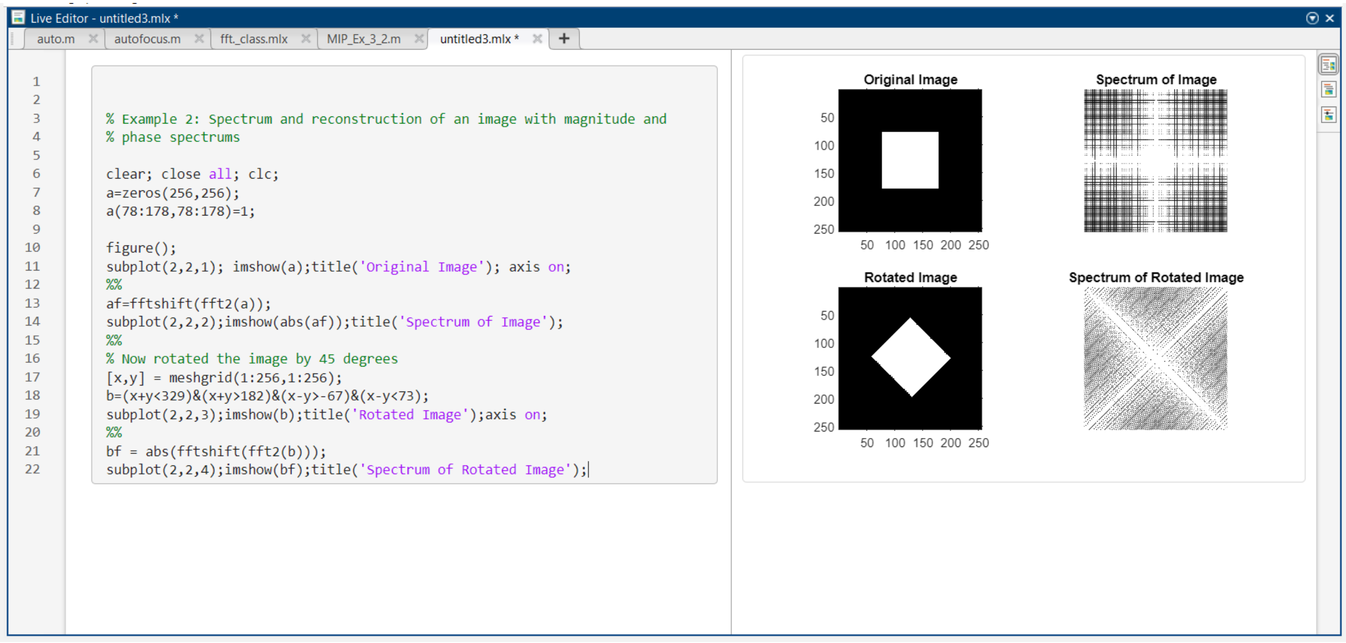
Live Editor - E:\image procesing lab\class 2\3_1.mlx

```

% Spectrum of an image
% Create an image with a white rectangle and black background.
clear; close all; clc;

% Generate an image
im = zeros(30,30);
%%
im(5:24,13:17)=1;
%%
figure();
imshow(im); title('Original Image'); axis on
%%
% display('Spectrum of the image');
% display('Press any Key');
% pause

% Find the Spectrum using FFT
imF = fft2(im);
%%
imF_mag = abs(imF);
figure(); imshow(imF_mag,[]);title('Magnitude Spectrum'); axis on
%%
% display('Spectrum of the image with fftshift');
% display('Press any Key');
%
% pause

% The zero-frequency coefficient is displayed in the upper left hand cc
```

Figure 1: use of fft2( ) and fftshift( ) function in MATLAB

CODE

```
% Example 2: Spectrum and reconstruction of an image with magnitude
and

% phase spectrums


clear; close all; clc;

a=zeros(256,256);

a(78:178,78:178)=1;


figure();

subplot(2,2,1); imshow(a);title('Original Image'); axis on;

%%

af=fftshift(fft2(a));

subplot(2,2,2);imshow(abs(af));title('Spectrum of Image');

%%

% Now rotated the image by 45 degrees

[x,y] = meshgrid(1:256,1:256);

b=(x+y<329)&(x+y>182)&(x-y>-67)&(x-y<73);

subplot(2,2,3);imshow(b);title('Rotated Image');axis on;

%%

bf = abs(fftshift(fft2(b)));

subplot(2,2,4);imshow(bf);title('Spectrum of Rotated Image');
```

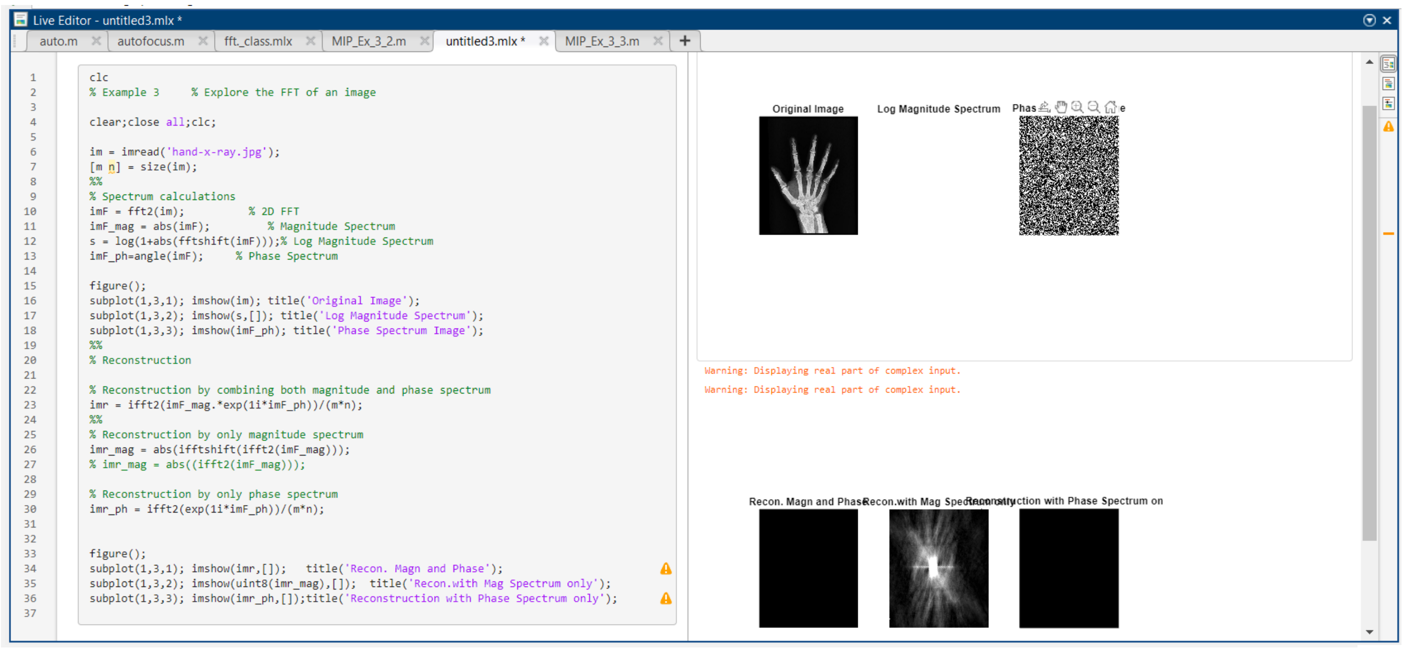


```

% Example 2: Spectrum and reconstruction of an image with magnitude and
% phase spectrums

clear; close all; clc;
a=zeros(256,256);
a(78:178,78:178)=1;

figure();
subplot(2,2,1); imshow(a);title('Original Image'); axis on;
%%
af=fftshift(fft2(a));
subplot(2,2,2);imshow(abs(af));title('Spectrum of Image');
%%
% Now rotated the image by 45 degrees
[x,y] = meshgrid(1:256,1:256);
b=(x+y<329)&(x+y>182)&(x-y>-67)&(x-y<73);
subplot(2,2,3);imshow(b);title('Rotated Image');axis on;
%%
bf = abs(fftshift(fft2(b)));
subplot(2,2,4);imshow(bf);title('Spectrum of Rotated Image');
```

Figure 2: use of fft2( ) and fftshift( ) function in MATLAB

CODE

```
% Example 3     % Explore the FFT of an image

clear;close all;clc;

im = imread('hand-x-ray.jpg');
[m n] = size(im);
%%
% Spectrum calculations
imF = fft2(im);          % 2D FFT
imF_mag = abs(imF);          % Magnitude Spectrum
s = log(1+abs(fftshift(imF)));% Log Magnitude Spectrum
imF_ph=angle(imF);     % Phase Spectrum

figure();
subplot(1,3,1); imshow(im); title('Original Image');
subplot(1,3,2); imshow(s,[]); title('Log Magnitude Spectrum');
subplot(1,3,3); imshow(imF_ph); title('Phase Spectrum Image');
%%
% Reconstruction

% Reconstruction by combining both magnitude and phase spectrum
imr = ifft2(imF_mag.*exp(1i*imF_ph))/(m*n);
%%
% Reconstruction by only magnitude spectrum
imr_mag = abs(ifftshift(ifft2(imF_mag)));
% imr_mag = abs((ifft2(imF_mag)));

% Reconstruction by only phase spectrum
imr_ph = ifft2(exp(1i*imF_ph))/(m*n);



figure();
subplot(1,3,1); imshow(imr,[]);   title('Recon. Magn and Phase');
subplot(1,3,2); imshow(uint8(imr_mag),[]);  title('Recon.with Mag
Spectrum only');
subplot(1,3,3); imshow(imr_ph,[]);title('Reconstruction with Phase
Spectrum only');
```



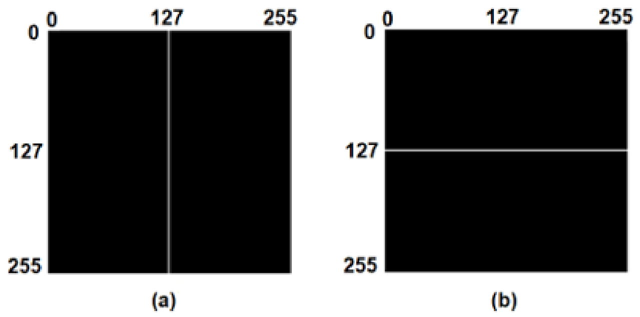
```
clc
% Example 3     % Explore the FFT of an image

clear;close all;clc;

im = imread('hand-x-ray.jpg');
[m n] = size(im);
%%
% Spectrum calculations
imF = fft2(im);          % 2D FFT
imF_mag = abs(imF);         % Magnitude Spectrum
s = log(1+abs(fftshift(imF)));% Log Magnitude Spectrum
imF_ph=angle(imF);     % Phase Spectrum

figure();
subplot(1,3,1); imshow(im); title('Original Image');
subplot(1,3,2); imshow(s,[]); title('Log Magnitude Spectrum');
subplot(1,3,3); imshow(imF_ph); title('Phase Spectrum Image');
%%
% Reconstruction

% Reconstruction by combining both magnitude and phase spectrum
imr = ifft2(imF_mag.*exp(1i*imF_ph))/(m*n);
%%
% Reconstruction by only magnitude spectrum
imr_mag = abs(ifftshift(ifft2(imF_mag)));
% imr_mag = abs((ifft2(imF_mag)));

% Reconstruction by only phase spectrum
imr_ph = ifft2(exp(1i*imF_ph))/(m*n);


figure();
subplot(1,3,1); imshow(imr,[]);   title('Recon. Magn and Phase');
subplot(1,3,2); imshow(uint8(imr_mag),[]);  title('Recon.with Mag Spectrum only');
subplot(1,3,3); imshow(imr_ph,[]);title('Reconstruction with Phase Spectrum only');
```

```
Warning: Displaying real part of complex input.
Warning: Displaying real part of complex input.
```

Figure 3: use of fft2( ) and fftshift( ) function in MATLAB

Exercise

Exercise1: (a) Write a Matlab code to generate the following images. Assume that the width of the white pixel for Fig(a) and height of the white pixel Fig(b) are unity.



(b) Find and display the magnitude and phase spectrums.

(c) Suppose the vertical line in Fig(a) and horizontal line in Fig(b) are rotated by (i) ±30^O, (ii) ±45^O and (iii) ±90^O. Find and display the magnitude and phase spectrums. Comment on the results.

CODE

```
% x=zeros(255,255);
y=zeros(255,255);


    y(127:128,1:255)=1
    x(1:255,127:128)=1;



subplot(3,2,1);imshow(x);title('horizontal white line of height 1px');
subplot(3,2,2);imshow(y);title('vertical white line of width 1px');


fft_x=fft2(x);
fft_shift_x=fftshift(fft_x);
abs_fft_shift_x=abs(fft_shift_x);
abs_fft_x=abs(fft_x);
fft_y=fft2(y);
fft_shift_y=fftshift(fft_y);
abs_fft_shift_y=abs(fft_shift_y);
abs_fft_y=abs(fft_y);
subplot(3,2,3);imshow(abs_fft_x);title('horizontal mag spectrum
without shift');
subplot(3,2,4);imshow(abs_fft_y);title('vertical mag spectrum without
shift');
subplot(3,2,5);imshow(abs_fft_shift_x);title('horizontal  mag spectrum
with shift');
subplot(3,2,6);imshow(abs_fft_shift_y);title('vertical  mag spectrum
with shift');
```

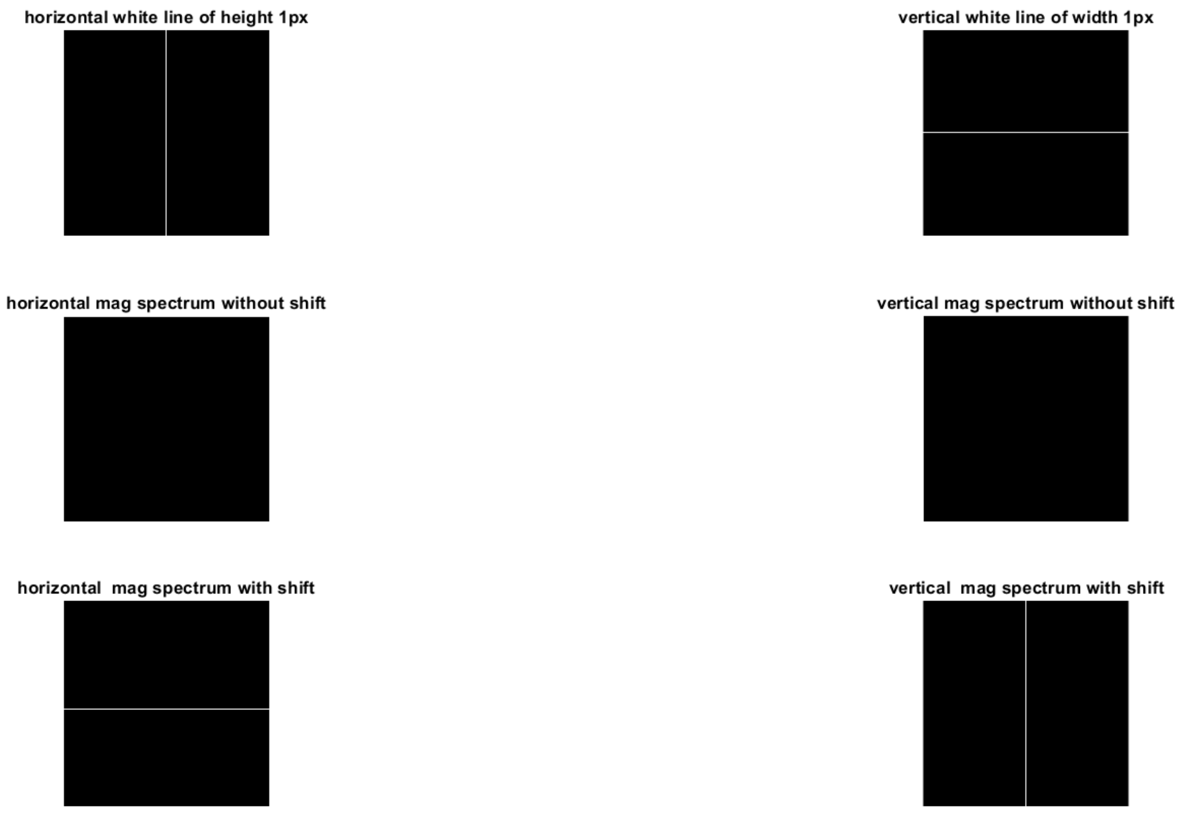
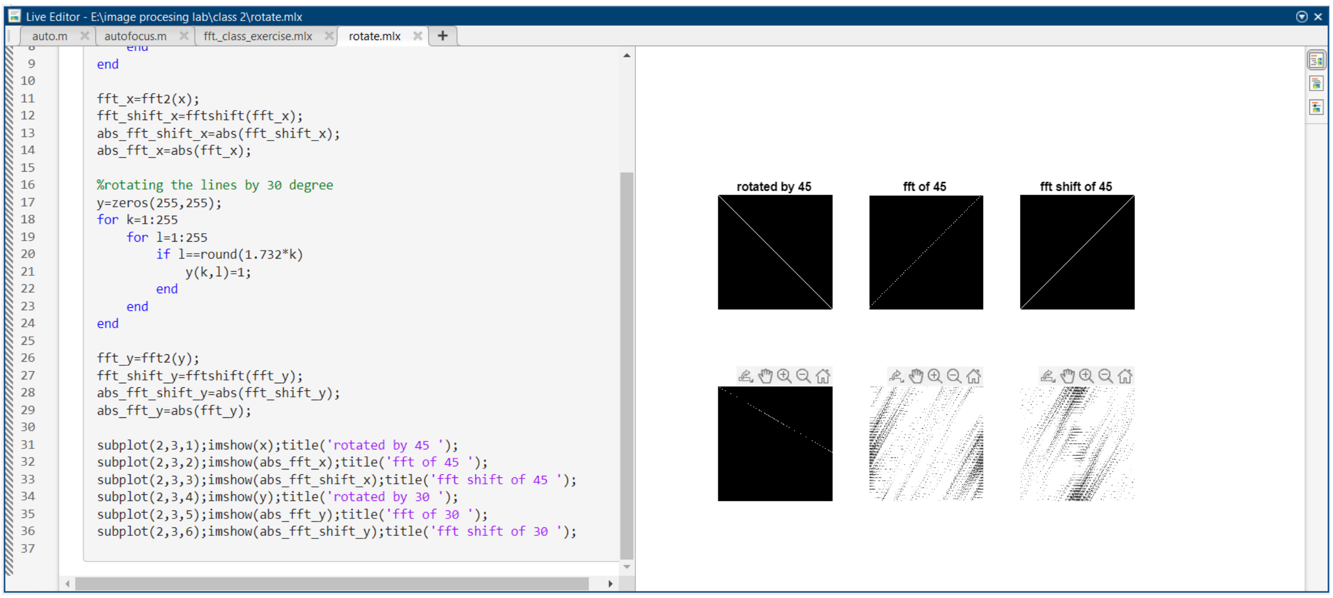

horizontal white line of height 1px
vertical white line of width 1px
horizontal mag spectrum without shift
vertical mag spectrum without shift
horizontal mag spectrum with shift
vertical mag spectrum with shift

Figure 4: use of fft2( ) and fftshift( ) function in MATLAB

CODE

```
% %rotating the lines by 45 degree
x=zeros(255,255);
for i=1:255
    for j=1:255
        if j==i
            x(i,j)=1;
        end
    end
end

fft_x=fft2(x);
fft_shift_x=fftshift(fft_x);
abs_fft_shift_x=abs(fft_shift_x);
abs_fft_x=abs(fft_x);

%rotating the lines by 30 degree
y=zeros(255,255);
for k=1:255
    for l=1:255
        if l==round(1.732*k)
            y(k,l)=1;
        end
    end
end

fft_y=fft2(y);
fft_shift_y=fftshift(fft_y);
abs_fft_shift_y=abs(fft_shift_y);
abs_fft_y=abs(fft_y);

subplot(2,3,1);imshow(x);title('rotated by 45 ');
subplot(2,3,2);imshow(abs_fft_x);title('fft of 45 ');
subplot(2,3,3);imshow(abs_fft_shift_x);title('fft shift of 45 ');
subplot(2,3,4);imshow(y);title('rotated by 30 ');
subplot(2,3,5);imshow(abs_fft_y);title('fft of 30 ');
subplot(2,3,6);imshow(abs_fft_shift_y);title('fft shift of 30 ');
```



```
    end
end

fft_x=fft2(x);
fft_shift_x=fftshift(fft_x);
abs_fft_shift_x=abs(fft_shift_x);
abs_fft_x=abs(fft_x);

%rotating the lines by 30 degree
y=zeros(255,255);
for k=1:255
    for l=1:255
        if l==round(1.732*k)
            y(k,l)=1;
        end
    end
end

fft_y=fft2(y);
fft_shift_y=fftshift(fft_y);
abs_fft_shift_y=abs(fft_shift_y);
abs_fft_y=abs(fft_y);

subplot(2,3,1);imshow(x);title('rotated by 45 ');
subplot(2,3,2);imshow(abs_fft_x);title('fft of 45 ');
subplot(2,3,3);imshow(abs_fft_shift_x);title('fft shift of 45 ');
subplot(2,3,4);imshow(y);title('rotated by 30 ');
subplot(2,3,5);imshow(abs_fft_y);title('fft of 30 ');
subplot(2,3,6);imshow(abs_fft_shift_y);title('fft shift of 30 ');
```

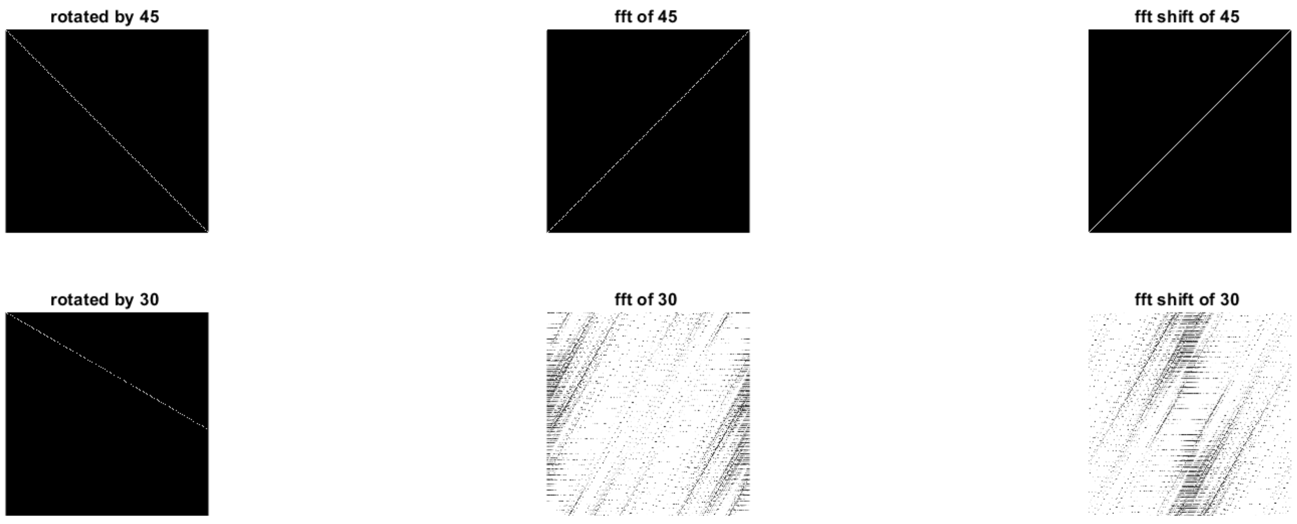
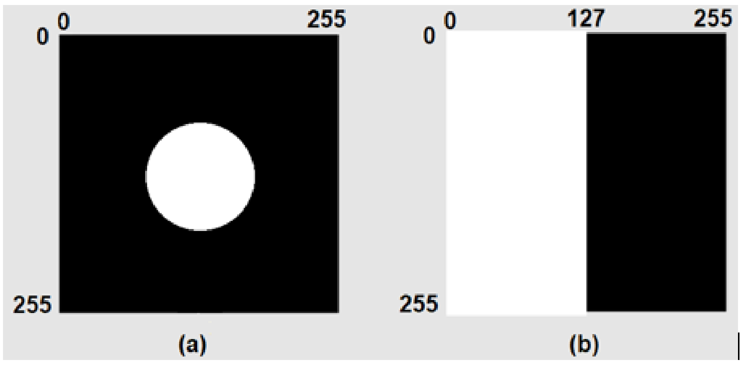


Figure 5: use of fft2( ) and fftshift( ) function in MATLAB

Exercise2: (a) Write a Matlab code to generate the following images. Assume that the radius of circle is 32 for Fig(a).

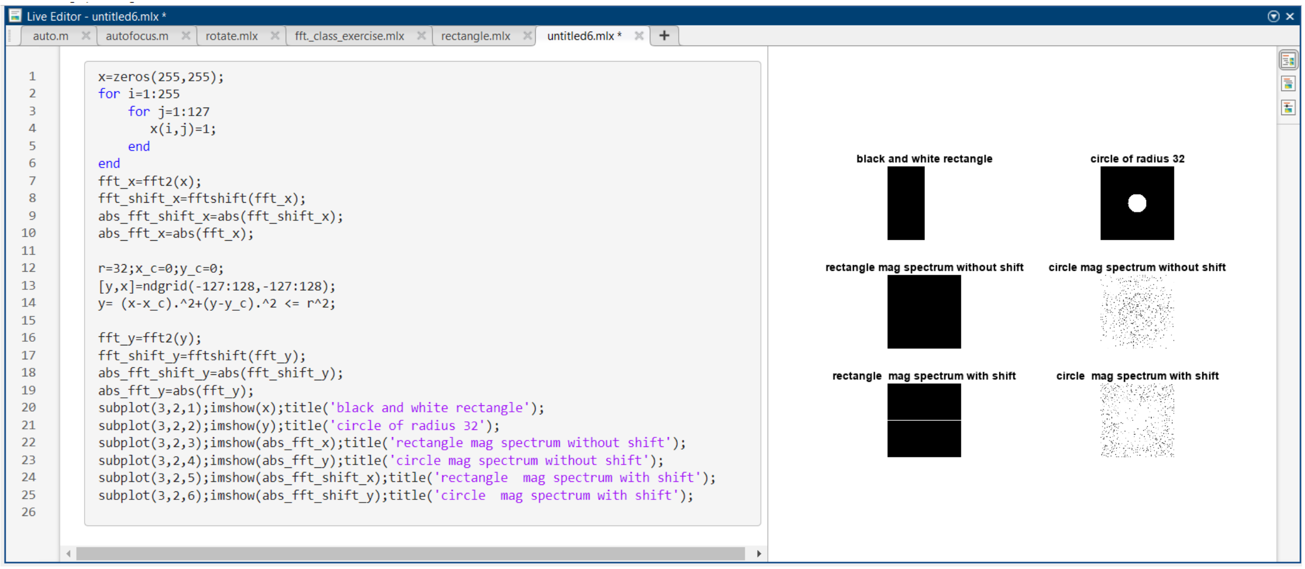
(b) Find and display the magnitude and phase spectrums.



CODE

```
x x=zeros(255,255);
for i=1:255
```

```
    for j=1:127
        x(i,j)=1;
    end
end
fft_x=fft2(x);
fft_shift_x=fftshift(fft_x);
abs_fft_shift_x=abs(fft_shift_x);
abs_fft_x=abs(fft_x);

r=32;x_c=0;y_c=0;
[y,x]=ndgrid(-127:128,-127:128);
y= (x-x_c).^2+(y-y_c).^2 <= r^2;

fft_y=fft2(y);
fft_shift_y=fftshift(fft_y);
abs_fft_shift_y=abs(fft_shift_y);
abs_fft_y=abs(fft_y);
subplot(3,2,1);imshow(x);title('black and white rectangle');
subplot(3,2,2);imshow(y);title('circle of radius 32');
subplot(3,2,3);imshow(abs_fft_x);title('rectangle mag spectrum without
shift');
subplot(3,2,4);imshow(abs_fft_y);title('circle mag spectrum without
shift');
subplot(3,2,5);imshow(abs_fft_shift_x);title('rectangle  mag spectrum
with shift');
subplot(3,2,6);imshow(abs_fft_shift_y);title('circle  mag spectrum
with shift');
```



Live Editor - untitled6.mlx *
auto.m
autofocus.m
rotate.mlx
fft_class_exercise.mlx
rectangle.mlx
untitled6.mlx *
black and white rectangle
circle of radius 32
rectangle mag spectrum without shift
circle mag spectrum without shift
rectangle mag spectrum with shift
circle mag spectrum with shift

```
x=zeros(255,255);
for i=1:255
    for j=1:127
      x(i,j)=1;
    end
end
fft_x=fft2(x);
fft_shift_x=fftshift(fft_x);
abs_fft_shift_x=abs(fft_shift_x);
abs_fft_x=abs(fft_x);

r=32;x_c=0;y_c=0;
[y,x]=ndgrid(-127:128,-127:128);
y= (x-x_c).^2+(y-y_c).^2 <= r^2;

fft_y=fft2(y);
fft_shift_y=fftshift(fft_y);
abs_fft_shift_y=abs(fft_shift_y);
abs_fft_y=abs(fft_y);
subplot(3,2,1);imshow(x);title('black and white rectangle');
subplot(3,2,2);imshow(y);title('circle of radius 32');
subplot(3,2,3);imshow(abs_fft_x);title('rectangle mag spectrum without shift');
subplot(3,2,4);imshow(abs_fft_y);title('circle mag spectrum without shift');
subplot(3,2,5);imshow(abs_fft_shift_x);title('rectangle  mag spectrum with shift');
subplot(3,2,6);imshow(abs_fft_shift_y);title('circle  mag spectrum with shift');
```

Figure 6: use of fft2( ) and fftshift( ) function in MATLAB

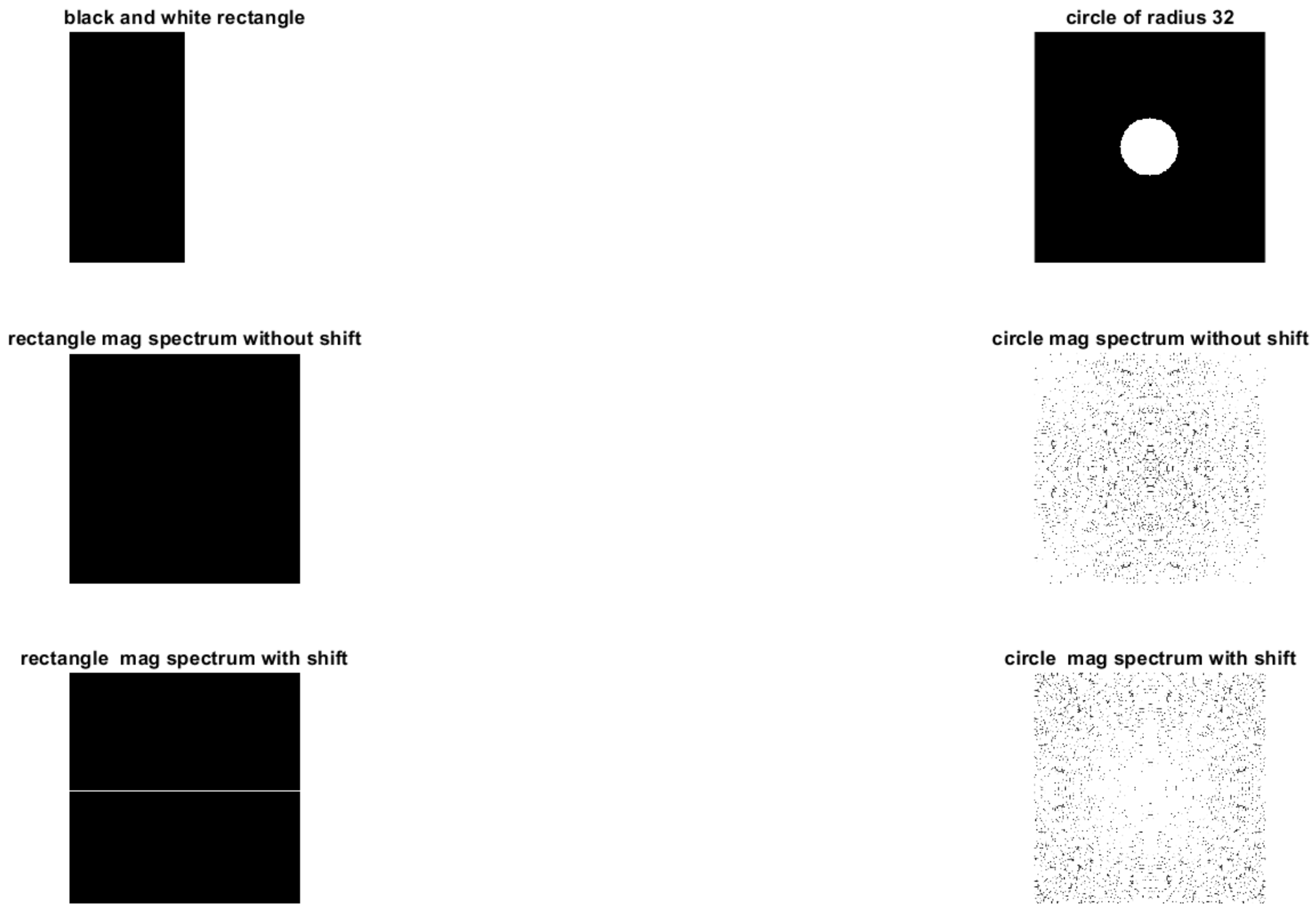
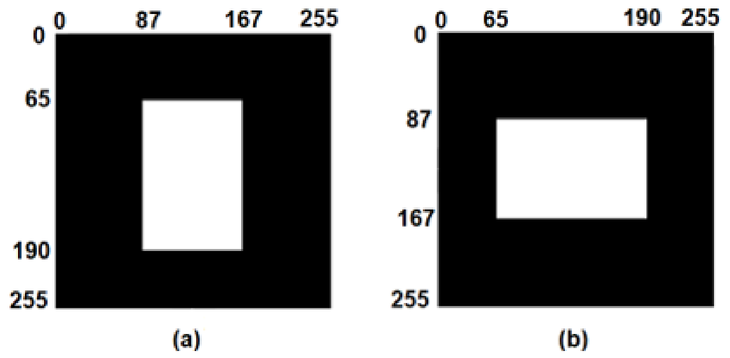

black and white rectangle
circle of radius 32
rectangle mag spectrum without shift
circle mag spectrum without shift
rectangle mag spectrum with shift
circle mag spectrum with shift

Figure 7: use of fft2( ) and fftshift( ) function in MATLAB

Exercise5: (a) Write a Matlab code to generate the following images.



