Experiment – I

1

IMAGE ENHANCEMENT

Spatial Domain and Frequency Domain Techniques

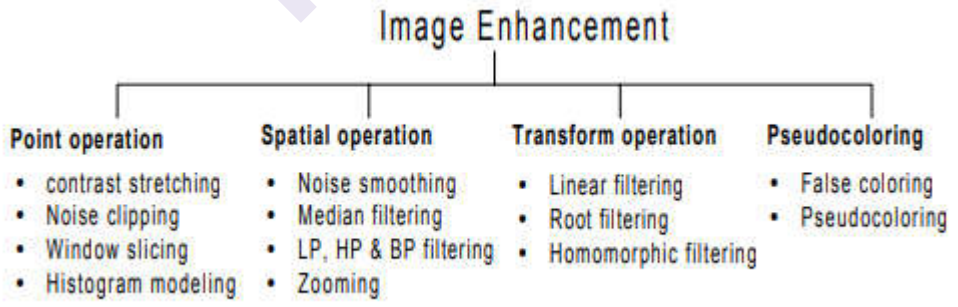
1.1.0 Objectives

The aim of image enhancement is to improve the interpretability or perception of information in images for human viewers, or to provide better' input for other automated image processing techniques. Image enhancement techniques can be divided into two broad categories:

1. Spatial domain methods, which operate directly on pixels, and
2. Frequency domain methods, which operate on the Fourier transform of an image.

1.1.1 Introduction to Image Enhancement

- The principal objective of image enhancement is to process a given image so that the result is more suitable than the original image for a specific application.
- It accentuates or sharpens image features such as edges, boundaries, or contrast to make a graphic display more helpful for display and analysis.
- The enhancement doesn't increase the inherent information content of the data, but it increases the dynamic range of the chosen features so that they can be detected easily



- The greatest difficulty in image enhancement is quantifying the criterion for enhancement and, therefore, a large number of image enhancement techniques are empirical and require interactive procedures to obtain satisfactory results.
- Image enhancement methods can be based on either spatial or frequency domain techniques.

1.1.2 Spatial Filtering:

The use of spatial masks for image processing is called spatial filtering. The masks used are called spatial filters.

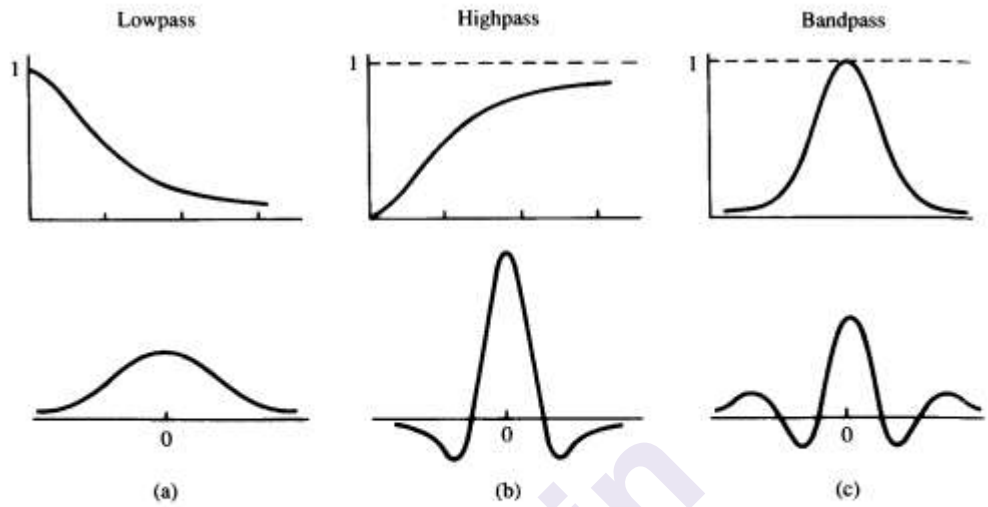


Fig 1 Top: cross sections of basic shapes for circularly symmetric frequency domain filter. Bottom: cross sections of corresponding spatial domain filters. (a) lowpass, (b) bandpass and (c) highpass filters.

- The basic approach is to sum products between the mask coefficients and the intensities of the pixels under the mask at a specific location in the image. (2D convolution).

$$R(x,y)=\sum_{i=-d}^{d}\sum_{-d}^{d} w(i,j)f(x-i,y-j)$$

where (2d+1)X(2d+1) is the mask size, w(i,j)'s are weights of the mask, f(x,y) is input pixel at coordinates (x,y), R(x,y) is the output value at (x,y).

If the center of the mask is at location (x,y) in the image, the gray level of the pixel located at (x,y) is replaced by R, the mask is then moved to the next location in the image and the process is repeated. This continues until all pixel locations have been covered.

1.1.2.1 Smoothing filter

1. Smoothing filters are used for blurring and for noise reduction.
2. Blurring is used in preprocessing steps, such as removal of small details from an image prior to object extraction, and bridging of small gaps in lines or curves.
3. Noise reduction can be accomplished by blurring with a linear filter and also by nonlinear filtering.

a. Low pass filtering:

- The key requirement is that all coefficients are positive.
- Neighborhood averaging is a special case of LPF where all coefficients are equal.
- It blurs edges and other sharp details in the image.

$$\text{Example: } \frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

b. Median filtering:

If the objective is to achieve noise reduction instead of blurring, this method should be used. This method is particularly effective when the noise pattern consists of strong, spike-like components and the characteristic to be preserved is edge sharpness. It is a nonlinear operation. For each input pixel f(x,y), we sort the values of the pixel and its neighbors to determine their median and assign its value to the output pixel g(x,y).

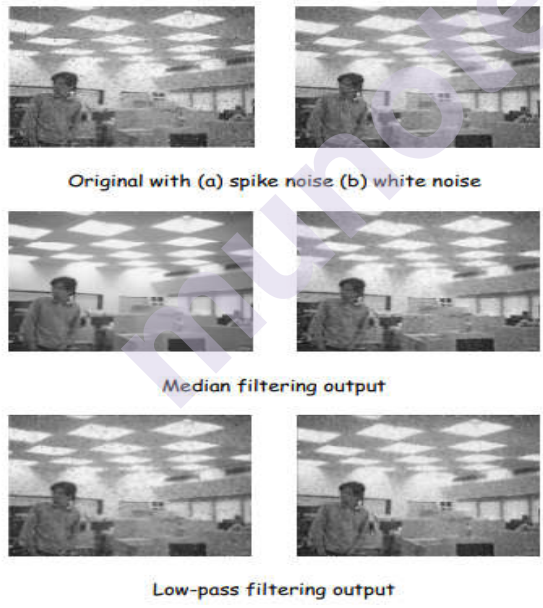


Fig.2: Median Filter

1.1.2.2 Sharpening Filters

To highlight fine detail in an image or to enhance detail that has been blurred, either in error or as a natural effect of a particular method of image acquisition. Uses of image sharpening vary and include applications ranging from electronic printing and medical imaging to industrial inspection and autonomous target detection in smart weapons.

a. Basic high pass spatial filter:

The shape of the impulse response needed to implement a high pass spatial filter indicates that the filter should have positive coefficients near its center, and negative coefficients in the outer periphery.

Example : filter mask of a 3x3 sharpening filter

$$\frac{1}{9}\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

The filtering output pixels might be of a gray level exceeding [0,L-1].

The results of high pass filtering involve some form of scaling and/or clipping to make sure that the gray levels of the final results are within [0,L-1].

1.1.3 Frequency Domain Filtering

We simply compute the Fourier transform of the image to be enhanced, multiply the result by a filter transfer function, and take the inverse transform to produce the enhanced image.

Spatial domain: $g(x,y)=f(x,y)*h(x,y)$

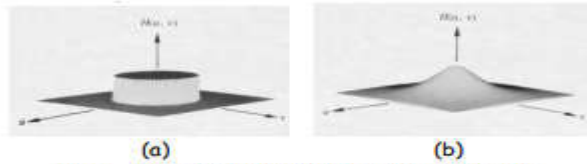
Frequency domain: $G(w1,w2)=F(w1,w2)H(w1,w2)$

a. Low Pass filtering:

Edges and sharp transitions in the gray levels contribute to the high frequency content of its Fourier transform, so a low pass filter smoothes an image.

Formula of ideal LPF:

$$H(u,v)=\begin{cases} 1 & if\ D(u,v)\le D_o \\ 0 & else \end{cases}$$



(a) (b)

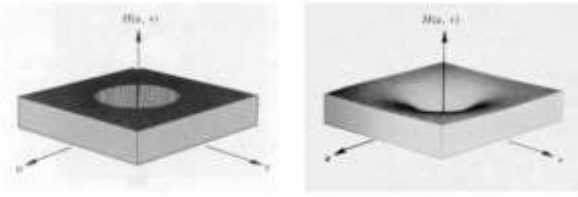
Fig 3. (a) Ideal LPF; (b) Butterworth LPF.

b. High Pass filtering:

A high pass filter attenuates the low frequency components without disturbing the high frequency information in the Fourier transform domain and can sharpen edges.

Formula of ideal HPF function:

$$H(u,v)=\begin{cases}0 & if\ D(u,v)\le D_o\\ 1 & else\end{cases}$$



(a) (b)

Fig 4. (a) Ideal HPF; (b) Butterworth HPF.

Experiment 01

Aim :- Program for image enhancement (Smoothing & Sharpening) using spatial domain filters.

Objective :-

The purpose of this assignment is to study image filtering in the spatial domain. Spatial filtering is performed by convolving the image with a mask or a kernel.Spatial filters include sharpening, smoothing, edge detection, noise removal, etc. It consists of four parts: the first one discusses the spatial filtering of an image using a spatial mask .3x3, 5x5, and then this mask is used in a blurring filter . The second part studies the order statistics filters, specially the median filter.

Part I Smoothing spatial filter: The output of a smoothing spatial filter is simply the average of the pixels contained in the neighborhood of the filter mask. - These filters are sometimes called averaging filters and also low pass filters - Two types of masks of the spatial filter

$\frac{1}{9}\times$

1	1	1
1	1	1
1	1	1

$\frac{1}{16}\times$

1	2	1
2	4	2
1	2	1

Steps :-

1) Read input Image.

2) Add noise using the “imnoise()” function.

3) Define a (3 x 3) filter.

4)Use convolution function conv2() for filtering

• Order statistics filters are nonlinear spatial filters whose response is based on ordering (ranking) the pixels contained in an area covered by the filter

• The best known example in this category is median filter

• Median filter - Median filters replace the value of the pixel by the median of the gray levels in the neighborhood of that pixel

Part :-II Sharpening spatial filter:

Spatial domain sharpening filters are also called as High Pass Filters Laplacian Filters

Laplacian Filters

0	1	0	1	1	1
1	−4	1	1	−8	1
0	1	0	1	1	1
0	−1	0	−1	−1	−1
−1	4	−1	−1	8	−1
0	−1	0	−1	−1	−1

Gradient Filter

−1	0	0	−1
0	1	1	0

−1	−2	−1	−1	0	1
0	0	0	−2	0	2
1	2	1	−1	0	1

Part III :- Study of spatial domain filters

Study following functions and use them on various images with all possible parameters.

fspecial(), imfilter() Ordfilt2() Medfilt2() Imnoise() Median()

fspecial :- Create predefined 2-D filter

Syntax h = fspecial(type)

h = fspecial(type, parameters)

Description:

h = fspecial(type) create a two-dimensional filter h of the specified type. fspecial returns h as a correlation kernel,.

Value	**Description**
'average'	Averaging filter
'gaussian'	Gaussian lowpass filter
'laplacian'	Approximates the two-dimensional Laplacian operator
'prewitt'	Prewitt horizontal edge-emphasizing filter
‘sobel'	'Sobel horizontal edge-emphasizing filter
‘Unsharp’	"Unsharp contrast enhancement filter

Value Description 'average' Averaging filter 'gaussian' Gaussian lowpass filter

h = fspecial(type, parameters)

accepts the filter specified by type plus additional modifying parameters particular to the type of filter chosen. If you omit these arguments, fspecial uses default values for the parameters. The following list shows the syntax for each filter type. Where applicable,

h = fspecial('average', hsize)

returns an averaging filter h of size hsize. The argument hsize can be a vector specifying the number of rows and columns in h, or it can be a scalar, in which case h is a square matrix. The default value for hsize is [3 3].

h = fspecial ('gaussian', hsize, sigma) :-

returns a rotationally symmetric Gaussian lowpass filter of size hsize with standard deviation sigma (positive). hsize can be a vector specifying the number of rows and columns in h, or it can be a scalar, in which case h is a square matrix. The default value for hsize is [3 3]; the default value for sigma is 0.5.

h = fspecial('laplacian', alpha)

returns a 3-by-3 filter approximating the shape of the two-dimensional Laplacian operator. The parameter alpha controls the shape of the Laplacian and must be in the range 0.0 to 1.0. The default value for alpha is 0.2.

h = fspecial('log', hsize, sigma)

returns a rotationally symmetric Laplacian of Gaussian filter of size hsize with standard deviation sigma (positive). hsize can be a vector specifying the number of rows and columns in h, or it can be a scalar, in which case h is a square matrix. The default value for hsize is [5 5] and 0.5 for sigma.

h = fspecial('prewitt')

h = fspecial('sobel')

Median Filter :- Median filters replace the value of the pixel by the median of the gray levels in the neighborhood of that pixel

1) Open /Read an image in a matrix .

2) Create a 3x3 matrix B called a mask.

3) Read the first 3x3 pixel grid of the input image into B.

4) Sort the matrix B in ascending order.

5) Select the middle value and put that as the first pixel value in the output image matrix.

6) Repeat the procedure for the entire input image by reading the next 3x3 values from the input image and sort using mask B. This way Output image values are calculated.

7) Display the input image and Output Image.

Programs for image enhancement using spatial domain filters. %This program is for Averaging spatial Filter

%This program is for Averaging spatial Filter

```
a=imread('D:\DIP Course Material\DIP pract\Images\rose.jpg');

% Addition of noise to the input image

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

c=imnoise(a,'gaussian');

d=imnoise(a,'speckle');

% Defining 3x3 and 5x5 kernel

h1=1/9*ones(3,3); h2=1/25*ones(5,5);

% Attempt to recover the image

b1=conv2(b,h1,'same');

b2=conv2(b,h2,'same');

c1=conv2(c,h1,'same');

c2=conv2(c,h2,'same');

d1=conv2(d,h1,'same');

d2=conv2(d,h2,'same');

a=imread('D:\DIP Course Material\DIP pract\Images\rose.jpg');

% Addition of noise to the input image

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

c=imnoise(a,'gaussian');

d=imnoise(a,'speckle');

% Defining 3x3 and 5x5 kernel

 h1=1/9*ones(3,3);

 h2=1/25*ones(5,5);
```

```
% Attempt to recover the image
b1=conv2(b,h1,'same');
b2=conv2(b,h2,'same');
c1=conv2(c,h1,'same');
c2=conv2(c,h2,'same');
d1=conv2(d,h1,'same');
d2=conv2(d,h2,'same');
% displaying the result figure,
subplot(2,2,1),
imshow(a),
title('Original Image'),
subplot(2,2,2),
imshow(b),
title('Salt & Pepper noise'),
subplot(2,2,3),
imshow(uint8(b1)),
title('3 x 3 Averaging filter'),
subplot(2,2,4)
imshow(uint8(b2)),
title('5 x 5 Averaging filter')
 %..... figure,
subplot(2,2,1),
imshow(a),
title('Original Image'),
subplot(2,2,2),
imshow(c),
title('Gaussian noise'),
subplot(2,2,3),imshow(uint8(c1)),
title('3 x 3 Averaging filter'),
```

```
subplot(2,2,4),
imshow(uint8(c2)),
title('5 x 5 Averaging filter'),
%..... figure,
subplot(2,2,1),
imshow(a),
title('Original Image'),
subplot(2,2,2),
imshow(d),
title('Speckle noise'),
subplot(2,2,3),
imshow(uint8(d1)),
title('3 x 3 Averaging filter'),
subplot(2,2,4),
imshow(uint8(d2)),
title('5 x 5 Averaging filter'),
```

%this program is for comparing averaging & median filter

```
clc clear
all close all
a=imread('D:\DIP Course Material\DIP pract\Images\horse.jpg');
%Addition of salt and pepper noise b=imnoise(a,'salt & pepper',0.1);
%Defining the box and median filters
h1=1/9*ones(3,3);
h2=1/25*ones(5,5);
c1=conv2(b,h1,'same');
c2=conv2(b,h2,'same');
c3=medfilt2(b,[3 3]);
c4=medfilt2(b,[5 5]);
subplot(3,2,1),
```

```
imshow(a),
title('Original image')
subplot(3,2,2),
imshow(b),
title('Salt & pepper noise')
subplot(3,2,3),
imshow(uint8(c1)),
title('3 x 3 smoothing')
subplot(3,2,4),
imshow(uint8(c2)),
title('5 x 5 smoothing')
subplot(3,2,5),
imshow(uint8(c3)),
title('3x 3 Median filter')
 subplot(3,2,6),
imshow(uint8(c4)),
title('5 x 5 Median filter')
```

% this program is for sharpening spatial domain filter

```
%Sharpening Filters
A=ones(200,200);
A(30:60,30:60)=0;
A(70:150,50:170)=0
figure(1) ,
subplot(1,2,1)
 imshow(A)
AM=[1 1 1;1 -8 1;1 1 1];
 B=conv2(A,AM);
subplot(1,2,2),
 imshow(B)
```

% this program is for sharpening spatial domain filter

```
a=imread('D:\horse.jpg');
%Defining the laplacian filters
 h1=[0 -1 0;-1 4 -1;0 -1 0]
h2=[-1 -1 -1;-1 8 -1; -1 -1 -1];
h3=[-1 -1 -1;-1 9 -1; -1 -1 -1];
c1=conv2(a,h1,'same');
c2=conv2(a,h2,'same');
c3=conv2(a,h3,'same');
subplot(2,2,1),imshow(a),
title('Original image')
subplot(2,2,2),
imshow(uint8(c1)),
title('Laplacian sharpening 4 at center')
subplot(2,2,3),imshow(uint8(c2)),
title('Laplacian sharpening 8 at center ')
subplot(2,2,4),
imshow(uint8(c3)),
title(' Laplacian sharpening 9 at center')
%Averaging Filter
 A=ones(200,200);
A(30:60,30:60)=0;
A(70:150,50:170)=0
figure(1)
subplot(1,2,1)
imshow(A)
AM=1/9.*[1 1 1;1 1 1;1 1 1];
B=conv2(A,AM);
subplot(1,2,2)
imshow(B)
```

Experiment 02

AIM: To Implement smoothing or averaging filters in spatial domain.