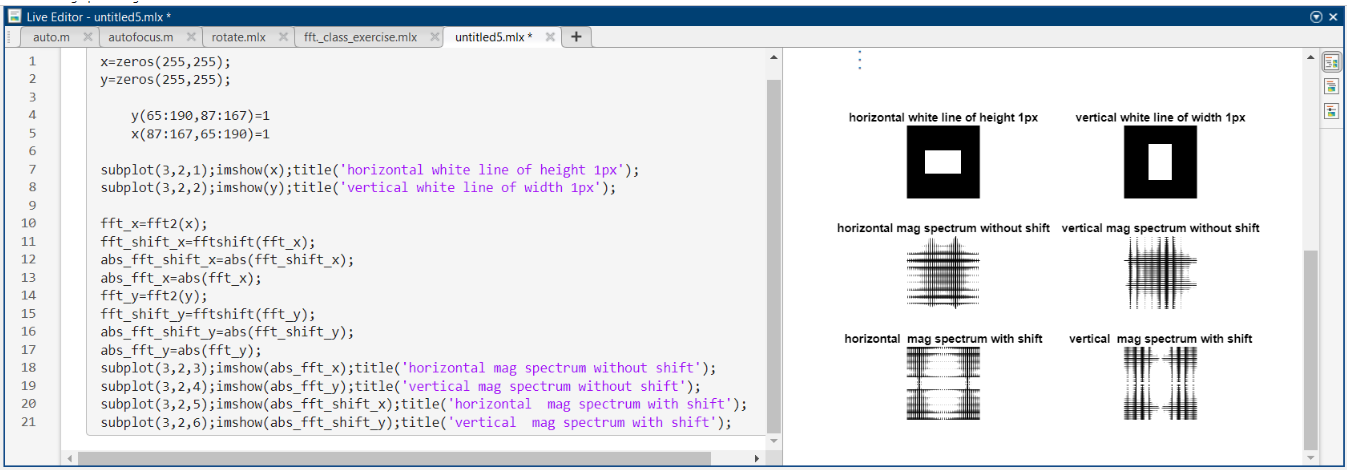
(b) Find and display the magnitude and phase spectrums.

(c) Suppose the white rectangular images are rotated by

(i) ±45^{O} and (ii) ±120^{O}. Find and display the magnitude and phase spectrums. Comment on the results.

CODE

```
x=zeros(255,255);

y=zeros(255,255);


    y(65:190,87:167)=1

    x(87:167,65:190)=1


subplot(3,2,1);imshow(x);title('horizontal white line of height 1px');

subplot(3,2,2);imshow(y);title('vertical white line of width 1px');


fft_x=fft2(x);

fft_shift_x=fftshift(fft_x);

abs_fft_shift_x=abs(fft_shift_x);

abs_fft_x=abs(fft_x);

fft_y=fft2(y);

fft_shift_y=fftshift(fft_y);

abs_fft_shift_y=abs(fft_shift_y);

abs_fft_y=abs(fft_y);


%ploting all the images and there fft

subplot(3,2,3);imshow(abs_fft_x);title('horizontal mag spectrum
without shift');

subplot(3,2,4);imshow(abs_fft_y);title('vertical mag spectrum without
shift');

subplot(3,2,5);imshow(abs_fft_shift_x);title('horizontal  mag spectrum
with shift');

subplot(3,2,6);imshow(abs_fft_shift_y);title('vertical  mag spectrum
with shift');end
```


Live Editor - untitled5.mlx *
auto.m
autofocus.m
rotate.mlx
fft_class_exercise.mlx
untitled5.mlx *
horizontal white line of height 1px
vertical white line of width 1px
horizontal mag spectrum without shift
vertical mag spectrum without shift
horizontal mag spectrum with shift
vertical mag spectrum with shift

```
x=zeros(255,255);
y=zeros(255,255);

    y(65:190,87:167)=1
    x(87:167,65:190)=1

subplot(3,2,1);imshow(x);title('horizontal white line of height 1px');
subplot(3,2,2);imshow(y);title('vertical white line of width 1px');

fft_x=fft2(x);
fft_shift_x=fftshift(fft_x);
abs_fft_shift_x=abs(fft_shift_x);
abs_fft_x=abs(fft_x);
fft_y=fft2(y);
fft_shift_y=fftshift(fft_y);
abs_fft_shift_y=abs(fft_shift_y);
abs_fft_y=abs(fft_y);
subplot(3,2,3);imshow(abs_fft_x);title('horizontal mag spectrum without shift');
subplot(3,2,4);imshow(abs_fft_y);title('vertical mag spectrum without shift');
subplot(3,2,5);imshow(abs_fft_shift_x);title('horizontal  mag spectrum with shift');
subplot(3,2,6);imshow(abs_fft_shift_y);title('vertical  mag spectrum with shift');
```

Figure 8: use of fft2( ) and fftshift( ) function in MATLAB

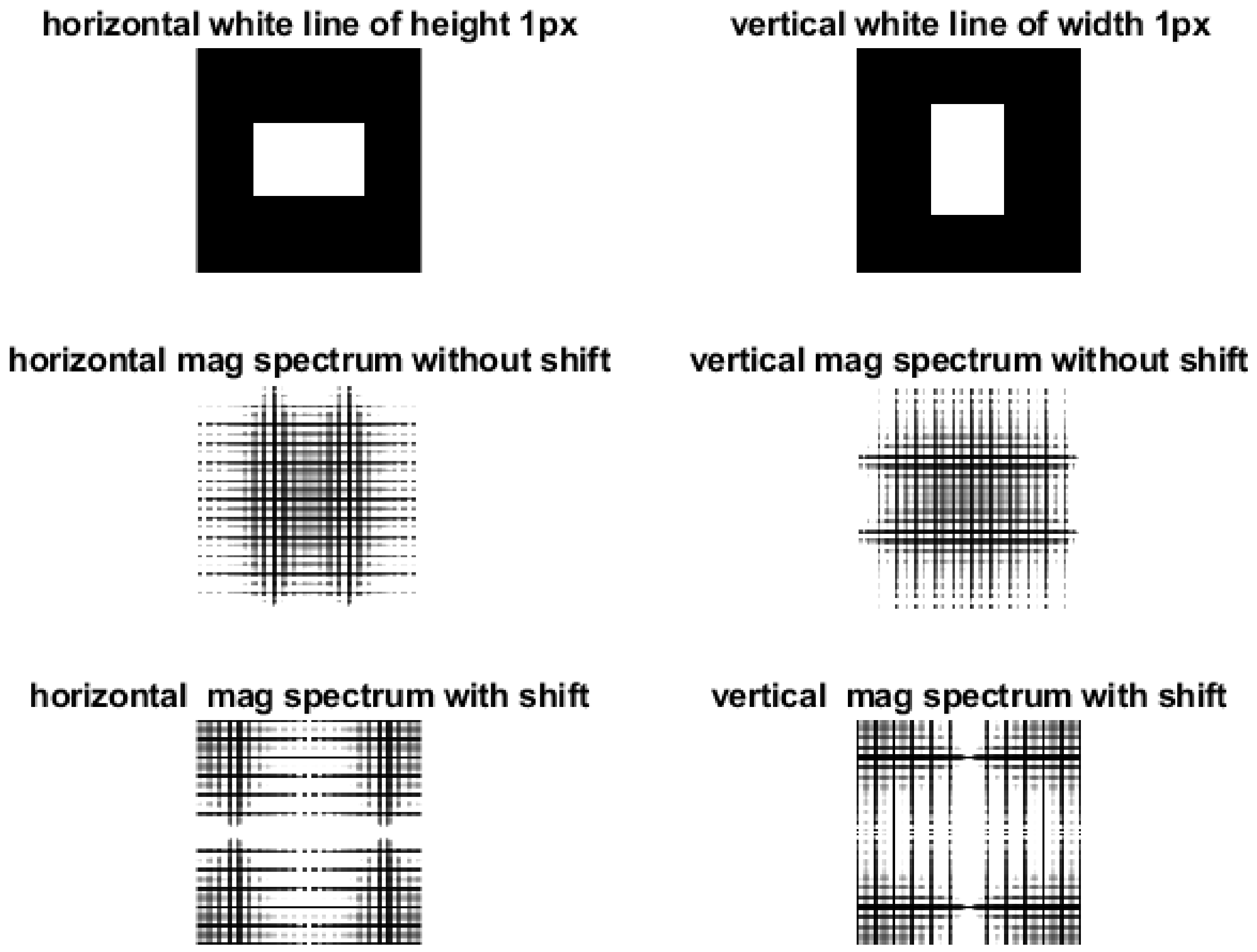

horizontal white line of height 1px
vertical white line of width 1px
horizontal mag spectrum without shift
vertical mag spectrum without shift
horizontal mag spectrum with shift
vertical mag spectrum with shift

Figure 9: use of fft2( ) and fftshift( ) function in MATLAB

Shreenandan Sahu |120BM0806

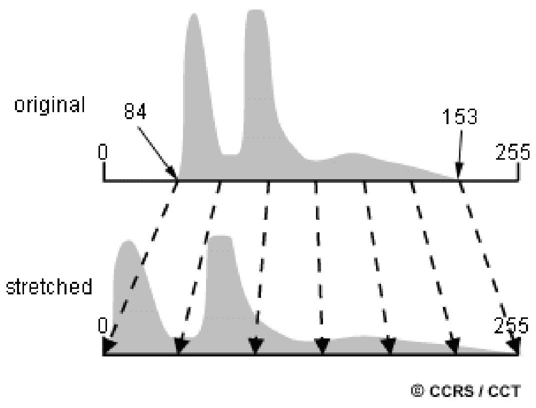
Lab Report 3

Aim

Performing histogram Equalization on images by developing algorithm for it.

Theory

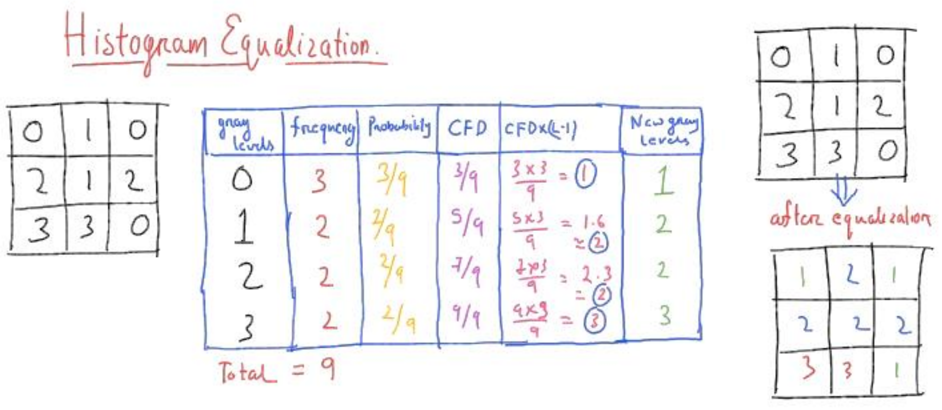
Histogram Equalization



- Histogram Equalization is a computer image processing technique used to improve contrast in images. It accomplishes this by effectively spreading out the most frequent intensity values, i.e. stretching out the intensity range of the image. This method usually increases the global contrast of images when its usable data is represented by close contrast values. This allows for areas of lower local contrast to gain a higher contrast.

Figure 1: graphical depiction of histogram equalization.

- Histogram equalization is a technique used in image processing to enhance the contrast of an image. It works by redistributing the pixel intensities in an image so that they are more evenly distributed across the full range of intensity values.
- The basic idea behind histogram equalization is to compute a histogram of the pixel intensities in the image, and then to use that histogram to create a mapping function that will transform the original image so that it has a more uniform distribution of pixel intensities.
- The mapping function used in histogram equalization is typically a cumulative distribution function (CDF), which represents the cumulative frequency of each intensity value in the histogram. The CDF is then normalized so that its values range from 0 to 1, and this normalized CDF is used to map the pixel intensities in the original image to new intensity values that are more evenly distributed.
- The result of histogram equalization is an image with improved contrast, where the darker and lighter areas are more pronounced, making it easier to distinguish details in the image. However, it is important to note that histogram equalization may also introduce artifacts or noise in the image if not applied properly, and it may not be appropriate for all types of images.



Histogram Equalization.

0	1	0
2	1	2
3	3	0

gray levels	frequency	Probability	CFD	CFD×(L-1)	New gray levels
0	3	3/9	3/9	$\frac{3\times3}{9}=\boxed{1}$	1
1	2	2/9	5/9	$\frac{5\times3}{9}=1.6\simeq\boxed{2}$	2
2	2	2/9	7/9	$\frac{7\times3}{9}=2.3=\boxed{2}$	2
3	2	2/9	9/9	$\frac{9\times3}{9}=\boxed{3}$	3

Total = 9

0	1	0
2	1	2
3	3	0

$\Downarrow$ after equalization

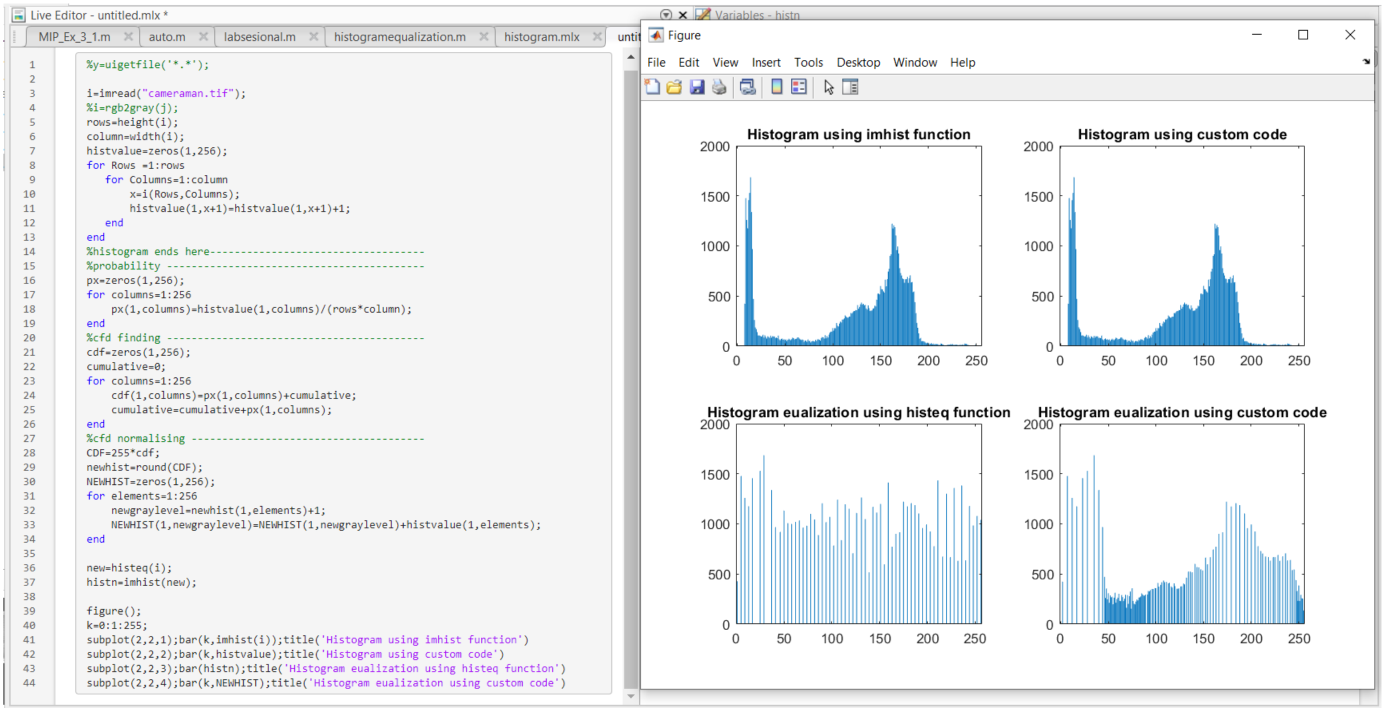
1	2	1
2	2	2
3	3	1

Figure 2: process of finding histogram equalization.

CODE

```
y=uigetfile('*.*');
j=imread(y);
i=rgb2gray(j);
rows=height(i);
column=width(i);
histvalue=zeros(1,256);
for Rows =1:rows
   for Columns=1:column
       x=i(Rows,Columns);
       histvalue(1,x+1)=histvalue(1,x+1)+1;
   end
end
%histogram ends here-----------------------------------
%probability ------------------------------------------
px=zeros(1,256);
for columns=1:256
    px(1,columns)=histvalue(1,columns)/(rows*column);
end
%cfd finding ------------------------------------------
cdf=zeros(1,256);
cumulative=0;
for columns=1:256
    cdf(1,columns)=px(1,columns)+cumulative;
    cumulative=cumulative+px(1,columns);
end
%cfd normalising --------------------------------------
CDF=255*cdf;
newhist=round(CDF);
NEWHIST=zeros(1,256);
for elements=1:256
    newgraylevel=newhist(1,elements)+1;

NEWHIST(1,newgraylevel)=NEWHIST(1,newgraylevel)+histvalue(1,elements);
end


new=histeq(i);
```

```
histn=imhist(new);


figure();
k=0:1:255;
subplot(2,2,1);bar(k,imhist(i));title('Histogram using imhist function')
subplot(2,2,2);bar(k,histvalue);title('Histogram using custom code')
subplot(2,2,3);bar(histn);title('Histogram eualization using histeq function')
subplot(2,2,4);bar(k,NEWHIST);title('Histogram eualization using custom code')
```


Live Editor - untitled.mlx *
MIP_Ex_3_1.m
auto.m
labsesional.m
histogramequalization.m
histogram.mlx
Variables - histn
Figure
File
Edit
View
Insert
Tools
Desktop
Window
Help
Histogram using imhist function
Histogram using custom code
Histogram eualization using histeq function
Histogram eualization using custom code

```
%y=uigetfile('*.*');

i=imread("cameraman.tif");
%i=rgb2gray(j);
rows=height(i);
column=width(i);
histvalue=zeros(1,256);
for Rows =1:rows
   for Columns=1:column
       x=i(Rows,Columns);
       histvalue(1,x+1)=histvalue(1,x+1)+1;
   end
end
%histogram ends here-----------------------------------
%probability ------------------------------------------
px=zeros(1,256);
for columns=1:256
    px(1,columns)=histvalue(1,columns)/(rows*column);
end
%cfd finding ------------------------------------------
cdf=zeros(1,256);
cumulative=0;
for columns=1:256
    cdf(1,columns)=px(1,columns)+cumulative;
    cumulative=cumulative+px(1,columns);
end
%cfd normalising --------------------------------------
CDF=255*cdf;
newhist=round(CDF);
NEWHIST=zeros(1,256);
for elements=1:256
    newgraylevel=newhist(1,elements)+1;
    NEWHIST(1,newgraylevel)=NEWHIST(1,newgraylevel)+histvalue(1,elements);
end

new=histeq(i);
histn=imhist(new);

figure();
k=0:1:255;
subplot(2,2,1);bar(k,imhist(i));title('Histogram using imhist function')
subplot(2,2,2);bar(k,histvalue);title('Histogram using custom code')
subplot(2,2,3);bar(histn);title('Histogram eualization using histeq function')
subplot(2,2,4);bar(k,NEWHIST);title('Histogram eualization using custom code')
```

Figure 3: plots showing histograms before and after equalization.

Shreenandan Sahu |120BM0806

Lab Report 4

Aim

To apply spatial domain filters on the images and enhance the images in MATLAB.

Theory

Spatial filters are one of the most commonly used techniques in image processing. They are used to enhance or suppress certain features in an image by altering the pixel values within a given neighbourhood of each pixel in the image.

A spatial filter works by replacing the value of each pixel with a weighted average of the values of its neighbouring pixels. The weights assigned to each neighbouring pixel depend on the type of filter being used and the distance of the pixel from the centre pixel.

There are two types of spatial filters: linear and non-linear. Linear filters are those that use a fixed kernel or matrix to calculate the weighted average of the neighbouring pixels. Examples of linear filters include the mean filter, median filter, and Gaussian filter. These filters are widely used for noise reduction, image smoothing, and edge detection.

Non-linear filters, on the other hand, use a varying kernel to calculate the weighted average of neighbouring pixels. Examples of non-linear filters include the maximum filter, minimum filter, and bilateral filter. These filters are used for tasks such as image sharpening, feature extraction, and image segmentation.

In summary, spatial filters are a powerful tool in image processing for enhancing, filtering, and segmenting images. The choice of filter depends on the specific task at hand and the characteristics of the image being processed.

Mean filter: This filter is used to reduce noise in an image by replacing each pixel value with the average value of the neighbouring pixels within a given kernel. The mean filter is a linear filter and is commonly used for smoothing and blurring an image.

Median filter: This filter is used to remove salt-and-pepper noise in an image by replacing each pixel value with the median value of the neighbouring pixels within a given kernel. The median filter is a non-linear filter and is effective in preserving edges in an image.

Gaussian filter: This filter is used to blur an image by replacing each pixel value with the weighted average of the neighbouring pixels within a Gaussian kernel. The Gaussian filter is a linear filter and is commonly used for smoothing an image while preserving its edges.

Laplacian filter: This filter is used for edge detection in an image by highlighting areas of high spatial frequency. The Laplacian filter is a non-linear filter and is commonly used for sharpening an image.

MEAN AND WEIGHTED-MEAN FILTER

Mean Filter:

The mean filter is a popular type of spatial filter used in image processing to remove noise and smooth out an image. It works by taking the average of the pixel values within a given kernel or window and replacing the central pixel with this average value. The size of the kernel determines the degree of smoothing that is applied to the image. The mean filter is a type of linear filter because the output value is a linear combination of the input values. The filter kernel is typically a square or rectangular matrix of equal size to the kernel window. The mean filter is easy to implement and computationally efficient, making it a popular choice for basic image processing tasks.

Weighted Mean Filter:

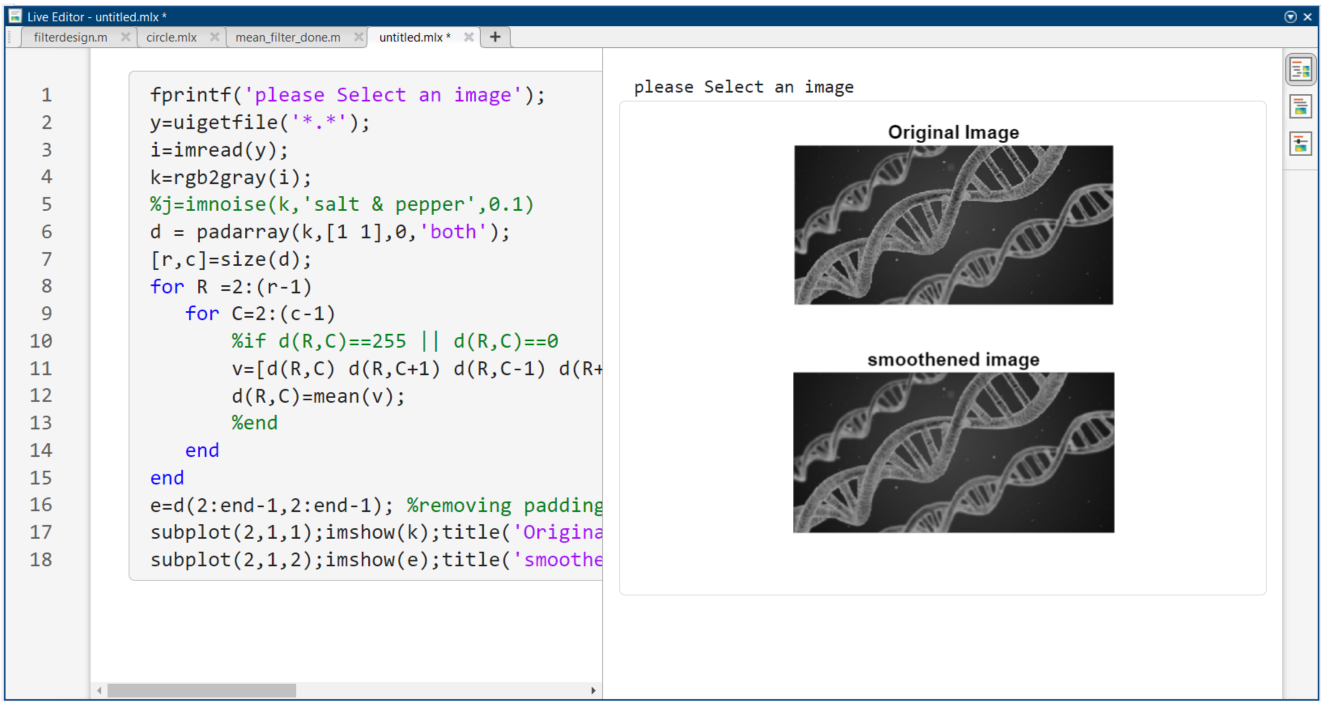
The weighted mean filter is a variation of the mean filter that assigns different weights to the pixel values within the kernel window. The weights are based on a predefined kernel function that assigns higher weights to the pixels closer to the centre of the kernel and lower weights to the pixels further away from the centre. The weighted mean filter is also a type of linear filter because it calculates a weighted average of the input pixel values. However, the weights are different for each pixel and are based on the kernel function.

The advantage of using a weighted mean filter over a regular mean filter is that it can better preserve edges and other fine details in the image, while still reducing noise and smoothing out the overall image. The choice of kernel function and its parameters can have a significant impact on the output of the filter, and the selection of the appropriate kernel function depends on the specific characteristics of the image being processed.

CODE

```
% MEAN FILTER
fprintf('please Select an image');
y=uigetfile('*.*');
i=imread(y);
k=rgb2gray(i);
%j=imnoise(k,'salt & pepper',0.1)
d = padarray(k,[1 1],0,'both');
[r,c]=size(d);
for R =2:(r-1)
   for C=2:(c-1)
       %if d(R,C)==255 || d(R,C)==0
       v=[d(R,C) d(R,C+1) d(R,C-1) d(R+1,C) d(R+1,C-1) d(R+1,C+1) d(R-1,C) d(R-1,C-1) d(R-1,C+1)];
       d(R,C)=mean(v);
       %end
```

```
    end
end
e=d(2:end-1,2:end-1); %removing padding
subplot(1,2,1);imshow(k);title('Original Image');
subplot(1,2,2);imshow(e);title('smoothened image');
```



```
fprintf('please Select an image');
y=uigetfile('*.*');
i=imread(y);
k=rgb2gray(i);
%j=imnoise(k,'salt & pepper',0.1)
d = padarray(k,[1 1],0,'both');
[r,c]=size(d);
for R =2:(r-1)
   for C=2:(c-1)
       %if d(R,C)==255 || d(R,C)==0
       v=[d(R,C) d(R,C+1) d(R,C-1) d(R+
       d(R,C)=mean(v);
       %end
   end
end
e=d(2:end-1,2:end-1); %removing padding
subplot(2,1,1);imshow(k);title('Origina
subplot(2,1,2);imshow(e);title('smoothe
```

```
please Select an image
```

Figure 1: use of mean( ) and imnoise( ) function in MATLAB

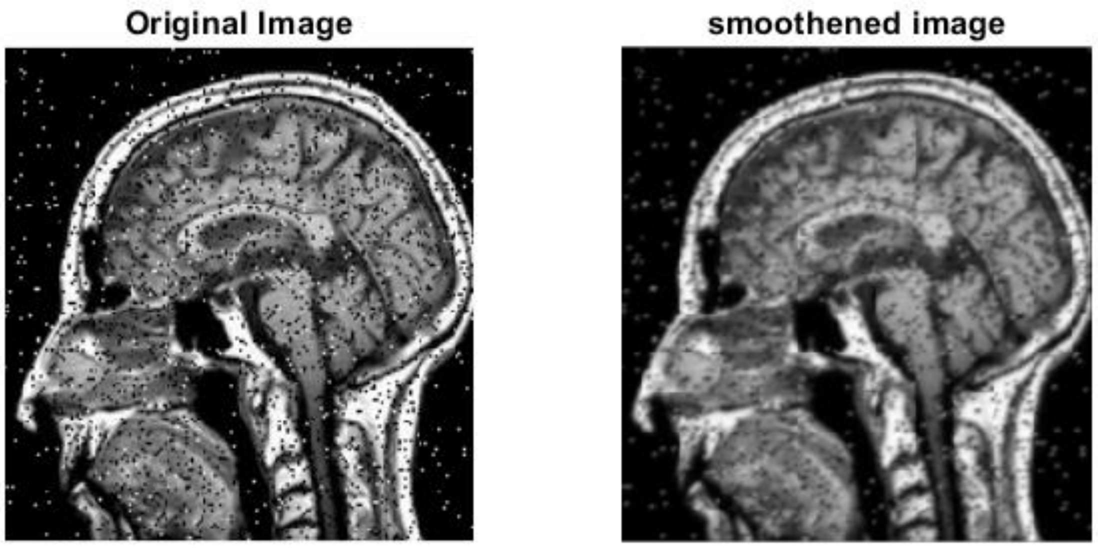
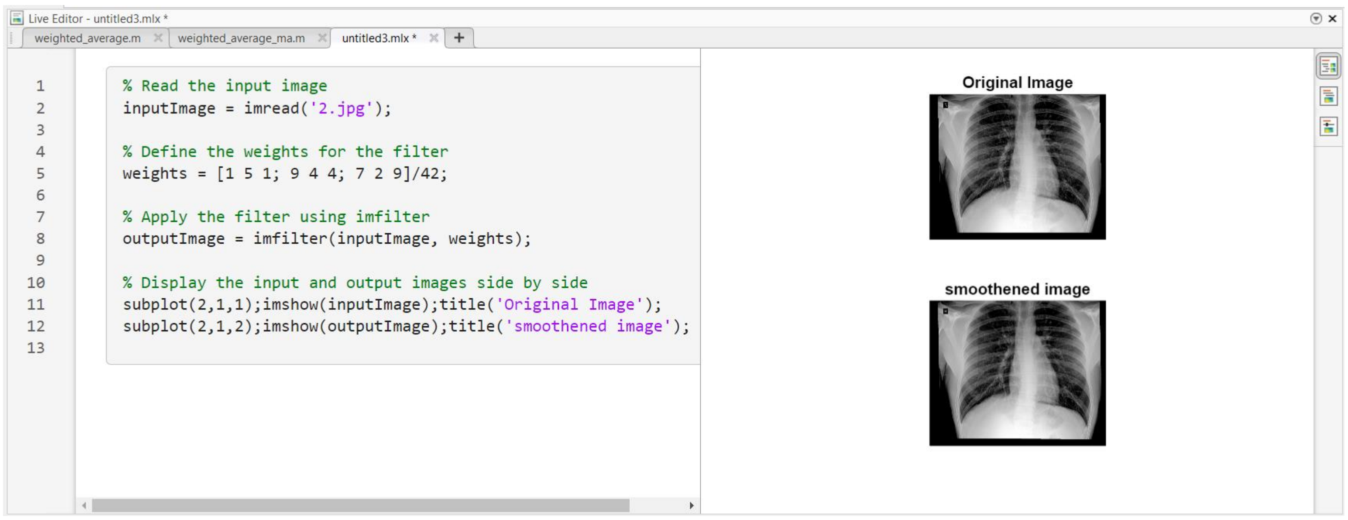


Figure 2: image smoothening using mean filter in MATLAB

CODE

```
% WEIGHTED MEAN FILTER
% Read the input image
inputImage = imread('s.jpg');

% Define the weights for the filter
weights = [1 5 1; 9 4 4; 7 2 9]/42;

% Apply the filter using imfilter
outputImage = imfilter(inputImage, weights);

% Display the input and output images side by side
subplot(1,2,1);imshow(inputImage);title('Original Image');
subplot(1,2,2);imshow(outputImage);title('smoothened image');
```



```
% Read the input image
inputImage = imread('2.jpg');

% Define the weights for the filter
weights = [1 5 1; 9 4 4; 7 2 9]/42;

% Apply the filter using imfilter
outputImage = imfilter(inputImage, weights);

% Display the input and output images side by side
subplot(2,1,1);imshow(inputImage);title('Original Image');
subplot(2,1,2);imshow(outputImage);title('smoothened image');
```

Figure 3: use of imfilter( ) function in MATLAB

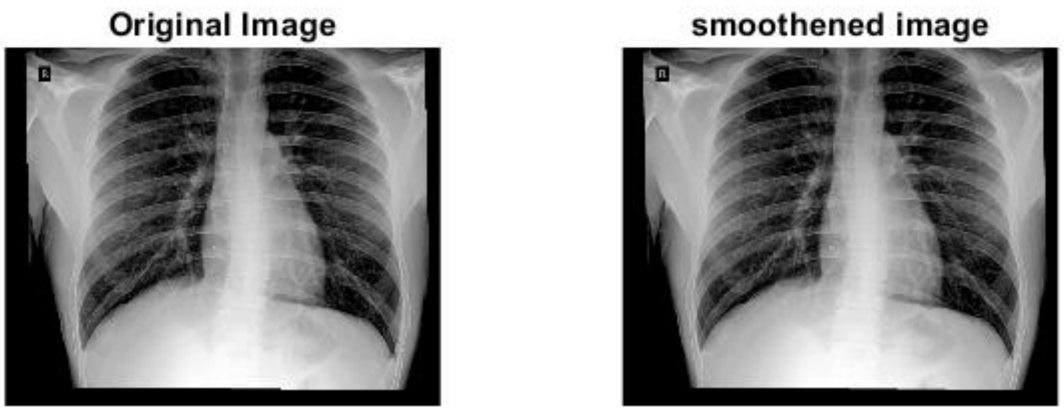


Figure 4: image smoothening using weighted mean filter in MATLAB

POWER LAW TRANFORMATION / GAMMA CORRECTION

Gamma Filter:

The Power law transformation, also known as gamma correction, is a type of image processing technique that alters the intensity values of pixels in an image using a power function. The power function can either increase or decrease the contrast of an image. In this technique, the image's pixel values are raised to a certain exponent (power) that changes the distribution of pixel intensities. The power function can be expressed as:

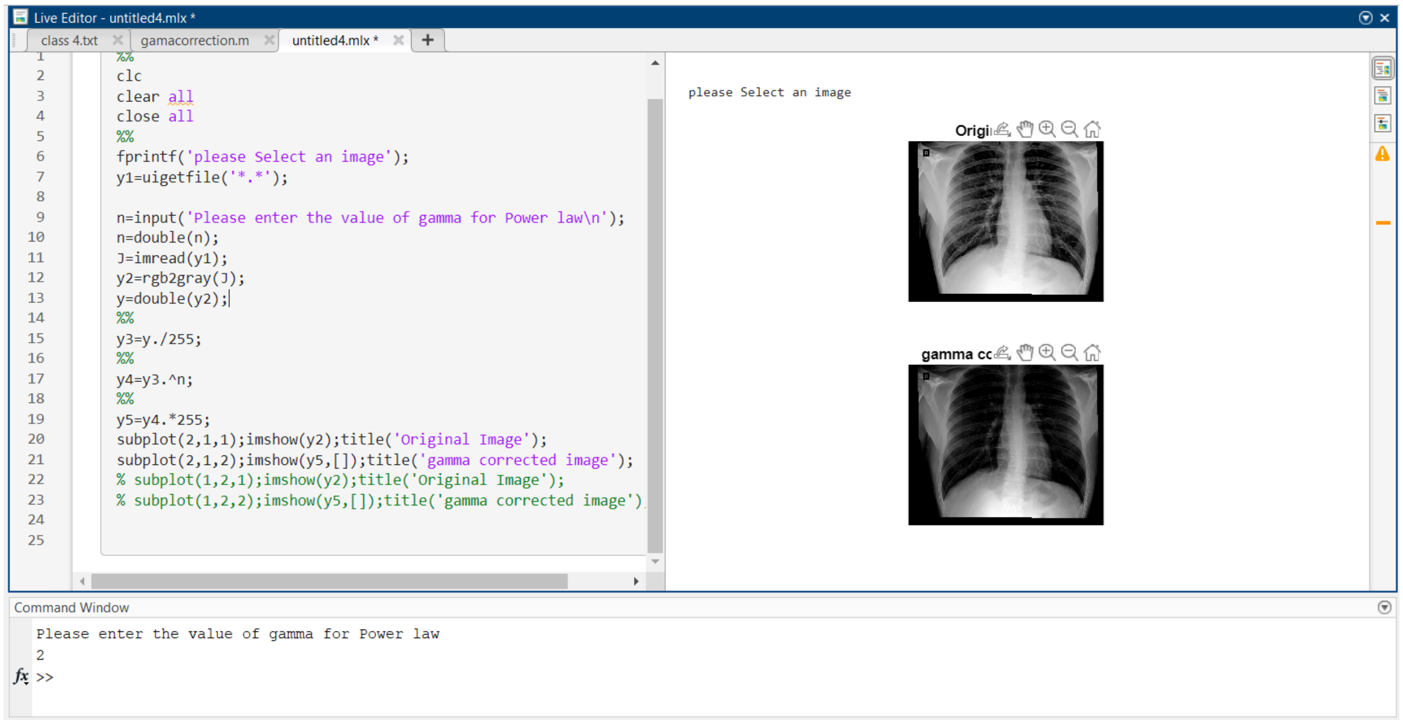
$$s = c * r^\gamma$$

Where s is the output pixel value, r is the input pixel value, γ is the gamma value, and c is a scaling constant. If the gamma value is less than 1, the resulting image will have increased contrast, making the darker pixels appear lighter and the lighter pixels appear darker. If the gamma value is greater than 1, the resulting image will have decreased contrast, making the darker pixels appear even darker and the lighter pixels even lighter.

Power law transformation is commonly used in image processing to correct for non-linearities in images caused by the imaging system or to enhance image contrast for better visualization.

CODE

```
% WEIGHTED MEAN FILTER

fprintf('please Select an image');
y1=uigetfile('*.*');

n=input('Please enter the value of gamma for Power law\n');
n=double(n);
J=imread(y1);
y2=rgb2gray(J);
y=double(y2);
%%
y3=y./255;
%%
y4=y3.^n;
%%
y5=y4.*255;
subplot(1,2,1);imshow(y2);title('Original Image');
subplot(1,2,2);imshow(y5,[]);title('gamma corrected image');
```



```
%%
clc
clear all
close all
%%
fprintf('please Select an image');
y1=uigetfile('*.*');

n=input('Please enter the value of gamma for Power law\n');
n=double(n);
J=imread(y1);
y2=rgb2gray(J);
y=double(y2);
%%
y3=y./255;
%%
y4=y3.^n;
%%
y5=y4.*255;
subplot(2,1,1);imshow(y2);title('Original Image');
subplot(2,1,2);imshow(y5,[]);title('gamma corrected image');
% subplot(1,2,1);imshow(y2);title('Original Image');
% subplot(1,2,2);imshow(y5,[]);title('gamma corrected image')
```

```
please Select an image
```

```
Please enter the value of gamma for Power law
2
>>
```

Figure 5: use of gamma filter in MATLAB

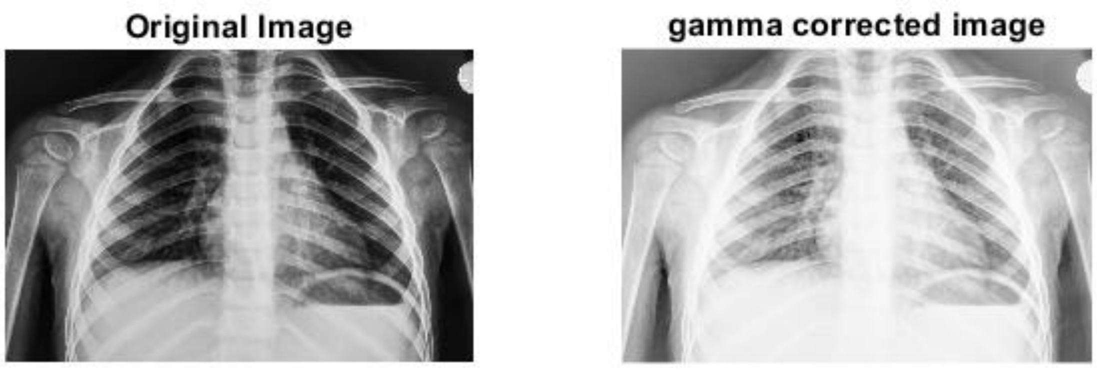


Figure 6: image enhancement using gamma filter in MATLAB

IMAGE NEGATIVE

Negative Filter:

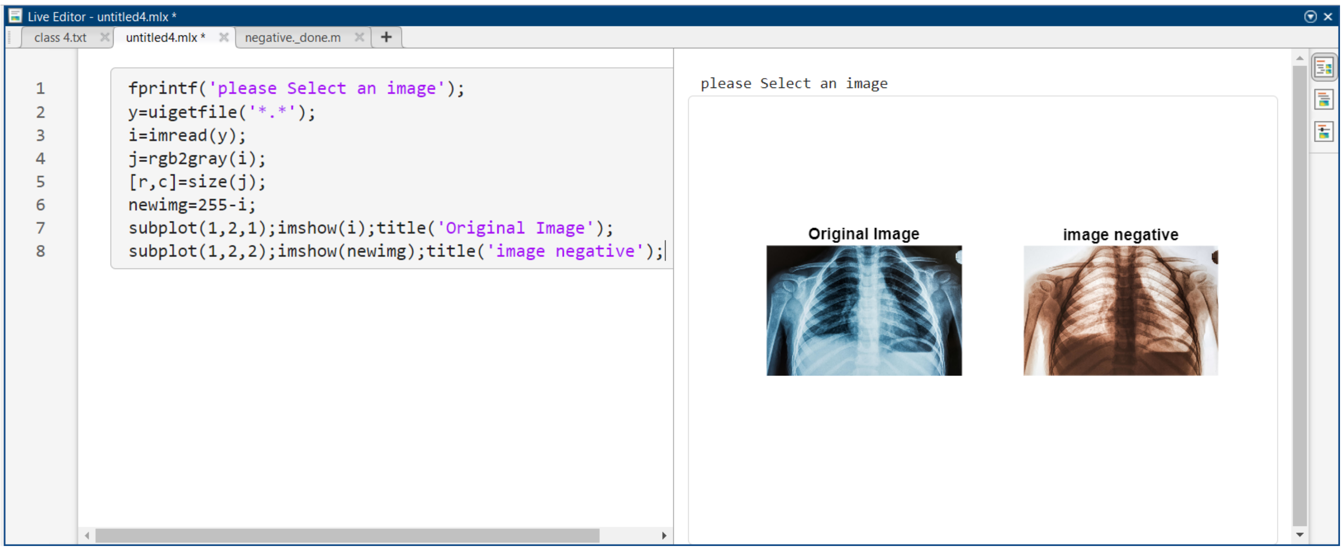
An image negative is an inverted version of an original image. In a negative image, the dark areas of the original image appear light, and the light areas appear dark. This effect is achieved by reversing the brightness and colour values of the original image. New intensity values are given as.

$$s = 255-i$$

In medical imaging, negative images can be useful for highlighting certain features in an image. For example, a negative image of a CT or MRI scan can be used to enhance the contrast between different tissues or structures, making it easier for a medical professional to identify abnormalities or areas of interest. Negative images can also be used in radiography to highlight the presence of foreign objects, such as metal or glass, that may be difficult to see in a regular X-ray image. In these cases, the negative image can help to differentiate the foreign object from surrounding tissue or bone. Additionally, negative images can be useful for enhancing the visibility of certain types of medical images, such as angiograms or mammograms. By creating a negative image, it is possible to enhance the contrast between blood vessels or breast tissue and background structures, making it easier to identify potential abnormalities.

CODE

```
% NEGATIVE FILTER
fprintf('please Select an image');
y=uigetfile('*.*');
i=imread(y);
j=rgb2gray(i);
[r,c]=size(j);
newimg=255-i;
subplot(1,2,1);imshow(i);title('Original Image');
subplot(1,2,2);imshow(newimg);title('image negative');
```



```
fprintf('please Select an image');
y=uigetfile('*.*');
i=imread(y);
j=rgb2gray(i);
[r,c]=size(j);
newimg=255-i;
subplot(1,2,1);imshow(i);title('Original Image');
subplot(1,2,2);imshow(newimg);title('image negative');
```

```
please Select an image
```

Figure 7: use of image negative filter in MATLAB

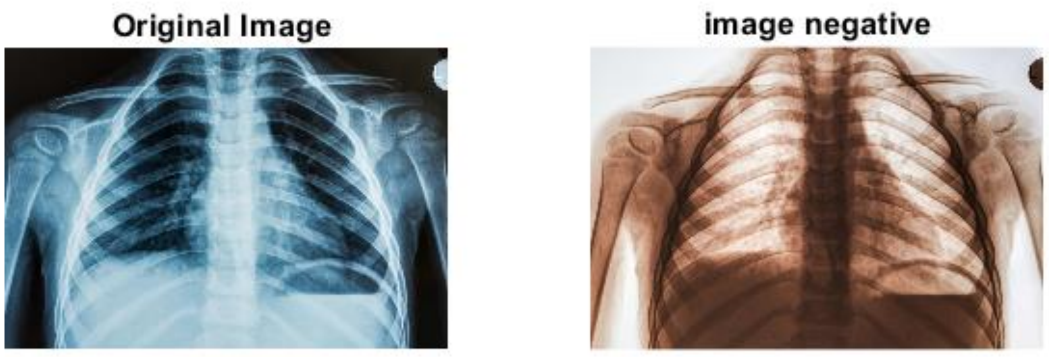


Figure 8: image enhancement using gamma filter in MATLAB

BIT PLANE SLICING

Bit Plane Slicing:

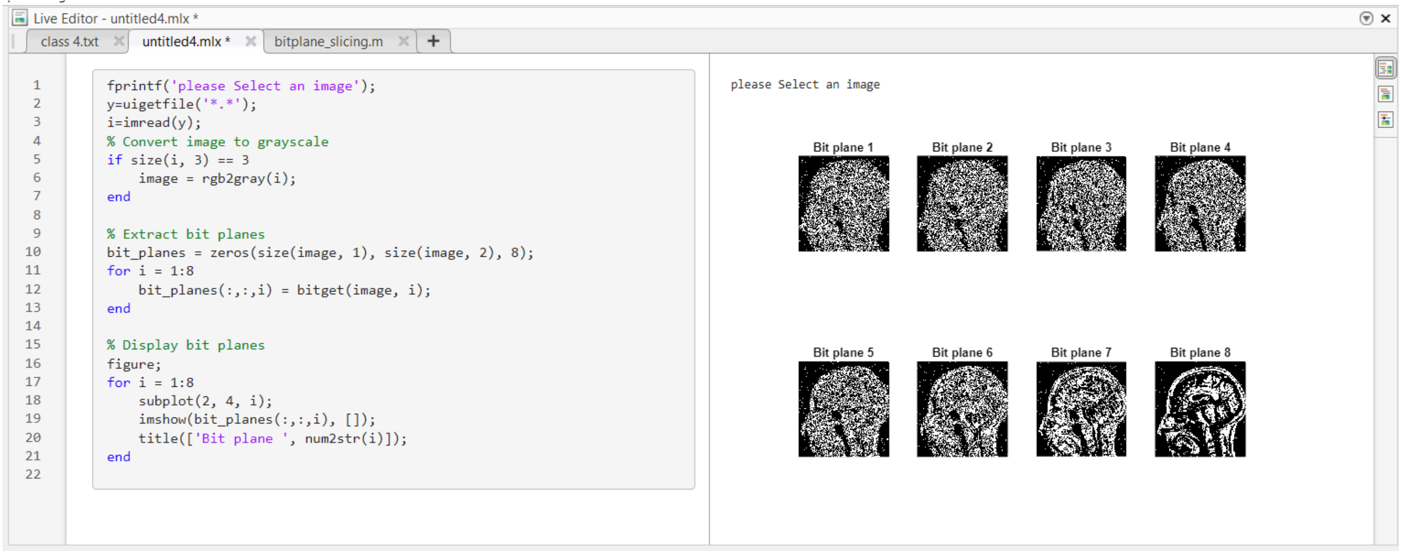
Bit plane slicing is a digital image processing technique used to separate the binary components of an image by extracting each bit plane. In digital imaging, each pixel in an image is represented by a binary code that can be divided into multiple bits. By using bit plane slicing, we can isolate each bit of the binary code, which results in a series of binary images. These binary images represent the contribution of each bit to the original image. For example, in an 8-bit grayscale image, the pixel values range from 0 to 255. We can slice the image into eight-bit planes, where the first bit plane represents the least significant bit (LSB) and the eighth bit plane represents the most significant bit (MSB). The MSB is the most important bit as it has the highest weight and contributes the most to the final pixel value. Each bit plane can be visualized as a binary image, where black pixels represent the bits that have a value of 0, and white pixels represent the bits that have a value of 1. By displaying each bit plane individually or in combination with other bit planes, we can enhance different features of the original image. For instance, the first bit plane would highlight the noise in the image, while the higher bit planes can help us to focus on the edges and details of the image.

Bit plane slicing is used in various applications such as image compression, feature extraction, and image enhancement.

CODE

```
% BIT PLANE SLICING

fprintf('please Select an image');
y=uigetfile('*.*');
i=imread(y);
% Convert image to grayscale
if size(i, 3) == 3
    image = rgb2gray(i);
end



% Extract bit planes
bit_planes = zeros(size(image, 1), size(image, 2), 8);
for i = 1:8
    bit_planes(:,:,i) = bitget(image, i);
end

% Display bit planes
```

```
figure;
for i = 1:8
    subplot(2, 4, i);
    imshow(bit_planes(:,:,i), []);
    title(['Bit plane ', num2str(i)]);
```



```
fprintf('please Select an image');
y=uigetfile('*.*');
i=imread(y);
% Convert image to grayscale
if size(i, 3) == 3
    image = rgb2gray(i);
end

% Extract bit planes
bit_planes = zeros(size(image, 1), size(image, 2), 8);
for i = 1:8
    bit_planes(:,:,i) = bitget(image, i);
end

% Display bit planes
figure;
for i = 1:8
    subplot(2, 4, i);
    imshow(bit_planes(:,:,i), []);
    title(['Bit plane ', num2str(i)]);
end
```

```
please Select an image
```

Figure 9: use of image bit slicing in MATLAB

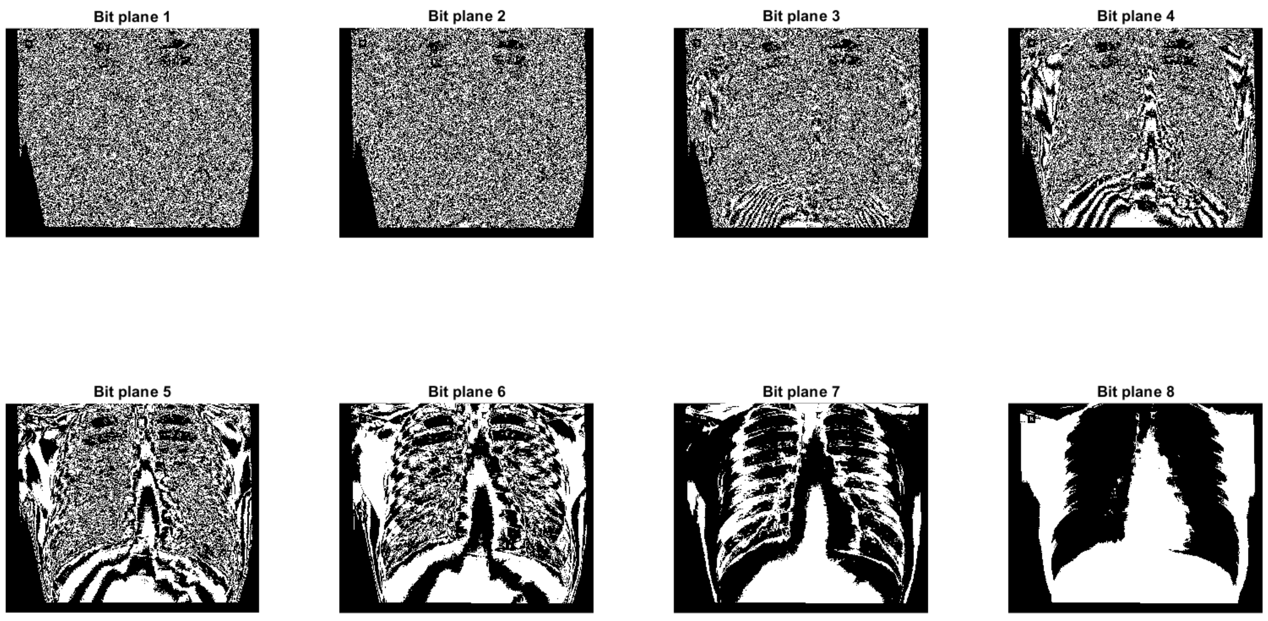


Figure 10: 8 bit sliced image in MATLAB

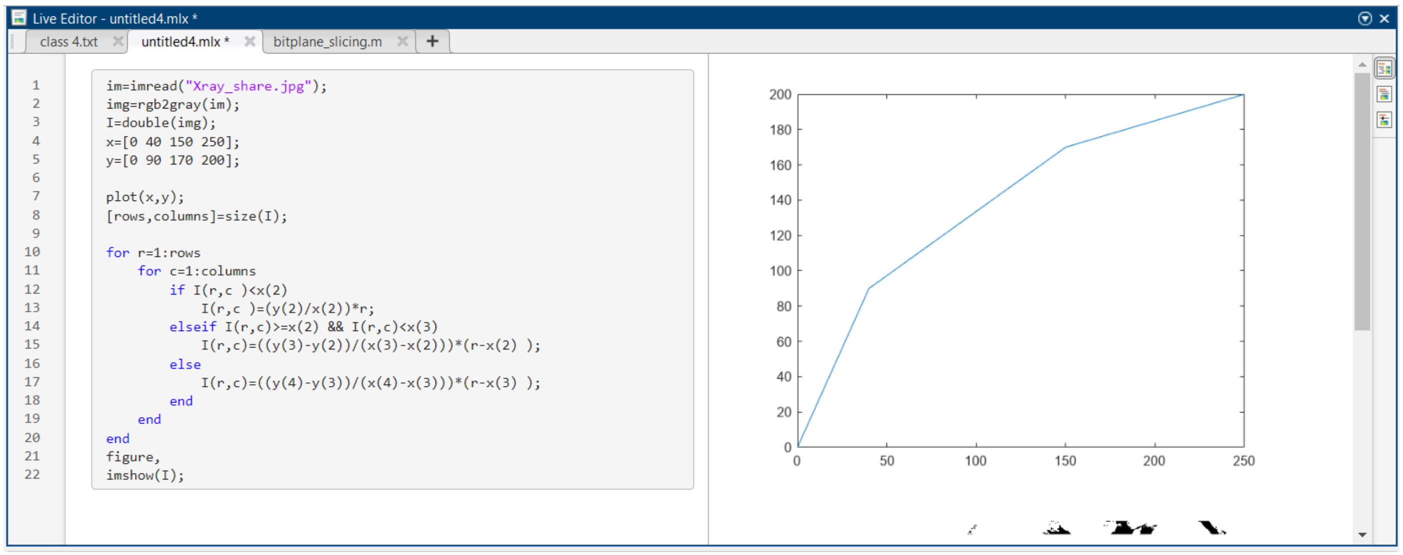
CONTRAST STRETCHING

Contrast stretching:

Contrast stretching is a technique used in image processing to enhance the contrast of an image. The technique works by expanding the dynamic range of the image so that the brightest and darkest pixels are spread over the entire range of pixel values, thus increasing the contrast. The process involves finding the minimum and maximum pixel values in the image, and then mapping the values in between to a new range. This new range is typically the full range of values that the image can represent, such as 0 to 255 for an 8-bit grayscale image. There are several methods for implementing contrast stretching, including linear stretching, piecewise linear stretching, and histogram equalization. Linear stretching involves simply scaling the pixel values between the minimum and maximum values to fill the entire range. Piecewise linear stretching involves dividing the range into several segments and applying a linear function to each segment. Histogram equalization involves mapping the histogram of the image to a uniform distribution. Contrast stretching is a useful technique for enhancing the visibility of details in an image, particularly when the contrast is low. However, it can also lead to artifacts and noise amplification if not applied carefully. Therefore, it is important to choose an appropriate method and parameter settings for each specific image.

CODE

```
% CONTRAST STRETCHING

im=imread("Xray_share.jpg");
img=rgb2gray(im);
I=double(img);
x=[0 40 150 250];
y=[0 90 170 200];

plot(x,y);
[rows,columns]=size(I);

for r=1:rows
    for c=1:columns
        if I(r,c )<x(2)
            I(r,c )=(y(2)/x(2))*r;
        elseif I(r,c)>=x(2) && I(r,c)<x(3)
            I(r,c)=((y(3)-y(2))/(x(3)-x(2)))*(r-x(2) );
        else
            I(r,c)=((y(4)-y(3))/(x(4)-x(3)))*(r-x(3) );
        end
    end
end
figure,
imshow(I);
```



```
im=imread("Xray_share.jpg");
img=rgb2gray(im);
I=double(img);
x=[0 40 150 250];
y=[0 90 170 200];

plot(x,y);
[rows,columns]=size(I);

for r=1:rows
    for c=1:columns
        if I(r,c )<x(2)
            I(r,c )=(y(2)/x(2))*r;
        elseif I(r,c)>=x(2) && I(r,c)<x(3)
            I(r,c)=((y(3)-y(2))/(x(3)-x(2)))*(r-x(2) );
        else
            I(r,c)=((y(4)-y(3))/(x(4)-x(3)))*(r-x(3) );
        end
    end
end
figure,
imshow(I);
```

Figure 11: use of contrast stretching in MATLAB

MIN AND MAX FILTER

Min and Max filter:

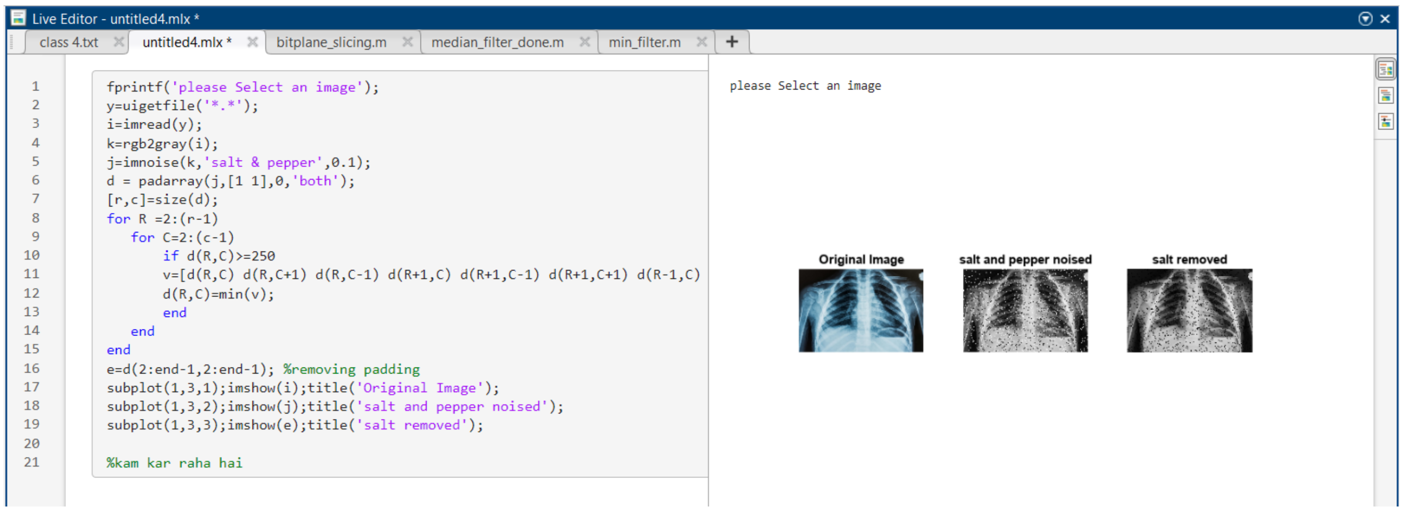
In image processing, a min filter (also known as erosion) is an operation that replaces each pixel in an image with the minimum value of its neighbouring pixels, using a kernel or structuring element. The result is an image that has had small features or details removed, effectively shrinking the image. On the other hand, a max filter (also known as dilation) replaces each pixel in an image with the maximum value of its neighbouring pixels. This operation has the opposite effect of the min filter, and can be used to enhance or highlight the edges and boundaries in an image.

Both min and max filters are used in image processing for various purposes, such as noise reduction, feature extraction, and object detection. They can be applied to grayscale as well as colour images. The size and shape of the kernel or structuring element used for the filtering process determines the degree of smoothing or sharpening in the resulting image..

CODE

```
% MIN FILTER
fprintf('please Select an image');
y=uigetfile('*.*');
i=imread(y);
k=rgb2gray(i);
j=imnoise(k,'salt & pepper',0.1);
```

```
d = padarray(j,[1 1],0,'both');
[r,c]=size(d);
for R =2:(r-1)
   for C=2:(c-1)
       if d(R,C)>=250
       v=[d(R,C) d(R,C+1) d(R,C-1) d(R+1,C) d(R+1,C-1) d(R+1,C+1) d(R-1,C) d(R-1,C-1) d(R-1,C+1)];
       d(R,C)=min(v);
       end
   end
end
e=d(2:end-1,2:end-1); %removing padding
subplot(1,3,1);imshow(i);title('Original Image');
subplot(1,3,2);imshow(j);title('salt and pepper noised');
subplot(1,3,3);imshow(e);title('salt removed');
```


Live Editor - untitled4.mlx *

class 4.txt | untitled4.mlx * | bitplane_slicing.m | median_filter_done.m | min_filter.m

```
fprintf('please Select an image');
y=uigetfile('*.*');
i=imread(y);
k=rgb2gray(i);
j=imnoise(k,'salt & pepper',0.1);
d = padarray(j,[1 1],0,'both');
[r,c]=size(d);
for R =2:(r-1)
    for C=2:(c-1)
        if d(R,C)>=250
        v=[d(R,C) d(R,C+1) d(R,C-1) d(R+1,C) d(R+1,C-1) d(R+1,C+1) d(R-1,C)
        d(R,C)=min(v);
        end
    end
end
e=d(2:end-1,2:end-1); %removing padding
subplot(1,3,1);imshow(i);title('Original Image');
subplot(1,3,2);imshow(j);title('salt and pepper noised');
subplot(1,3,3);imshow(e);title('salt removed');

%kam kar raha hai
```

```
please Select an image
```

Figure 12: use of min() function for making min filter in MATLAB

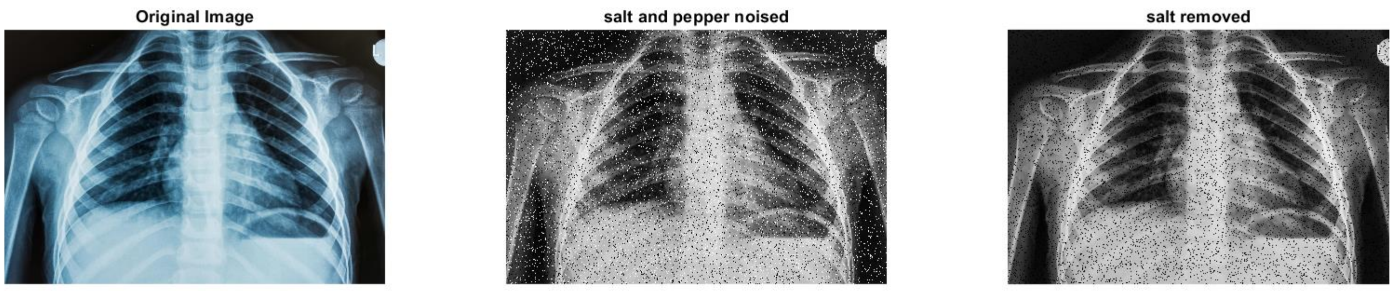
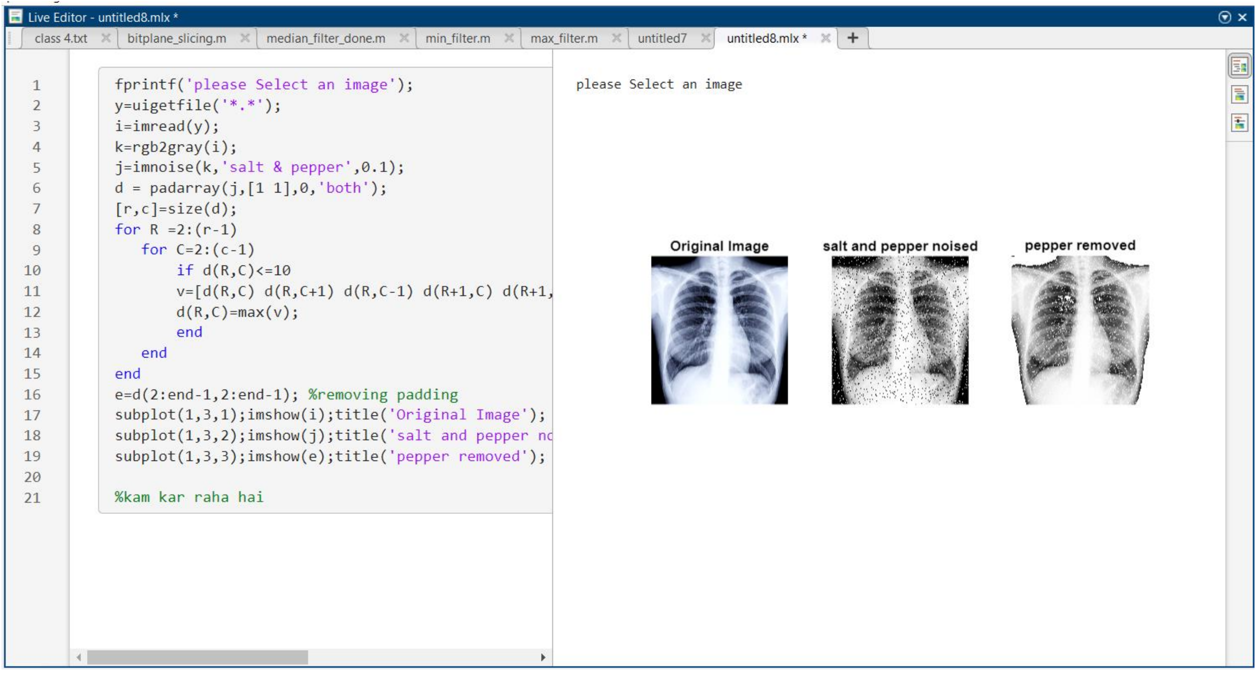


Figure 13: min filter in MATLAB is able to filter only salt noise.