OBJECTIVE: To Implement smoothing or averaging filters in spatial domain.

TOOLS REQUIRED: MATLAB

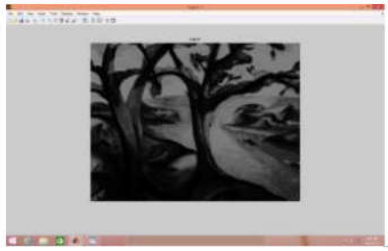
THEORY:

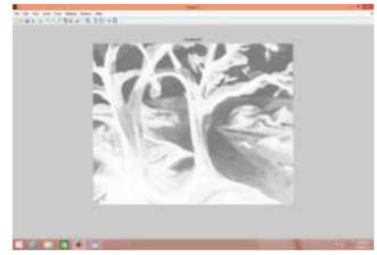
Filtering is a technique for modifying or enhancing an image. Masks or filters will be defined. The general process of convolution and correlation will be introduced via an example. Also smoothing linear filters such as box and weighted average filters will be introduced. In statistics and image processing, to smooth a data set is to create an approximating function that attempts to capture important patterns in the data, while leaving out noise or other fine-scale structures/rapid phenomena. In smoothing, the data points of a signal are modified so individual points (presumably because of noise) are reduced, and points that are lower than the adjacent points are increased leading to a smoother signal. Smoothing may be used in two important ways that can aid in data analysis by being able to extract more information from the data as long as the assumption of smoothing is reasonable by being able to provide analyses that are both flexible and robust. different algorithms are used in smoothing.

% Program for implementation of smoothing or averaging filter in spatial domain

```
 I=imread('trees.tif');
subplot(2,2,1);
imshow(J);
title('original image');
 f=ones(3,3)/9;
 h=imfilter(I,f,'circular');
subplot(2,2,2);
 imshow(h);
title('averaged image');
```

Result:





Conclusion: Thus we have performed the smoothing or averaging filter operation on the Original image and we get a filtered image.

Discrete Fourier Transform:

The general idea is that the image (***f(x,y)*** of size ***M*** x ***N***) will be represented in the frequency domain (***F(u,v)***). The equation for the two-dimensional discrete Fourier transform (DFT) is:

$$F(u,v) = \sum_{x=0}^{M-1}\sum_{y=0}^{N-1} f(x,y)e^{-i2\pi\left(\frac{ux}{M}+\frac{vy}{N}\right)}$$

The concept behind the Fourier transform is that any waveform can be constructed using a sum of sine and cosine waves of different frequencies. The exponential in the above formula can be expanded into sines and cosines with the variables ***u*** and ***v*** determining these frequencies.

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

$$f(x,y) = \frac{1}{MN}\sum_{u=0}^{M-1}\sum_{v=0}^{N-1} F(u,v)e^{i2\pi\left(\frac{ux}{M}+\frac{vy}{N}\right)}$$

Thus, if we have ***F(u,v)***, we can obtain the corresponding image (***f(x,y)***) using the inverse, discrete Fourier transform.

Things to note about the discrete Fourier transform are the following:

- the value of the transform at the origin of the frequency domain, at ***F(0,0)***, is called the dc component
    - ***F(0,0)*** is equal to ***MN*** times the average value of ***f(x,y)***
    - in MATLAB, ***F(0,0)*** is actually ***F(1,1)*** because array indices in MATLAB start at 1 rather than 0

- The values of the Fourier transform are complex, meaning they have real and imaginary parts. The imaginary parts are represented by i, which is defined solely by the property that its square is −1, ie:

$$i^2 = -1$$

- We visually analyze a Fourier transform by computing a **Fourier spectrum** (the magnitude of ***F(u,v)***) and display it as an image.

    1. the Fourier spectrum is symmetric about the origin

- The fast Fourier transform (FFT) is a fast algorithm for computing the discrete Fourier transform.
- MATLAB has three functions to compute the DFT:

    1. fft -for one dimension (useful for audio)
    2. fft2 -for two dimensions (useful for images)
    3. fftn -for n dimensions

- MATLAB has three related functions that compute the inverse DFT:

    1. ifft
    2. ifft2
    3. ifftn

How to Display a Fourier Spectrum using MATLAB?

```
%Create a black 30x30 image
f=zeros(30,30);
%With a white rectangle in it.
f(5:24,13:17)=1;
imshow(f,'InitialMagnification', 'fit')
%Calculate the DFT.
F=fft2(f);
%There are real and imaginary parts to F.
%Use the abs function to compute the magnitude
%of the combined components.
F2=abs(F);
```

```
figure, imshow(F2,[], 'InitialMagnification','fit')
%To create a finer sampling of the Fourier transform,
%you can add zero padding to f when computing its DFT
%Also note that we use a power of 2, 2^256
%This is because the FFT -Fast Fourier Transform -
%is fastest when the image size has many factors.
F=fft2(f, 256, 256);
F2=abs(F);
figure, imshow(F2, [])
%The zero-frequency coefficient is displayed in the
%upper left hand  corner. To display it in the center,
%you can use the function fftshift.
F2=fftshift(F);
F2=abs(F2);
figure,imshow(F2,[])
%In Fourier transforms, high peaks are so high they
%hide details. Reduce contrast with the log function.
F2=log(1+F2);
figure,imshow(F2,[])
```

To get the results shown in the last image of the table, you can also combine MATLAB calls as in:

```
f=zeros(30,30);
f(5:24,13:17)=1;
F=fft2(f, 256,256);
F2=fftshift(F);
figure,imshow(log(1+abs(F2)),[])
```

Notice in these calls to imshow, the second argument is empty square brackets. This maps the minimum value in the image to black and the maximum value in the image to white.

1.1.4 References

- ***Digital Image Processing, Using MATLAB***, by Rafael C. Gonzalez, Richard E. Woods, and Steven L. Eddins
- ***Image Processing Toolbox, For Use with MATLAB*** (MATLAB's documentation)--available through MATLAB's help menu or online at: http://www.mathworks.com/access/helpdesk/help/toolbox/images/
- Frequency Domain Processing: www.cs.uregina.ca/Links/class-info/425/Lab5/index.html

❖❖❖❖

Experiment -II

2

DISCRETE FOURIER TRANSFORMATION

Aim: To find DFT/FFT forward and inverse transform of image.

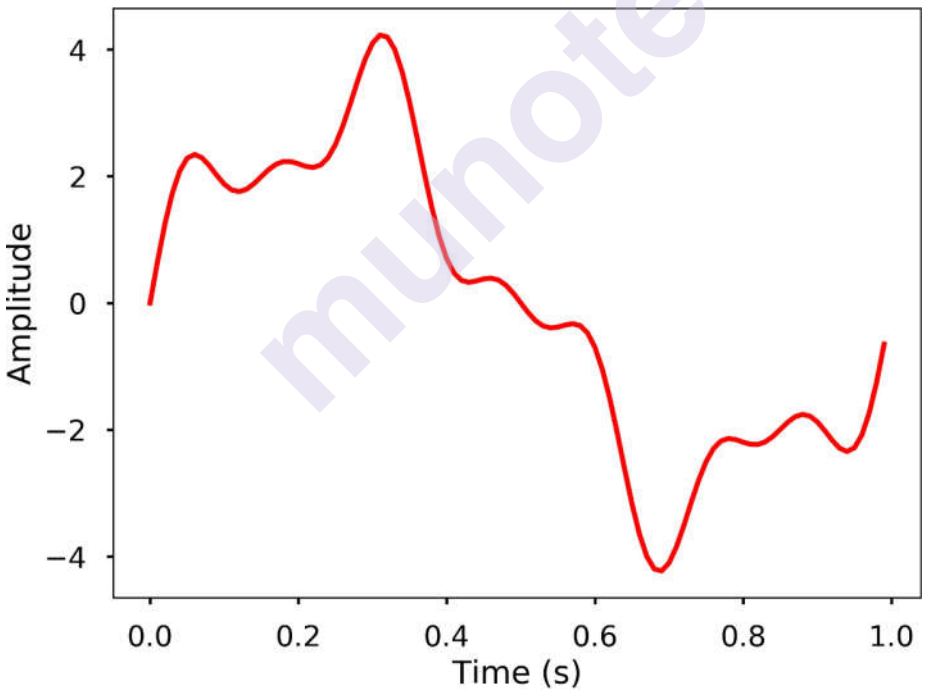
Theory:

FFT: fast Fourier transform.

IFFT: Inverse fast Fourier transform.

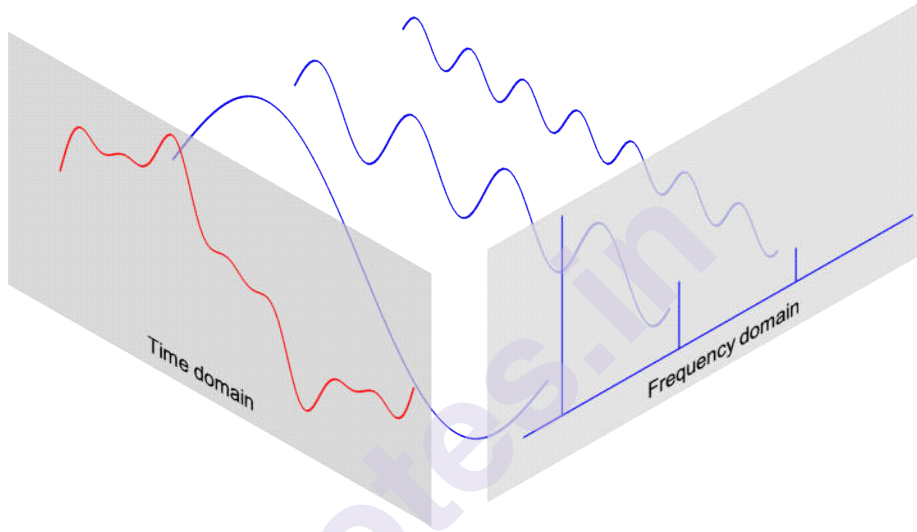
Discrete Fourier Transform (DFT)

From the previous section, we learned how we can easily characterize a wave with period/frequency, amplitude, phase. But these are easy for simple periodic signal, such as sine or cosine waves. For complicated waves, it is not easy to characterize like that. For example, the following is a relatively more complicate waves, and it is hard to say what's the frequency, amplitude of the wave, right?



There are more complicated cases in real world, it would be great if we have a method that we can use to analyze the characteristics of the wave. The **Fourier Transform** can be used for this purpose, which it decompose any signal into a sum of simple sine and cosine waves that we can easily measure the frequency, amplitude and phase. The Fourier transform can be applied to continuous or discrete waves, in this chapter, we will only talk about the Discrete Fourier Transform (DFT).

Using the DFT, we can compose the above signal to a series of sinusoids and each of them will have a different frequency. The following 3D figure shows the idea behind the DFT, that the above signal is actually the results of the sum of 3 different sine waves. The time domain signal, which is the above signal we saw can be transformed into a figure in the frequency domain called DFT amplitude spectrum, where the signal frequencies are showing as vertical bars. The height of the bar after normalization is the amplitude of the signal in the time domain. You can see that the 3 vertical bars are corresponding the 3 frequencies of the sine wave, which are also plotted in the figure.



In this section, we will learn how to use DFT to compute and plot the DFT amplitude spectrum.

DFT

The DFT can transform a sequence of evenly spaced signal to the information about the frequency of all the sine waves that needed to sum to the time domain signal. It is defined as:

$$Xk=\sum n=0N-1xn \cdot e-i2\pi kn/N=\sum n=0N-1xn[\cos(2\pi kn/N)-i \cdot \sin(2\pi kn/N)]$$
$$Xk=\sum n=0N-1xn \cdot e-i2\pi kn/N=\sum n=0N-1xn[\cos(2\pi kn/N)-i \cdot \sin(2\pi kn/N)]$$

where

- N = number of samples
- n = current sample
- k = current frequency, where $k \in [0,N-1]k \in [0,N-1]$
- xnxn = the sine value at sample n
- XkXk = The DFT which include information of both amplitude and phase

Also, the last expression in the above equation derived from the *Euler's formula*, which links the trigonometric functions to the complex exponential function: ei□x=cosx+i□sinxei□x=cosx+i□sinx

Due to the nature of the transform, X0=∑N−1n=0xnX0=∑n=0N−1xn. If NN is an odd number, the elements X1,X2,...,X(N−1)/2X1, X2,...,X(N−1)/2 contain the positive frequency terms and the elements X(N+1)/2,...,XN−1X(N+1)/2,...,XN−1 contain the negative frequency terms, in order of decreasingly negative frequency. While if NN is even, the elements X1,X2,...,XN/2−1X1,X2,...,XN/2−1 contain the positive frequency terms, and the elements XN/2, ...,XN−1XN/2,..., XN−1 contain the negative frequency terms, in order of decreasingly negative frequency. In the case that our input signal xx is a real-valued sequence, the DFT output XnXn for positive frequencies is the conjugate of the values XnXn for negative frequencies, the spectrum will be symmetric. Therefore, usually we only plot the DFT corresponding to the positive frequencies.

Note that the XkXk is a complex number that encodes both the amplitude and phase information of a complex sinusoidal component ei□2πkn/Nei□ 2πkn/ N of function xnxn. The amplitude and phase of the signal can be calculated as:

amp=|Xk|N=Re(Xk)2+Im(Xk)2−−−−−−−−−−−−−−−−−√Namp=|Xk|N=Re(Xk)2+Im(Xk)2N

phase=atan2(Im(Xk),Re(Xk))phase=atan2(Im(Xk),Re(Xk))

where Im(Xk)Im(Xk) and Re(Xk)Re(Xk) are the imagery and real part of the complex number, atan2atan2 is the two-argument form of the arctanarctan function.

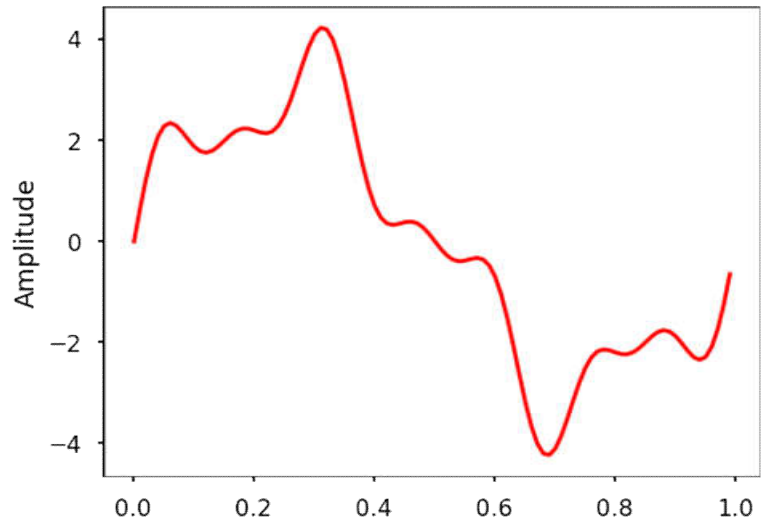
The amplitudes returned by DFT equal to the amplitudes of the signals fed into the DFT if we normalize it by the number of sample points. Note that doing this will divide the power between the positive and negative sides, if the input signal is real-valued sequence as we described above, the spectrum of the positive and negative frequencies will be symmetric, therefore, we will only look at one side of the DFT result, and instead of divide NN, we divide N/2N/2 to get the amplitude corresponding to the time domain signal.

Now that we have the basic knowledge of DFT, let's see how we can use it.

TRY IT! Generate 3 sine waves with frequencies 1 Hz, 4 Hz, and 7 Hz, amplitudes 3, 1 and 0.5, and phase all zeros. Add this 3 sine waves together with a sampling rate 100 Hz, you will see that it is the same signal we just shown at the beginning of the section.

```
import matplotlib.pyplot as plt
import numpy as np
plt.style.use('seaborn-poster')
%matplotlib inline
# sampling rate
sr = 100
# sampling interval
ts = 1.0/sr
t = np.arange(0,1,ts)
freq = 1.
x = 3*np.sin(2*np.pi*freq*t)
freq = 4
x += np.sin(2*np.pi*freq*t)
freq = 7
x += 0.5* np.sin(2*np.pi*freq*t)
plt.figure(figsize = (8, 6))
plt.plot(t, x, 'r')
plt.ylabel('Amplitude')
plt.show()
```

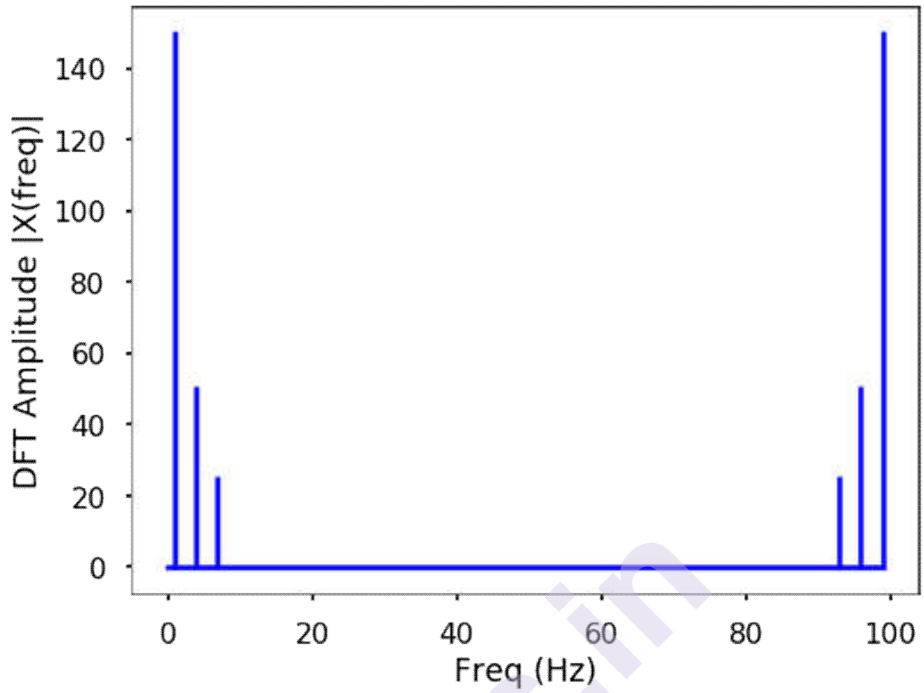
Output:



TRY IT! Write a function *DFT(x)* which takes in one argument, *x* - input 1 dimensional real-valued signal. The function will calculate the DFT of the signal and return the DFT values. Apply this function to the signal we generated above and plot the result.

```
def DFT(x):
    """
    Function to calculate the
    discrete Fourier Transform
    of a 1D real-valued signal x
    """
    N = len(x)
    n = np.arange(N)
    k = n.reshape((N, 1))
    e = np.exp(-2j * np.pi * k * n / N)
       X = np.dot(e, x)
       return X


       X = DFT(x)
# calculate the frequency
N = len(X)
n = np.arange(N)
T = N/sr
freq = n/T
plt.figure(figsize = (8, 6))
plt.stem(freq, abs(X), 'b', \
        markerfmt=" ", basefmt="-b")
plt.xlabel('Freq (Hz)')
plt.ylabel('DFT Amplitude |X(freq)|')
plt.show()
```