CODE

```
% MAX FILTER
fprintf('please Select an image');
y=uigetfile('*.*');
i=imread(y);
k=rgb2gray(i);
j=imnoise(k,'salt & pepper',0.1);
d = padarray(j,[1 1],0,'both');
[r,c]=size(d);
for R =2:(r-1)
   for C=2:(c-1)
       if d(R,C)<=10
       v=[d(R,C) d(R,C+1) d(R,C-1) d(R+1,C) d(R+1,C-1) d(R+1,C+1) d(R-
1,C) d(R-1,C-1) d(R-1,C+1)];
       d(R,C)=max(v);
       end
   end
end
e=d(2:end-1,2:end-1); %removing padding
subplot(1,3,1);imshow(i);title('Original Image');
subplot(1,3,2);imshow(j);title('salt and pepper noised');
subplot(1,3,3);imshow(e);title('pepper removed');
```



```
fprintf('please Select an image');
y=uigetfile('*.*');
i=imread(y);
k=rgb2gray(i);
j=imnoise(k,'salt & pepper',0.1);
d = padarray(j,[1 1],0,'both');
[r,c]=size(d);
for R =2:(r-1)
   for C=2:(c-1)
       if d(R,C)<=10
       v=[d(R,C) d(R,C+1) d(R,C-1) d(R+1,C) d(R+1,
       d(R,C)=max(v);
       end
   end
end
e=d(2:end-1,2:end-1); %removing padding
subplot(1,3,1);imshow(i);title('Original Image');
subplot(1,3,2);imshow(j);title('salt and pepper nc
subplot(1,3,3);imshow(e);title('pepper removed');

%kam kar raha hai
```

```
please Select an image
```

Figure 14: use of max() function for making max filter in MATLAB

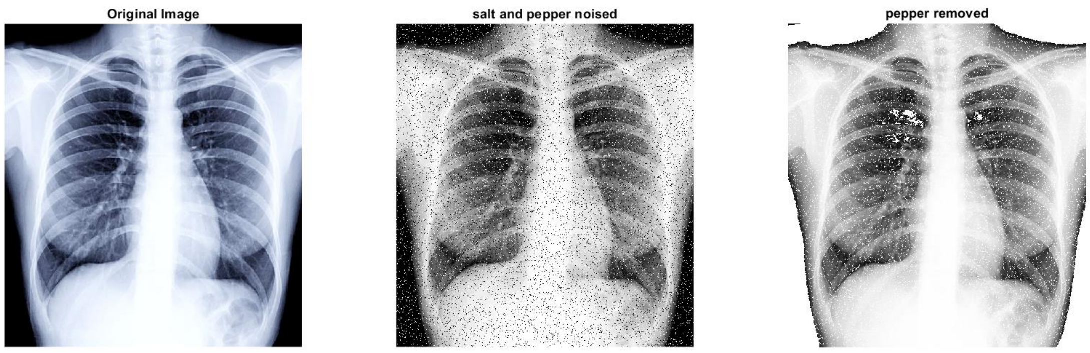


Figure 15: max filter in MATLAB is able to filter only pepper noise.

MEDIAN FILTER

Min and Max filter:

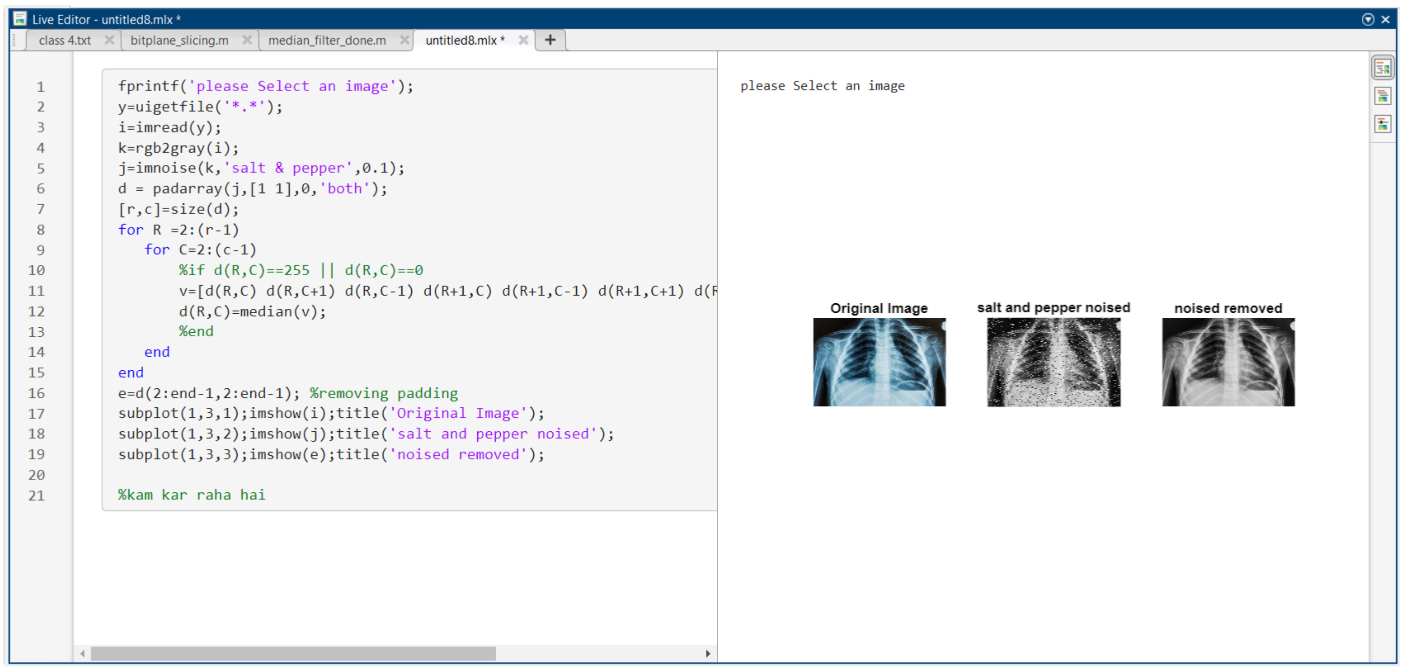
A median filter is a digital signal processing technique used to remove noise from a signal or an image. It works by replacing each pixel's value with the median value of its neighbouring pixels. The process of median filtering involves sliding a window (typically a square or rectangular shape) over the input signal or image. For each pixel within the window, the median value of the pixel's neighbourhood is calculated and used as the new value for that pixel. The size of the window determines the extent of smoothing applied to the signal or image.

Median filtering is commonly used in image processing to remove salt and pepper noise, which appears as random black and white pixels in an image. The median filter can effectively remove this type of noise without blurring or distorting the image's edges and details, making it a popular choice for image denoising.

CODE

```
% MEDIAN FILTER
fprintf('please Select an image');
y=uigetfile('*.*');
i=imread(y);
k=rgb2gray(i);
j=imnoise(k,'salt & pepper',0.1);
d = padarray(j,[1 1],0,'both');
[r,c]=size(d);
for R =2:(r-1)
   for C=2:(c-1)
       %if d(R,C)==255 || d(R,C)==0
       v=[d(R,C) d(R,C+1) d(R,C-1) d(R+1,C) d(R+1,C-1) d(R+1,C+1) d(R-1,C) d(R-1,C-1) d(R-1,C+1)];
       d(R,C)=median(v);
```

```
        %end
    end
end
e=d(2:end-1,2:end-1); %removing padding
subplot(1,3,1);imshow(i);title('Original Image');
subplot(1,3,2);imshow(j);title('salt and pepper noised');
subplot(1,3,3);imshow(e);title('noised removed');
```



```
fprintf('please Select an image');
y=uigetfile('*.*');
i=imread(y);
k=rgb2gray(i);
j=imnoise(k,'salt & pepper',0.1);
d = padarray(j,[1 1],0,'both');
[r,c]=size(d);
for R =2:(r-1)
   for C=2:(c-1)
       %if d(R,C)==255 || d(R,C)==0
       v=[d(R,C) d(R,C+1) d(R,C-1) d(R+1,C) d(R+1,C-1) d(R+1,C+1) d(F
       d(R,C)=median(v);
       %end
   end
end
e=d(2:end-1,2:end-1); %removing padding
subplot(1,3,1);imshow(i);title('Original Image');
subplot(1,3,2);imshow(j);title('salt and pepper noised');
subplot(1,3,3);imshow(e);title('noised removed');

%kam kar raha hai
```

```
please Select an image
```

Figure 16: use of median() function for making median filter in MATLAB

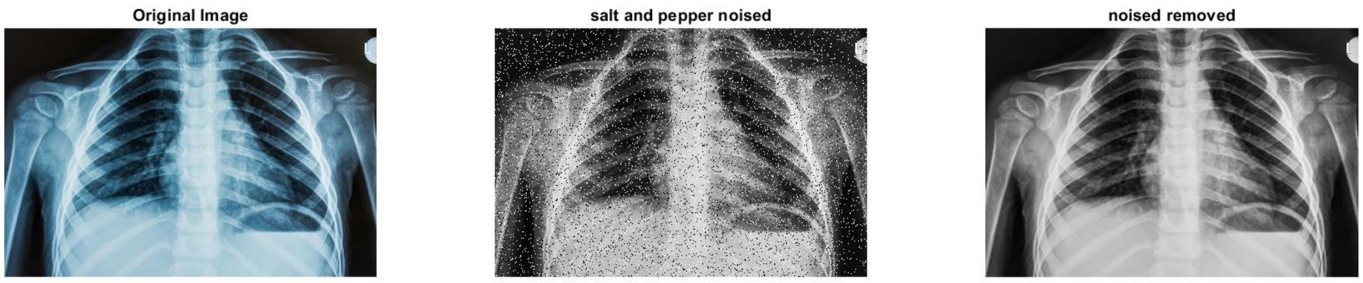


Figure 17: median filter in MATLAB is able to filter both salt and pepper noise.

Shreenandan Sahu |120BM0806

Lab Report 5

Aim

To apply frequency domain filters for image enhancement in MATLAB.

Theory

In image processing, frequency domain filters are used to modify or enhance the frequency content of an image. An image can be represented in the frequency domain using the Fourier transform, which decomposes the image into its constituent frequencies. Frequency domain filters in image processing can be broadly classified into two categories: low-pass filters and high-pass filters. Low-pass filters attenuate high-frequency components in the image, while high-pass filters attenuate low-frequency components. Low-pass filters are commonly used in image smoothing, where they are used to remove high-frequency noise from the image while preserving the lower-frequency content. The most common type of low-pass filter used in image processing is the Gaussian filter, which attenuates high-frequency components according to a Gaussian distribution. High-pass filters, on the other hand, are used to enhance the high-frequency content in an image, making the edges and details of the image more visible. Commonly used high-pass filters in image processing include the Laplacian filter, the Sobel filter, and the Canny filter. Other types of frequency domain filters used in image processing include band-pass filters, which allow a certain range of frequencies to pass through while attenuating other frequencies, and band-stop filters, which attenuate a certain range of frequencies while passing other frequencies. Frequency domain filters in image processing are useful in a wide range of applications, including medical imaging, remote sensing, and industrial inspection, where image quality is critical and noise or interference can have a significant impact on the results.

LOW PASS AND HIGH PASS FILTER

Low Pass Filter:

In image processing, low-pass filters are used to remove high-frequency noise and to smooth images by attenuating high-frequency components while preserving low-frequency components. The most common type of low-pass filter used in image processing is the Gaussian filter. The Gaussian filter is a frequency domain filter that attenuates high-frequency components according to a Gaussian distribution. It is a popular choice for image smoothing because it has a simple mathematical form, is easy to implement, and produces visually pleasing results. The Gaussian filter works by convolving the image with a Gaussian kernel, which is a two-dimensional bell-shaped curve centred at the origin. The size of the kernel and the standard deviation of the Gaussian distribution determine the degree of smoothing and the amount of detail preserved in the image. Another commonly used low-pass filter in image processing is the mean filter, which replaces each pixel in the image with the average of its neighbouring pixels. The size of the neighbourhood or the window used for averaging determines the degree of smoothing, with larger neighbourhoods resulting in greater smoothing and more detail loss. Low-pass filters in image processing can be used to reduce noise, blur, or hide details in an image. They are commonly used in applications such as image denoising, feature extraction, and image segmentation. It is important to note that excessive smoothing can result in loss of important details and edges in the image. Therefore, the choice of the filter and the parameters used for filtering should be carefully tuned to achieve the desired level of smoothing while preserving important features in the image.

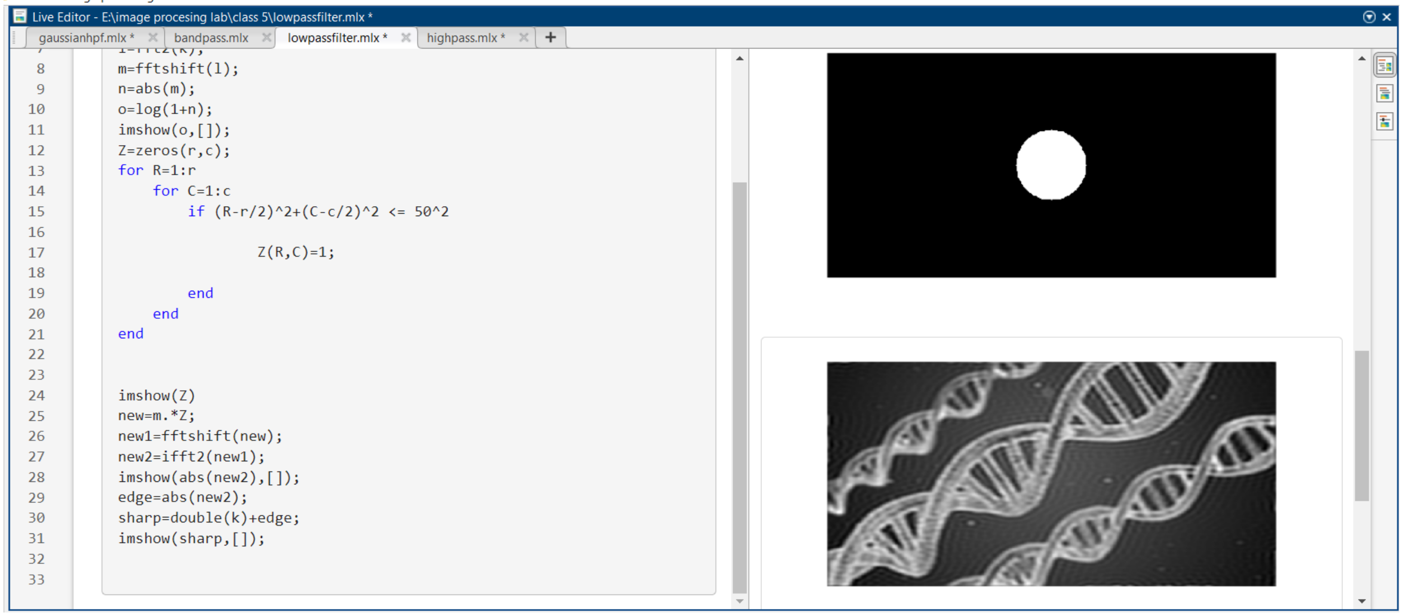
High Pass Filter:

In image processing, high-pass filters are used to enhance the high-frequency content of an image by attenuating the low-frequency components. High-pass filters are commonly used for edge detection and sharpening of images. The most common type of high-pass filter used in image processing is the Laplacian filter. The Laplacian filter enhances edges and details in an image by detecting areas where the brightness changes rapidly. It does this by convolving the image with the second derivative of the Gaussian function, which is a measure of the rate of change of the image intensity. Another commonly used high-pass filter is the Sobel filter, which is used for edge detection in images. The Sobel filter works by computing the gradient of the image intensity in the x and y directions, and then combining these gradients to obtain a measure of the edge strength. Other types of high-pass filters used in image processing include the Prewitt filter, the Roberts cross filter, and the Canny filter. Each of these filters has its own advantages and disadvantages, and the choice of filter depends on the specific application and the desired outcome. High-pass filters can be useful in a wide range of applications, including medical imaging, industrial inspection, and remote sensing, where image detail and clarity are critical. However, it is important to note that excessive filtering can result in the loss of important image information, such as subtle details and textures. Therefore, the choice of filter and the parameters used for filtering should be carefully tuned to achieve the desired level of enhancement while preserving important image features.

CODE

```
% LOW PASS FILTER

fprintf("PLEASE SELECT AN IMAGE\n")
i=uigetfile('*.*');
j=imread(i)
k=rgb2gray(j);
imshow(k)
[r,c]=size(k)
l=fft2(k);
m=fftshift(l);
n=abs(m);
o=log(1+n);
imshow(o,[]);
Z=zeros(r,c);
for R=1:r
    for C=1:c
        if (R-r/2)^2+(C-c/2)^2 <= 50^2
                    Z(R,C)=1;
```

```
            end
        end
end
imshow(Z)
new=m.*Z;
new1=fftshift(new);
new2=ifft2(new1);
imshow(abs(new2),[]);
edge=abs(new2);
sharp=double(k)+edge;
imshow(sharp,[]);
```



Live Editor - E:\image procesing lab\class 5\lowpassfilter.mlx *

```
m=fftshift(1);
n=abs(m);
o=log(1+n);
imshow(o,[]);
Z=zeros(r,c);
for R=1:r
    for C=1:c
        if (R-r/2)^2+(C-c/2)^2 <= 50^2

                Z(R,C)=1;


        end
    end
end


imshow(Z)
new=m.*Z;
new1=fftshift(new);
new2=ifft2(new1);
imshow(abs(new2),[]);
edge=abs(new2);
sharp=double(k)+edge;
imshow(sharp,[]);
```

Figure 1: use of low pass filter to smoothen image in MATLAB

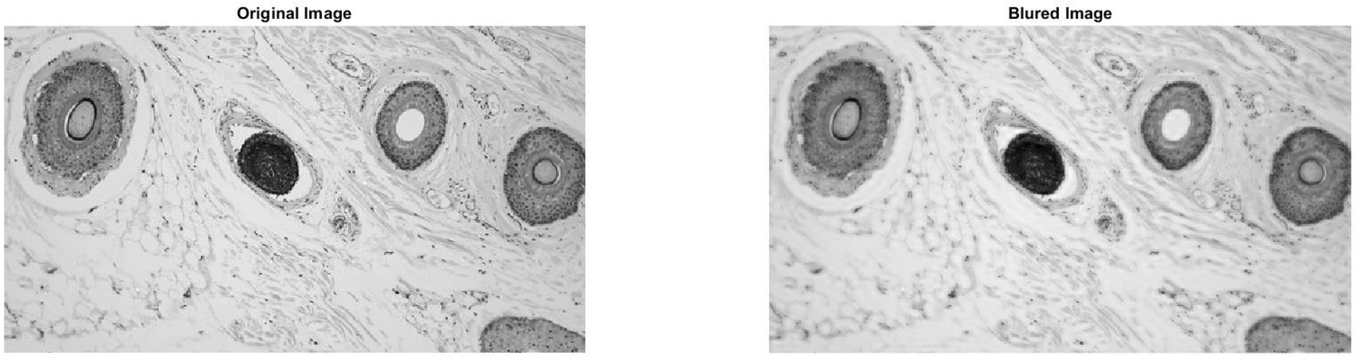
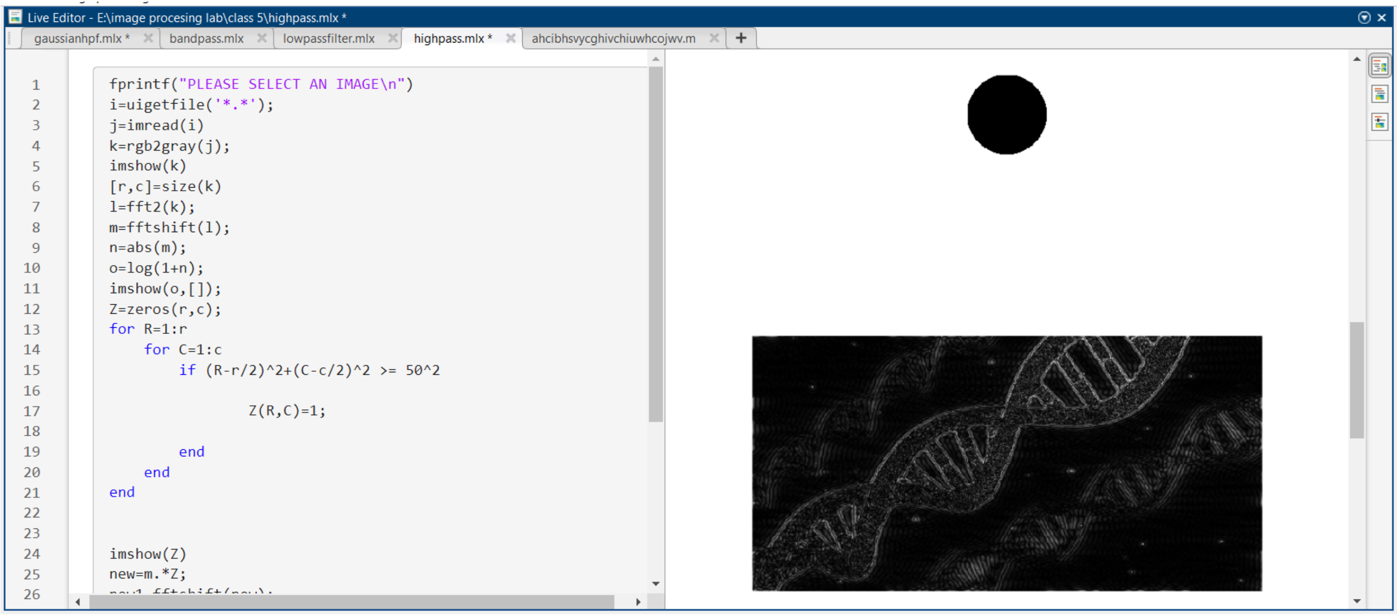


Figure 2: image smoothening using low pass filter in MATLAB

CODE

```
% HIGH PASS FILTER
fprintf("PLEASE SELECT AN IMAGE\n")
i=uigetfile('*.*');
j=imread(i)
k=rgb2gray(j);
imshow(k)
[r,c]=size(k)
l=fft2(k);
m=fftshift(l);
n=abs(m);
o=log(1+n);
imshow(o,[]);
Z=zeros(r,c);
for R=1:r

    for C=1:c

        if (R-r/2)^2+(C-c/2)^2 >= 50^2

        Z(R,C)=1;

        end

    end

end
imshow(Z)
new=m.*Z;
new1=fftshift(new);
new2=ifft2(new1);
imshow(abs(new2),[]);
edge=abs(new2);
sharp=double(k)+edge;
imshow(sharp,[]);
```


Live Editor - E:\image procesing lab\class 5\highpass.mlx *
gaussianhpf.mlx *
bandpass.mlx
lowpassfilter.mlx
highpass.mlx *
ahcibhsvycghivchiuwhcojwv.m

```
fprintf("PLEASE SELECT AN IMAGE\n")
i=uigetfile('*.*');
j=imread(i)
k=rgb2gray(j);
imshow(k)
[r,c]=size(k)
l=fft2(k);
m=fftshift(l);
n=abs(m);
o=log(1+n);
imshow(o,[]);
Z=zeros(r,c);
for R=1:r
    for C=1:c
        if (R-r/2)^2+(C-c/2)^2 >= 50^2

                Z(R,C)=1;

        end
    end
end


imshow(Z)
new=m.*Z;
```

Figure 3: use of high pass filter to detect the edges in MATLAB

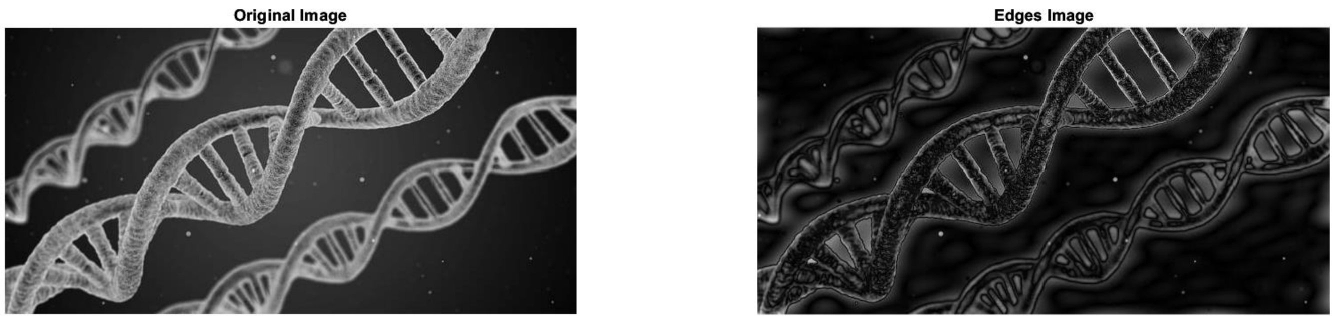


Figure 4: image edge detection using high pass filter in MATLAB

GAUSSIAN FILTERS

Gaussian Filter:

A Gaussian filter is a popular image processing technique used for smoothing or blurring an image by reducing the high-frequency components in the image. The filter is based on the Gaussian function, which is a bell-shaped curve that describes the probability distribution of a random variable.

In image processing, the Gaussian filter works by convolving each pixel of the image with a 2D Gaussian kernel, which is a matrix of values that represents the Gaussian function. The kernel size and standard deviation of the Gaussian function can be adjusted to control the amount of smoothing applied to the image.

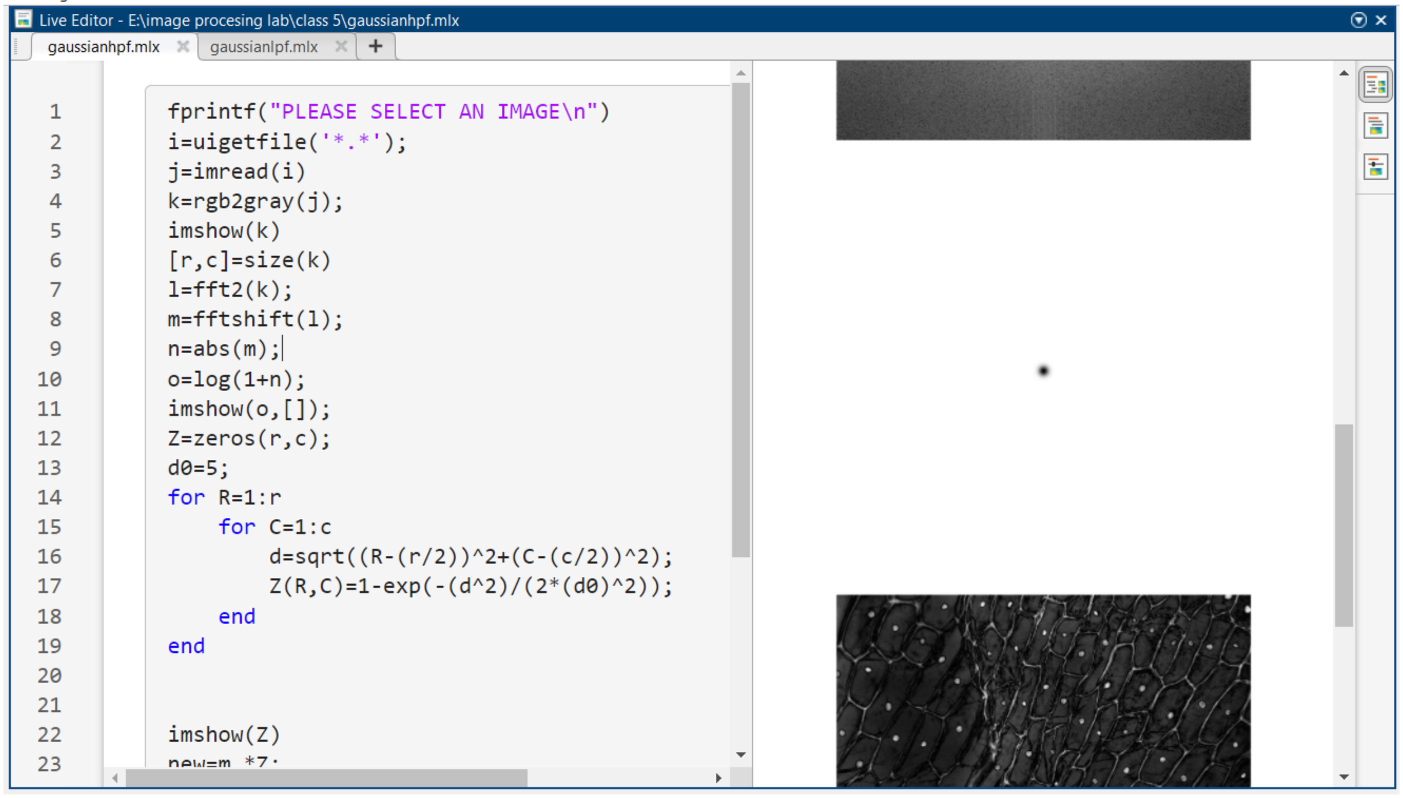
The Gaussian filter is commonly used in applications such as noise reduction, edge detection, and feature extraction. It is particularly useful for removing high-frequency noise from an image while preserving the edges and other important details.

Overall, the Gaussian filter is a simple and effective way to enhance the visual quality of images in a variety of applications.