Output:

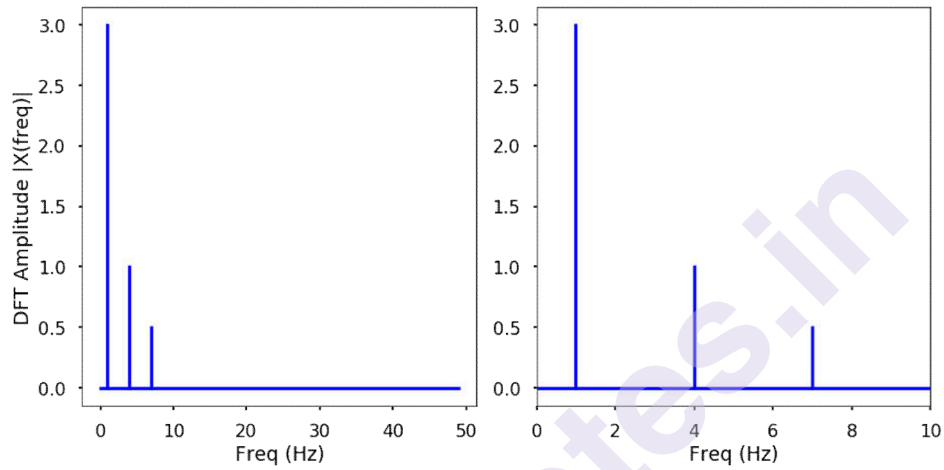


We can see from here that the output of the DFT is symmetric at half of the sampling rate (you can try different sampling rate to test). This half of the sampling rate is called **Nyquist frequency** or the folding frequency, it is named after the electronic engineer Harry Nyquist. He and Claude Shannon have the Nyquist-Shannon sampling theorem, which states that a signal sampled at a rate can be fully reconstructed if it contains only frequency components below half that sampling frequency, thus the highest frequency output from the DFT is half the sampling rate.

```
n_oneside = N//2

# get the one side frequency

f_oneside = freq[:n_oneside]

# normalize the amplitude

X_oneside =X[:n_oneside]/n_oneside

plt.figure(figsize = (12, 6))

plt.subplot(121)

plt.stem(f_oneside, abs(X_oneside), 'b', \

        markerfmt=" ", basefmt="-b")

plt.xlabel('Freq (Hz)')

plt.ylabel('DFT Amplitude |X(freq)|')

plt.subplot(122)
```

```
plt.stem(f_oneside, abs(X_oneside), 'b', \
        markerfmt=" ", basefmt="-b")
plt.xlabel('Freq (Hz)')
plt.xlim(0, 10)
plt.tight_layout()
plt.show()
```

Output:



We can see by plotting the first half of the DFT results, we can see 3 clear peaks at frequency 1 Hz, 4 Hz, and 7 Hz, with amplitude 3, 1, 0.5 as expected. This is how we can use the DFT to analyze an arbitrary signal by decomposing it to simple sine waves.

The inverse DFT

Of course, we can do the inverse transform of the DFT easily.

xn=1N∑k=0N−1Xk□ei□2πkn/Nxn=1N∑k=0N−1Xk· ei· 2πkn/N

We will leave this as an exercise for you to write a function.

The limit of DFT

The main issue with the above DFT implementation is that it is not efficient if we have a signal with many data points. It may take a long time to compute the DFT if the signal is large.

TRY IT Write a function to generate a simple signal with different sampling rate, and see the difference of computing time by varying the sampling rate.

```
def gen_sig(sr):
    '''
```

```
    function to generate
    a simple 1D signal with
    different sampling rate
    '''
    ts = 1.0/sr
    t = np.arange(0,1,ts)
    freq = 1.
    x = 3*np.sin(2*np.pi*freq*t)
    return x
# sampling rate =2000
sr = 2000
%timeit DFT(gen_sig(sr))
```

Output:

120 ms ± 8.27 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)

```
# sampling rate 20000
sr = 20000
%timeit DFT(gen_sig(sr))
```

Output:

15.9 s ± 1.51 s per loop (mean ± std. dev. of 7 runs, 1 loop each)

Example 1 :

```
# import sympy
from sympy import fft
# sequence
seq = [15, 21, 13, 44]
# fft
transform = fft(seq)
print (transform)
```

Output :

FFT : [93, 2 - 23*I, -37, 2 + 23*I]

Example 2 :

```
# import sympy
from sympy import fft
# sequence
seq = [15, 21, 13, 44]
decimal_point = 4
# fft
transform = fft(seq, decimal_point )

  print ("FFT : ", transform)
```

Output :

```
FFT :  [93, 2.0 - 23.0*I, -37, 2.0 + 23.0*I]
```

Fast Fourier Transform (FFT)

The **Fast Fourier Transform (FFT)** is an efficient algorithm to calculate the DFT of a sequence. It is described first in Cooley and Tukey's classic paper in 1965, but the idea actually can be traced back to Gauss's unpublished work in 1805. It is a divide and conquer algorithm that recursively breaks the DFT into smaller DFTs to bring down the computation. As a result, it successfully reduces the complexity of the DFT from O(n2)O(n2) to O(nlogn)O(nlogn), where nn is the size of the data. This reduction in computation time is significant especially for data with large NN, therefore, making FFT widely used in engineering, science and mathematics. The FFT algorithm is the Top 10 algorithm of 20th century by the journal Computing in Science & Engineering.

In this section, we will introduce you how does the FFT reduces the computation time. The content of this section is heavily based on this great tutorial put together by Jake VanderPlas.

Symmetries in the DFT

The answer to how FFT speedup the computing of DFT lies in the exploitation of the symmetries in the DFT. Let's take a look of the symmetries in the DFT. From the definition of the DFT equation

$$Xk=\sum n=0N-1xne-i2\pi kn/NXk=\sum n=0N-1xne-i2\pi kn/N$$

we can calculate the

$$Xk+N=\sum n=0N-1xne-i2\pi(k+N)n/N=\sum n=0N-1xne-i2\pi ne-i2\pi kn/NXk+N=\sum n=0N-1xne-i2\pi(k+N)n/N=\sum n=0N-1xne-i2\pi ne-i2\pi kn/N$$

Note that, $e-i2\pi n=1e-i2\pi n=1$, therefore, we have

Xk+N=∑n=0N−1xn□e−i2πkn/N=XkXk+N=∑n=0N−1xne−i2πkn/N=Xk

with a little extension, we can have

Xk+iN=Xk, for any integer iXk+iN=Xk, for any integer i

This means that within the DFT, we clearly have some symmetries that we can use to reduce the computation.

Tricks in FFT

Since we know there are symmetries in the DFT, we can consider to use it reduce the computation, because if we need to calculate both XkXk and Xk+NXk+N, we only need to do this once. This is exactly the idea behind the FFT. Cooley and Tukey showed that we can calculate DFT more efficiently if we continue to divide the problem into smaller ones. Let's first divide the whole series into two parts, i.e. the even number part and the odd number part:

Xk===∑n=0N−1xn□e−i2πkn/N∑m=0N/2−1x2m□e−i2πk(2m)/N+∑m=0N/2−1x2m+1□e−i2πk(2m+1)/N∑m=0N/2−1x2m□e−i2πkm/(N/2)+e−i2πk/N∑m=0N/2−1x2m+1□e−i2πkm/(N/2)Xk=∑n=0N−1xn□e−i2πkn/N=∑m=0N/2−1x2m□e−i2πk(2m)/N+∑m=0N/2−1x2m+1□e−i2πk(2m+1)/N=∑m=0N/2−1x2m□e−i2πkm/(N/2)+e−i2πk/N∑m=0N/2−1x2m+1□e−i2πkm/(N/2)

We can see that, the two smaller terms which only have half of the size (N2N2) in the above equation are two smaller DFTs. For each term, the 0≤m≤N20≤m≤N2, but 0≤k≤N0≤k≤N, therefore, we can see that half of the values will be the same due to the symmetry properties we described above. Thus, we only need to calculate half of the fields in each term. Of course, we don't need to stop here, we can continue to divide each term into half with the even and odd values until it reaches the last two numbers, then calculation will be really simple.

This is how FFT works using this recursive approach. Let's see a quick and dirty implementation of the FFT. Note that, the input signal to FFT should have a length of power of 2. If the length is not, usually we need to fill up zeros to the next power of 2 size.

```
import matplotlib.pyplot as plt

import numpy as np

plt.style.use('seaborn-poster')

%matplotlib inline

def FFT(x):
    """
```

```
    A recursive implementation of
    the 1D Cooley-Tukey FFT, the
    input should have a length of
    power of 2.
    """
    N = len(x)
        if N == 1:
        return x
    else:
        X_even = FFT(x[::2])
        X_odd = FFT(x[1::2])
        factor = \
          np.exp(-2j*np.pi*np.arange(N)/ N)
              X = np.concatenate(\
            [X_even+factor[:int(N/2)]*X_odd,
             X_even+factor[int(N/2):]*X_odd])
        return X

        # sampling rate
sr = 128
# sampling interval
ts = 1.0/sr
t = np.arange(0,1,ts)
freq = 1.
x = 3*np.sin(2*np.pi*freq*t)
freq = 4
x += np.sin(2*np.pi*freq*t)
freq = 7
x += 0.5* np.sin(2*np.pi*freq*t)
plt.figure(figsize = (8, 6))
plt.plot(t, x, 'r')
plt.ylabel('Amplitude')
plt.show()
```

TRY IT! Use the FFT function to calculate the Fourier transform of the above signal. Plot the amplitude spectrum for both the two-sided and one-side frequencies.

```
X=FFT(x)
# calculate the frequency
N = len(X)
n = np.arange(N)
T = N/sr
freq = n/T
plt.figure(figsize = (12, 6))
plt.subplot(121)
plt.stem(freq, abs(X), 'b', \
        markerfmt=" ", basefmt="-b")
plt.xlabel('Freq (Hz)')
plt.ylabel('FFT Amplitude |X(freq)|')
# Get the one-sided specturm
n_oneside = N//2
# get the one side frequency
f_oneside = freq[:n_oneside]
# normalize the amplitude
X_oneside =X[:n_oneside]/n_oneside
plt.subplot(122)
plt.stem(f_oneside, abs(X_oneside), 'b', \
        markerfmt=" ", basefmt="-b")
plt.xlabel('Freq (Hz)')
plt.ylabel('Normalized FFT Amplitude |X(freq)|')
plt.tight_layout()
plt.show()
```

TRY IT! Generate a simple signal for length 2048, and time how long it will run the FFT and compare the speed with the DFT.

```
def gen_sig(sr):
    '''
    function to generate
    a simple 1D signal with
    different sampling rate
    '''
    ts = 1.0/sr
    t = np.arange(0,1,ts)
    freq = 1.
    x = 3*np.sin(2*np.pi*freq*t)
    return x
# sampling rate =2048
sr = 2048
%timeit FFT(gen_sig(sr))
```

16.9 ms ± 1.3 ms per loop (mean ± std. dev. of 7 runs, 100 loops each)

We can see that, for a signal with length 2048 (about 2000), this implementation of F

Example 1:

```
# import sympy
from sympy import ifft
# sequence
seq = [15, 21, 13, 44]
# fft
transform = ifft(seq)
print ("Inverse FFT : ", transform)
```

Output:

Inverse FFT : [93/4, 1/2 + 23*I/4, -37/4, 1/2 - 23*I/4]

Example 2:

```
# import sympy
from sympy import ifft
# sequence
seq = [15, 21, 13, 44]
decimal_point = 4
# fft
transform = ifft(seq, decimal_point )
print ("Inverse FFT : ", transform)
```

Output:

```
Inverse FFT :  [23.25, 0.5 + 5.75*I, -9.250, 0.5 - 5.75*I]
```

❖❖❖❖

Experiment III

3

DISCRETE COSINE TRANSFORM

Aim: To find DCT forward and inverse transform of image.

Theory:

DCT: Discrete cosine transform.

IDCT: Inverse discrete cosine transform.

The DCT (Discrete Cosine Transform)

An explanation and illustration of the math behind the Discrete Cosine Transform and the concepts used in lossy JPEG image compression - low pass filtering.

In [23]:

 # imports

import numpy as np

from numpy import *

import matplotlib.pyplot as plt

from matplotlib.pyplot import *

import matplotlib.image as mpimg

%**matplotlib** inline

 # software versions:

 # python 3.6, numpy 1.15, matplotlib 3.0.2, Pillow 5.4.1 (python imaging library)

The basic linear algebra with N = 2

You can think of a vector - a list of numbers - as coefficients times basis vectors.

f0[10]+f1[01]f0[10]+f1[01]

Using a different basis, different coefficients can describe the same vector.

G012–√[11]+G112–√[1−1]G012[11]+G112[1−1]

(The sqrt(2)'s give the basis vectors length 1, i.e. "normalizes" them.)

This transormation **f** to **G** is a DCT (Discrete Cosine Transform). For a vector with 2 components, this perhaps isn't all that exciting, but does still transform the original (f0,f1)(f0,f1) into low and high frequency components (G0,G1)(G0,G1).

the matrix math

This transform can be written as a matrix multiplication.

f0[10]+f1[01]=[f0f1]=G012–√[11]+G112–√[1−1]=12–√[111−1][G0G1]f0[10]+f1[01]=[f0f1]=G012[11]+G112[1−1]=12[111−1][G0G1]

Moreover, this orthnormal matrix has the interesting and useful property that its transpose is its inverse. That makes the equation easy to invert.

two dimensions

The same idea can be applied to 2D images rather than 1D vectors, by applying the 1D transform to each row and column of the image.

The 2D basis images for N=2 are then the outer products of the 1D basis vectors. From lowest (0,0) to highest (1,1) spatial frequency these basis images are :

In [2]:

```
basis = (1/sqrt(2) * array([1, 1]), 1/sqrt(2) * array([1, -1]))

for i in [0,1]:
    for j in [0,1]:
        print("{}, {} :".format(i,j))
        print(outer(basis[i], basis[j]))
        print()
```

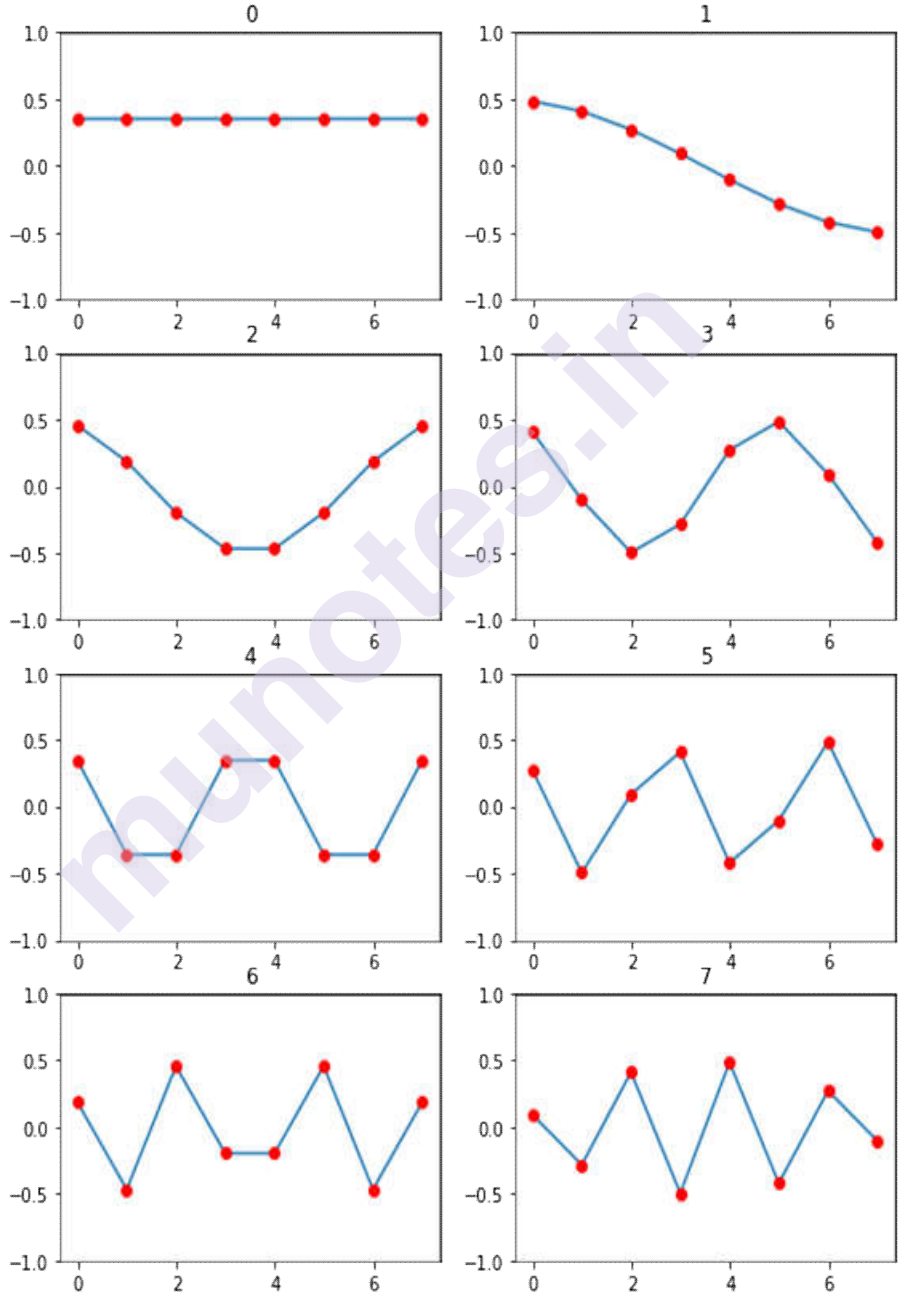
```
0, 0 :

[[0.5 0.5]
 [0.5 0.5]]

0, 1 :

[[ 0.5 -0.5]
 [ 0.5 -0.5]]

1, 0 :

[[ 0.5  0.5]
```

```
 [-0.5 -0.5]]
1, 1 :
[[ 0.5 -0.5]
 [-0.5  0.5]]
In [3]:
# The 8 x 8 DCT matrix thus looks like this.
N = 8
dct = np.zeros((N, N))
for x in range(N):
    dct[0,x] = sqrt(2.0/N) / sqrt(2.0)
for u in range(1,N):
    for x in range(N):
        dct[u,x] = sqrt(2.0/N) * cos((pi/N) * u * (x + 0.5) )
        np.set_printoptions(precision=3)
dct
Out[3]:
array([[ 0.354,  0.354,  0.354,  0.354,  0.354,  0.354,  0.354,  0.354],
       [ 0.49 ,  0.416,  0.278,  0.098, -0.098, -0.278, -0.416, -0.49 ],
       [ 0.462,  0.191, -0.191, -0.462, -0.462, -0.191,  0.191,  0.462],
       [ 0.416, -0.098, -0.49 , -0.278,  0.278,  0.49 ,  0.098, -0.416],
       [ 0.354, -0.354, -0.354,  0.354,  0.354, -0.354, -0.354,  0.354],
       [ 0.278, -0.49 ,  0.098,  0.416, -0.416, -0.098,  0.49 , -0.278],
       [ 0.191, -0.462,  0.462, -0.191, -0.191,  0.462, -0.462,  0.191],
       [ 0.098, -0.278,  0.416, -0.49 ,  0.49 , -0.416,  0.278, -0.098]])
```

The corresponding eight 1D basis vectors (the matrix rows) oscillate with successively higher spatial frequencies.

```
In [4]:
# Here's what they look like.
figure(figsize=(9,12))
for u in range(N):
```

```
subplot(4, 2, u+1)
ylim((-1, 1))
title(str(u))
plot(dct[u, :])
plot(dct[u, :],'ro')
```



Like the N=2 case, the vectors are orthnormal. In other words, their dot products are 0, and each has length 1. Here are a few illustrative products.