CODE

```
% GAUSSIAN HIGH PASS FILTER

fprintf("PLEASE SELECT AN IMAGE\n")
i=uigetfile('*.*');
j=imread(i)
k=rgb2gray(j);
imshow(k)
[r,c]=size(k)
l=fft2(k);
m=fftshift(l);
n=abs(m);
o=log(1+n);
imshow(o,[]);
Z=zeros(r,c);
d0=5;
for R=1:r
    for C=1:c
        d=sqrt((R-(r/2))^2+(C-(c/2))^2);
        Z(R,C)=1-exp(-(d^2)/(2*(d0)^2));
    end
end



imshow(Z)
new=m.*Z;
new1=fftshift(new);
new2=ifft2(new1);
imshow(abs(new2),[]);
edge=abs(new2);
```

```
sharp=double(k)+edge;
subplot(1,2,1); imshow(sharp,[]); title('Sharpened Image');

subplot(1,2,2); imshow(k,[]); title('Original Image');
```



Live Editor - E:\image procesing lab\class 5\gaussianhpf.mlx

```
fprintf("PLEASE SELECT AN IMAGE\n")
i=uigetfile('*.*');
j=imread(i)
k=rgb2gray(j);
imshow(k)
[r,c]=size(k)
l=fft2(k);
m=fftshift(l);
n=abs(m);
o=log(1+n);
imshow(o,[]);
Z=zeros(r,c);
d0=5;
for R=1:r
    for C=1:c
        d=sqrt((R-(r/2))^2+(C-(c/2))^2);
        Z(R,C)=1-exp(-(d^2)/(2*(d0)^2));
    end
end


imshow(Z)
new=m.*Z;
```

Figure 5: use of gaussian high pass filter in MATLAB

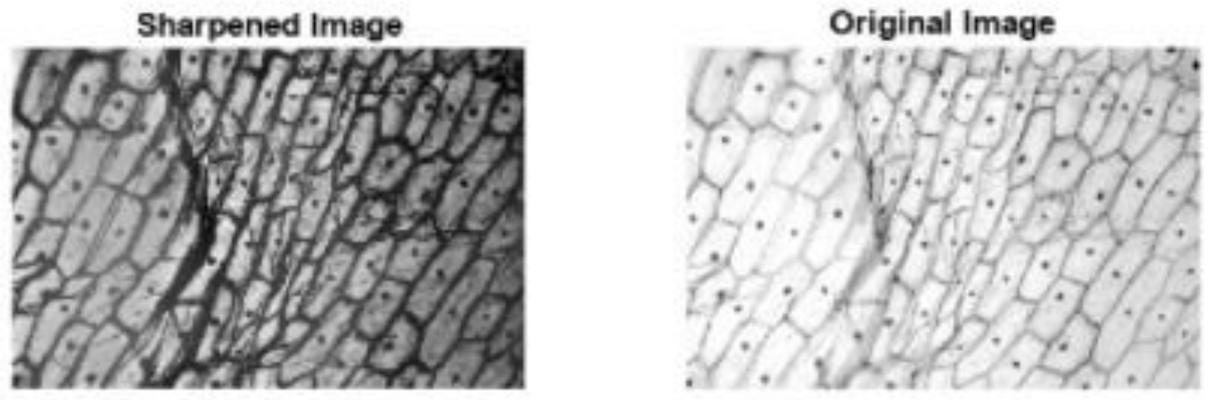
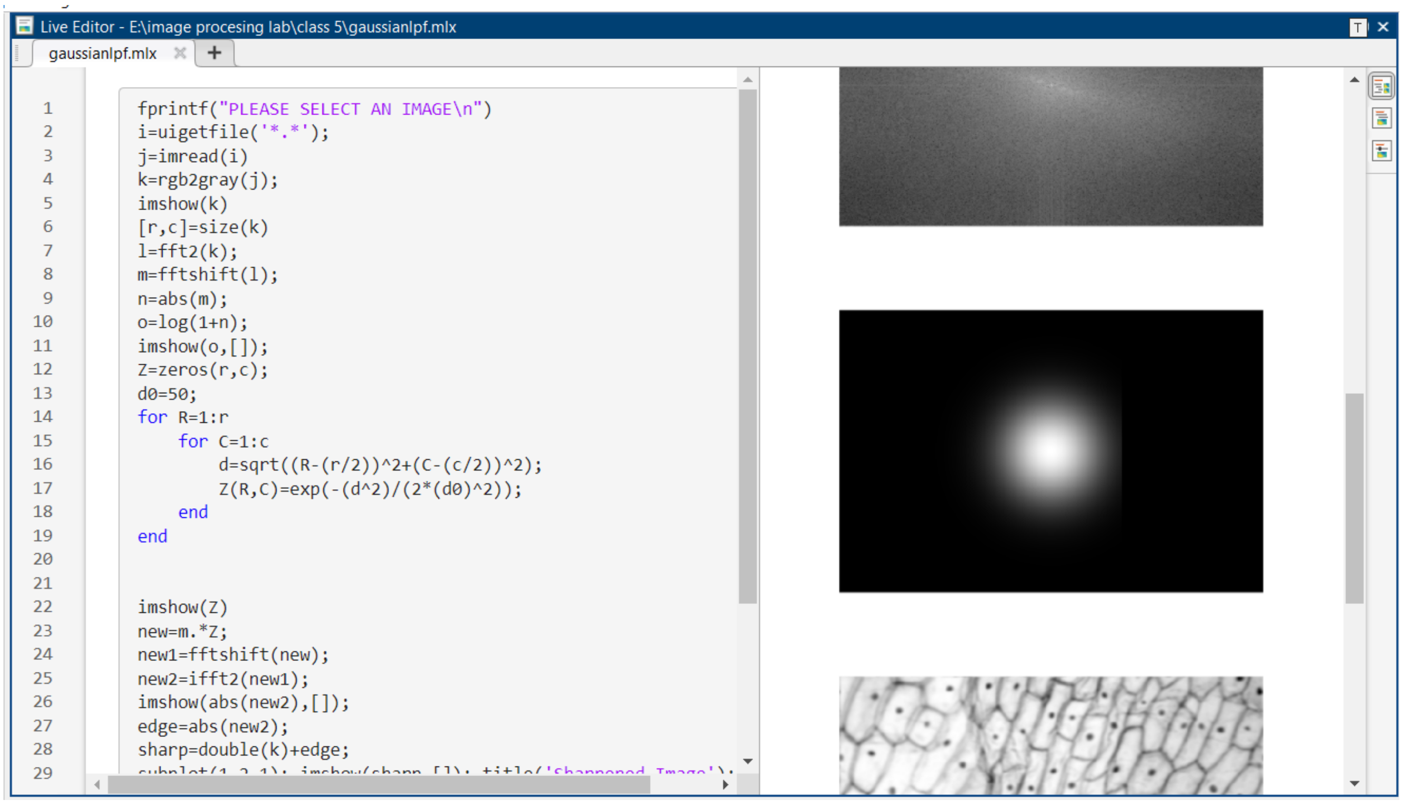


Figure 6: image processing using high pass filter in MATLAB

CODE

```
% GAUSSIAN LOW PASS FILTER
```

```
fprintf("PLEASE SELECT AN IMAGE\n")
i=uigetfile('*.*');
j=imread(i)
k=rgb2gray(j);
imshow(k)
[r,c]=size(k)
l=fft2(k);
m=fftshift(l);
n=abs(m);
o=log(1+n);
imshow(o,[]);
Z=zeros(r,c);
d0=50;
for R=1:r
    for C=1:c
        d=sqrt((R-(r/2))^2+(C-(c/2))^2);
        Z(R,C)=exp(-(d^2)/(2*(d0)^2));
    end
end


imshow(Z)
new=m.*Z;
new1=fftshift(new);
new2=ifft2(new1);
imshow(abs(new2),[]);
edge=abs(new2);
sharp=double(k)+edge;
subplot(1,2,1); imshow(sharp,[]); title('Sharpened Image');

subplot(1,2,2); imshow(k,[]); title('Original Image');
```


Live Editor - E:\image procesing lab\class 5\gaussianlpf.mlx
gaussianlpf.mlx

```
fprintf("PLEASE SELECT AN IMAGE\n")
i=uigetfile('*.*');
j=imread(i)
k=rgb2gray(j);
imshow(k)
[r,c]=size(k)
l=fft2(k);
m=fftshift(l);
n=abs(m);
o=log(1+n);
imshow(o,[]);
Z=zeros(r,c);
d0=50;
for R=1:r
    for C=1:c
        d=sqrt((R-(r/2))^2+(C-(c/2))^2);
        Z(R,C)=exp(-(d^2)/(2*(d0)^2));
    end
end


imshow(Z)
new=m.*Z;
new1=fftshift(new);
new2=ifft2(new1);
imshow(abs(new2),[]);
edge=abs(new2);
sharp=double(k)+edge;
```

Figure 7: use of gaussian high low filter in MATLAB

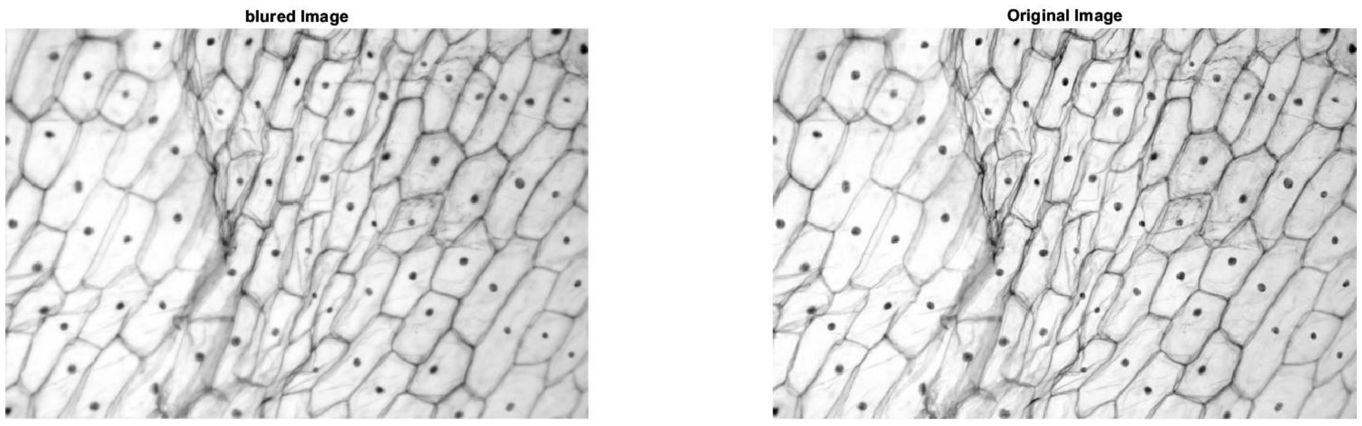


Figure 8: image procesing using gaussian low pass filter in MATLAB

BAND FILTER

Band Pass and Band Reject Filter:

In image processing, a band-pass filter is a type of filter that allows a specific range of frequencies to pass through while attenuating (reducing) frequencies outside that range. Similarly, a band-reject filter, also known as a notch filter, attenuates a specific range of frequencies while allowing frequencies outside that range to pass through.

A band-pass filter is often used to extract certain features or details from an image that exist in a particular frequency range. For example, a band-pass filter may be used to enhance the edges or texture of an image, or to remove certain types of noise that exist in a specific frequency range.

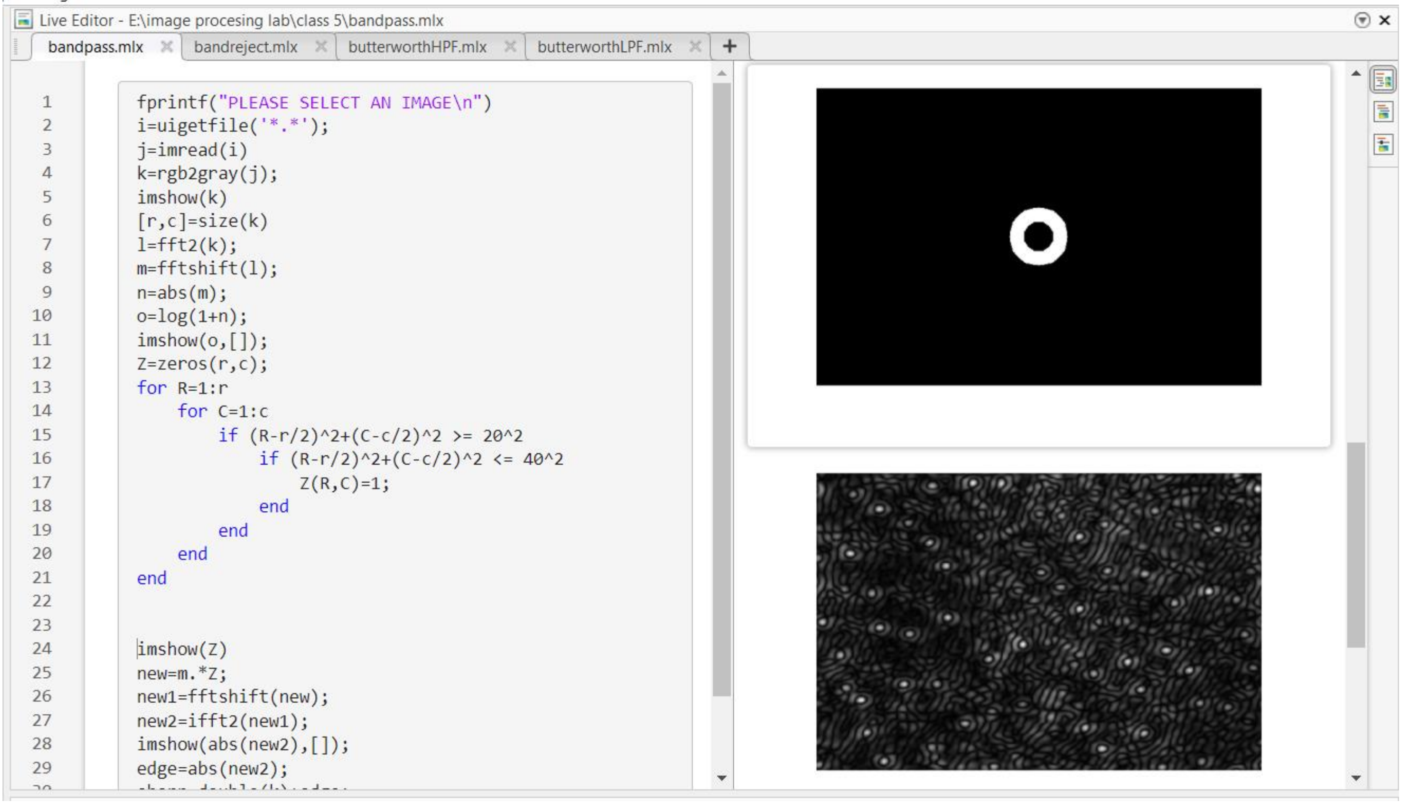
A band-reject filter, on the other hand, is often used to remove unwanted features or artifacts from an image. For example, a band-reject filter may be used to remove periodic noise or interference that exists in a certain frequency range.

Both band-pass and band-reject filters can be implemented using a variety of techniques, such as Fourier transforms or digital signal processing algorithms. The specific parameters of the filter, such as the cutoff frequencies or filter order, can be adjusted to achieve the desired filtering effect.

CODE

```
% BAND PASS FILTER

fprintf("PLEASE SELECT AN IMAGE\n")
i=uigetfile('*.*');
j=imread(i)
k=rgb2gray(j);
imshow(k)
[r,c]=size(k)
l=fft2(k);
m=fftshift(l);
n=abs(m);
o=log(1+n);
imshow(o,[]);
Z=zeros(r,c);
for R=1:r
    for C=1:c
        if (R-r/2)^2+(C-c/2)^2 >= 20^2
            if (R-r/2)^2+(C-c/2)^2 <= 40^2
                Z(R,C)=1;
            end
        end
    end
end
```

```
imshow(Z)
new=m.*Z;
new1=fftshift(new);
new2=ifft2(new1);
imshow(abs(new2),[]);
edge=abs(new2);
sharp=double(k)+edge;
imshow(sharp,[]);
```



Live Editor - E:\image procesing lab\class 5\bandpass.mlx

bandpass.mlx | bandreject.mlx | butterworthHPF.mlx | butterworthLPF.mlx

```
fprintf("PLEASE SELECT AN IMAGE\n")
i=uigetfile('*.*');
j=imread(i)
k=rgb2gray(j);
imshow(k)
[r,c]=size(k)
l=fft2(k);
m=fftshift(l);
n=abs(m);
o=log(1+n);
imshow(o,[]);
Z=zeros(r,c);
for R=1:r
    for C=1:c
        if (R-r/2)^2+(C-c/2)^2 >= 20^2
            if (R-r/2)^2+(C-c/2)^2 <= 40^2
                Z(R,C)=1;
            end
        end
    end
end


imshow(Z)
new=m.*Z;
new1=fftshift(new);
new2=ifft2(new1);
imshow(abs(new2),[]);
edge=abs(new2);
```

Figure 9: use of image band pass filter in MATLAB

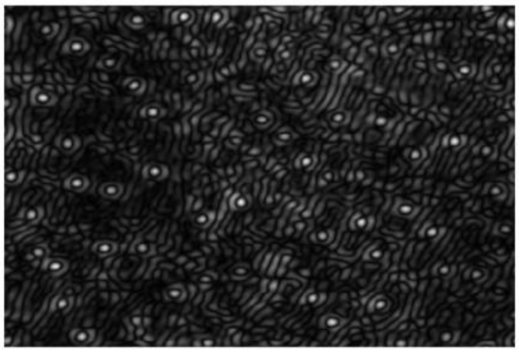
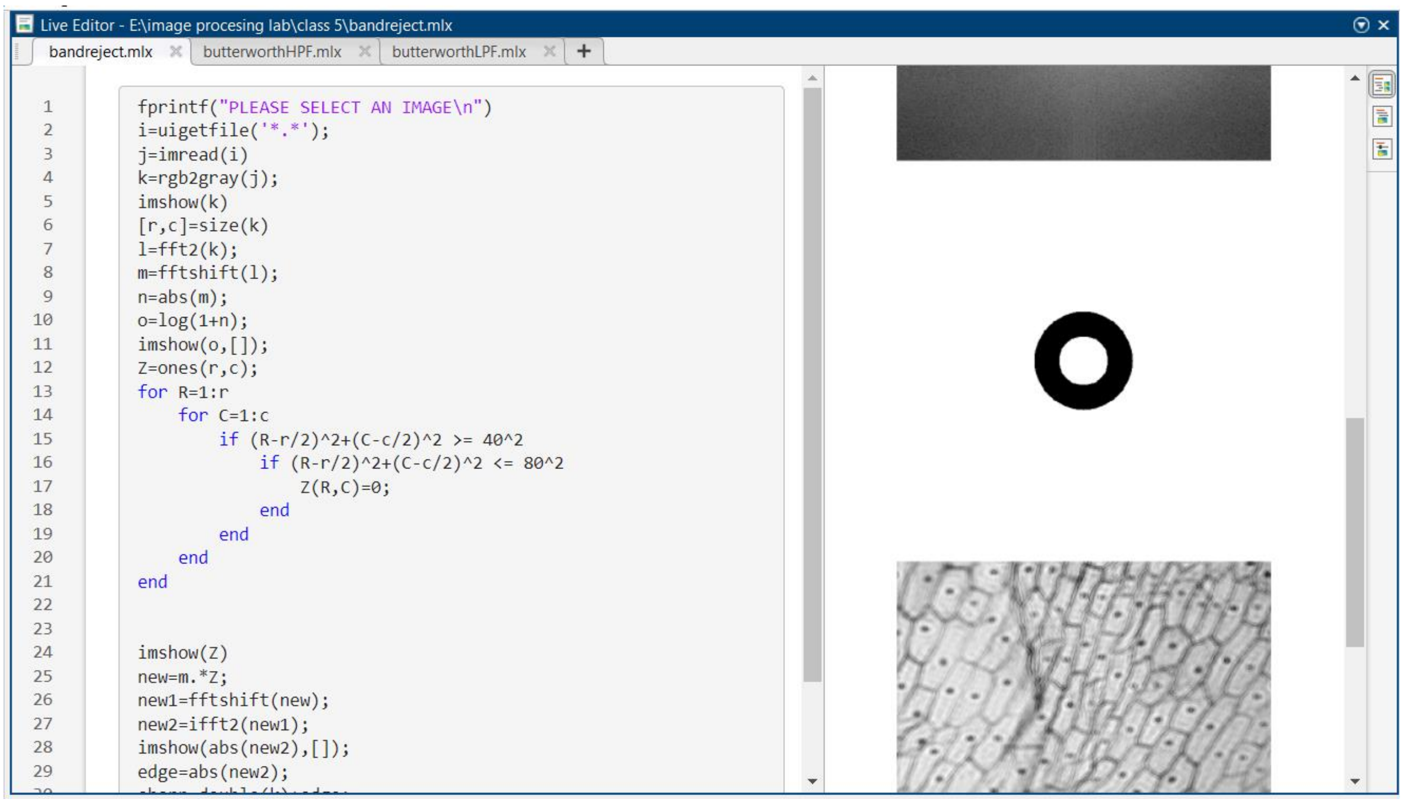


Figure 10: image processing using band pass filter in MATLAB

CODE

```
% BAND REJECT FILTER

fprintf("PLEASE SELECT AN IMAGE\n")
i=uigetfile('*.*');
j=imread(i)
k=rgb2gray(j);
imshow(k)
[r,c]=size(k)
l=fft2(k);
m=fftshift(l);
n=abs(m);
o=log(1+n);
imshow(o,[]);
Z=ones(r,c);
for R=1:r
    for C=1:c
        if (R-r/2)^2+(C-c/2)^2 >= 40^2
            if (R-r/2)^2+(C-c/2)^2 <= 80^2
                Z(R,C)=0;
            end
        end
    end
end


imshow(Z)
new=m.*Z;
new1=fftshift(new);
new2=ifft2(new1);
imshow(abs(new2),[]);
edge=abs(new2);
sharp=double(k)+edge;
imshow(sharp,[]);
```



```
fprintf("PLEASE SELECT AN IMAGE\n")
i=uigetfile('*.*');
j=imread(i)
k=rgb2gray(j);
imshow(k)
[r,c]=size(k)
l=fft2(k);
m=fftshift(l);
n=abs(m);
o=log(1+n);
imshow(o,[]);
Z=ones(r,c);
for R=1:r
    for C=1:c
        if (R-r/2)^2+(C-c/2)^2 >= 40^2
            if (R-r/2)^2+(C-c/2)^2 <= 80^2
                Z(R,C)=0;
            end
        end
    end
end


imshow(Z)
new=m.*Z;
new1=fftshift(new);
new2=ifft2(new1);
imshow(abs(new2),[]);
edge=abs(new2);
```

Figure 11: use of image band reject filter in MATLAB

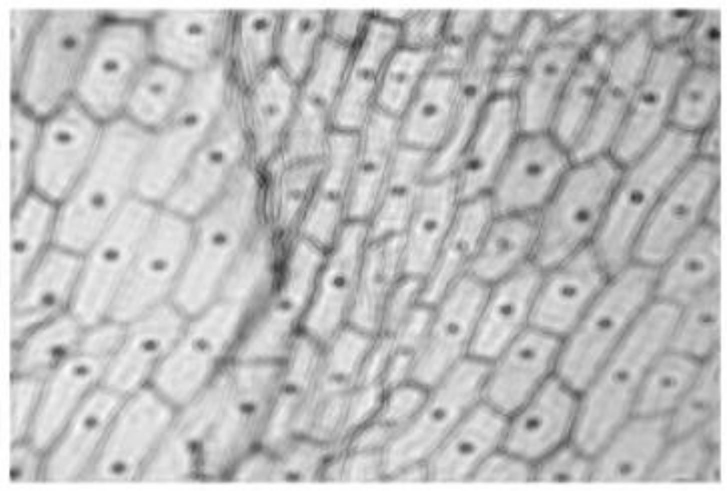


Figure 12: image processing using band reject filter in MATLAB

BUTTER WORTH FILTERS

Butter Worth High and Low pass filter:

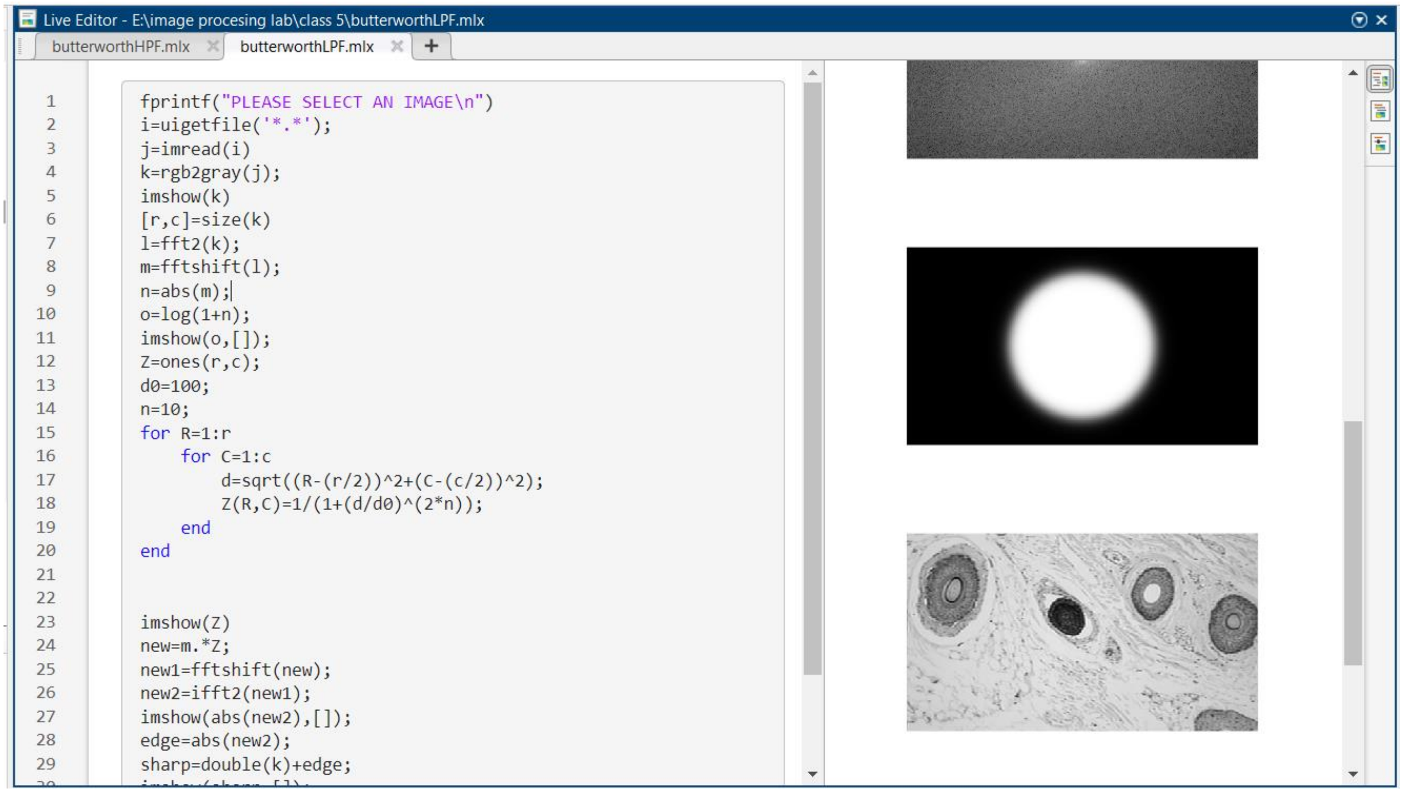
A Butterworth filter is a type of digital filter used in image processing to enhance or suppress certain frequency components in an image. It is a low-pass filter that is designed to allow low-frequency components to pass through while attenuating high-frequency components. The filter is named after the British engineer and physicist Stephen Butterworth, who developed it in the 1930s. It is widely used in image processing applications such as image smoothing, noise reduction, and edge detection.

The Butterworth filter is designed based on a specific order and cut-off frequency. The order of the filter determines how sharply the filter cuts off the high-frequency components. The cut-off frequency is the frequency at which the filter begins to attenuate the high-frequency components. In image processing, the Butterworth filter can be applied to the Fourier transform of an image, which represents the image's frequency content. The filter's cut-off frequency and order can be chosen based on the specific application requirements. The Butterworth filter has several advantages over other types of filters, including its smooth transition between passband and stopband, and its ability to attenuate high-frequency noise without affecting the image's edges. However, it can also introduce ringing artifacts and may not be suitable for all types of image processing applications. Overall, the Butterworth filter is a useful tool in image processing for selectively enhancing or suppressing frequency components in an image, and its specific design parameters can be adjusted to achieve the desired results.

CODE

```
% BUTTER WORTH LOW PASS FILTER


fprintf("PLEASE SELECT AN IMAGE\n")
i=uigetfile('*.*');
j=imread(i)
k=rgb2gray(j);
imshow(k)
[r,c]=size(k)
l=fft2(k);
m=fftshift(l);
n=abs(m);
o=log(1+n);
imshow(o,[]);
Z=ones(r,c);
d0=100;
n=10;
for R=1:r
    for C=1:c
        d=sqrt((R-(r/2))^2+(C-(c/2))^2);
        Z(R,C)=1/(1+(d/d0)^(2*n));
    end
end



imshow(Z)
```

```
new=m.*Z;
new1=fftshift(new);
new2=ifft2(new1);
imshow(abs(new2),[]);
edge=abs(new2);
sharp=double(k)+edge;
imshow(sharp,[]);
```


Live Editor - E:\image procesing lab\class 5\butterworthLPF.mlx
butterworthHPF.mlx
butterworthLPF.mlx

```
fprintf("PLEASE SELECT AN IMAGE\n")
i=uigetfile('*.*');
j=imread(i)
k=rgb2gray(j);
imshow(k)
[r,c]=size(k)
l=fft2(k);
m=fftshift(l);
n=abs(m);
o=log(1+n);
imshow(o,[]);
Z=ones(r,c);
d0=100;
n=10;
for R=1:r
    for C=1:c
        d=sqrt((R-(r/2))^2+(C-(c/2))^2);
        Z(R,C)=1/(1+(d/d0)^(2*n));
    end
end


imshow(Z)
new=m.*Z;
new1=fftshift(new);
new2=ifft2(new1);
imshow(abs(new2),[]);
edge=abs(new2);
sharp=double(k)+edge;
```

Figure 13: use of butter worth low pass filter in MATLAB

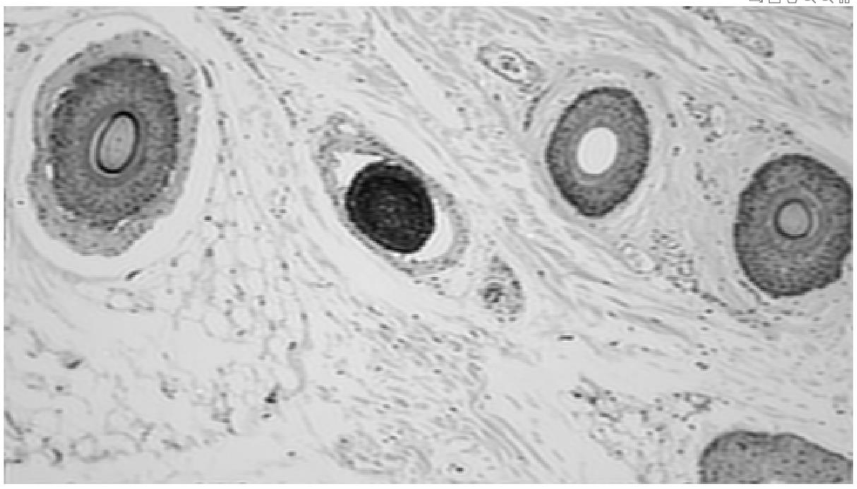
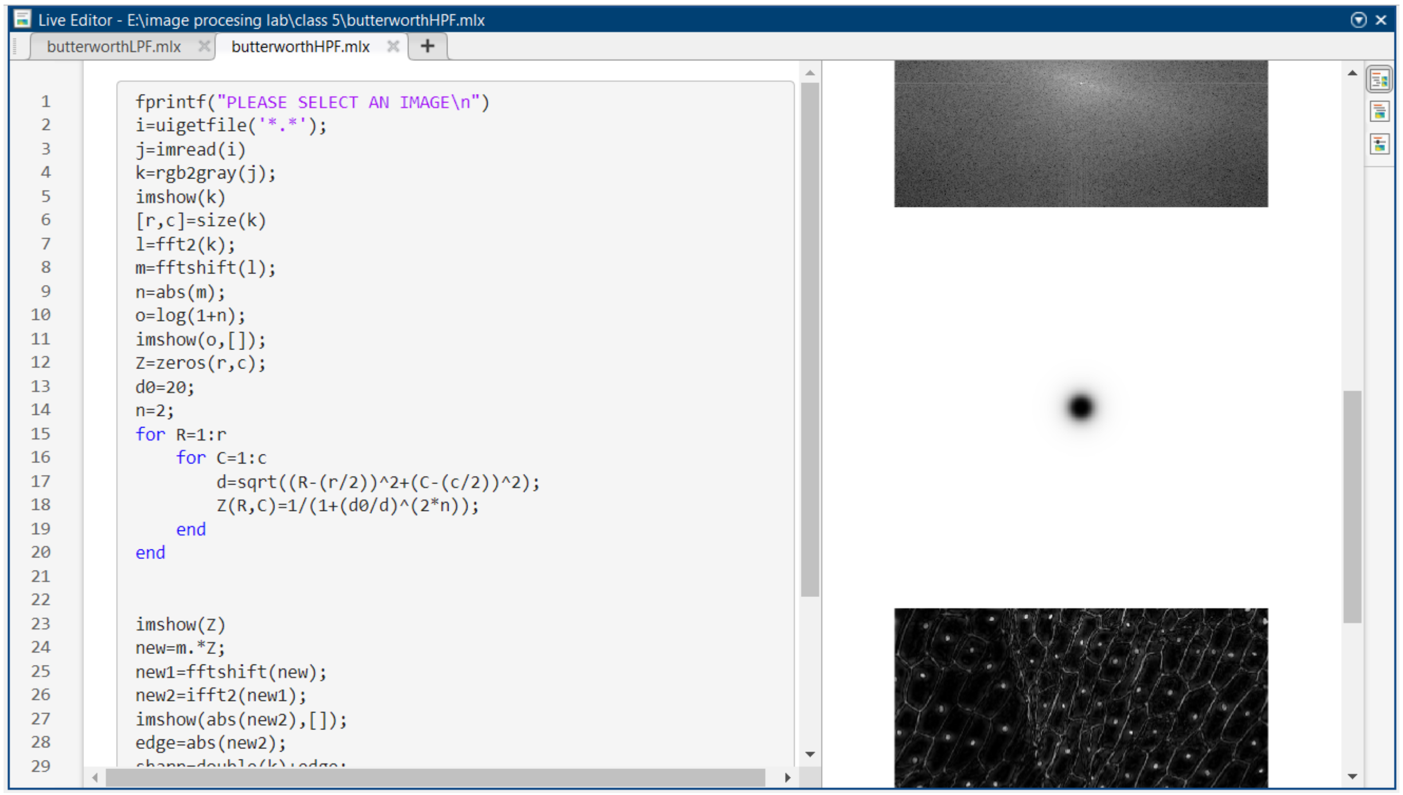


Figure 14: image processing butter worth low pass filter MATLAB

CODE

```
% BUTTER WORTH HIGH PASS FILTER

fprintf("PLEASE SELECT AN IMAGE\n")
i=uigetfile('*.*');
j=imread(i)
k=rgb2gray(j);
imshow(k)
[r,c]=size(k)
l=fft2(k);
m=fftshift(l);
n=abs(m);
o=log(1+n);
imshow(o,[]);
Z=zeros(r,c);
d0=20;
n=2;
for R=1:r
    for C=1:c
        d=sqrt((R-(r/2))^2+(C-(c/2))^2);
        Z(R,C)=1/(1+(d0/d)^(2*n));
    end
end



imshow(Z)
new=m.*Z;
new1=fftshift(new);
new2=ifft2(new1);
imshow(abs(new2),[]);
edge=abs(new2);
sharp=double(k)+edge;
subplot(1,2,1); imshow(sharp,[]); title('Sharpened Image');

subplot(1,2,2); imshow(k,[]); title('Original Image');
```


Live Editor - E:\image procesing lab\class 5\butterworthHPF.mlx
butterworthLPF.mlx
butterworthHPF.mlx

```
fprintf("PLEASE SELECT AN IMAGE\n")
i=uigetfile('*.*');
j=imread(i)
k=rgb2gray(j);
imshow(k)
[r,c]=size(k)
l=fft2(k);
m=fftshift(l);
n=abs(m);
o=log(1+n);
imshow(o,[]);
Z=zeros(r,c);
d0=20;
n=2;
for R=1:r
    for C=1:c
        d=sqrt((R-(r/2))^2+(C-(c/2))^2);
        Z(R,C)=1/(1+(d0/d)^(2*n));
    end
end


imshow(Z)
new=m.*Z;
new1=fftshift(new);
new2=ifft2(new1);
imshow(abs(new2),[]);
edge=abs(new2);
sharp=double(k)+edge;
```

Figure 15: use of butter worth high pass filter in MATLAB

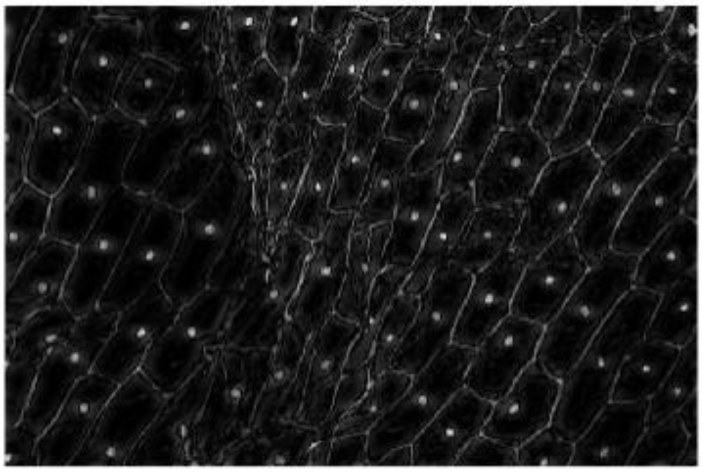


Figure 16: image processing butter worth high pass filter MATLAB

Shreenandan Sahu |120BM0806

Lab Report 6

Aim

Using different mean filters for removal of noise in MATLAB.