In [5]:

```
def rowdot(i,j):
    return dot(dct[i, :], dct[j, :])
rowdot(0,0), rowdot(3,3), rowdot(0,3), rowdot(1, 7), rowdot(1,5)
```

Out[5]:

```
(0.9999999999999998,
 0.9999999999999998,
 6.938893903907228e-17,
 1.942890293094024e-16,
 -2.498001805406602e-16)
```

This also implies the inverse of this matrix is just its transpose.

In [6]:

```
dct_transpose = dct.transpose()
dct_transpose
```

Out[6]:

```
array([[ 0.354,  0.49 ,  0.462,  0.416,  0.354,  0.278,  0.191,  0.098],
       [ 0.354,  0.416,  0.191, -0.098, -0.354, -0.49 , -0.462, -0.278],
       [ 0.354,  0.278, -0.191, -0.49 , -0.354,  0.098,  0.462,  0.416],
       [ 0.354,  0.098, -0.462, -0.278,  0.354,  0.416, -0.191, -0.49 ],
       [ 0.354, -0.098, -0.462,  0.278,  0.354, -0.416, -0.191,  0.49 ],
       [ 0.354, -0.278, -0.191,  0.49 , -0.354, -0.098,  0.462, -0.416],
       [ 0.354, -0.416,  0.191,  0.098, -0.354,  0.49 , -0.462,  0.278],
       [ 0.354, -0.49 ,  0.462, -0.416,  0.354, -0.278,  0.191, -0.098]])
```

In [7]:

```
# Is the dot product of dct and its transpose the identity?
maybe_identity = dot(dct, dct_transpose)
# Since there are many nearly zero like 3.2334e-17 in this numerical result,
# the output will look much nicer if we round them all of to (say) 6 places.
roundoff = vectorize(lambda m: round(m, 6))
```

```
roundoff(maybe_identity)

Out[7]:

array([[ 1.,  0., -0.,  0.,  0.,  0., -0., -0.],
       [ 0.,  1.,  0., -0.,  0., -0.,  0.,  0.],
       [-0.,  0.,  1.,  0., -0.,  0.,  0.,  0.],
       [ 0., -0.,  0.,  1.,  0.,  0., -0.,  0.],
       [ 0.,  0., -0.,  0.,  1.,  0., -0., -0.],
       [ 0., -0.,  0.,  0.,  0.,  1.,  0., -0.],
       [-0.,  0.,  0., -0., -0.,  0.,  1.,  0.],
       [-0.,  0.,  0.,  0., -0., -0.,  0.,  1.]])
```

playing with a real image

To make all this more concrete, let's apply the 2D DCT transform to part of a real image.

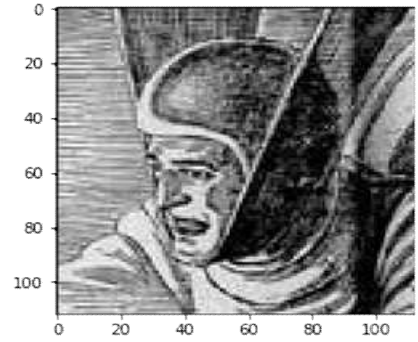
Here's one, takenly randomly from the web.

In [10]:

```
# See http://matplotlib.org/users/image_tutorial.html for the image manipulation syntax.

# The image itself is a small piece from http://www.cordwainer-smith.com/virgil_finlay.htm.

img = mpimg.imread('stormplanet112.jpg')

plt.imshow(img)
```

Out[10]:

```
<matplotlib.image.AxesImage at 0x7f9b20830da0>
```



In [11]:

 # The image itself contains 3 dimensions: rows, columns, and colors

```
img.shape
```

Out[11]:

(112, 112, 3)

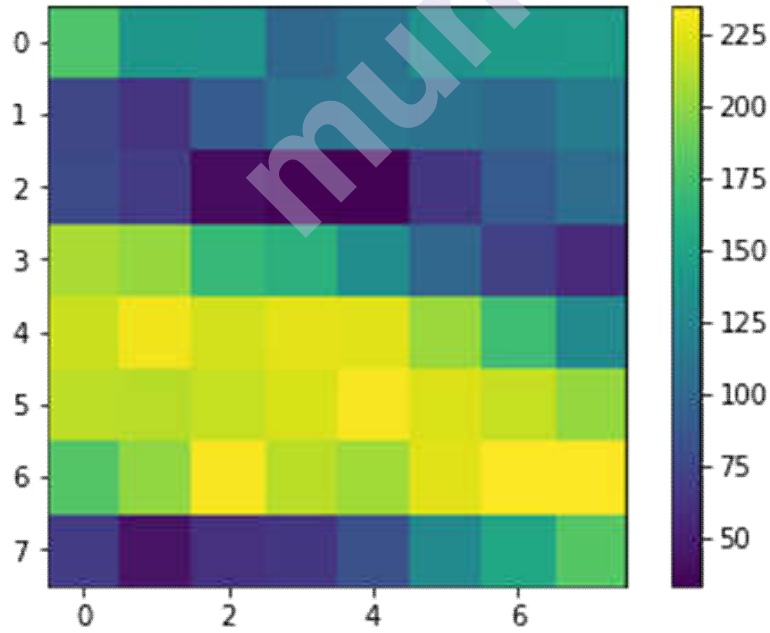
All three of the R,G,B color values in the greyscale image are the same for each pixel.

Let's just look at values from one tiny 8 x 8 block (which is what's used JPEG compression) near his nose.

(The next images use a false color spectrum to display pixel intensity.)

In [12]:

```
tiny = img[40:48, 40:48, 0]     # a tiny 8 x 8 block, in the color=0 (Red) channel

def show_image(img):
    plt.imshow(img)
    plt.colorbar()

show_image(tiny)
```



In [13]:

 # And here are the numbers.

tiny

Out[13]:

```
array([[179, 140, 138, 101, 110, 135, 143, 144],
       [ 76,  64,  91, 110, 113, 109, 104, 118],
       [ 78,  68,  40,  34,  33,  66,  90, 105],
       [209, 204, 168, 163, 132, 100,  73,  57],
       [219, 231, 221, 227, 226, 205, 172, 130],
       [215, 213, 217, 223, 232, 224, 217, 203],
       [181, 202, 233, 214, 207, 226, 235, 235],
       [ 69,  44,  62,  66,  83, 129, 153, 182]], dtype=uint8)
```

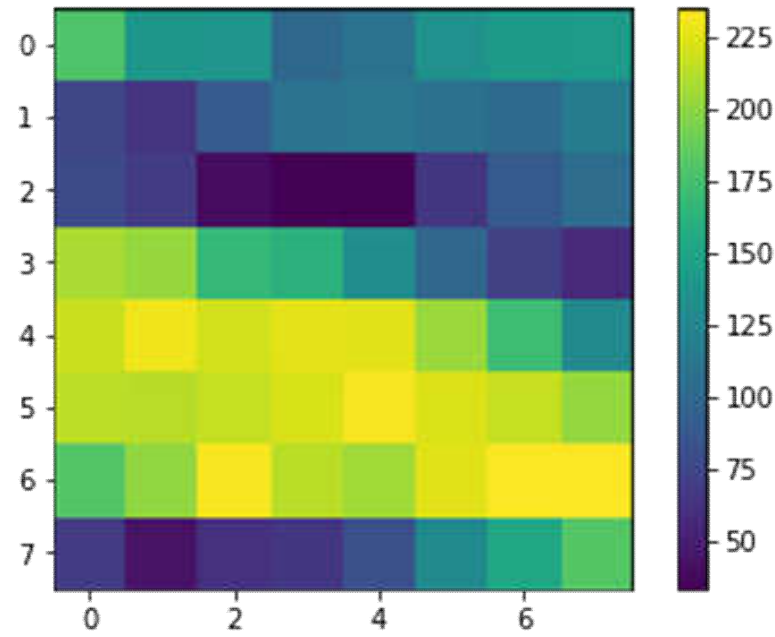
Now we define the 2D version of the N=8 DCT described above.

The trick is to apply the 1D DCT to every column, *and* then also apply it to every row, i.e.

$$G = DCT \cdot f \cdot DCTTG = DCT \cdot f \cdot DCTT$$

In [14]:

```
def doDCT(grid):
    return dot(dot(dct, grid), dct_transpose)
def undoDCT(grid):
    return dot(dot(dct_transpose, grid), dct)
# test : do DCT, then undo DCT; should get back the same image.
tiny_do_undo = undoDCT(doDCT(tiny))
show_image(tiny_do_undo) # Yup, looks the same.
```



In [15]:

 # And the numbers are the same.

tiny_do_undo

Out[15]:

array([[179., 140., 138., 101., 110., 135., 143., 144.],

[76., 64., 91., 110., 113., 109., 104., 118.],

[78., 68., 40., 34., 33., 66., 90., 105.],

[209., 204., 168., 163., 132., 100., 73., 57.],

[219., 231., 221., 227., 226., 205., 172., 130.],

[215., 213., 217., 223., 232., 224., 217., 203.],

[181., 202., 233., 214., 207., 226., 235., 235.],

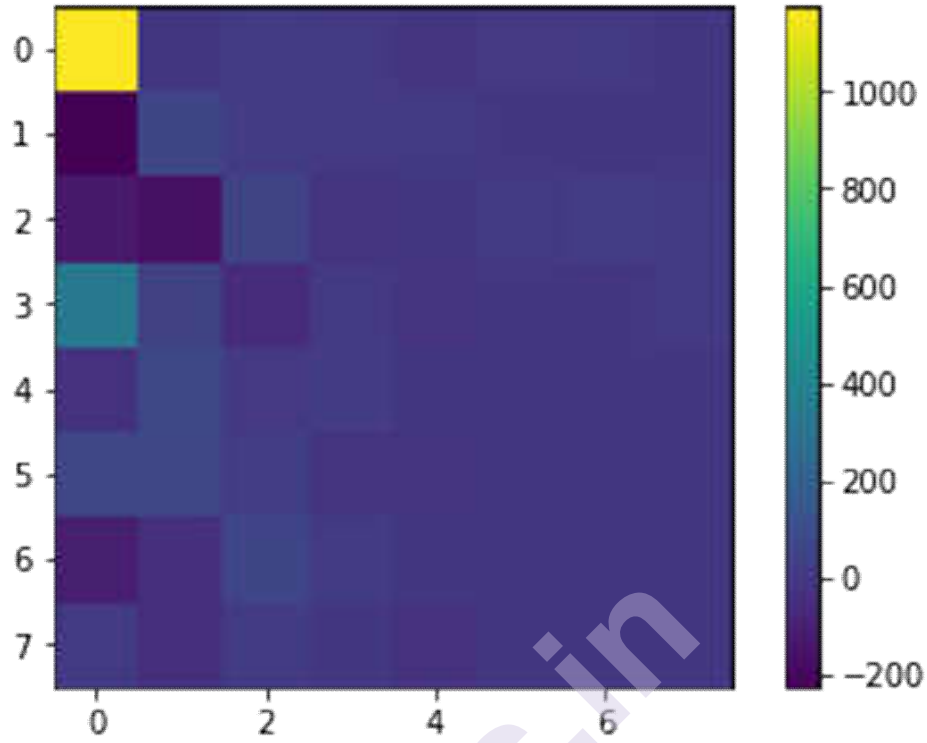
[69., 44., 62., 66., 83., 129., 153., 182.]])

The DCT transform looks like this. Note that most of the intensity is at the top left, in the lowest frequencies.

In [16]:

tinyDCT = doDCT(tiny)

show_image(tinyDCT)



In [17]:

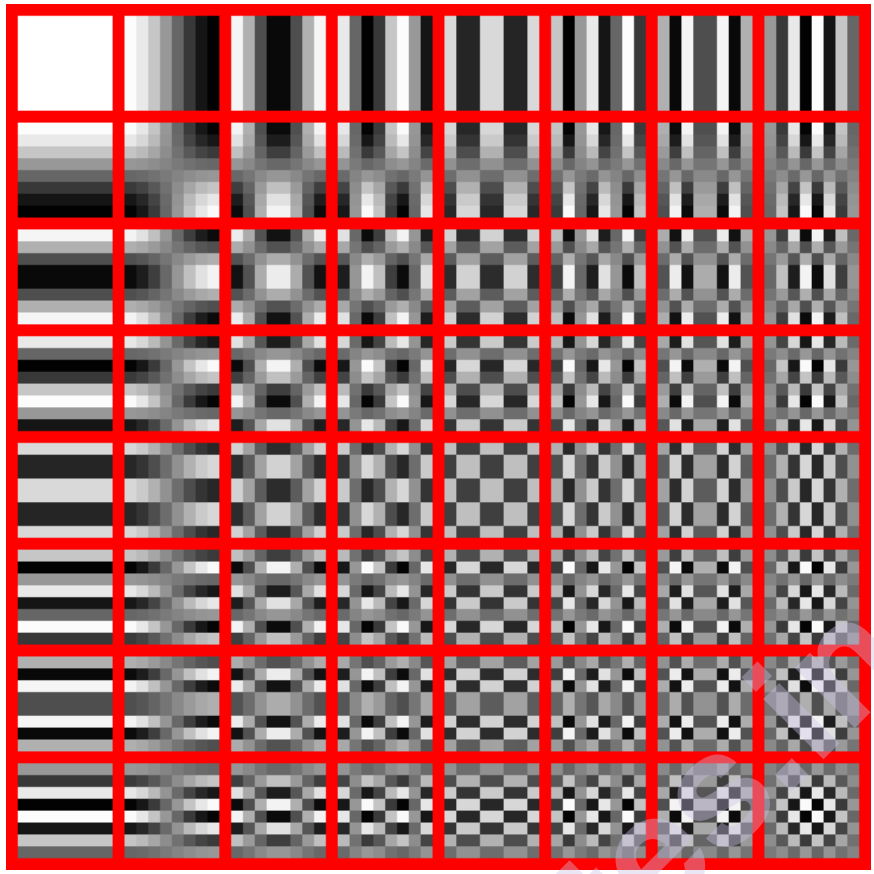
set_printoptions(linewidth=100) *# output line width (default is 75)*

round6 = vectorize(**lambda** m: '**{:6.1f}**'.format(m))

round6(tinyDCT)

Out[17]:

```
array([['1173.9', '  3.6', ' 19.8', ' 12.3', ' -5.4', '  8.2', ' 10.3', ' -0.0'],
       ['-225.9', ' 64.1', ' 24.2', ' 12.2', '  9.9', ' -0.2', '  0.0', '  0.1'],
       ['-122.7', '-161.8', ' 63.2', '-15.0', '  0.3', ' 11.1', ' 28.5', ' 10.7'],
       [' 341.9', ' 50.8', '-48.4', ' 12.0', '-10.2', ' -0.4', '  0.1', ' 12.1'],
       [' -20.1', ' 80.2', '  6.9', ' 22.1', '  0.1', ' -0.1', ' -0.0', ' -0.3'],
       [' 74.4', ' 69.9', ' 32.9', '-13.0', '-16.3', ' -0.4', ' -0.2', ' -0.0'],
       ['-100.6', '-38.9', ' 64.3', ' 17.2', ' -0.3', '  0.5', ' -0.2', ' -0.1'],
       [' 13.8', '-36.5', ' 18.5', ' -0.4', '-21.6', '  0.1', '  0.3', '  0.2']],
      dtype='<U6')
```



The grid positions in that last image correspond to spatial frequencies, with the lowest DC component at the top left, and the highest vertical and horizontal frequency at the bottom right.

These 2D basis functions can be visualized with the image shown which is from wikimedia commons.

The details of what I'm doing here don't really match the JPEG transformations: I haven't done the color space transforms, and I haven't handled the DC offsets as the JPEG spec does (which centers the values around 0 explicitly.)

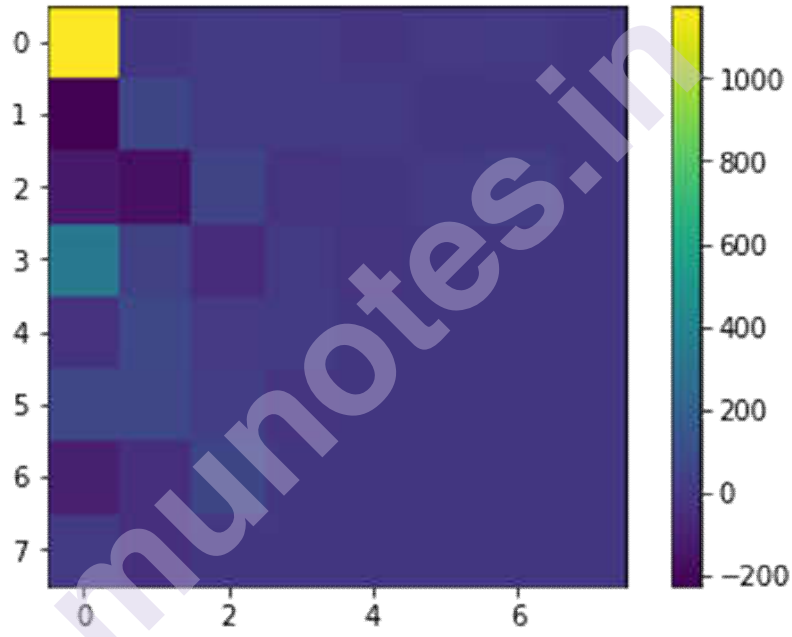
But the concept is visible in the last two pictures: after the DCT, most of the power is in fewer pixels, which are typically near the top left DC part of the grid.