Theory

Different mean filters can be used to remove noise from an image. The most common types of mean filters are the arithmetic mean filter, geometric mean filter, and harmonic mean filter.

Arithmetic mean filter: This filter replaces each pixel with the average value of the neighboring pixels. It is a simple and effective filter for removing random noise. However, it can blur edges and reduce image details.

Geometric mean filter: This filter replaces each pixel with the geometric mean of the neighboring pixels. It is effective at removing multiplicative noise, such as speckle noise in ultrasound or radar images. However, it can also blur the image if the kernel size is too large.

Harmonic mean filter: This filter replaces each pixel with the harmonic mean of the neighboring pixels. It is effective at removing salt-and-pepper noise and preserving edges, but it can produce artifacts in uniform areas of the image.

Alpha-trimmed mean filter: This filter removes a certain percentage of the highest and lowest pixel values in the neighbourhood before calculating the mean. It is effective at removing impulse noise and preserving edges and details, but it may also introduce some smoothing.

Adaptive local noise reduction filter: This filter estimates the local noise level in the image and adapts the filter parameters accordingly. It is effective at removing noise while preserving details and edges.

Midpoint filter: This filter replaces each pixel with the midpoint value of the neighboring pixels. It is effective at removing noise while preserving edges and details, but it may also produce some smoothing.

Contraharmonic mean filter: This filter replaces each pixel with the Contraharmonic mean of the neighboring pixels. It is effective at removing noise of a certain type, such as Gaussian or impulse noise, but it may also produce some artifacts and reduce image sharpness.

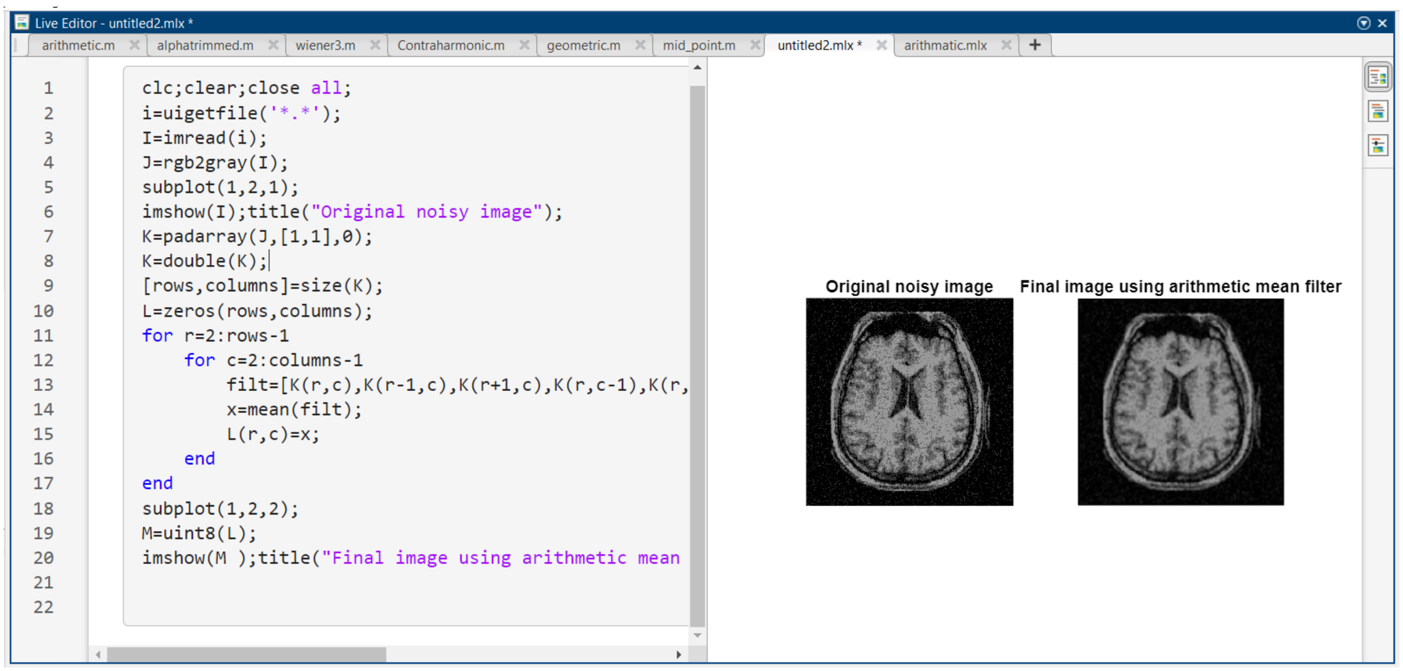
To apply a mean filter to an image, a kernel or mask of a specified size is moved over each pixel in the image. The value of each pixel in the new image is then calculated as the mean value of the values of all the pixels within the kernel. The size of the kernel determines the amount of smoothing and the level of detail preservation.

It is important to note that while mean filters are effective at removing noise, they may also introduce unwanted artifacts and reduce image sharpness. It is therefore important to carefully choose the filter type and parameters to achieve the desired level of noise reduction while preserving image details.

CODE

```
% ARITHMATIC MEAN FILTER


clc;clear;close all;
i=uigetfile('*.*');
```

```
I=imread(i);
J=rgb2gray(I);
subplot(1,2,1);
imshow(I);title("Original noisy image");
K=padarray(J,[1,1],0);
K=double(K);
[rows,columns]=size(K);
L=zeros(rows,columns);
for r=2:rows-1
    for c=2:columns-1
        filt=[K(r,c),K(r-1,c),K(r+1,c),K(r,c-1),K(r,c+1),K(r-1,c-
1),K(r-1,c+1),K(r+1,c-1),K(r+1,c+1)];
        x=mean(filt);
        L(r,c)=x;
    end
end
subplot(1,2,2);
M=uint8(L);
imshow(M );title("Final image using arithmetic mean filter");
```



```
clc;clear;close all;
i=uigetfile('*.*');
I=imread(i);
J=rgb2gray(I);
subplot(1,2,1);
imshow(I);title("Original noisy image");
K=padarray(J,[1,1],0);
K=double(K);
[rows,columns]=size(K);
L=zeros(rows,columns);
for r=2:rows-1
    for c=2:columns-1
        filt=[K(r,c),K(r-1,c),K(r+1,c),K(r,c-1),K(r,
        x=mean(filt);
        L(r,c)=x;
    end
end
subplot(1,2,2);
M=uint8(L);
imshow(M );title("Final image using arithmetic mean
```

Figure 1: use of arithmetic mean filter to process image in MATLAB

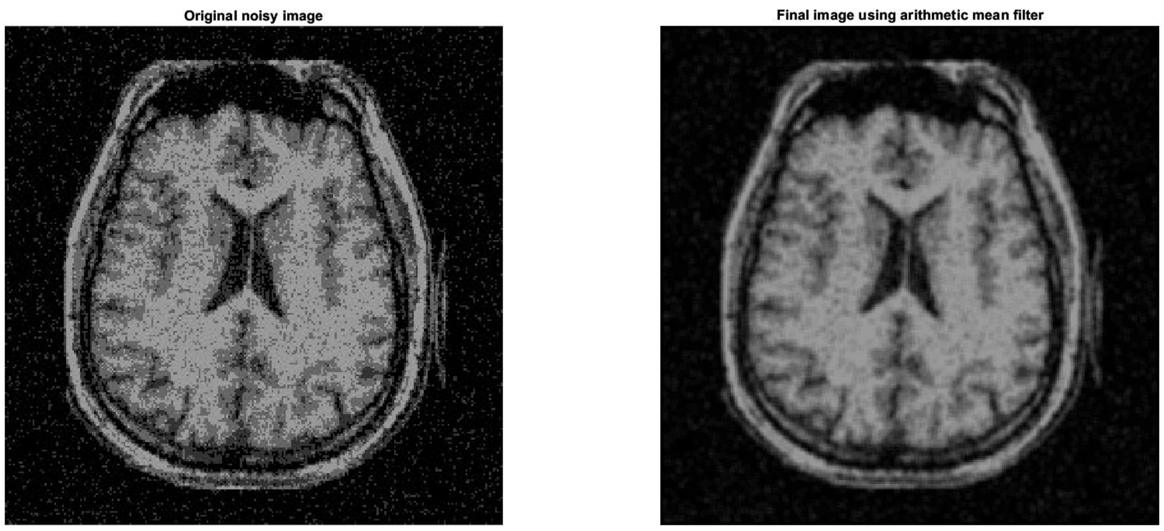
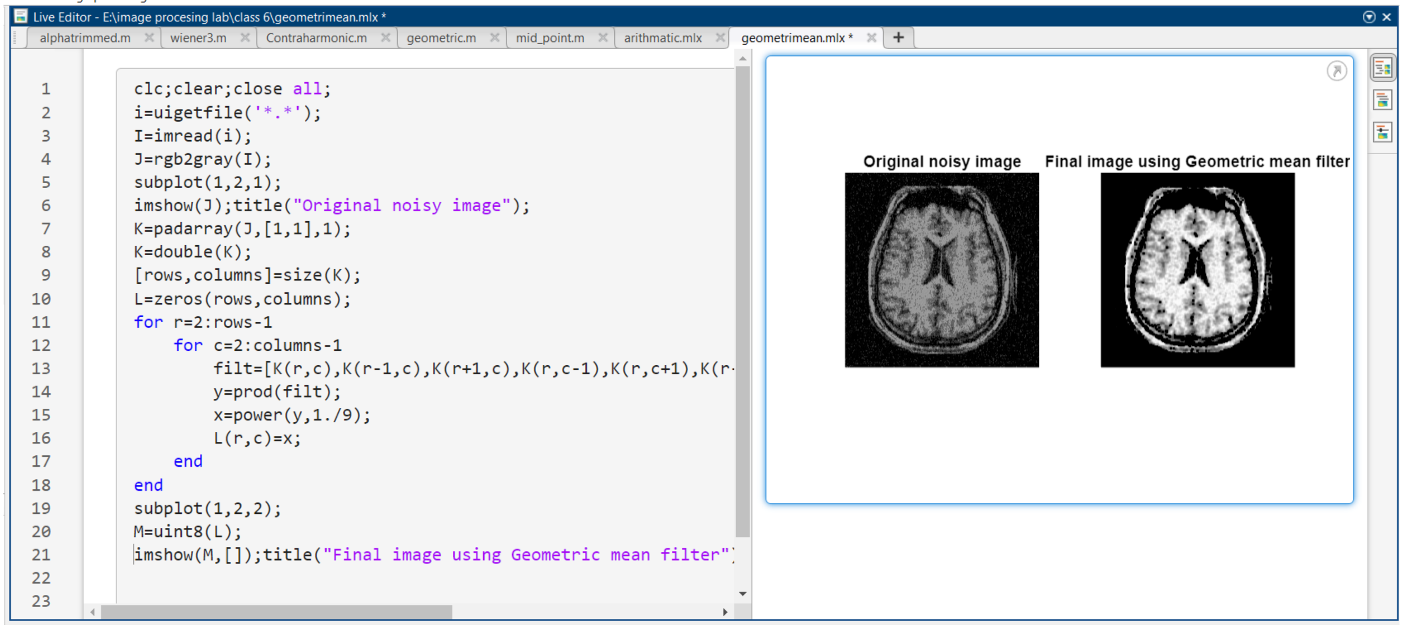


Figure 2: image processing using arithmetic mean filter in MATLAB

CODE

```
% GEOMETRIC MEAN FILTER
clc;clear;close all;
i=uigetfile('*.*');
I=imread(i);
J=rgb2gray(I);
subplot(1,2,1);
imshow(J);title("Original noisy image");
K=padarray(J,[1,1],1);
K=double(K);
[rows,columns]=size(K);
L=zeros(rows,columns);
for r=2:rows-1
    for c=2:columns-1
        filt=[K(r,c),K(r-1,c),K(r+1,c),K(r,c-1),K(r,c+1),K(r-1,c-
1),K(r-1,c+1),K(r+1,c-1),K(r+1,c+1)];
        y=prod(filt);
        x=power(y,1./9);
        L(r,c)=x;
    end
end
subplot(1,2,2);
M=uint8(L);
imshow(M,[]);title("Final image using Geometric mean filter");
```



Live Editor - E:\image procesing lab\class 6\geometrimean.mlx *

alphatrimmed.m | wiener3.m | Contraharmonic.m | geometric.m | mid_point.m | arithmatic.mlx | geometrimean.mlx *

```
clc;clear;close all;
i=uigetfile('*.*');
I=imread(i);
J=rgb2gray(I);
subplot(1,2,1);
imshow(J);title("Original noisy image");
K=padarray(J,[1,1],1);
K=double(K);
[rows,columns]=size(K);
L=zeros(rows,columns);
for r=2:rows-1
    for c=2:columns-1
        filt=[K(r,c),K(r-1,c),K(r+1,c),K(r,c-1),K(r,c+1),K(r
        y=prod(filt);
        x=power(y,1./9);
        L(r,c)=x;
    end
end
subplot(1,2,2);
M=uint8(L);
imshow(M,[]);title("Final image using Geometric mean filter")
```

Figure 3: use of geometric mean filter to process image in MATLAB

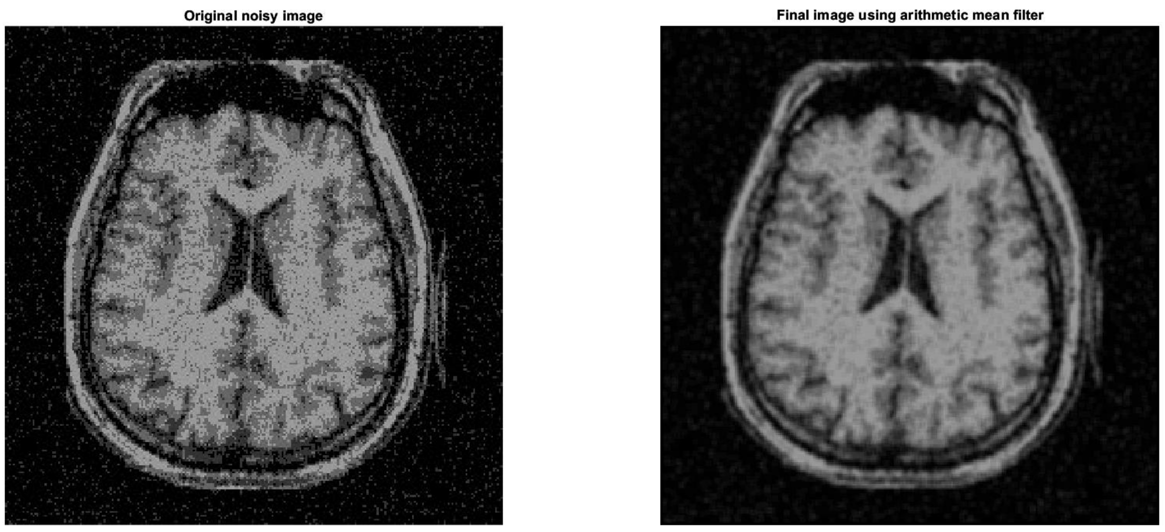
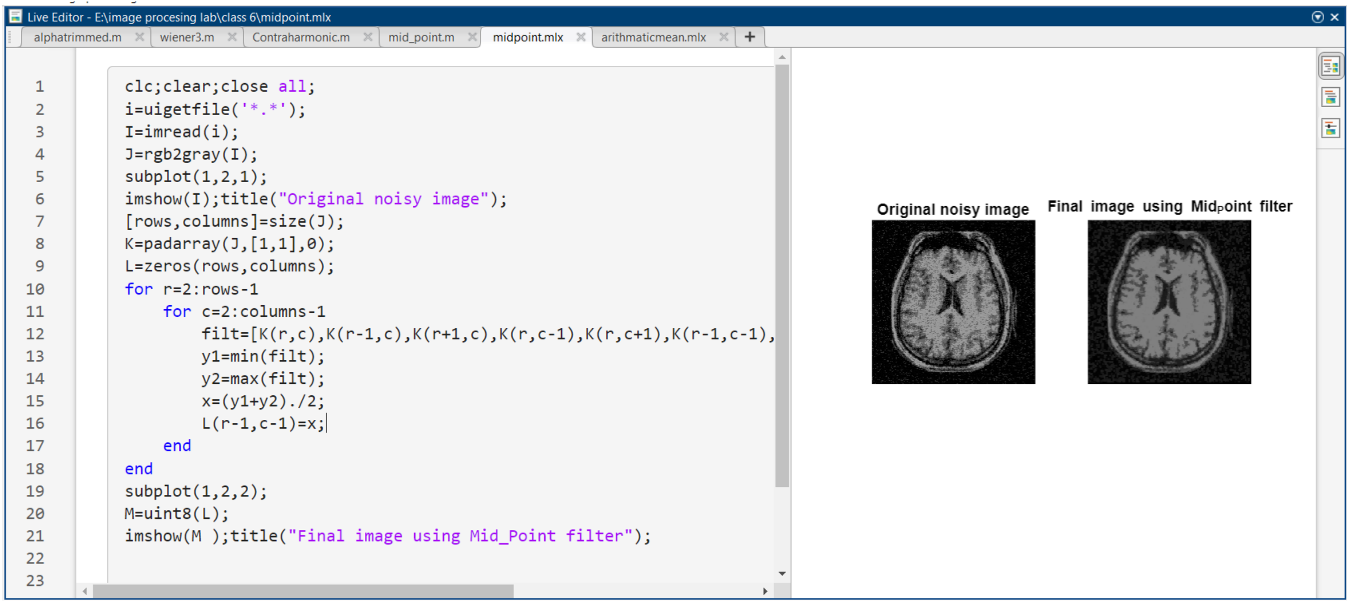


Figure 4: image processing using geometric mean filter in MATLAB

CODE

```
% MID POINT FILTER
clc;clear;close all;
i=uigetfile('*.*');
I=imread(i);
J=rgb2gray(I);
subplot(1,2,1);
imshow(I);title("Original noisy image");
```

```
[rows,columns]=size(J);
K=padarray(J,[1,1],0);
L=zeros(rows,columns);
for r=2:rows-1
    for c=2:columns-1
        filt=[K(r,c),K(r-1,c),K(r+1,c),K(r,c-1),K(r,c+1),K(r-1,c-
1),K(r-1,c+1),K(r+1,c-1),K(r+1,c+1)];
        y1=min(filt); y2=max(filt); x=(y1+y2)./2; L(r-1,c-1)=x;
    end
end
subplot(1,2,2);
M=uint8(L);
imshow(M );title("Final image using Mid_Point filter");
```



Live Editor - E:\image procesing lab\class 6\midpoint.mlx

alphatrimmed.m | wiener3.m | Contraharmonic.m | mid_point.m | midpoint.mlx | arithmaticmean.mlx

```
clc;clear;close all;
i=uigetfile('*.*');
I=imread(i);
J=rgb2gray(I);
subplot(1,2,1);
imshow(I);title("Original noisy image");
[rows,columns]=size(J);
K=padarray(J,[1,1],0);
L=zeros(rows,columns);
for r=2:rows-1
    for c=2:columns-1
        filt=[K(r,c),K(r-1,c),K(r+1,c),K(r,c-1),K(r,c+1),K(r-1,c-1),
        y1=min(filt);
        y2=max(filt);
        x=(y1+y2)./2;
        L(r-1,c-1)=x;
    end
end
subplot(1,2,2);
M=uint8(L);
imshow(M );title("Final image using Mid_Point filter");
```

Figure 5: use of mid point filter to process image in MATLAB

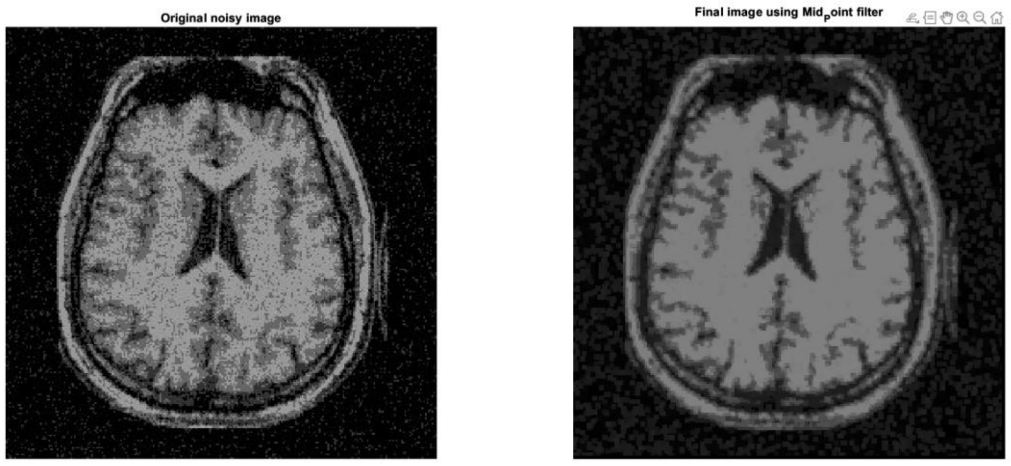
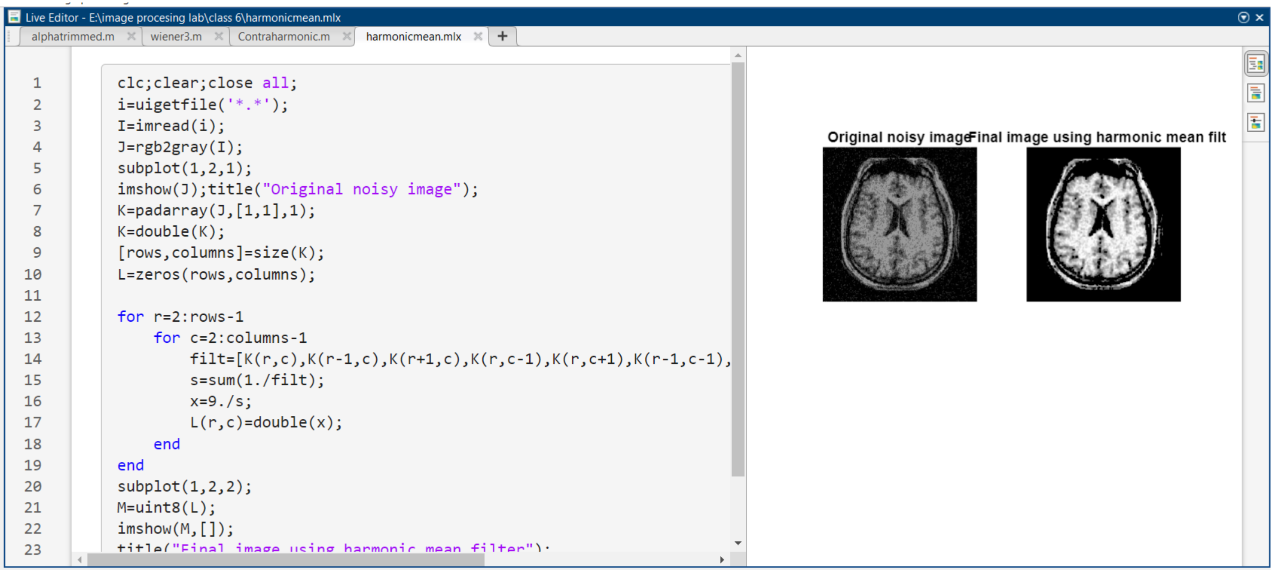


Figure 6: image processing using mid point filter in MATLAB

CODE

```
% HARMONIC MEAN FILTER
clc;clear;close all;
i=uigetfile('*.*');
I=imread(i);
J=rgb2gray(I);
subplot(1,2,1);
imshow(J);title("Original noisy image");
K=padarray(J,[1,1],1);
K=double(K);
[rows,columns]=size(K);
L=zeros(rows,columns);

for r=2:rows-1  for c=2:columns-1
        filt=[K(r,c),K(r-1,c),K(r+1,c),K(r,c-1),K(r,c+1),K(r-1,c-
1),K(r-1,c+1),K(r+1,c-1),K(r+1,c+1)];
        s=sum(1./filt);
        x=9./s;
        L(r,c)=double(x); end end
subplot(1,2,2);
M=uint8(L);
imshow(M,[]);
title("Final image using harmonic mean filter");
```


Live Editor - E:\image procesing lab\class 6\harmonicmean.mlx
alphatrimmed.m
wiener3.m
Contraharmonic.m
harmonicmean.mlx
Original noisy image
Final image using harmonic mean filt

```
clc;clear;close all;
i=uigetfile('*.*');
I=imread(i);
J=rgb2gray(I);
subplot(1,2,1);
imshow(J);title("Original noisy image");
K=padarray(J,[1,1],1);
K=double(K);
[rows,columns]=size(K);
L=zeros(rows,columns);

for r=2:rows-1
    for c=2:columns-1
        filt=[K(r,c),K(r-1,c),K(r+1,c),K(r,c-1),K(r,c+1),K(r-1,c-1),
        s=sum(1./filt);
        x=9./s;
        L(r,c)=double(x);
    end
end
subplot(1,2,2);
M=uint8(L);
imshow(M,[]);
title("Final image using harmonic mean filter");
```

Figure 7: use of harmonic mean filter to process image in MATLAB

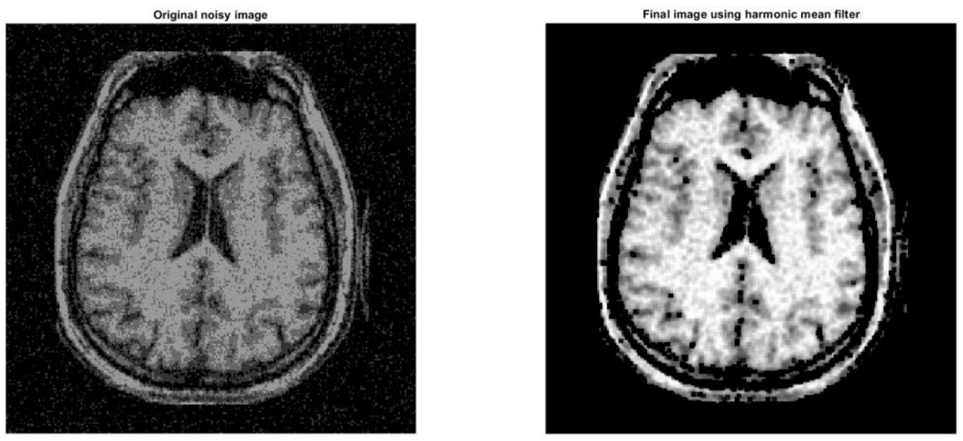
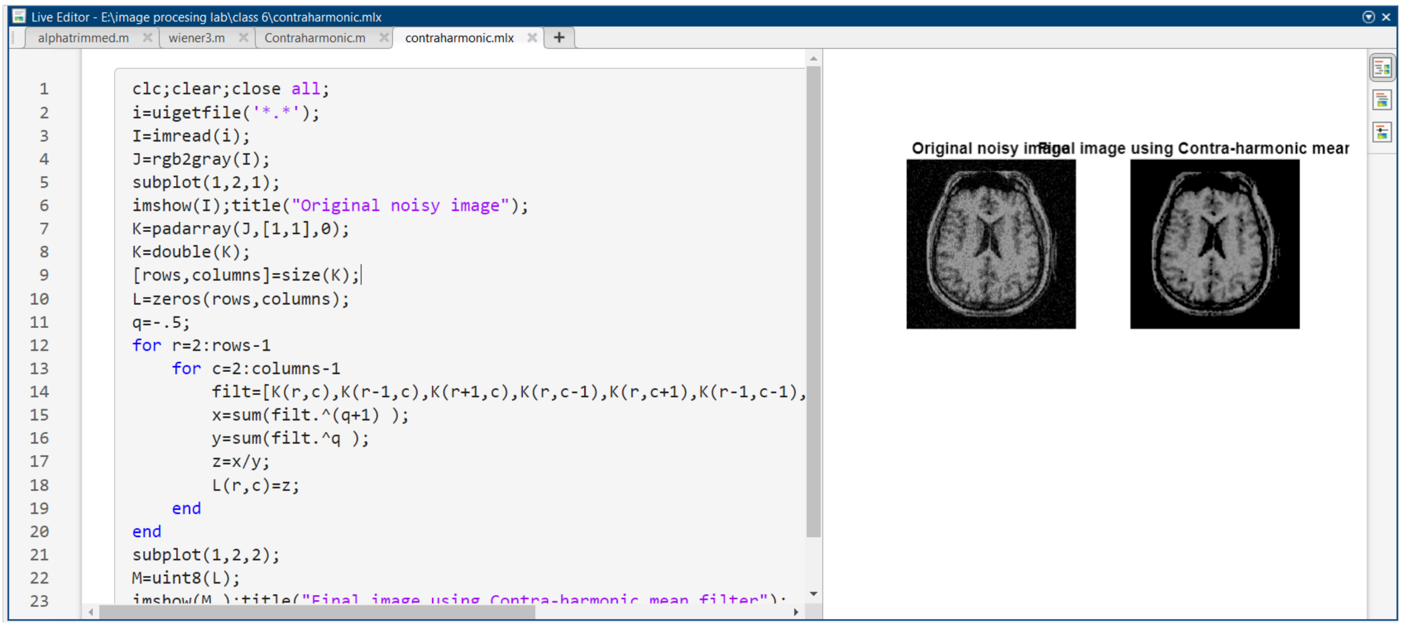


Figure 8: image processing using harmonic mean filter in MATLAB

CODE

```
% CONTRA HARMONIC MEAN FILTER
clc;clear;close all;
i=uigetfile('*.*');
I=imread(i);
J=rgb2gray(I);
subplot(1,2,1);
imshow(I);title("Original noisy image");
K=padarray(J,[1,1],0);
K=double(K);
[rows,columns]=size(K);
L=zeros(rows,columns);
q=-.5;
for r=2:rows-1
    for c=2:columns-1
        filt=[K(r,c),K(r-1,c),K(r+1,c),K(r,c-1),K(r,c+1),K(r-1,c-
1),K(r-1,c+1),K(r+1,c-1),K(r+1,c+1)];
        x=sum(filt.^(q+1) );
        y=sum(filt.^q );
        z=x/y;
        L(r,c)=z;
    end
end
subplot(1,2,2);
M=uint8(L);
imshow(M );title("Final image using Contra-harmonic mean filter");
```



Live Editor - E:\image procesing lab\class 6\contraharmonic.mlx

```
clc;clear;close all;
i=uigetfile('*.*');
I=imread(i);
J=rgb2gray(I);
subplot(1,2,1);
imshow(I);title("Original noisy image");
K=padarray(J,[1,1],0);
K=double(K);
[rows,columns]=size(K);
L=zeros(rows,columns);
q=-.5;
for r=2:rows-1
    for c=2:columns-1
        filt=[K(r,c),K(r-1,c),K(r+1,c),K(r,c-1),K(r,c+1),K(r-1,c-1),
        x=sum(filt.^(q+1) );
        y=sum(filt.^q );
        z=x/y;
        L(r,c)=z;
    end
end
subplot(1,2,2);
M=uint8(L);
imshow(M );title("Final image using Contra-harmonic mean filter");
```