So here's a simple lossy "low pass filter" of the data : let's chop some of the high frequency numbers. One (somewhat arbitrary) choice to to set the frequencies higher than the (1,7) to (7,1) line, to zero.

This is a lossy transormation since we're throwing away information - it can't be undone. But since there are fewer numbers, it's a form of compression.

In [19]:

```
# First make a copy to work on.
tinyDCT_chopped = tinyDCT.copy()
# Then zero the pieces below the x + y = 8 line.
for x in range(N):
    for u in range(N):
        if x + u > 8:
            tinyDCT_chopped[x,u] = 0.0
show_image(tinyDCT_chopped)
```



In [20]:

```
round6(tinyDCT_chopped)
```

 # Notice all the zeros at the bottom right - those are the chopped high frequences.

 # We've essentially done a "low pass filter" on the spacial frequencies.

Out[20]:

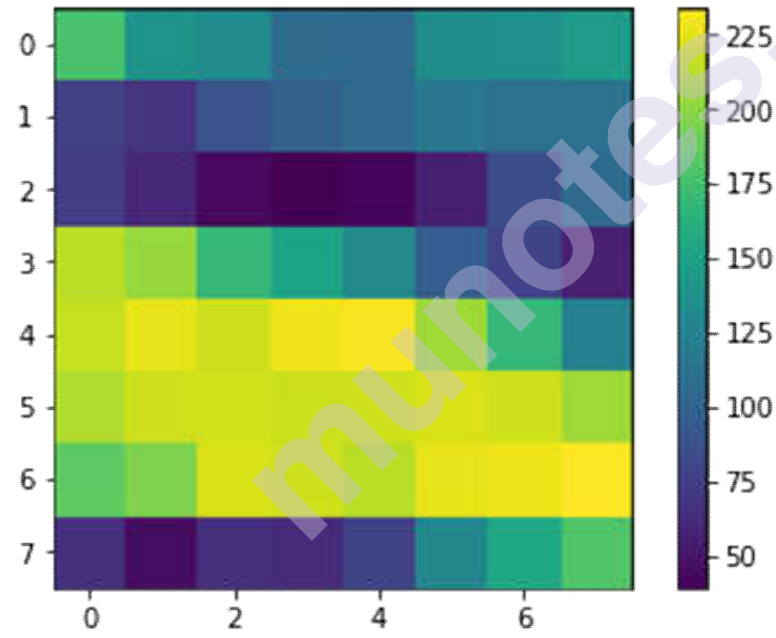
```
array([['1173.9', '  3.6', ' 19.8', ' 12.3', ' -5.4', '  8.2', ' 10.3', ' -0.0'],
       ['-225.9', ' 64.1', ' 24.2', ' 12.2', '  9.9', ' -0.2', '  0.0', '  0.1'],
       ['-122.7', '-161.8', ' 63.2', '-15.0', '  0.3', ' 11.1', ' 28.5', '  0.0'],
       [' 341.9', ' 50.8', '-48.4', ' 12.0', '-10.2', ' -0.4', '  0.0', '  0.0'],
```

```
       [' -20.1', '  80.2', '   6.9', '  22.1', '   0.1', '   0.0', '   0.0', '   0.0'],
       [' 74.4', '  69.9', '  32.9', ' -13.0', '   0.0', '   0.0', '   0.0', '   0.0'],
       ['-100.6', ' -38.9', '  64.3', '   0.0', '   0.0', '   0.0', '   0.0', '   0.0'],
       [' 13.8', ' -36.5', '   0.0', '   0.0', '   0.0', '   0.0', '   0.0', '   0.0']],
      dtype='<U6')
```

To see what this did to the original, we just transform it back.

In [21]:

```
tiny_chopped_float = undoDCT(tinyDCT_chopped)
# Also convert the floats back to uint8, which was the original format
tiny_chopped = vectorize(lambda x: uint8(x))(tiny_chopped_float)
show_image(tiny_chopped)
```



In [22]:

```
tiny_chopped
```

Out[22]:

```
array([[178, 140, 133, 109, 107, 135, 137, 147],
       [ 76,  69,  90, 100, 107, 117, 110, 112],
       [ 75,  61,  44,  39,  42,  56,  86, 107],
       [214, 204, 169, 152, 131,  97,  78,  57],
```

[217, 227, 220, 230, 233, 206, 169, 125],

[211, 220, 221, 219, 220, 223, 220, 206],

[186, 196, 223, 220, 214, 227, 229, 234],

[66, 46, 65, 63, 79, 129, 155, 181]], dtype=uint8)

And we have something close to the original back again - even though close to half of the transformed image was set to zero.

conclusions

The procedue here isn't what happens in JPEG compression, but does illustrate one of the central concepts - throwing away some of higher spatial frequency information after a DCT transform.

In the real JPEG lossy compression algorithm, the steps are

- the color space is transformed from R,G,B to Y,Cb,Cr to take advantage of human visual prejudices
- the values are shifted so that they center around zero
- the values after the DCT are "quantized" (i.e. rounded off) by different amounts at different spots in the grid. *(*This* is the lossy step, and how lossy depends on the JPEG quality.)
- a zigzag (keeping similar frequencies together) pattern turns this to a 1D stream of 64 values
- which are then huffman encoded by, typically by a pre-chosen code (part of the JPEG standard

❖❖❖❖

Experiment IV

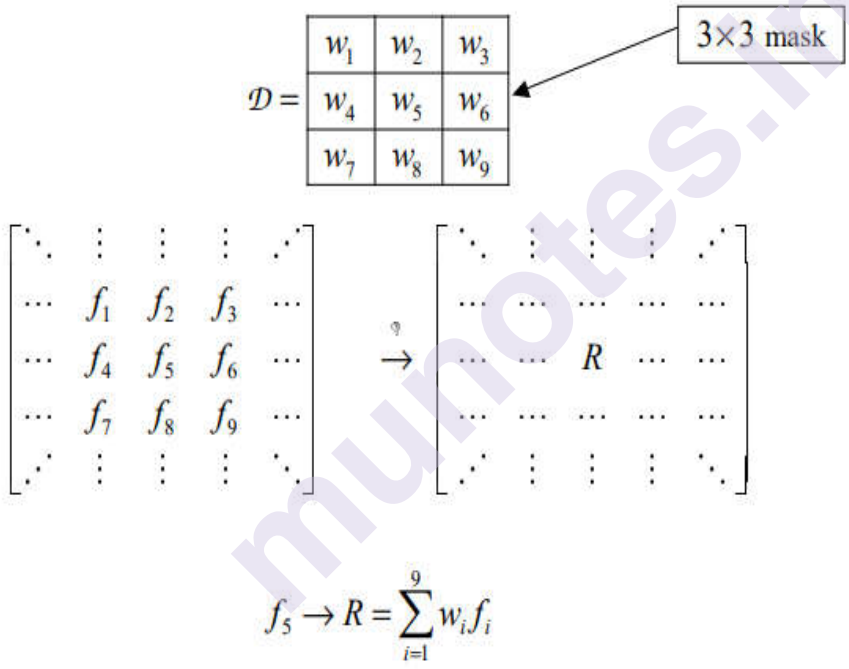
4

IMAGE SEGMENTATION AND IMAGE RESTORATION

Aim: The detection of discontinuities – Point, Line and Edge detections, Hough transform, Thresholding, Region based segmentation chain codes.

Theory:

• This is usually accomplished by applying a suitable mask to the image.



$$f_5 \to R = \sum_{i=1}^{9} w_i f_i$$

- The mask output or response at each pixel is computed by centering centering the mask on the pixel location.
- When the mask is centered at a point on the image boundary, the mask response or output is computed using suitable boundary condition. Usually, the mask is truncated.

Point Detection

This is used to detect isolated spots in an image.

- The graylevel of an isolated point will be very different from its neighbors.
- It can be accomplished using the following 3×3 mask:

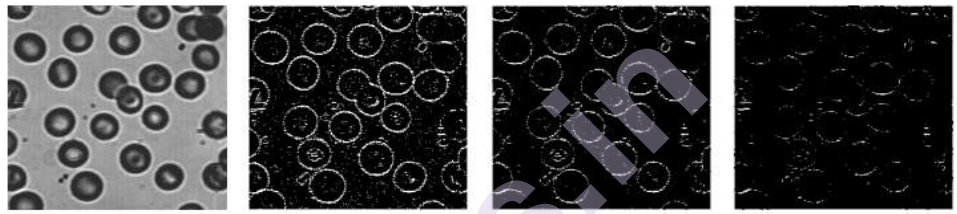
$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

The output of the mask operation is usually thresholded.

• We say that an isolated point has been detected if

$$|8f_5 - (f_1 + f_2 + f_3 + f_4 + f_6 + f_7 + f_8 + f_9)| > T$$

for some pre-specified non-negative threshold T.



Orig. Image $T = 0.25$ $T = 0.35$ $T = 0.5$

Detection of lines

• This is used to detect lines in an image.

• It can be done using the following four masks:

$$\mathcal{D}_{0^\circ} = \begin{bmatrix} -1 & -1 & -1 \\ 2 & 2 & 2 \\ -1 & -1 & -1 \end{bmatrix}$$ Detects horizontal lines

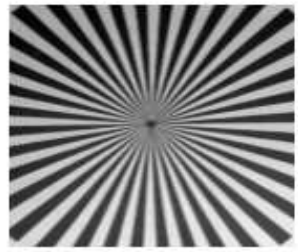
$$\mathcal{D}_{45^\circ} = \begin{bmatrix} -1 & -1 & 2 \\ -1 & 2 & -1 \\ 2 & -1 & -1 \end{bmatrix}$$ Detects 45° lines

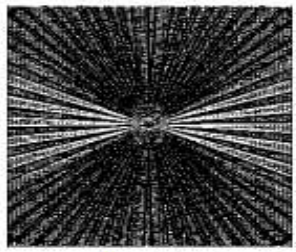
$$\mathcal{D}_{90^\circ} = \begin{bmatrix} -1 & 2 & -1 \\ -1 & 2 & -1 \\ -1 & 2 & -1 \end{bmatrix}$$ Detects vertical lines

$$\mathcal{D}_{135^\circ} = \begin{bmatrix} 2 & -1 & -1 \\ -1 & 2 & -1 \\ -1 & -1 & 2 \end{bmatrix}$$ Detects 135° lines

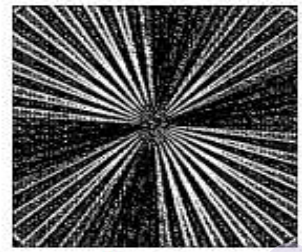
- Let 0 R , 45 R , 90 R , and 135 R , respectively be the response to masks 0 D , 45 D , 90 D , and 135 D , respectively. At a given pixel

(m,n), if 135 R is the maximum among { 0 R , 45 R , 90 R , 135 R }, we say that a 135 line is most likely passing through that pixel

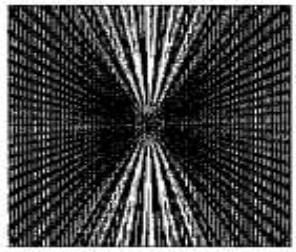




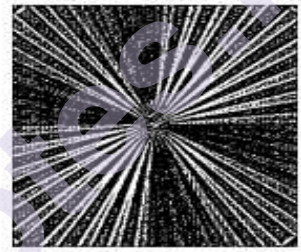
$R_{0'} = \max\{R_{0'}, R_{45'}, R_{90'}, R_{135'}\}$



$R_{45'} = \max\{R_{0'}, R_{45'}, R_{90'}, R_{135'}\}$



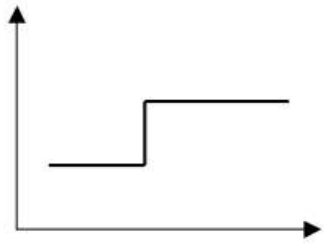
$R_{90'} = \max\{R_{0'}, R_{45'}, R_{90'}, R_{135'}\}$



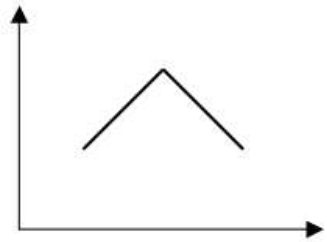
$R_{135'} = \max\{R_{0'}, R_{45'}, R_{90'}, R_{135'}\}$

Edge Detection

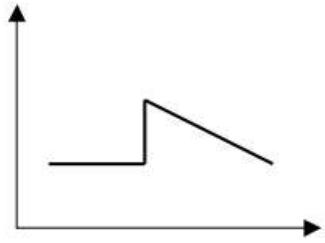
- Isolated points and thin lines do not occur frequently in most practical applications.
- For image segmentation, we are mostly interested in detecting the boundary between two regions with relatively distinct gray-level properties.
- We assume that the regions in question are sufficiently homogeneous so that the transition between two regions can be determined on the basis of gray-level discontinuities alone.
- An edge in an image may be defined as a discontinuity or abrupt change in gray level.



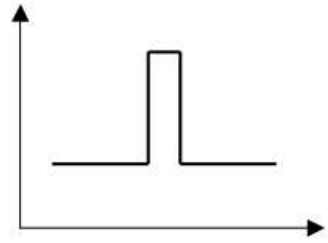
Ideal Step Edge



Ideal Roof Edge

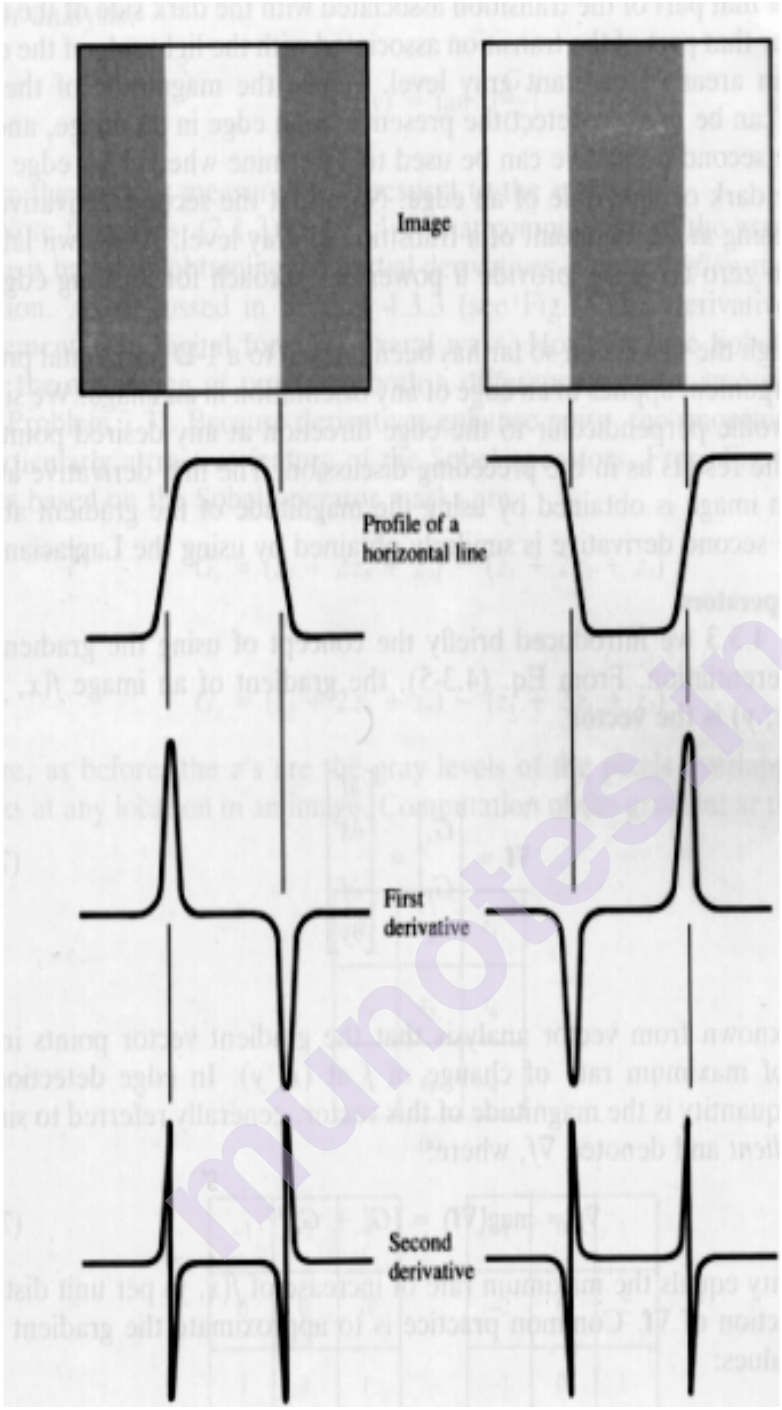


Combination of Step and Roof Edges



Ideal Spike Edge

- These are ideal situations that do not frequently occur in practice. Also, in two dimensions edges may occur at any orientation.
- Edges may not be represented by perfect discontinuities. Therefore, the task of edge detection is much more difficult than what it looks like.
- A useful mathematical tool for developing edge detectors is the first and second derivative operators.