Figure 9: use of contra harmonic mean filter to process image in MATLAB

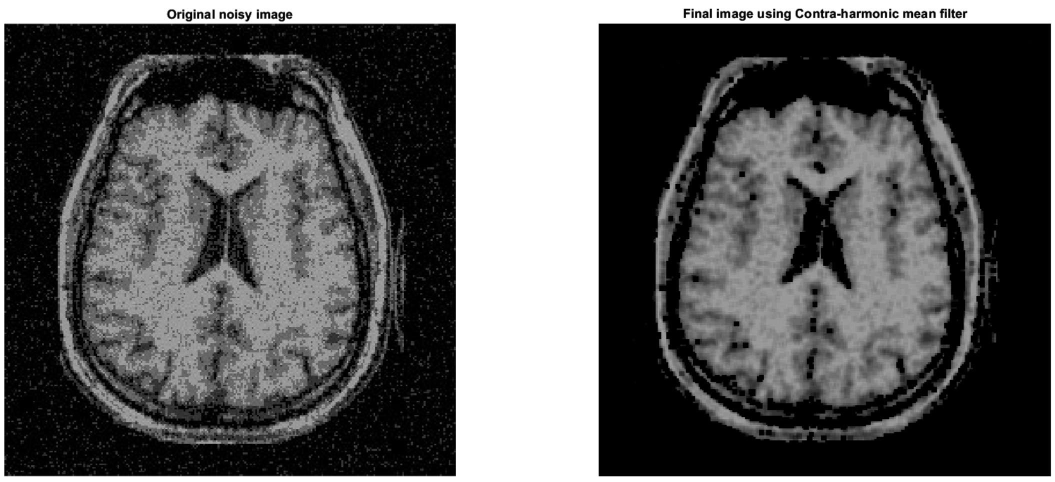
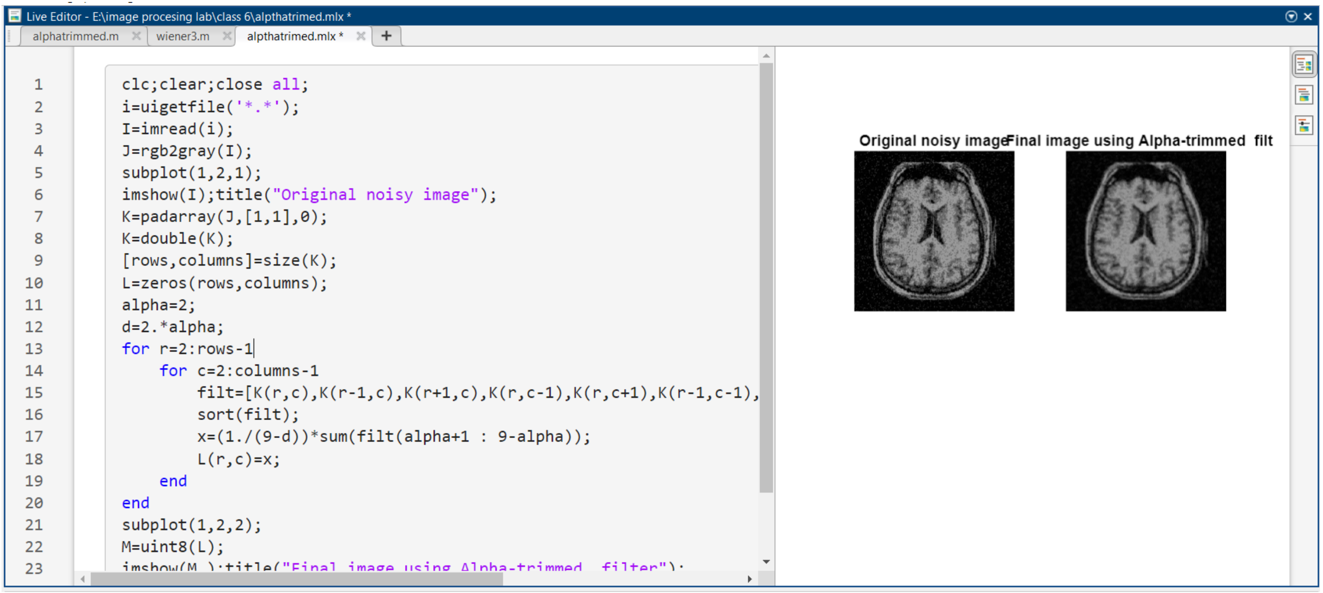


Figure 10: image processing using contra harmonic mean filter in MATLAB

CODE

```
% ALPHA TRIMED FILTER
clc;clear;close all;
i=uigetfile('*.*');
I=imread(i);
J=rgb2gray(I);
subplot(1,2,1);
imshow(I);title("Original noisy image");
K=padarray(J,[1,1],0);
```

```
K=double(K);
[rows,columns]=size(K);
L=zeros(rows,columns);
alpha=2;
d=2.*alpha;
for r=2:rows-1
    for c=2:columns-1
        filt=[K(r,c),K(r-1,c),K(r+1,c),K(r,c-1),K(r,c+1),K(r-1,c-
1),K(r-1,c+1),K(r+1,c-1),K(r+1,c+1)];
        sort(filt);
        x=(1./(9-d))*sum(filt(alpha+1 : 9-alpha));
        L(r,c)=x;
    end
end
subplot(1,2,2);
M=uint8(L);
imshow(M );title("Final image using Alpha-trimmed  filter");
```


Live Editor - E:\image procesing lab\class 6\alpthatrimed.mlx *
alphatrimmed.m
wiener3.m
alpthatrimed.mlx *
Original noisy image
Final image using Alpha-trimmed filt

```
clc;clear;close all;
i=uigetfile('*.*');
I=imread(i);
J=rgb2gray(I);
subplot(1,2,1);
imshow(I);title("Original noisy image");
K=padarray(J,[1,1],0);
K=double(K);
[rows,columns]=size(K);
L=zeros(rows,columns);
alpha=2;
d=2.*alpha;
for r=2:rows-1
    for c=2:columns-1
        filt=[K(r,c),K(r-1,c),K(r+1,c),K(r,c-1),K(r,c+1),K(r-1,c-1),
        sort(filt);
        x=(1./(9-d))*sum(filt(alpha+1 : 9-alpha));
        L(r,c)=x;
    end
end
subplot(1,2,2);
M=uint8(L);
imshow(M );title("Final image using Alpha-trimmed  filter");
```

Figure 11: use of alpha trimmed filter to process image in MATLAB

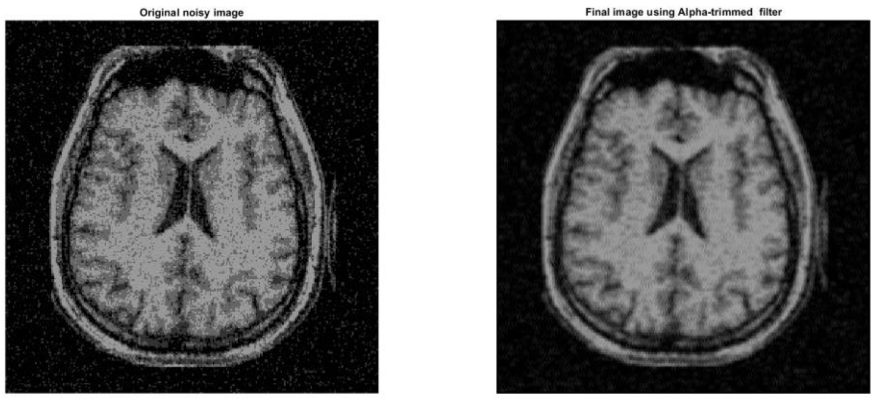


Figure 12: image processing alpha trimmed filter *in MATLAB*

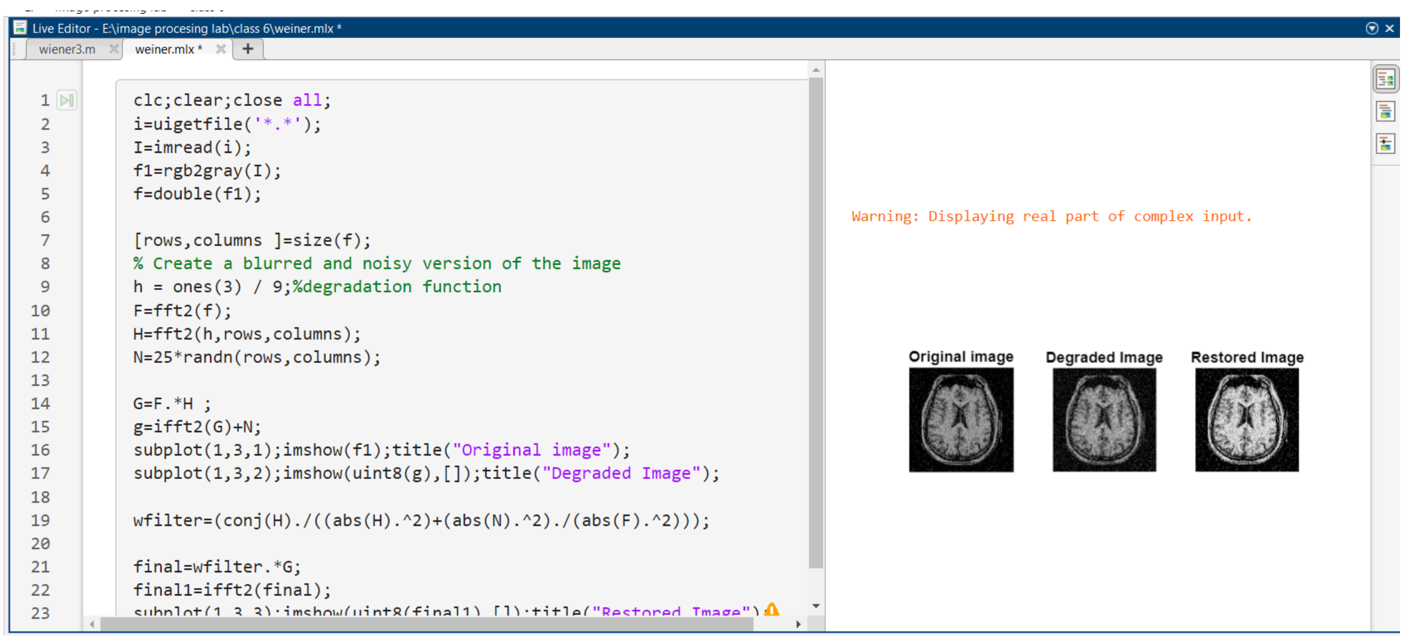
WEINER INVEERSE FILTER

The Weiner inverse filter is a signal processing technique used to restore an image or signal that has been degraded by a known linear time-invariant (LTI) system. The filter is named after Norbert Weiner, who first proposed it in the 1940s. The Weiner inverse filter is designed to remove the effects of degradation caused by an LTI system, such as blurring, distortion, or noise, by filtering the degraded signal. The filter works by estimating the original signal's frequency spectrum and then applying a weighted inverse filter to the degraded signal. The filter's design involves two main steps: estimation of the power spectral density (PSD) of the original signal and estimation of the PSD of the degradation process. The PSD estimates are then used to derive a filter that minimizes the mean square error between the original and filtered signals. The Weiner inverse filter is an optimal linear filter that takes into account the statistical properties of the original signal and the noise present in the degraded signal. It is a powerful tool for signal restoration but can be sensitive to errors in the PSD estimates, which can lead to instability and noise amplification. The Weiner inverse filter is widely used in image processing, audio signal processing, and other fields where signal restoration is necessary. However, it is important to note that the filter is only effective when the degradation process is known and can be modelled accurately.

CODE

```
% WEINER INVEERSE FILTER
clc;clear;close all;
i=uigetfile('*.*');
I=imread(i);
f1=rgb2gray(I);
f=double(f1);

[rows,columns ]=size(f);
% Create a blurred and noisy version of the image
```

```
h = ones(3) / 9;%degradation function
F=fft2(f);
H=fft2(h,rows,columns);
N=25*randn(rows,columns);

G=F.*H ;
g=ifft2(G)+N;
subplot(1,3,1);imshow(f1);title("Original image");
subplot(1,3,2);imshow(uint8(g),[]);title("Degraded Image");

wfilter=(conj(H)./((abs(H).^2)+(abs(N).^2)./(abs(F).^2)));

final=wfilter.*G;
final1=ifft2(final);
subplot(1,3,3);imshow(uint8(final1),[]);title("Restored Image");
```



Live Editor - E:\image procesing lab\class 6\weiner.mlx *

```
clc;clear;close all;
i=uigetfile('*.*');
I=imread(i);
f1=rgb2gray(I);
f=double(f1);

[rows,columns ]=size(f);
% Create a blurred and noisy version of the image
h = ones(3) / 9;%degradation function
F=fft2(f);
H=fft2(h,rows,columns);
N=25*randn(rows,columns);

G=F.*H ;
g=ifft2(G)+N;
subplot(1,3,1);imshow(f1);title("Original image");
subplot(1,3,2);imshow(uint8(g),[]);title("Degraded Image");

wfilter=(conj(H)./((abs(H).^2)+(abs(N).^2)./(abs(F).^2)));

final=wfilter.*G;
final1=ifft2(final);
subplot(1,3,3);imshow(uint8(final1),[]);title("Restored Image")
```

```
Warning: Displaying real part of complex input.
```

Figure 13: use of weiner inverse filter to process image in MATLAB

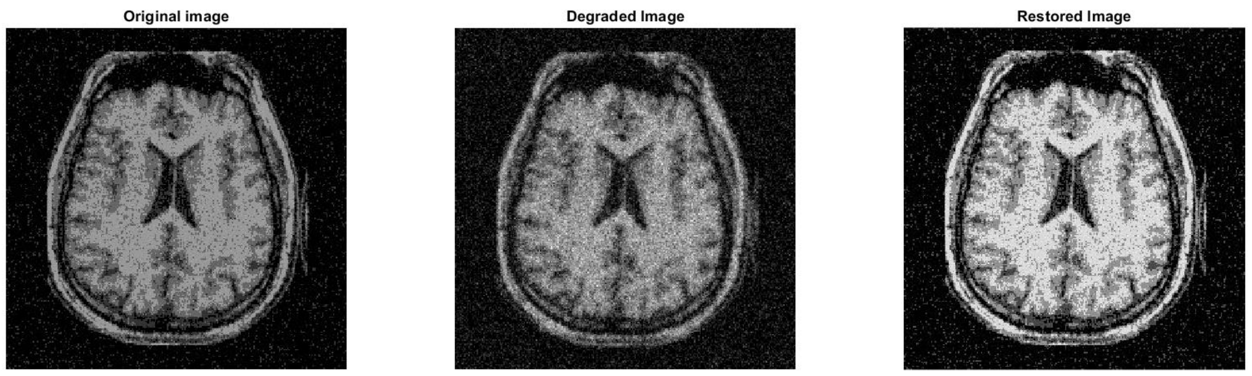


Figure 14: image processing weiner inverse filter in MATLAB

Shreenandan Sahu |120BM0806

Lab Report 7

Aim

Image segmentation in MATLAB.

Theory

Image registration is a technique used to align two or more images of the same scene taken at different times, viewpoints or by different sensors. In MATLAB, there are several ways to perform image registration. Here is a general overview of the steps involved in image registration using MATLAB: Load the images: The first step is to load the images that you want to register using the imread function in MATLAB. Pre-processing: Depending on the quality and size of the images, pre-processing may be necessary. Common pre-processing techniques include filtering, resizing, and normalization. Feature detection: The next step is to detect features in the images that can be used for registration. MATLAB has built-in functions like detect SURF Features and detect BRISK Features that can be used for feature detection. Feature extraction: After detecting features, the next step is to extract descriptors that can be used to match the features in the two images. MATLAB provides functions like extract Features to extract descriptors. Feature matching: The extracted features and descriptors are then matched using functions like match Features in MATLAB. This step aims to find the corresponding points between the two images. Transformation estimation: Once the corresponding points are found, a transformation model is estimated to align the two images. MATLAB has functions like estimate Geometric Transform and fitgeotrans for transformation estimation. Image registration: Finally, the registered images are obtained by applying the estimated transformation to one of the images. The imwarp function in MATLAB can be used for image warping.

Image registration:

• Overlapping 2 images so that we can visualize the differences between the 2 images.

• Applications - What are the effects of a drug that can be visualized by overlapping 2 images.

1)Different transforms:

a) Affine transformation:

i. Scaling

ii. Rotation

iii. Translation

iv. Changing the transparency of the base image/registered image

2)Basically there are 2 images:

a) Base Image (Untreated/before)

b) Unregistered image (After treatment)

3)How to do scaling:

a) Mark 3 non-collinear points which we feel have not varied much in base and unregistered image.

b) Take the Euclidean distance between any 2 of these points in both images. (base image - d1 , unregistered image - d2)

c) Scale the unregistered image so that it can be overlapped. Scaling Factor =d1/d2)

4)How to do rotation:

• Take three points in each image

• Find the angle between the two lines in both images using theta = tan^(-1) (y2-y2/x2-x1)

• Rotation angle = | theta2 - theta1 |

5) How to do translation:

• Take the same three points in both the base and the unregistered image.

• Here we are going to shift the origin of the unregistered image

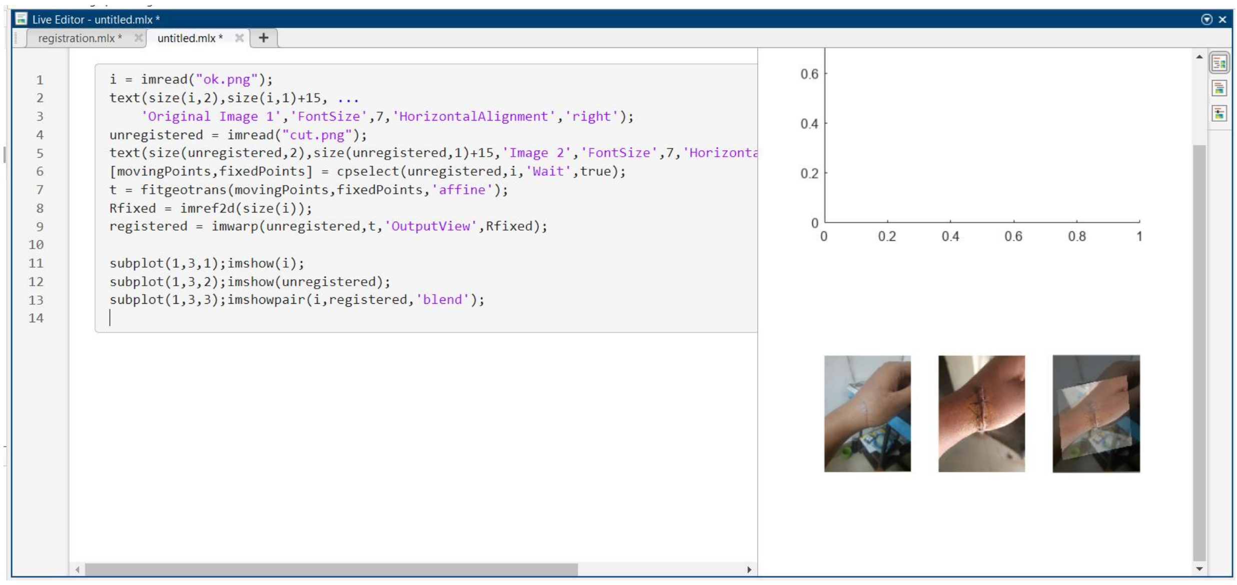
• Take a point on the base image, then take the same point in the unregistered image.

• How these 2 images should have the same spatial location for being overlapped.

• Hence we shift the origin of the unregistered image so that the point in the unregistered image should have the same spatial location as the base image.

CODE

```
% IMAGE REGISTRATION


i = imread("ok.png");
text(size(i,2),size(i,1)+15, ...
    'Original Image 1','FontSize',7,'HorizontalAlignment','right');
unregistered = imread("cut.png");
text(size(unregistered,2),size(unregistered,1)+15,'Image
2','FontSize',7,'HorizontalAlignment','right');
[movingPoints,fixedPoints] = cpselect(unregistered,i,'Wait',true);
t = fitgeotrans(movingPoints,fixedPoints,'affine');
Rfixed = imref2d(size(i));
registered = imwarp(unregistered,t,'OutputView',Rfixed);


subplot(1,3,1);imshow(i);
subplot(1,3,2);imshow(unregistered);
subplot(1,3,3);imshowpair(i,registered,'blend');
```



```
i = imread("ok.png");
text(size(i,2),size(i,1)+15, ...
    'Original Image 1','FontSize',7,'HorizontalAlignment','right');
unregistered = imread("cut.png");
text(size(unregistered,2),size(unregistered,1)+15,'Image 2','FontSize',7,'Horizonta
[movingPoints,fixedPoints] = cpselect(unregistered,i,'Wait',true);
t = fitgeotrans(movingPoints,fixedPoints,'affine');
Rfixed = imref2d(size(i));
registered = imwarp(unregistered,t,'OutputView',Rfixed);

subplot(1,3,1);imshow(i);
subplot(1,3,2);imshow(unregistered);
subplot(1,3,3);imshowpair(i,registered,'blend');
```

Figure 1: image registration in MATLAB

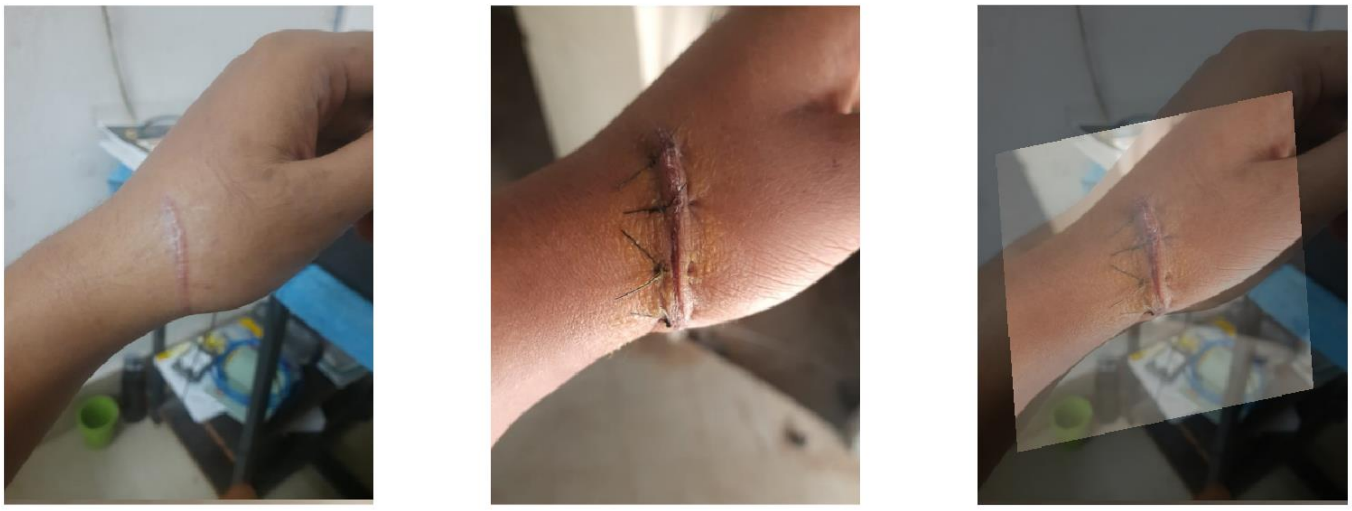


Figure 2: image registration of injured and healed hand in MATLAB

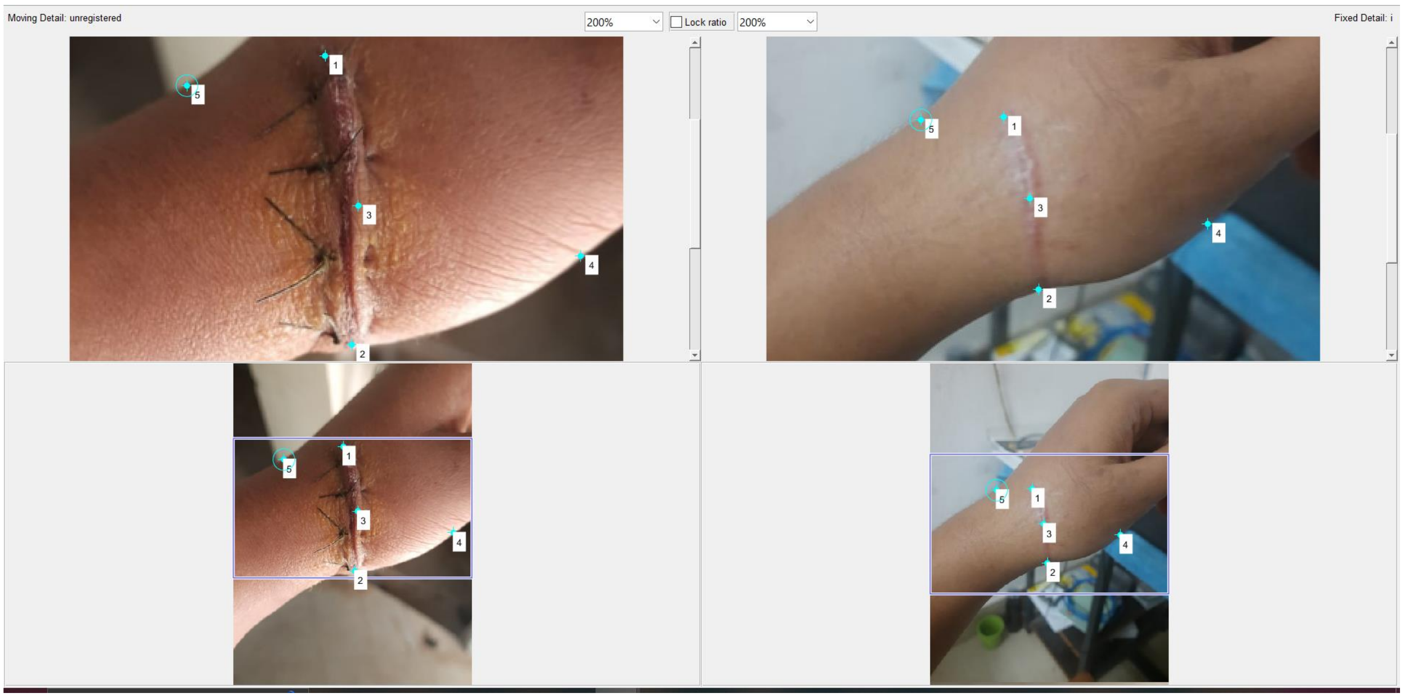


Figure 3: point selection for image registration of injured and healed hand in MATLAB

Shreenandan Sahu |120BM0806

Lab Report 8

Aim

Image segmentation using point segmentation, line segmentation and edge detection.

Theory

Image segmentation is the process of dividing an image into multiple segments or regions, each of which corresponds to a different object or part of the image. Point segmentation, line segmentation, and edge detection are all techniques that can be used to perform image segmentation.

Point segmentation involves identifying individual points or pixels within an image that belong to a particular segment or object. This can be achieved using techniques such as clustering or thresholding, where pixels with similar characteristics are grouped together.

Line segmentation involves identifying linear features within an image, such as edges or contours, and using these features to separate the image into different segments. This can be achieved using techniques such as the Hough transform, which can detect lines or curves within an image.

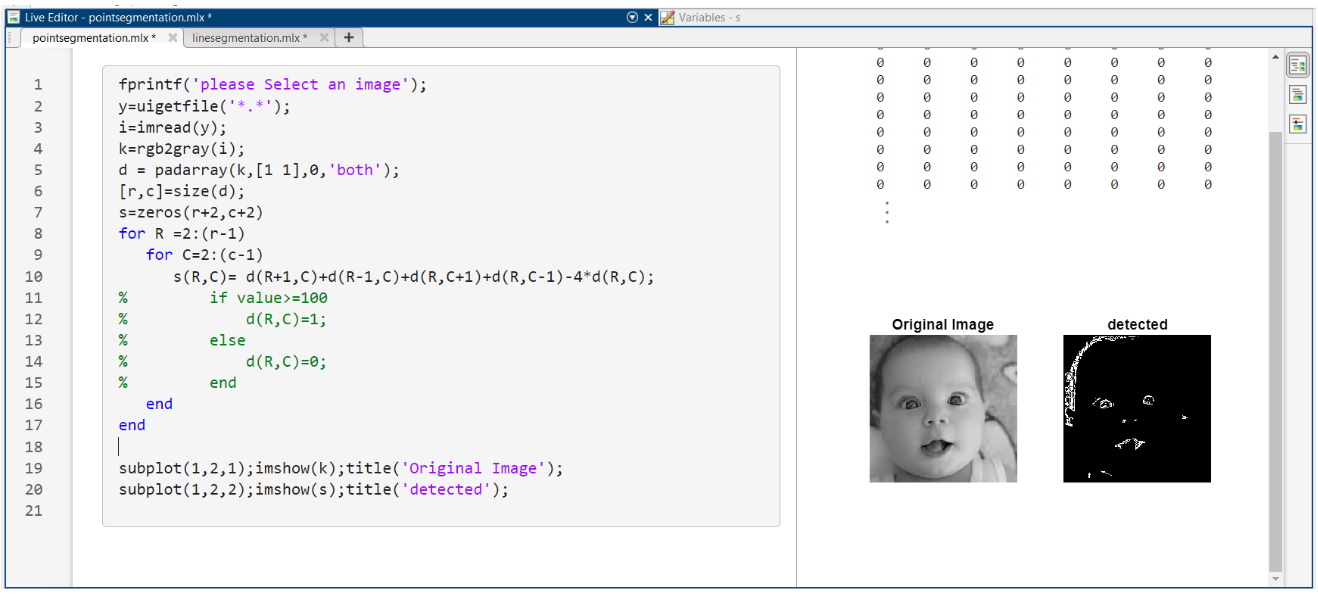
Edge detection involves identifying abrupt changes in brightness or color within an image, which can be used to locate the boundaries between different objects or segments. This can be achieved using techniques such as the Sobel operator, which highlights areas of the image with high spatial gradients.

All three of these techniques can be used in combination to perform image segmentation. For example, edge detection can be used to identify the boundaries between different segments, while point segmentation and line segmentation can be used to group pixels or linear features within each segment. The choice of technique will depend on the specific requirements of the image segmentation task, as well as the characteristics of the image being segmented.

CODE

```
% POINT SEGMENTATION

fprintf('please Select an image');
y=uigetfile('*.*');
i=imread(y);
k=rgb2gray(i);
d = padarray(k,[1 1],0,'both');
[r,c]=size(d);
s=zeros(r+2,c+2)
```

```
for R =2:(r-1)
   for C=2:(c-1)
      s(R,C)= d(R+1,C)+d(R-1,C)+d(R,C+1)+d(R,C-1)-4*d(R,C);
%          if value>=100
%             d(R,C)=1;
%          else
%             d(R,C)=0;
%          end
   end
end

subplot(1,2,1);imshow(k);title('Original Image');
subplot(1,2,2);imshow(s);title('detected');
```



```
fprintf('please Select an image');
y=uigetfile('*.*');
i=imread(y);
k=rgb2gray(i);
d = padarray(k,[1 1],0,'both');
[r,c]=size(d);
s=zeros(r+2,c+2)
for R =2:(r-1)
   for C=2:(c-1)
      s(R,C)= d(R+1,C)+d(R-1,C)+d(R,C+1)+d(R,C-1)-4*d(R,C);
%        if value>=100
%           d(R,C)=1;
%        else
%           d(R,C)=0;
%        end
   end
end

subplot(1,2,1);imshow(k);title('Original Image');
subplot(1,2,2);imshow(s);title('detected');
```

```
0    0    0    0    0    0    0    0
0    0    0    0    0    0    0    0
0    0    0    0    0    0    0    0
0    0    0    0    0    0    0    0
0    0    0    0    0    0    0    0
0    0    0    0    0    0    0    0
0    0    0    0    0    0    0    0
0    0    0    0    0    0    0    0
⋮
```

Figure 1: use of point segmentation to detect points in image in MATLAB

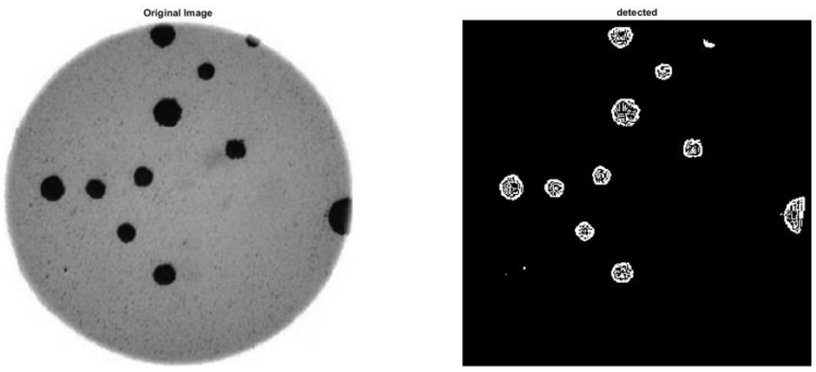
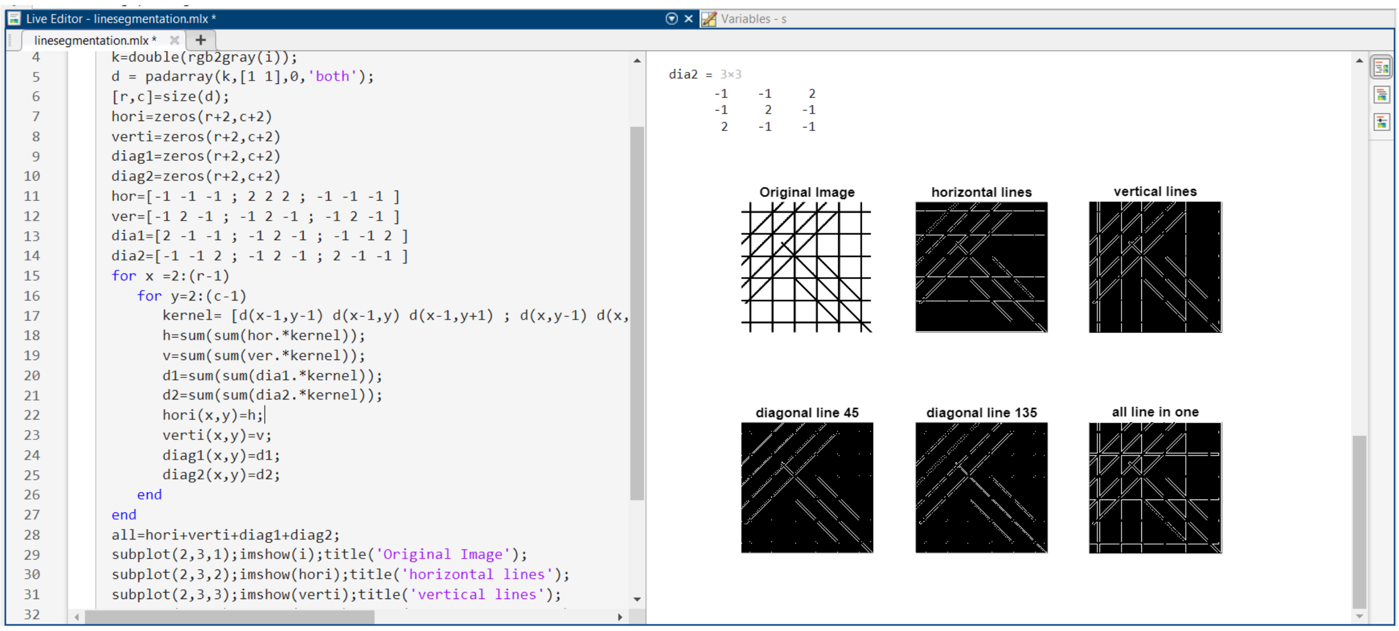


Figure 2: point detection using point segmentation in MATLAB

CODE

```
% LINE SEGMENTATION
fprintf('please Select an image');
y=uigetfile('*.*');
i=imread(y);
k=double(rgb2gray(i));
d = padarray(k,[1 1],0,'both');
[r,c]=size(d);
hori=zeros(r+2,c+2)
verti=zeros(r+2,c+2)
diag1=zeros(r+2,c+2)
diag2=zeros(r+2,c+2)
hor=[-1 -1 -1 ; 2 2 2 ; -1 -1 -1 ] % horizontal filter
ver=[-1 2 -1 ; -1 2 -1 ; -1 2 -1 ] % Vertical filter
dia1=[2 -1 -1 ; -1 2 -1 ; -1 -1 2 ] % diagonal 45 degree filter
dia2=[-1 -1 2 ; -1 2 -1 ; 2 -1 -1 ] % diagonal 135 degree filter
for x =2:(r-1)
   for y=2:(c-1)
      kernel= [d(x-1,y-1) d(x-1,y) d(x-1,y+1) ; d(x,y-1) d(x,y) 
d(x,y+1) ; d(x+1,y-1) d(x+1,y) d(x+1,y+1)];
      h=sum(sum(hor.*kernel));
      v=sum(sum(ver.*kernel));
      d1=sum(sum(dia1.*kernel));
      d2=sum(sum(dia2.*kernel));
      hori(x,y)=h;
      verti(x,y)=v;
      diag1(x,y)=d1;
      diag2(x,y)=d2;
   end
end
all=hori+verti+diag1+diag2;
subplot(2,3,1);imshow(i);title('Original Image');
subplot(2,3,2);imshow(hori);title('horizontal lines');
subplot(2,3,3);imshow(verti);title('vertical lines');
subplot(2,3,4);imshow(diag1);title('diagonal line 45');
subplot(2,3,5);imshow(diag2);title('diagonal line 135');
subplot(2,3,6);imshow(all);title('all line in one');
```



```matlab
    k=double(rgb2gray(i));
    d = padarray(k,[1 1],0,'both');
    [r,c]=size(d);
    hori=zeros(r+2,c+2)
    verti=zeros(r+2,c+2)
    diag1=zeros(r+2,c+2)
    diag2=zeros(r+2,c+2)
    hor=[-1 -1 -1 ; 2 2 2 ; -1 -1 -1 ]
    ver=[-1 2 -1 ; -1 2 -1 ; -1 2 -1 ]
    dia1=[2 -1 -1 ; -1 2 -1 ; -1 -1 2 ]
    dia2=[-1 -1 2 ; -1 2 -1 ; 2 -1 -1 ]
    for x =2:(r-1)
       for y=2:(c-1)
          kernel= [d(x-1,y-1) d(x-1,y) d(x-1,y+1) ; d(x,y-1) d(x,
          h=sum(sum(hor.*kernel));
          v=sum(sum(ver.*kernel));
          d1=sum(sum(dia1.*kernel));
          d2=sum(sum(dia2.*kernel));
          hori(x,y)=h;
          verti(x,y)=v;
          diag1(x,y)=d1;
          diag2(x,y)=d2;
       end
    end
    all=hori+verti+diag1+diag2;
    subplot(2,3,1);imshow(i);title('Original Image');
    subplot(2,3,2);imshow(hori);title('horizontal lines');
    subplot(2,3,3);imshow(verti);title('vertical lines');
```

```
dia2 = 3×3
      -1    -1     2
      -1     2    -1
       2    -1    -1
```

Figure 3: use of line segmentation to detect lines in MATLAB

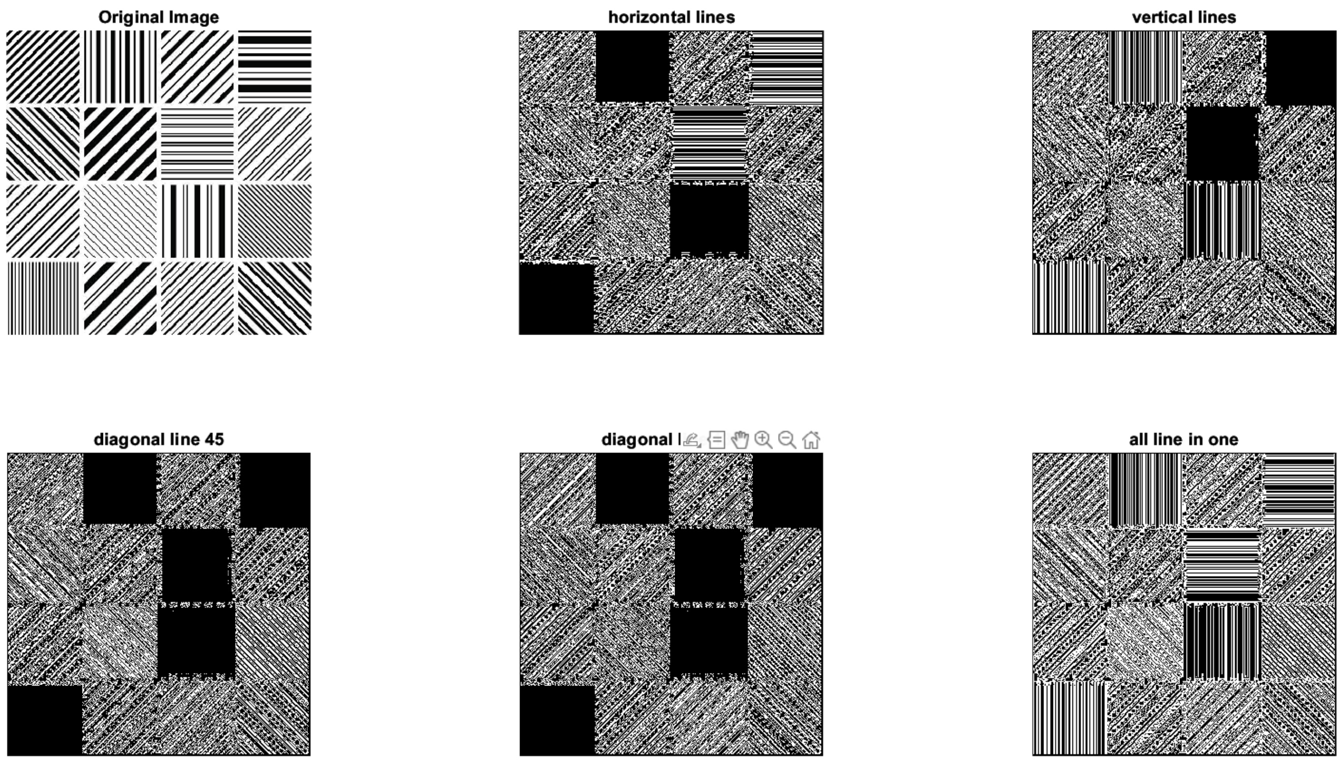


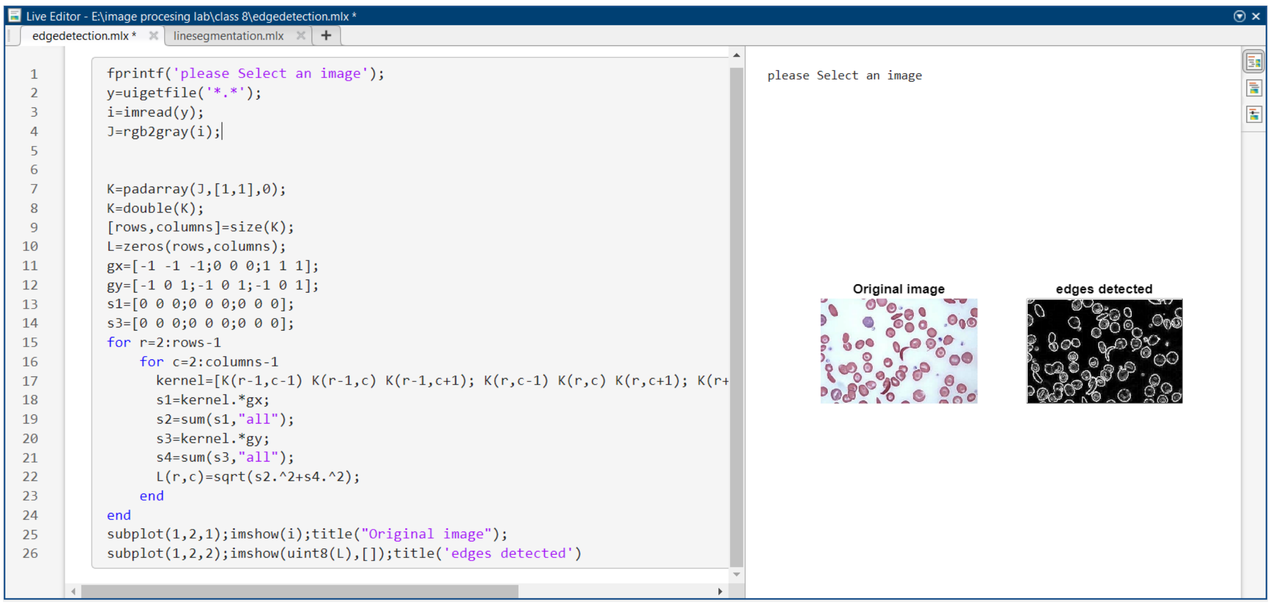
Figure 4: detection of various types of lines in MATLAB

EDGE DETECTION

Edge detection is a computer vision technique that involves identifying the boundaries of objects within an image. It is a crucial step in many image processing applications, such as object recognition, image segmentation, and feature extraction. Edge detection algorithms typically work by analyzing the variations in brightness, colour, or texture across an image. The most commonly used edge detection algorithms include the Sobel, Prewitt, and Canny edge detection algorithms. The Sobel and Prewitt algorithms use convolution filters to identify edges based on the intensity gradients in the image. The Canny algorithm, on the other hand, uses a more sophisticated approach that involves detecting edges at multiple scales and suppressing false positives. Once the edges have been detected, they can be further processed to extract useful information, such as the shape and size of objects in the image. This information can then be used for a wide range of applications, such as object tracking, face recognition, and autonomous driving.

CODE

```
% EDGE DETECTION
fprintf('please Select an image');
y=uigetfile('*.*');
i=imread(y);
J=rgb2gray(i);



K=padarray(J,[1,1],0);
K=double(K);
[rows,columns]=size(K);
L=zeros(rows,columns);
gx=[-1 -1 -1;0 0 0;1 1 1];
gy=[-1 0 1;-1 0 1;-1 0 1];
s1=[0 0 0;0 0 0;0 0 0];
s3=[0 0 0;0 0 0;0 0 0];
for r=2:rows-1
    for c=2:columns-1
      kernel=[K(r-1,c-1) K(r-1,c) K(r-1,c+1); K(r,c-1) K(r,c) 
K(r,c+1); K(r+1,c-1) K(r+1,c) K(r+1,c+1)];
      s1=kernel.*gx;
      s2=sum(s1,"all");
      s3=kernel.*gy;
      s4=sum(s3,"all");
      L(r,c)=sqrt(s2.^2+s4.^2);
    end
end
subplot(1,2,1);imshow(i);title("Original image");
subplot(1,2,2);imshow(uint8(L),[]);title('edges detected');
```


Live Editor - E:\image procesing lab\class 8\edgedetection.mlx *
edgedetection.mlx *
linesegmentation.mlx

```
fprintf('please Select an image');
y=uigetfile('*.*');
i=imread(y);
J=rgb2gray(i);


K=padarray(J,[1,1],0);
K=double(K);
[rows,columns]=size(K);
L=zeros(rows,columns);
gx=[-1 -1 -1;0 0 0;1 1 1];
gy=[-1 0 1;-1 0 1;-1 0 1];
s1=[0 0 0;0 0 0;0 0 0];
s3=[0 0 0;0 0 0;0 0 0];
for r=2:rows-1
    for c=2:columns-1
      kernel=[K(r-1,c-1) K(r-1,c) K(r-1,c+1); K(r,c-1) K(r,c) K(r,c+1); K(r+
      s1=kernel.*gx;
      s2=sum(s1,"all");
      s3=kernel.*gy;
      s4=sum(s3,"all");
      L(r,c)=sqrt(s2.^2+s4.^2);
    end
end
subplot(1,2,1);imshow(i);title("Original image");
subplot(1,2,2);imshow(uint8(L),[]);title('edges detected')
```

```
please Select an image
```

Figure 5: use of edge detection code to detect edges in MATLAB

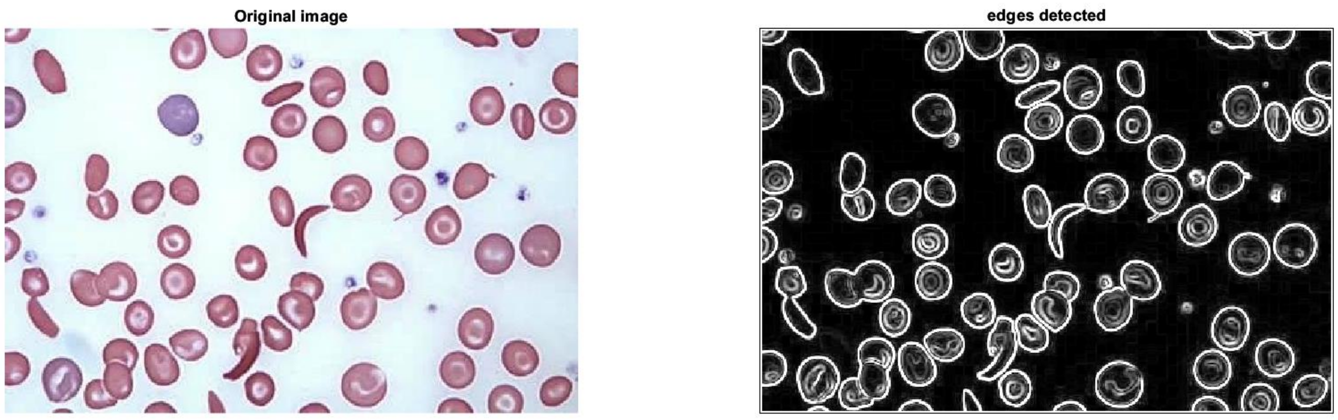


Figure 6 edge detection *in MATLAB*

THRESHOLD BASED SEGMENTATION

Threshold-based segmentation is a commonly used technique in image processing and computer vision for segmenting an image into different regions based on their intensity values. The basic idea behind threshold-based segmentation is to set a threshold value, and then classify each pixel in the image based on whether its intensity value is above or below the threshold.

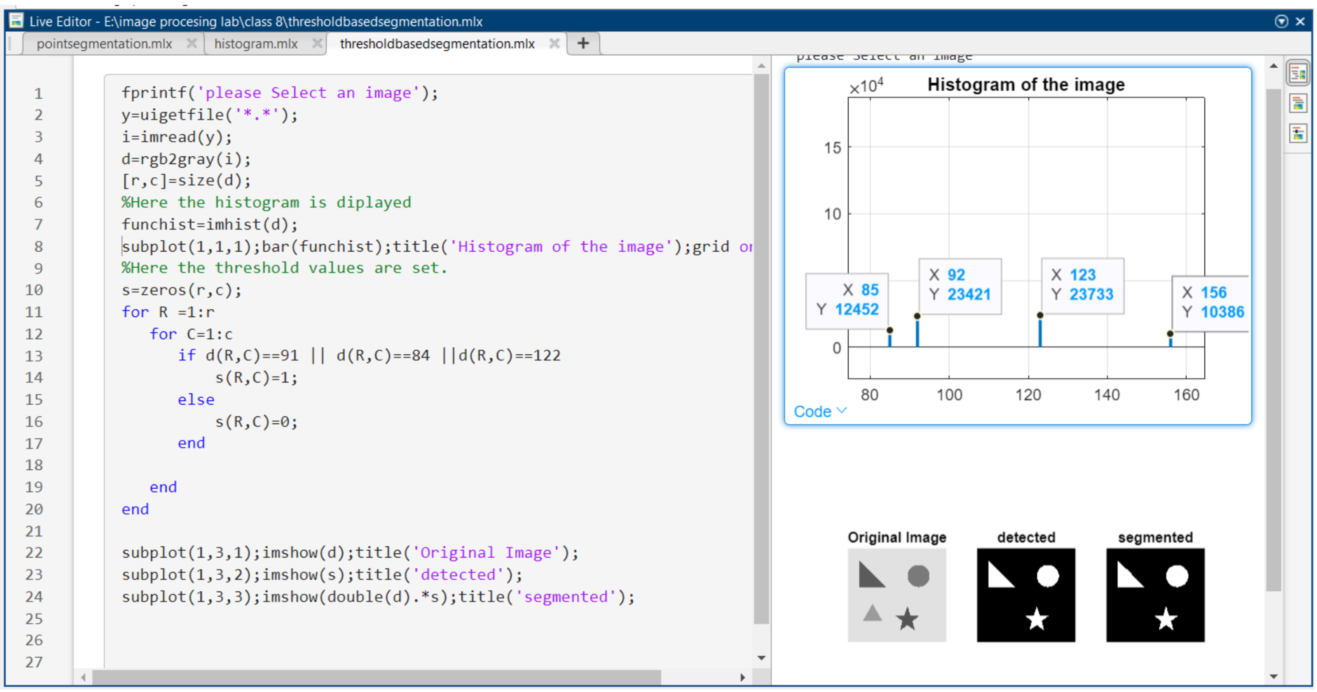
The threshold value can be chosen manually or automatically, depending on the specific application and the characteristics of the image being segmented. If the threshold is chosen manually, it is typically based on some prior knowledge of the image or the desired segmentation result. If the threshold is chosen automatically, there are several methods that can be used, such as Otsu's method, which selects a threshold that minimizes the variance between the two classes of pixels (above and below the threshold).

Once the threshold value is set, each pixel in the image is compared to the threshold, and is classified as either part of the foreground (above the threshold) or part of the background (below the threshold). The result is a binary image, where the foreground pixels are represented as white and the background pixels are represented as black.

Threshold-based segmentation is a simple and computationally efficient technique, but it has some limitations. It works well when there is a clear contrast between the foreground and background, but may fail when there is significant overlap in intensity values between the two classes. In such cases, more advanced techniques such as edge detection and region-growing may be necessary.

CODE

```
% THRESHOLD-BASED SEGMENTATION
fprintf('please Select an image');
y=uigetfile('*.*');
i=imread(y);
d=rgb2gray(i);
[r,c]=size(d);
%Here the histogram is displayed
funchist=imhist(d);
subplot(1,1,1);bar(funchist);title('Histogram of the image');grid on;
%Here the threshold values are set.
s=zeros(r,c);
for R =1:r
   for C=1:c
      if d(R,C)==91 || d(R,C)==84 ||d(R,C)==122
          s(R,C)=1;
      else
          s(R,C)=0;
      end

   end
end

subplot(1,3,1);imshow(d);title('Original Image');
subplot(1,3,2);imshow(s);title('detected');
subplot(1,3,3);imshow(double(d).*s);title('segmented');
```



```matlab
fprintf('please Select an image');
y=uigetfile('*.*');
i=imread(y);
d=rgb2gray(i);
[r,c]=size(d);
%Here the histogram is diplayed
funchist=imhist(d);
subplot(1,1,1);bar(funchist);title('Histogram of the image');grid o
%Here the threshold values are set.
s=zeros(r,c);
for R =1:r
   for C=1:c
      if d(R,C)==91 || d(R,C)==84 ||d(R,C)==122
          s(R,C)=1;
      else
          s(R,C)=0;
      end

   end
end

subplot(1,3,1);imshow(d);title('Original Image');
subplot(1,3,2);imshow(s);title('detected');
subplot(1,3,3);imshow(double(d).*s);title('segmented');
```

Figure 7: use of threshold-based segmentation in MATLAB

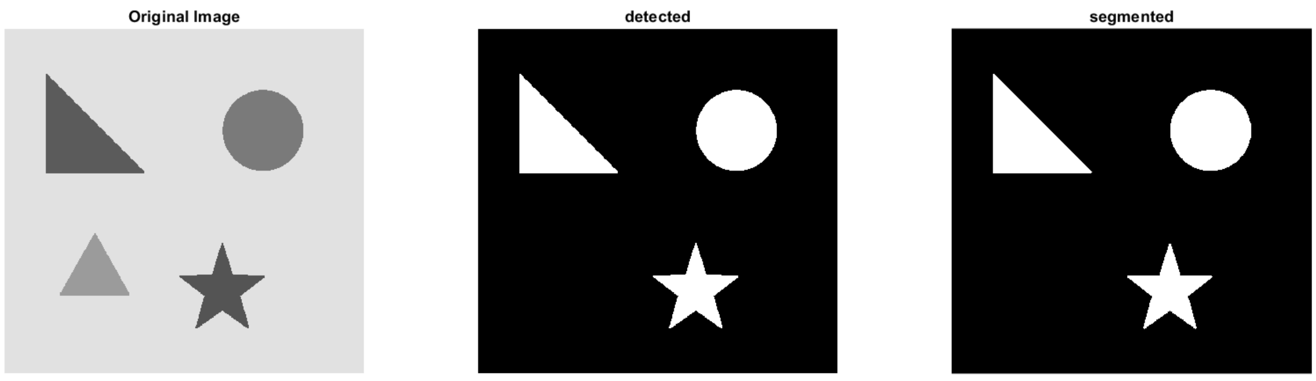


Figure 8: segmented image in MATLAB

REGION BASED SEGMENTATION

Region-based segmentation is a type of image segmentation technique that involves partitioning an image into multiple regions or segments based on certain characteristics or features of the image. The goal of region-based segmentation is to identify and separate different objects or regions within an image based on their similarities or differences.

There are several approaches to region-based segmentation, including thresholding, edge detection, and clustering. In thresholding, the image is converted into a binary image by selecting a threshold value that separates the image pixels into foreground and background. In edge detection, the boundaries of different regions are detected based on changes in intensity or colour. Clustering algorithms group similar pixels together based on their features or attributes, such as colour, texture, or intensity.

Once the image has been segmented into regions, further analysis can be performed on each region separately. This can include feature extraction, object recognition, and classification. Region-based segmentation is widely used in applications such as image processing, computer vision, and medical imaging, where identifying and analysing specific regions of an image is important.

REGION GROWING SEGMENTATION

Region growing is a type of region-based segmentation technique that involves starting with a seed point or region and gradually expanding the region by including neighbouring pixels or regions that have similar properties or characteristics. The process continues until the entire region of interest has been segmented. The region growing algorithm works by selecting a seed point or region and then examining its neighbouring pixels or regions. The algorithm checks if the neighbouring pixels or regions meet a certain similarity criterion, such as having similar colour, intensity, or texture. If they do, they are added to the growing region, and the algorithm continues to examine their neighbouring pixels or regions. This process continues until no more pixels or regions meet the similarity criterion, and the growing region is complete.

One advantage of region growing is that it can handle images with variable illumination and shading. However, the algorithm can be sensitive to the choice of seed point or region, as the resulting segmented region can vary depending on the starting point. To address this issue, multiple seed points can be used, and the final segmented region can be obtained by merging the results of multiple regions growing processes. Region growing is widely used in medical imaging applications, such as segmenting tumours in MRI or CT scans. It is also used in computer vision applications, such as object recognition and tracking.

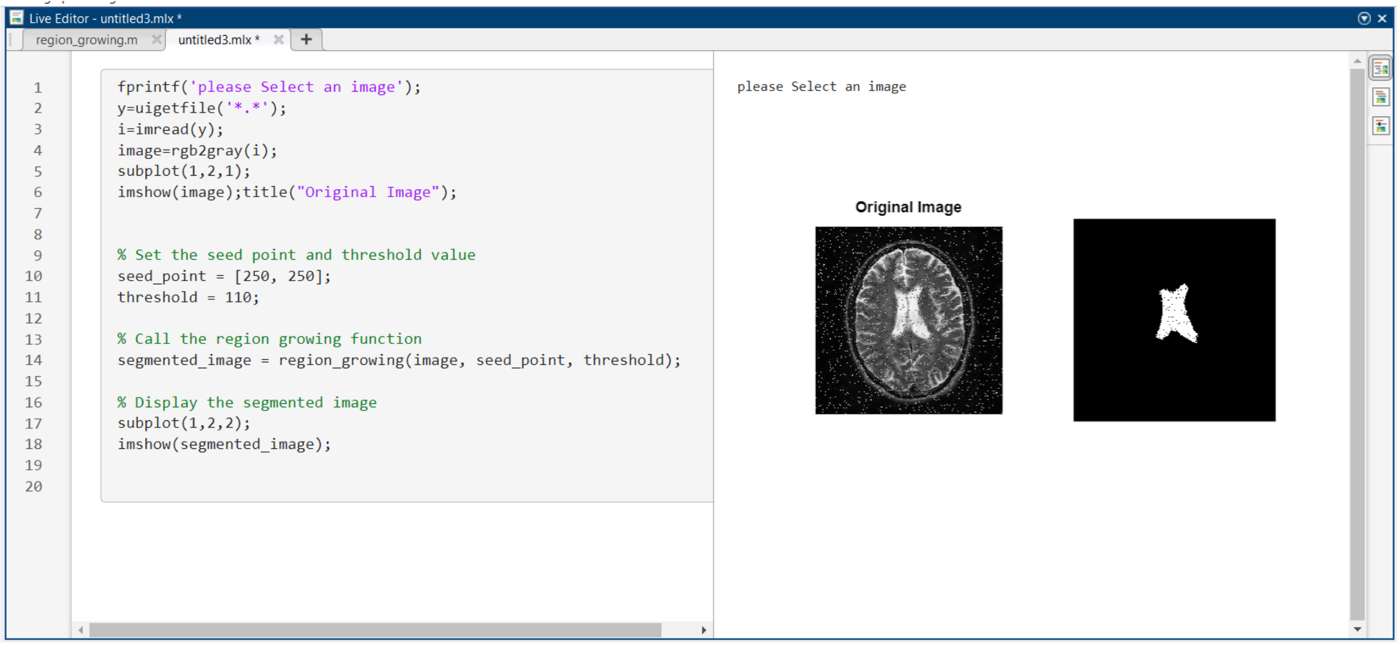
CODE

```
 FUNCTION FOR REGION GROWING SEGMENTATION


function [segmented_image] = region_growing(image, seed_point, threshold)
% Inputs:
%   - image: The input grayscale image
%   - seed_point: A 2-element vector containing the (x,y) coordinates of the seed point
%   - threshold: The threshold value for region growing
% Output:
%   - segmented_image: The output binary segmented image


% Initialize the segmented image to all zeros
segmented_image = zeros(size(image));


% Get the size of the image
[rows,cols] = size(image);


% Initialize the queue with the seed point
queue = [seed_point(1), seed_point(2)];
```

```
% Loop through the queue until it is empty
while ~isempty(queue)
    % Pop the first element from the queue
    current_point = queue(1,:);
    queue(1,:) = [];

    % Check if the current point is already segmented
    if segmented_image(current_point(2), current_point(1)) == 1
        continue;
    end

    % Check if the current point is within the image boundaries
    if current_point(1) < 1 || current_point(1) > cols || ...
            current_point(2) < 1 || current_point(2) > rows
        continue;
    end

    % Check if the intensity of the current point is below the
threshold
    if image(current_point(2), current_point(1)) < threshold
        continue;
    end

    % Mark the current point as segmented
    segmented_image(current_point(2), current_point(1)) = 1;

    % Add the neighbors of the current point to the queue
    queue(end+1,:) = [current_point(1)-1, current_point(2)-1];
    queue(end+1,:) = [current_point(1), current_point(2)-1];
    queue(end+1,:) = [current_point(1)+1, current_point(2)-1];
    queue(end+1,:) = [current_point(1)-1, current_point(2)];
    queue(end+1,:) = [current_point(1)+1, current_point(2)];
    queue(end+1,:) = [current_point(1)-1, current_point(2)+1];
    queue(end+1,:) = [current_point(1), current_point(2)+1];
    queue(end+1,:) = [current_point(1)+1, current_point(2)+1];
end
```

CODE

```
 REGION GROWING SEGMENTATION

fprintf('please Select an image');
y=uigetfile('*.*');
i=imread(y);
image=rgb2gray(i);
subplot(1,2,1);
imshow(image);title("Original Image");
% Set the seed point and threshold value
seed_point = [300    , 550];
threshold = 10;
% Call the region growing function
segmented_image = region_growing(image, seed_point, threshold);
% Display the segmented image
subplot(1,2,2);
imshow(segmented_image);
```



```
fprintf('please Select an image');
y=uigetfile('*.*');
i=imread(y);
image=rgb2gray(i);
subplot(1,2,1);
imshow(image);title("Original Image");


% Set the seed point and threshold value
seed_point = [250, 250];
threshold = 110;

% Call the region growing function
segmented_image = region_growing(image, seed_point, threshold);

% Display the segmented image
subplot(1,2,2);
imshow(segmented_image);
```

```
please Select an image
```

Figure 9: use of region growing segmentation in MATLAB

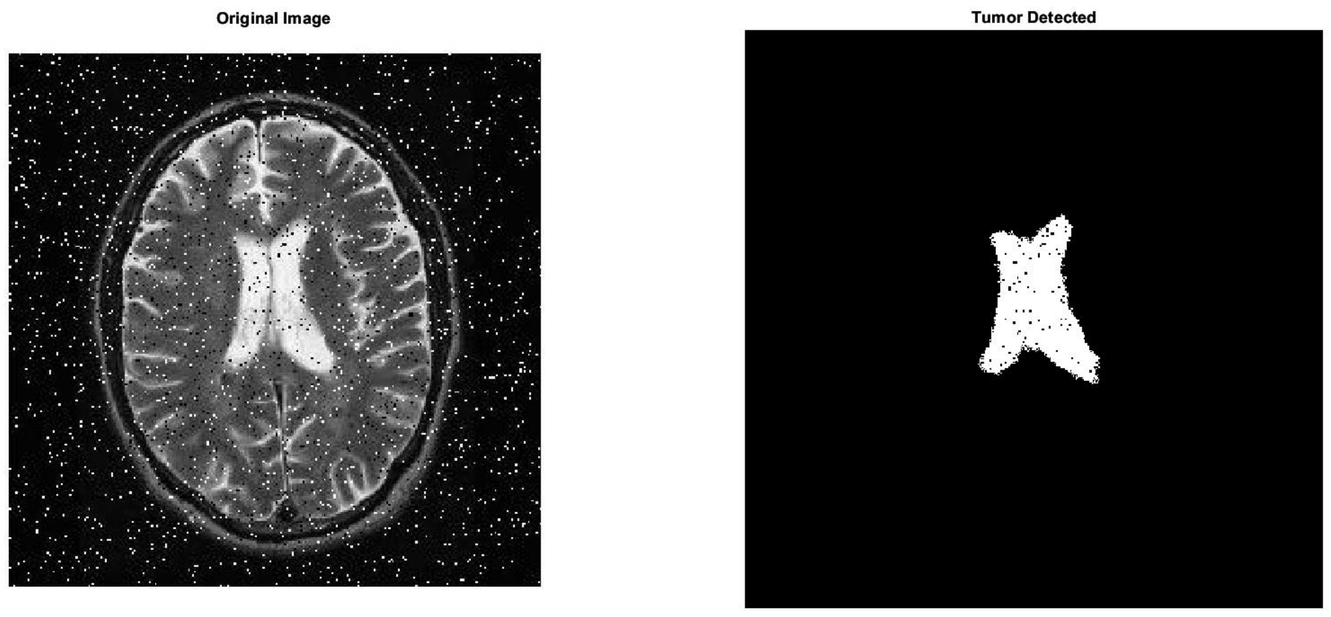


Figure 10: segmented image in MATLAB

Shreenandan Sahu |120BM0806