- From the example above, it is clear that the magnitude of the first derivative can be used to detect the presence of an edge in an image.
- The sign of the second derivative can be used to determine whether an edge pixel lies on the dark or light side of an edge.
- The zero crossings of the second derivative provide a powerful way of locating edges in an image.
- We would like to have small-sized masks in order to detect fine variation in graylevel distribution (i.e., micro-edges). •

- On the other hand, we would like to employ large-sized masks in order to detect coarse variation in graylevel distribution (i.e., macro-edges) and filter-out noise and other irregularities.
- We therefore need to find a mask size, which is a compromise between these two opposing requirements, or determine edge content by using different mask sizes
- Most common differentiation operator is the gradient.

$$\nabla f(x,y) = \begin{bmatrix} \dfrac{\partial f(x,y)}{\partial x} \\ \dfrac{\partial f(x,y)}{\partial y} \end{bmatrix}$$

• The magnitude of the gradient is:

$$|\nabla f(x,y)| = \left[\left(\frac{\partial f(x,y)}{\partial x} \right)^2 + \left(\frac{\partial f(x,y)}{\partial y} \right)^2 \right]^{1/2}$$

• The direction of the gradient is given by:

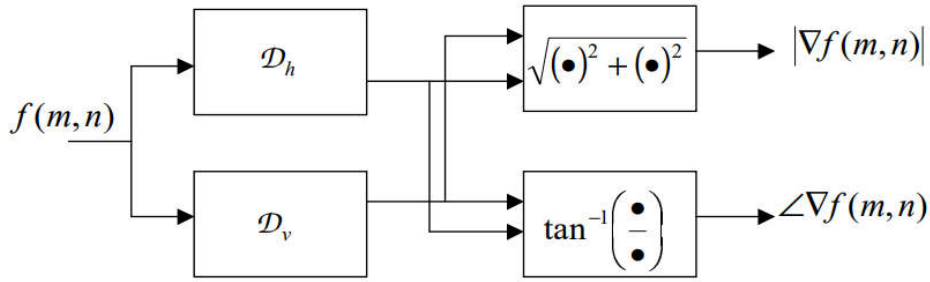
$$\angle \nabla f(x,y) = \tan^{-1} \left\{ \frac{\partial f}{\partial y} \Big/ \frac{\partial f}{\partial x} \right\}$$

• In practice, we use discrete approximations of the partial derivatives $\partial f/\partial x$ and $\partial f/\partial y$, which are implemented using the masks:

$$\mathcal{D}_h = \frac{1}{2}\{[-1 \quad 1 \quad 0] + [0 \quad -1 \quad 1]\} = \frac{1}{2}[-1 \quad 0 \quad 1]$$

$$\mathcal{D}_v = \frac{1}{2}\left\{ \begin{bmatrix} -1 \\ 1 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ -1 \\ 1 \end{bmatrix} \right\} = \frac{1}{2}\begin{bmatrix} -1 \\ 0 \\ 1 \end{bmatrix}$$

• The gradient can then be computed as follows:



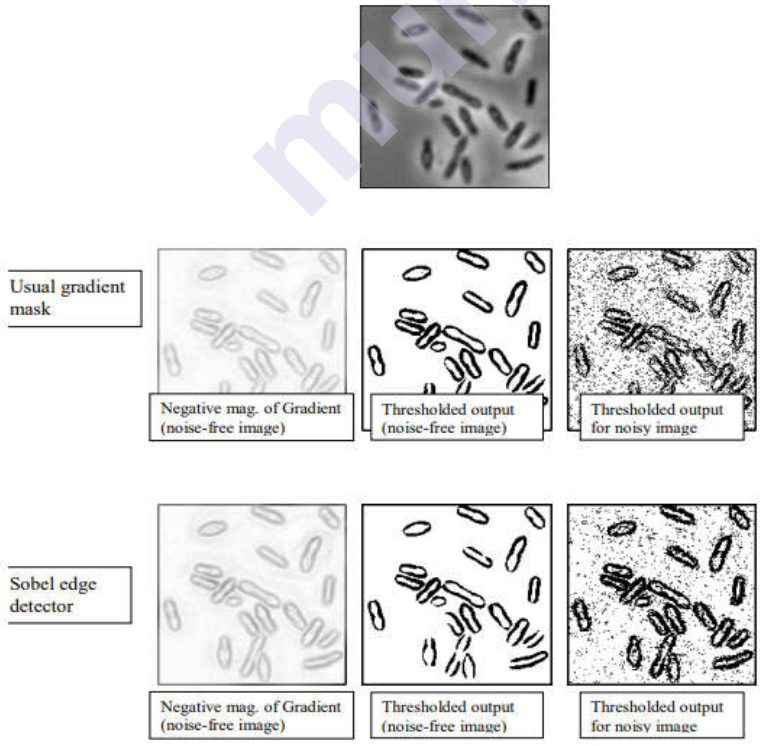
- Other discrete approximations to the gradient (more precisely, the appropriate partial derivatives) have been proposed (Roberts, Prewitt).
- Because derivatives enhance noise, the previous operators may not give good results if the input image is very noisy.
- One way to combat the effect of noise is by applying a smoothing mask. The Sobel edge detector combines this smoothing operation along with the derivative operation give the following masks:

Since the gradient edge detection methodology depends only on the relative magnitudes within an image, scalar multiplication by factors such as 1/2 or 1/8 play no essential role. The same is true for the signs of the mask entries. Therefore, masks like correspond to the same detector, namely the Sobel edge detector.

$$\frac{1}{8}\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \quad \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} \quad \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

- However, when the exact magnitude is important, the proper scalar multiplication factor should be used.
- All masks considered so far have entries that add up to zero. This is typical of any derivative mask.

Example:



The code will only compile in linux environment. Make sure that openCV is installed in your system before you run the program.

Steps to download the requirements below:

- Run the following command on your terminal to install it from the Ubuntu or Debian repository.
- sudo apt-get install libopencv-dev python-opencv
- OR In order to download OpenCV from the official site run the following command:
- bash install-opencv.sh
- on your terminal.
- Type your sudo password and you will have installed OpenCV.

Principle behind Edge Detection

Edge detection involves mathematical methods to find points in an image where the brightness of pixel intensities changes distinctly.

- The first thing we are going to do is find the **gradient** of the grayscale image, allowing us to find edge-like regions in the x and y direction. The gradient is a multi-variable generalization of the derivative. While a derivative can be defined on functions of a single variable, for functions of several variables, the gradient takes its place.
- The gradient is a vector-valued function, as opposed to a derivative, which is scalar-valued. Like the derivative, **the gradient represents the slope of the tangent of the graph of the function**. More precisely, the gradient points in the direction of the greatest rate of increase of the function, and its magnitude is the slope of the graph in that direction.

Note: In computer vision, transitioning from black-to-white is considered a positive slope, whereas a transition from white-to-black is a negative slope.

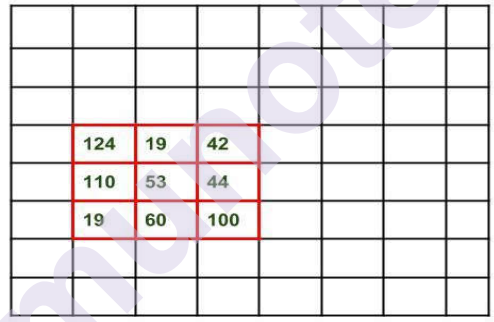
```
# Python program to  Edge detection
# using OpenCV in Python
# using Sobel edge detection
# and laplacian method
import cv2
import numpy as np

#Capture livestream video content from camera 0
cap = cv2.VideoCapture(0)

while(1):

    # Take each frame
    _, frame = cap.read()

    # Convert to HSV for simpler calculations
    hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)

    # Calculation of Sobelx
    sobelx = cv2.Sobel(frame,cv2.CV_64F,1,0,ksize=5)

    # Calculation of Sobely
    sobely = cv2.Sobel(frame,cv2.CV_64F,0,1,ksize=5)

    # Calculation of Laplacian
    laplacian = cv2.Laplacian(frame,cv2.CV_64F)

    cv2.imshow('sobelx',sobelx)
    cv2.imshow('sobely',sobely)
    cv2.imshow('laplacian',laplacian)
    k = cv2.waitKey(5) & 0xFF
    if k == 27:
        break

cv2.destroyAllWindows()

#release the frame
cap.release()
```

Calculation of the derivative of an image

A digital image is represented by a matrix that stores the RGB/BGR/HSV(whichever color space the image belongs to) value of each pixel in rows and columns.

The derivative of a matrix is calculated by an operator called the **Laplacian**. In order to calculate a Laplacian, you will need to calculate first two derivatives, called derivatives of **Sobel**, each of which takes into account the gradient variations in a certain direction: one horizontal, the other vertical.

- **Horizontal Sobel derivative (Sobel x)**: It is obtained through the convolution of the image with a matrix called kernel which has always odd size. The kernel with size 3 is the simplest case.
- **Vertical Sobel derivative (Sobel y)**: It is obtained through the convolution of the image with a matrix called kernel which has always odd size. The kernel with size 3 is the simplest case.
- **Convolution** is calculated by the following method: Image represents the original image matrix and filter is the kernel matrix.



0	-2	0
-2	11	-2
0	-2	0

- **Factor** = 11 – 2- 2- 2- 2- 2 = 3
  Offset = 0
  Weighted Sum = 124*0 + 19*(-2) + 110*(-2) + 53*11 + 44*(-2) + 19*0 + 60*(-2) + 100*0 = 117
  O[4,2] = (117/3) + 0 = 39

  So in the end to get the Laplacian (approximation) we will need to combine the two previous results (Sobelx and Sobely) and store it in laplacian.

Parameters:

- **cv2.Sobel():** The function cv2.Sobel(frame,cv2.CV_64F,1,0,ksize=5) can be written as cv2.Sobel (original_image, ddepth, xorder, yorder, kernelsize)
- where the first parameter is the original image, the second parameter is the depth of the destination image. When ddepth=-1/CV_64F, the destination image will have the same depth as the source. The third parameter is the order of the derivative x. The fourth parameter is the order of the derivative y. While calculating Sobelx we will set xorder as 1 and yorder as 0 whereas while calculating Sobely, the case will be reversed. The last parameter is the size of the extended Sobel kernel; it must be 1, 3, 5, or 7.
- **cv2.Laplacian**: In the function

  cv2.Laplacian(frame,cv2.CV_64F)
- the first parameter is the original image and the second parameter is the depth of the destination image.When depth=-1/CV_64F, the destination image will have the same depth as the source.

Edge Detection Applications

Reduce unnecessary information in an image while preserving the structure of image.

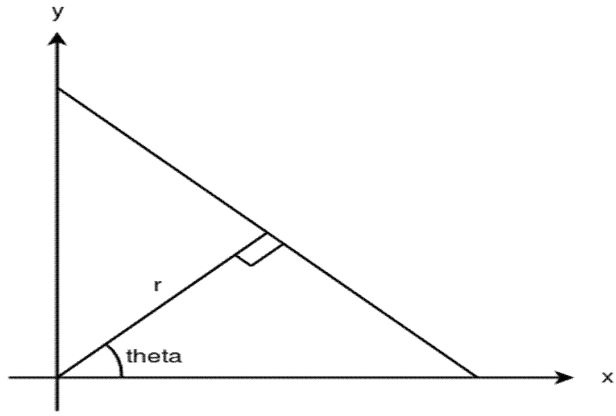
- Extract important features of image like curves, corners and lines.
- Recognizes objects, boundaries and segmentation.
- Plays a major role in computer vision and recognition

Line detection in python with OpenCV | Houghline method

The Hough Transform is a method that is used in image processing to detect any shape, if that shape can be represented in mathematical form. It can detect the shape even if it is broken or distorted a little bit. We will see how Hough transform works for line detection using the HoughLine transform method. To apply the Houghline method, first an edge detection of the specific image is desirable. For the edge detection technique go through the article Edge detection

Basics of Houghline Method

A line can be represented as $y = mx + c$ or in parametric form, as $r = x\cos\theta + y\sin\theta$ where r is the perpendicular distance from origin to the line, and θ is the angle formed by this perpendicular line and horizontal axis measured in counter-clockwise (That direction varies on how you represent the coordinate system. This representation is used in OpenCV).



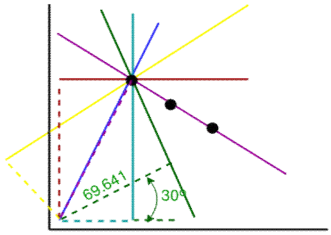
So Any line can be represented in these two terms, (r, θ).

Working of Houghline method:

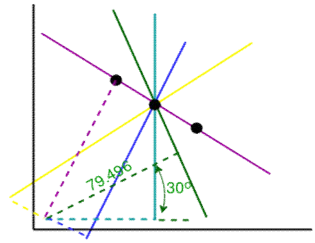
- First it creates a 2D array or accumulator (to hold values of two parameters) and it is set to zero initially.
- Let rows denote the r and columns denote the (θ)theta.
- Size of array depends on the accuracy you need. Suppose you want the accuracy of angles to be 1 degree, you need 180 columns(Maximum degree for a straight line is 180).
- For r, the maximum distance possible is the diagonal length of the image. So taking one pixel accuracy, number of rows can be diagonal length of the image.

Example:
Consider a 100×100 image with a horizontal line at the middle. Take the first point of the line. You know its (x,y) values. Now in the line equation, put the values θ(theta) = 0,1,2,….,180 and check the r you get. For every (r, 0) pair, you increment value by one in the accumulator in its corresponding (r,0) cells. So now in accumulator, the cell (50,90) = 1 along with some other cells.

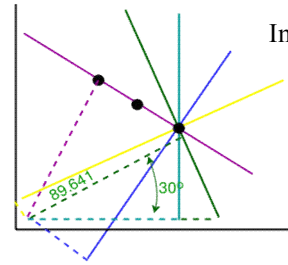
Now take the second point on the line. Do the same as above. Increment the values in the cells corresponding to (r,0) you got. This time, the cell (50,90) = 2. We are actually voting the (r,0) values. You continue this process for every point on the line. At each point, the cell (50,90) will be incremented or voted up, while other cells may or may not be voted up. This way, at the end, the cell (50,90) will have maximum votes. So if you search the accumulator for maximum votes, you get the value (50,90) which says, there is a line in this image at distance 50 from origin and at angle 90 degrees.



Angle	Dist.
0	40
30	69.6
60	81.2
120	40.6
150	0.4



Angle	Dist.
0	57.1
30	79.5
60	80.5
120	23.4
150	-19.5



Angle	Dist.
0	74.6
30	89.6
60	80.6
120	6.0
150	-39.6

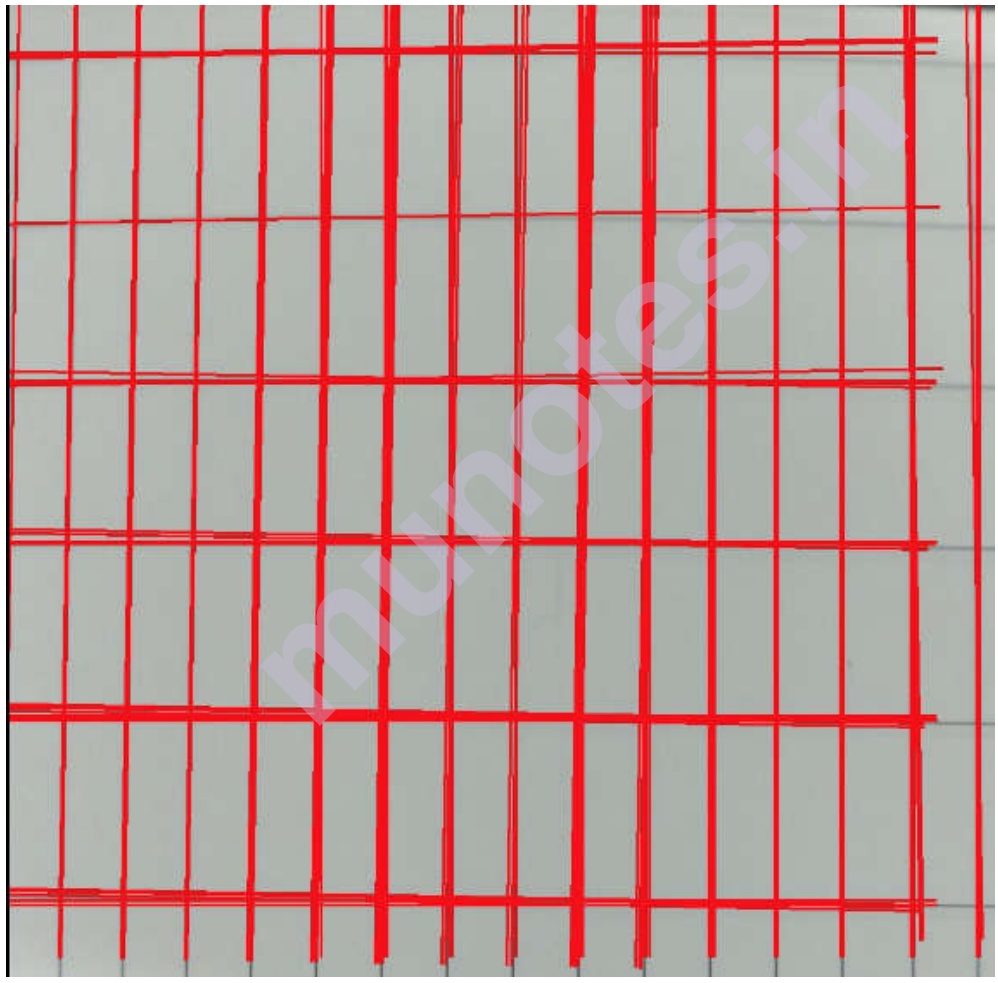
Everything explained above is encapsulated in the OpenCV function, cv2.HoughLines(). It simply returns an array of (r, 0) values. r is measured in pixels and 0 is measured in radians.

```
# Python program to illustrate HoughLine
# method for line detection
import cv2
import numpy as np
# Reading the required image in
# which operations are to be done.
# Make sure that the image is in the same
# directory in which this python program is
img = cv2.imread('image.jpg')

# Convert the img to grayscale
gray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)

# Apply edge detection method on the image
edges = cv2.Canny(gray,50,150,apertureSize = 3)

# This returns an array of r and theta values
lines = cv2.HoughLines(edges,1,np.pi/180, 200)
```

```
 # The below for loop runs till r and theta values
# are in the range of the 2d array
for r_theta in lines[0]:
    r,theta = r_theta[0]
    # Stores the value of cos(theta) in a
    a = np.cos(theta)
     # Stores the value of sin(theta) in b
    b = np.sin(theta)
        # x0 stores the value rcos(theta)
    x0 = a*r
        # y0 stores the value rsin(theta)
    y0 = b*r
        # x1 stores the rounded off value of (rcos(theta)-1000sin(theta))
    x1 = int(x0 + 1000*(-b))
        # y1 stores the rounded off value of (rsin(theta)+1000cos(theta))
    y1 = int(y0 + 1000*(a))
     # x2 stores the rounded off value of (rcos(theta)+1000sin(theta))
    x2 = int(x0 - 1000*(-b))
        # y2 stores the rounded off value of (rsin(theta)-1000cos(theta))
    y2 = int(y0 - 1000*(a))
        # cv2.line draws a line in img from the point(x1,y1) to (x2,y2).
    # (0,0,255) denotes the colour of the line to be
    #drawn. In this case, it is red.
    cv2.line(img,(x1,y1), (x2,y2), (0,0,255),2)
     # All the changes made in the input image are finally
# written on a new image houghlines.jpg
cv2.imwrite('linesDetected.jpg', img)
```

Elaboration of function(cv2.HoughLines (edges,1,np.pi/180, 200)):

1. First parameter, Input image should be a binary image, so apply threshold edge detection before finding applying hough transform.
2. Second and third parameters are r and θ(theta) accuracies respectively.
3. Fourth argument is the threshold, which means minimum vote it should get for it to be considered as a line.
4. Remember, number of votes depend upon number of points on the line. So it represents the minimum length of line that should be detected.



Alternate simpler method for directly extracting points:

```
Python3import cv2
import numpy as np
 # Read image
image = cv2.imread('path/to/image.png')
 # Convert image to grayscale
gray = cv2.cvtColor(image,cv2.COLOR_BGR2GRAY)
 # Use canny edge detection
edges = cv2.Canny(gray,50,150,apertureSize=3)
 # Apply HoughLinesP method to
# to directly obtain line end points
lines = cv2.HoughLinesP(
        edges, # Input edge image
        1, # Distance resolution in pixels
        np.pi/180, # Angle resolution in radians
        threshold=100, # Min number of votes for valid line
        minLineLength=5, # Min allowed length of line
        maxLineGap=10 # Max allowed gap between line for joining them
        )
 # Iterate over points
for points in lines:
     # Extracted points nested in the list
   x1,y1,x2,y2=points[0]
   # Draw the lines joing the points
   # On the original image
   cv2.line(image,(x1,y1),(x2,y2),(0,255,0),2)
   # Maintain a simples lookup list for points
   lines_list.append([(x1,y1),(x2,y2)])

# Save the result image
cv2.imwrite('detectedLines.png',image)
```

Summarizing the process

In an image analysis context, the coordinates of the point(s) of edge segments (i.e. X,Y) in the image are known and therefore serve as constants in the parametric line equation, while R(rho) and Theta(θ) are the unknown variables we seek.

- If we plot the possible (r) values defined by each (theta), points in cartesian image space map to curves (i.e. sinusoids) in the polar Hough parameter space. This point-to-curve transformation is the Hough transformation for straight lines.
- The transform is implemented by quantizing the Hough parameter space into finite intervals or accumulator cells. As the algorithm runs, each (X,Y) is transformed into a discretized (r,0) curve and the accumulator(2D array) cells which lie along this curve are incremented.
- Resulting peaks in the accumulator array represent strong evidence that a corresponding straight line exists in the image.

Applications of Hough transform:

1. It is used to isolate features of a particular shape within an image.
2. Tolerant of gaps in feature boundary descriptions and is relatively unaffected by image noise.
3. Used extensively in barcode scanning, verification and recognition

Thresholding techniques using OpenCV

Thresholding is a technique in OpenCV, which is the assignment of pixel values in relation to the threshold value provided. In thresholding, each pixel value is compared with the threshold value. If the pixel value is smaller than the threshold, it is set to 0, otherwise, it is set to a maximum value (generally 255). Thresholding is a very popular segmentation technique, used for separating an object considered as a foreground from its background. A threshold is a value which has two regions on its either side i.e. below the threshold or above the threshold.

In Computer Vision, this technique of thresholding is done on grayscale images. So initially, the image has to be converted in grayscale color space.

If $f(x, y) < T$

then $f(x, y) = 0$

else

$f(x, y) = 255$

where

f (x, y) = Coordinate Pixel Value

T = Threshold Value.

In Open CV with Python, the function **cv2.threshold** is used for thresholding.

Syntax: *cv2.threshold(source, threshold Value, max Val, thresholding Technique)*

Parameters:
*-> **source**: Input Image array (must be in Grayscale).*
*-> **thresholdValue**: Value of Threshold below and above which pixel values will change accordingly.*

*-> **maxVal**: Maximum value that can be assigned to a pixel.*
*-> **thresholdingTechnique**: The type of thresholding to be applied.*

Simple Thresholding

The basic Thresholding technique is Binary Thresholding. For every pixel, the same threshold value is applied. If the pixel value is smaller than the threshold, it is set to 0, otherwise, it is set to a maximum value. The different Simple Thresholding Techniques are:

- **cv2.THRESH_BINARY**: If pixel intensity is greater than the set threshold, value set to 255, else set to 0 (black).
- **cv2.THRESH_BINARY_INV**: Inverted or Opposite case of cv2.THRESH_BINARY.
- **cv.THRESH_TRUNC**: If pixel intensity value is greater than threshold, it is truncated to the threshold. The pixel values are set to be the same as the threshold. All other values remain the same.
- **cv.THRESH_TOZERO**: Pixel intensity is set to 0, for all the pixels intensity, less than the threshold value.
- **cv.THRESH_TOZERO_INV**: Inverted or Opposite case of cv2.THRESH_TOZERO.

Binary

$$\texttt{dst}(x,y) = \begin{cases} \texttt{maxval} & \text{if } \texttt{src}(x,y) > \texttt{thresh} \\ 0 & \text{otherwise} \end{cases}$$

Inverted Binary

$$\texttt{dst}(x,y) = \begin{cases} 0 & \text{if } \texttt{src}(x,y) > \texttt{thresh} \\ \texttt{maxval} & \text{otherwise} \end{cases}$$

Truncated

$$\texttt{dst}(x,y) = \begin{cases} \texttt{threshold} & \text{if } \texttt{src}(x,y) > \texttt{thresh} \\ \texttt{src}(x,y) & \text{otherwise} \end{cases}$$

To Zero

$$\texttt{dst}(x,y) = \begin{cases} \texttt{src}(x,y) & \text{if } \texttt{src}(x,y) > \texttt{thresh} \\ 0 & \text{otherwise} \end{cases}$$

To Zero Inverted

$$\texttt{dst}(x,y) = \begin{cases} 0 & \text{if } \texttt{src}(x,y) > \texttt{thresh} \\ \texttt{src}(x,y) & \text{otherwise} \end{cases}$$

Below is the Python code explaining different Simple Thresholding Techniques –

- Python3
  - # Python program to illustrate
  - # simple thresholding type on an image
  -
  - # organizing imports
  - import cv2
  - import numpy as np
  -
  - # path to input image is specified and
  - # image is loaded with imread command
  - image1 = cv2.imread('input1.jpg')
  -
  - # cv2.cvtColor is applied over the
  - # image input with applied parameters
  - # to convert the image in grayscale
  - img = cv2.cvtColor(image1, cv2.COLOR_BGR2GRAY)

```
# applying different thresholding
# techniques on the input image
# all pixels value above 120 will
# be set to 255
ret, thresh1 = cv2.threshold(img, 120, 255, cv2.THRESH_BINARY)
ret, thresh2 = cv2.threshold(img, 120, 255, cv2.THRESH_BINARY_INV)
ret, thresh3 = cv2.threshold(img, 120, 255, cv2.THRESH_TRUNC)
ret, thresh4 = cv2.threshold(img, 120, 255, cv2.THRESH_TOZERO)
ret, thresh5 = cv2.threshold(img, 120, 255, cv2.THRESH_TOZERO_INV)

# the window showing output images
# with the corresponding thresholding
# techniques applied to the input images
cv2.imshow('Binary Threshold', thresh1)
cv2.imshow('Binary Threshold Inverted', thresh2)
cv2.imshow('Truncated Threshold', thresh3)
cv2.imshow('Set to 0', thresh4)
cv2.imshow('Set to 0 Inverted', thresh5)

# De-allocate any associated memory usage
if cv2.waitKey(0) & 0xff == 27:
    cv2.destroyAllWindows()
```


Region and Edge Based Segmentation

Segmentation

Segmentation is the separation of one or more regions or objects in an image based on a discontinuity or a similarity criterion. A region in an image can be defined by its border (edge) or its interior, and the two representations are equal. There are prominently three methods of performing segmentation:

- Pixel Based Segmentation
- Region-Based Segmentation
- Edges based segmentation

Edges based segmentation

Edge-based segmentation contains 2 steps:

- **Edge Detection:** In edge detection, we need to find the pixels that are edge pixels of an object. There are many object detection methods such as Sobel operator, Laplace operator, Canny, etc.

1 0 -1

2 0 -2

1 0 -1

Sobel vertical Operator

+1 2 1

0 0 0

-1 -2 -1

Sobel Horizontal Operator

0 -1 0

-1 4 -1

0 -1 0

Negative Laplace Operator

- **Edge Linking:** In this step, we try to refine the edge detection by linking the adjacent edges and combine to form the whole object. The edge linking can be performed using any of the two methods below:
  - Local Processing: In this method, we used gradient and direction to link the neighborhood edges. If two edges have a similar direction vector then they can be linked.
  - Global processing: This method can be done using HOG transformation

- **Pros** :
  - This approach is similar to how the humans brain approaches the segmentation task.
  - Works well in images with good contrast between object and background.
- **Limitations:**
  - Does not work well on images with smooth transitions and low contrast.
  - Sensitive to noise.
  - Robust edge linking is not trivial and easy to perform.

Region-Based Segmentation

In this segmentation, we grow regions by recursively including the neighboring pixels that are similar and connected to the seed pixel. We use similarity measures such as differences in gray levels for regions with homogeneous gray levels. We use connectivity to prevent connecting different parts of the image.

There are two variants of region-based segmentation:

- **Top-down approach**
  - First, we need to define the predefined seed pixel. Either we can define all pixels as seed pixels or randomly chosen pixels. Grow regions until all pixels in the image belongs to the region.
- **Bottom-Up approach**
  - Select seed only from objects of interest. Grow regions only if the similarity criterion is fulfilled.
- **Similarity Measures:**
  - Similarity measures can be of different types: For the grayscale image the similarity measure can be the different textures and other spatial properties, intensity difference within a region or the distance b/w mean value of the region.
- **Region merging techniques:**
  - In the region merging technique, we try to combine the regions that contain the single object and separate it from the background.. There are many regions merging techniques such as Watershed algorithm, Split and merge algorithm, etc**.**

- **Pros:**
  - Since it performs simple threshold calculation, it is faster to perform.
  - Region-based segmentation works better when the object and background have high contrast.
- **Limitations:**
  - It did not produce many accurate segmentation results when there are no significant differences b/w pixel values of the object and the background.

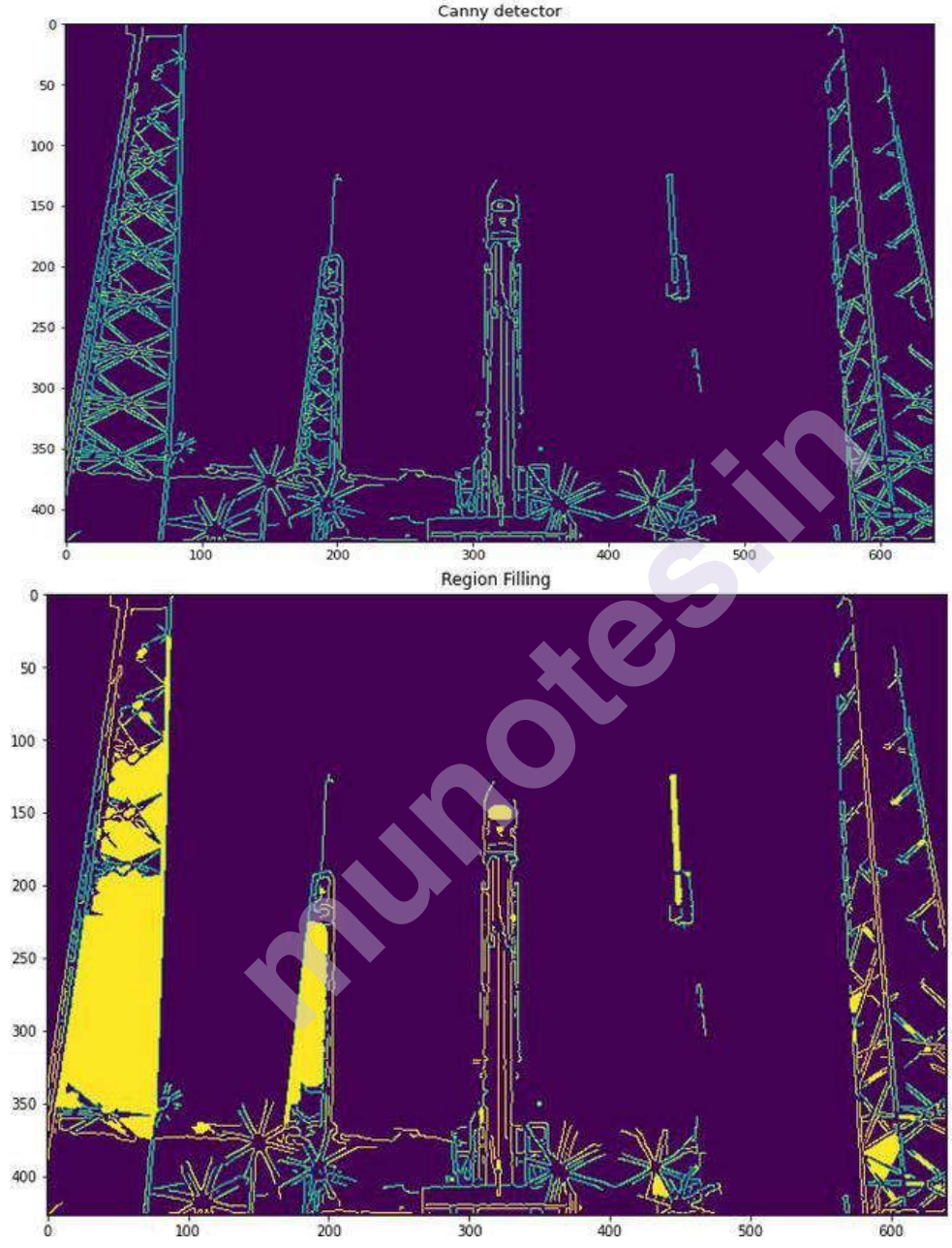
Implementation:

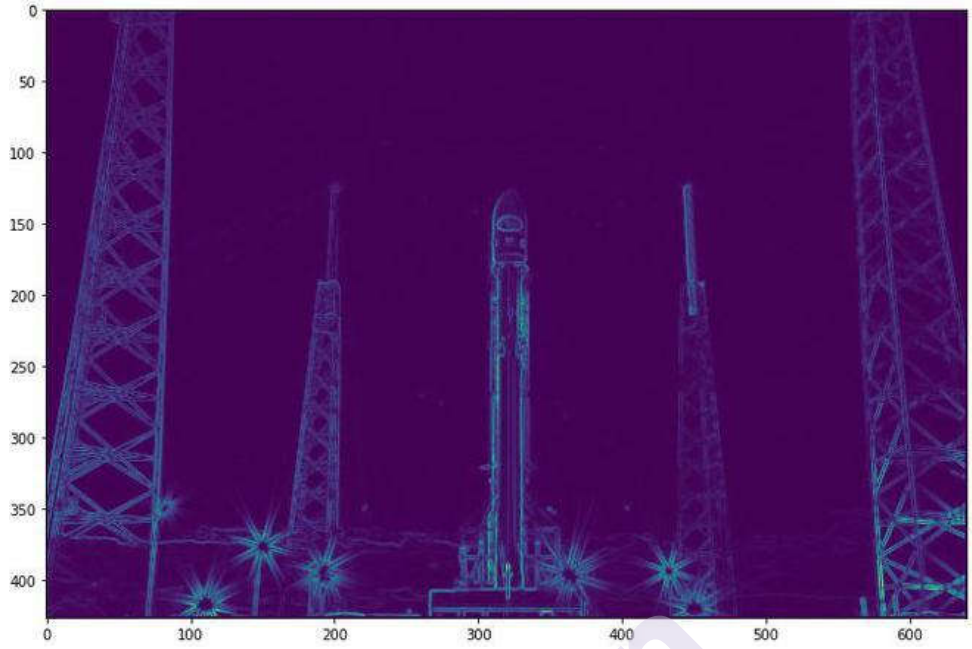
- In this implementation, we will be performing edge and region-based segmentation. We will be using scikit image module for that and an image from its dataset provided.
  - **# code**
  - import numpy as np
  - import matplotlib.pyplot as plt
  - from skimage.feature import canny
  - from skimage import data,morphology
  - from skimage.color import rgb2gray
  - import scipy.ndimage as nd
  - plt.rcParams["figure.figsize"] = (12,8)
  - %matplotlib inline
  - # load images and convert grayscale
  - rocket = data.rocket()
  - rocket_wh = rgb2gray(rocket)
  - # apply edge segmentation
  - # plot canny edge detection
  - edges = canny(rocket_wh)
  - plt.imshow(edges, interpolation='gaussian')
  - plt.title('Canny detector')
  - # fill regions to perform edge segmentation
  - fill_im = nd.binary_fill_holes(edges)
  - plt.imshow(fill_im)
  - plt.title('Region Filling')

- # Region Segmentation
- # First we print the elevation map
- elevation_map = sobel(rocket_wh)
- plt.imshow(elevation_map)

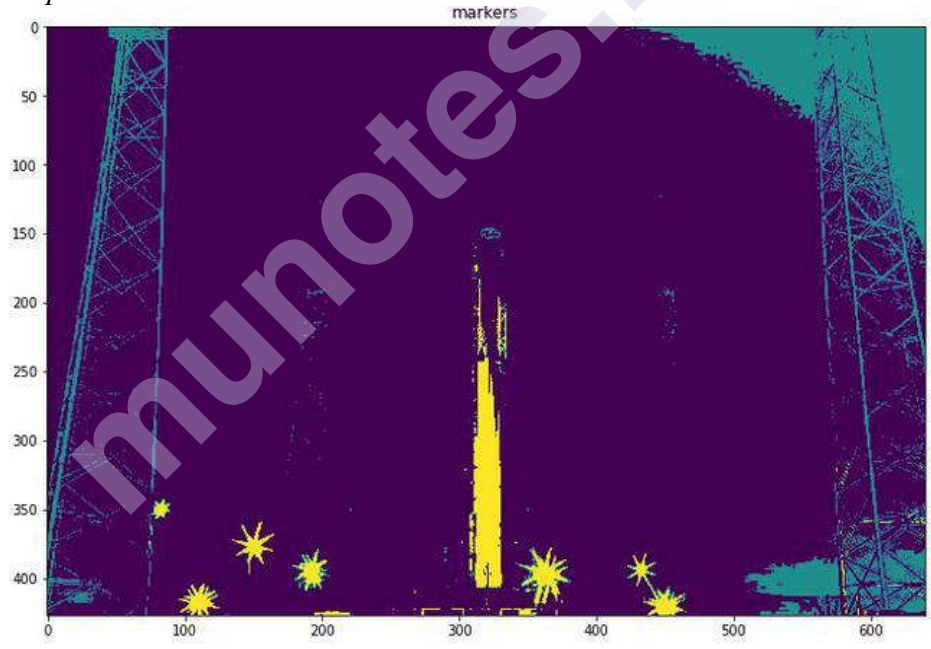
```
# Since, the contrast difference is not much. Anyways we will perform it
markers = np.zeros_like(rocket_wh)
markers[rocket_wh < 0.1171875] = 1 # 30/255
markers[rocket_wh > 0.5859375] = 2 # 150/255

plt.imshow(markers)
plt.title('markers')
# Perform watershed region segmentation
segmentation = morphology.watershed(elevation_map, markers)
plt.imshow(segmentation)
plt.title('Watershed segmentation')

# plot overlays and contour
segmentation = nd.binary_fill_holes(segmentation - 1)
label_rock, _ = nd.label(segmentation)
# overlay image with different labels
image_label_overlay = label2rgb(label_rock, image=rocket_wh)

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(24, 16), sharey=True)
ax1.imshow(rocket_wh)
ax1.contour(segmentation, [0.8], linewidths=1.8, colors='w')
ax2.imshow(image_label_overlay)

fig.subplots_adjust(**margins)
```

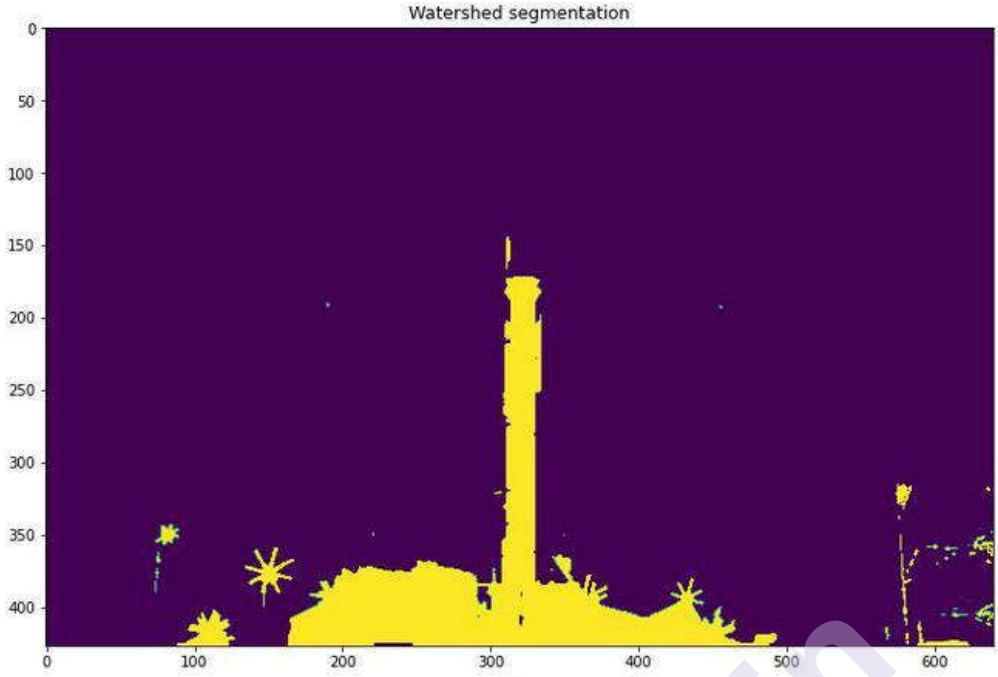
Output:



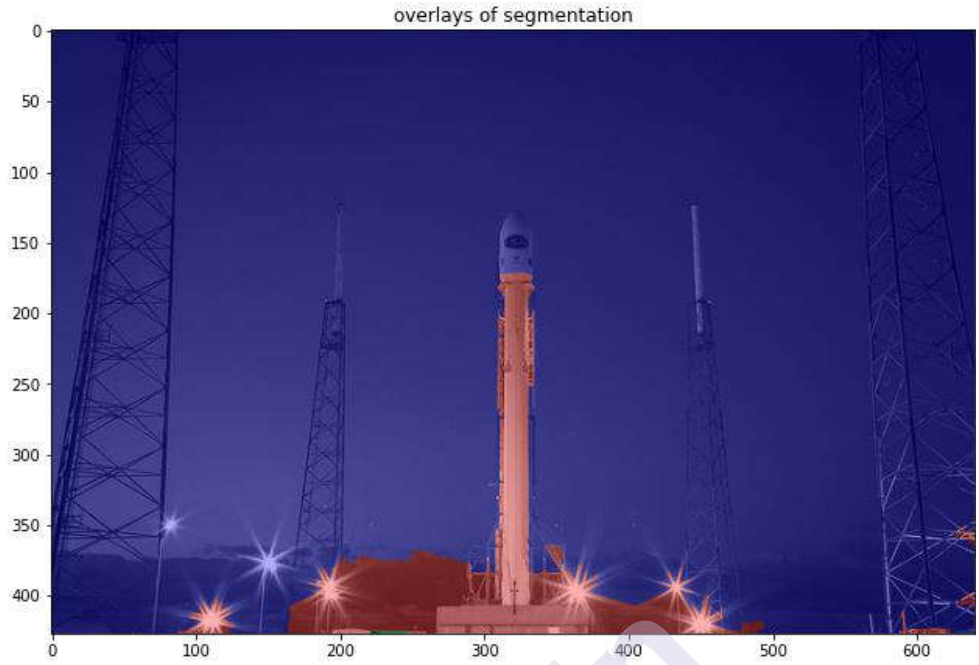


Elevation maps




Watershed segmentation
0
50
100
150
200
250
300
350
400
0
100
200
300
400
500
600


contours of segmentation
0
50
100
150
200
250
300
350
400


overlays of segmentation
0
50
100
150
200
250
300
350
400
0
100
200
300
400
500
600

Experiment V

5

IMAGE DATA COMPRESSION

Aim: Fundamentals of compression, Basic compression methods.

Theory:

In the field of Image processing, the compression of images is an important step before we start the processing of larger images or videos. The compression of images is carried out by an encoder and output a compressed form of an image. In the processes of compression, the mathematical transforms play a vital role. A flow chart of the process of the compression of the image can be represented as:

The general representation of the image in a computer is like a vector of pixels. Each pixel is represented by a fixed number of bits. These bits determine the intensity of the color (on grayscale if a black and white image and has three channels of RGB if colored images.)

Why Do We Need Image Compression?

Consider a black and white image that has a resolution of 1000*1000 and each pixel uses 8 bits to represent the intensity. So the total no of bits req= 1000*1000*8 = 80,00,000 bits per image. And **consider if it is a video with 30 frames per second of the above-mentioned type images then the total bits for a video of 3 secs is: 3*(30*(8, 000, 000))=720, 000, 000 bits**

As we see just to store a 3-sec video we need so many bits which is very huge. So, we need a way to have proper representation as well to store the information about the image in a minimum no of bits without losing the character of the image. Thus, image compression plays an important role.

Basic steps in image compression:

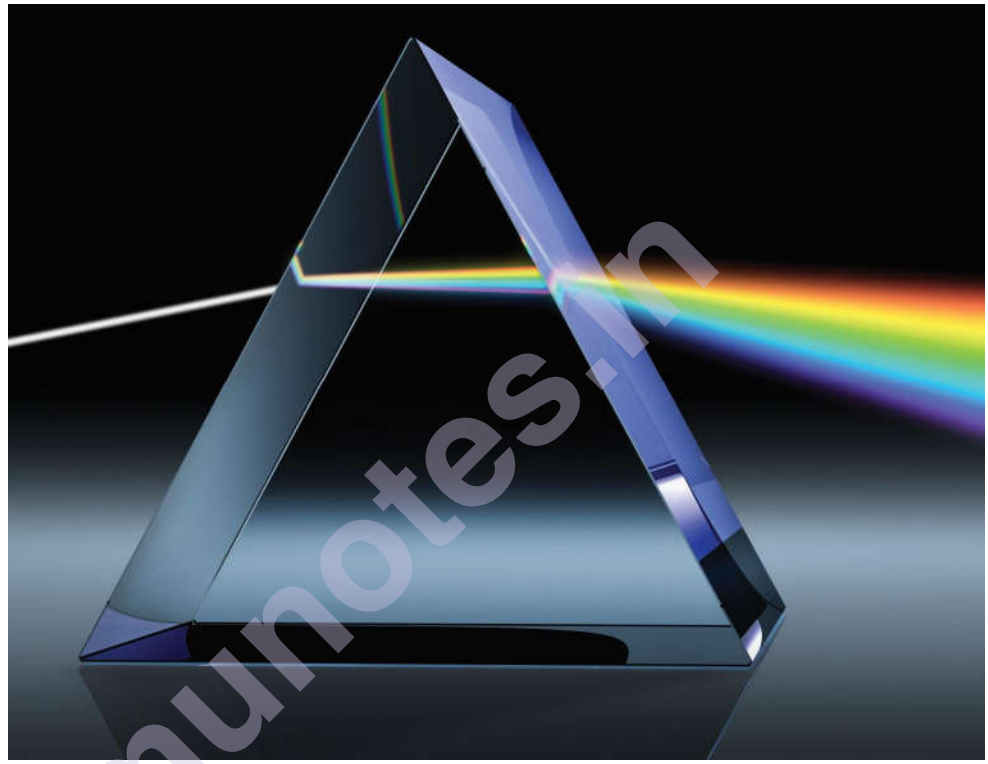
- Applying the image transform
- Quantization of the levels
- Encoding the sequences.
- Transforming The Image

What is a transformation(Mathematically)?

It a function that maps from one domain(vector space) to another domain(other vector space). Assume, **T is a transform, f(t):X->X' is a function then, T(f(t)) is called the transform of the function.**

We generally carry out the transformation of the function from one vector space to the other because when we do that in the newly projected vector space we infer more information about the function.

A real life example of a transform:



Here we can say that the prism is a transformation function in which it splits the white light (f(t)) into its components i.e the representation of the white light.

And we observe that we can infer more information about the light in its component representation than the white light one. This is how transforms help in understanding the functions in an efficient manner.

Transforms in Image Processing

The image is also a function of the location of the pixels. i.e I(x, y) where (x, y) are the coordinates of the pixel in the image. So, we generally transform an image from the spatial domain to the frequency domain.

Why Transformation of the Image is Important?

It becomes easy to know what all the principal components that make up the image and help in the compressed representation.

It makes the computations easy.

Example: finding convolution in the time domain before the transformation:

$$(f * g)(t) \triangleq \int_{-\infty}^{\infty} f(\tau) g(t - \tau)\, d\tau.$$

Finding convolution in the frequency domain after the transformation:

(f*g)(t)=F(s)G(s)

So we can see that the computation cost has reduced as we switched to the frequency domain. We can also see that in the time domain the convolution was equivalent to an integration operator but in the frequency domain, it becomes equal to the simple product of terms. So, this way the cost of computation reduces.

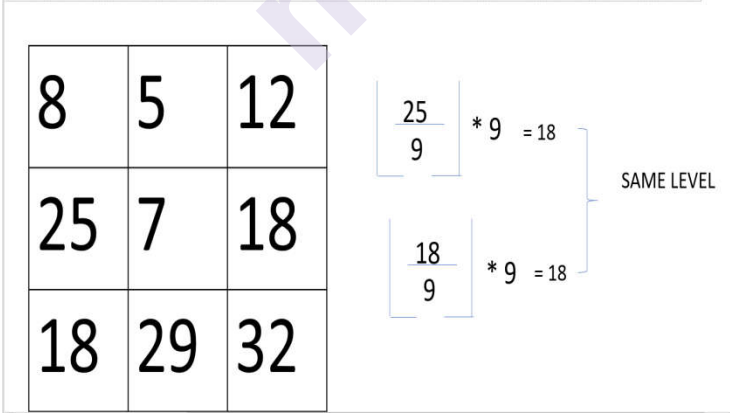
So this way when we transform the image from domain to the other carrying out the spatial filtering operations becomes easier.

Quantization

The process quantization is a vital step in which the various levels of intensity are grouped into a particular level based on the mathematical function defined on the pixels. Generally, the newer level is determined by taking a fixed filter size of "m" and dividing each of the "m" terms of the filter and rounding it its closest integer and again multiplying with "m".
Basic quantization Function: [pixelvalue/m] * m

So, the closest of the pixel values approximate to a single level hence as the no of distinct levels involved in the image becomes less. Hence we reduce the redundancy in the level of the intensity. So thus quantization helps in reducing the distinct levels.

Eg: (m=9)



Thus, we see in the above example both the intensity values round up to 18 thus we reduce the number of distinct levels(characters involved) in the image specification.

Symbol Encoding

The symbol stage involves where the distinct characters involved in the image are encoded in a way that the no. of bits required to represent a character is optimal based on the frequency of the character's occurrence. In simple terms, In this stage codewords are generated for the different characters present. By doing so we aim to reduce the no. of bits required to represent the intensity levels and represent them in an optimum number of bits.

There are many encoding algorithms. Some of the popular ones are:

Huffman variable-length encoding.

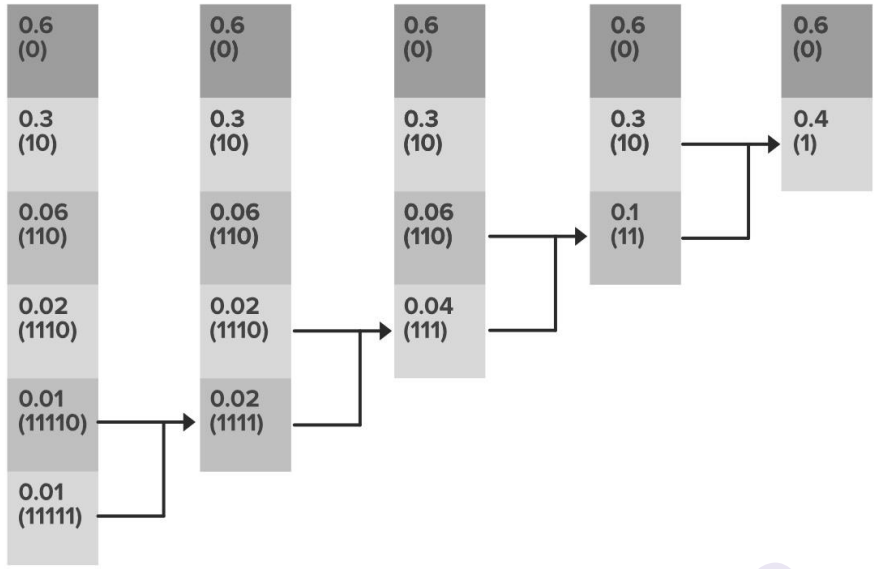
Run-length encoding.

In the Huffman coding scheme, we try to find the codes in such a way that none of the codes are the prefixes to the other. And based on the probability of the occurrence of the character the length of the code is determined. In order to have an optimum solution the most probable character has the smallest length code.

Example:

SYMBOLS (with instensity value gray scale)	Probability (arranged in decreasing order)	Binary Code	Huffman code	Length of Huffman code
a1 - 18	0.6	00010010	0	1
a2 - 25	0.3	00011001	10	2
a3 - 255	0.06	11111111	110	3
A4 - 128	0.02	10000000	1110	4
a5 - 200	0.01	11001000	11110	5
a6 - 140	0.01	10001100	11111	5

We see the actual 8-bit representation as well as the new smaller length codes. The mechanism of generation of codes is:



So we see how the storage requirement for the no of bits is decreased as:

Initial representation–average code length: 8 bits per intensity level.

After encoding–average code length:

*(0.6*1)+(0.3*2)+(0.06*3)+(0.02*4)+(0.01*5)+(0.01*5)=1.56 bits per intensity level*

Thus the no of bits required to represent the pixel intensity is drastically reduced.

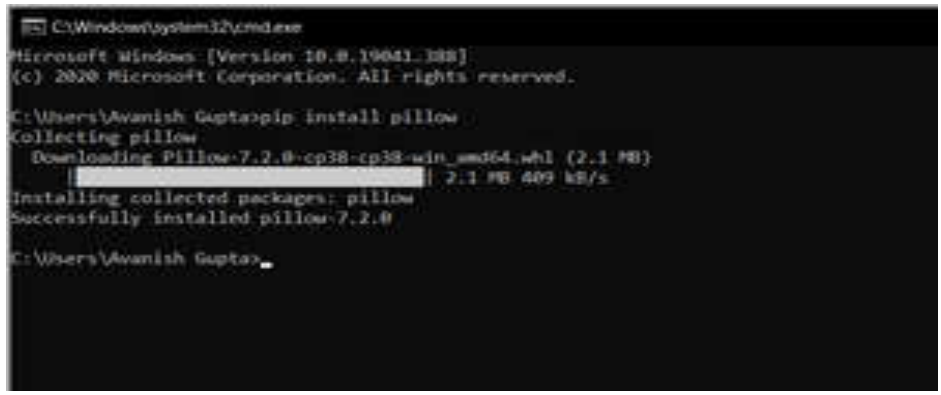
Thus in this way, the mechanism of quantization helps in compression. When the images are once compressed its easy for them to be stored on a device or to transfer them. And based on the type of transforms used, type of quantization, and the encoding scheme the decoders are designed based on the reversed logic of the compression so that the original image can be re-built based on the data obtained out of the compressed images.

There are organizations who receive data form lakhs or more persons, which is mostly in form of text, with a few images. Most of you know that the text part is stored in databases in the form of tables, but what about the images? The images are small compared to the textual data but constitute a much higher space in terms of storage. Hence, to save on the part of space and keep running the processes smoothly, they ask the users to submit the compressed images. As most of the readers have a bit of CS background(either in school or college), they understand that using online free tools to compress images is not a good practice for them.

Till Windows 7, Microsoft used to give MS Office Picture Manager which could be used to compress images till an extent, but it also had some limitations.

Those who know a bit of python can install python and use **pip install pillow** in command prompt(terminal for Linux users) to **install pillow fork**.

You'll get a screen like this



Assemble all the files in a folder and keep the file Compress.py in the same folder.

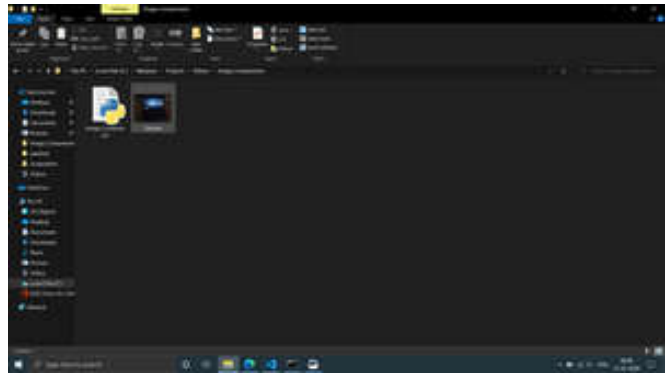
Run the python file with python.

Below is the Source Code of the file:

```
# run this in any directory
# add -v for verbose
# get Pillow (fork of PIL) from
# pip before running -->
# pip install Pillow

# import required libraries
import os
import sys
from PIL import Image

# define a function for
# compressing an image
def compressMe(file, verbose = False):
# Get the path of the file
    filepath = os.path.join (os.getcwd(), file)

            # open the image
    picture = Image.open(filepath)
```

```
        # Save the picture with desired quality
    # To change the quality of image,
    # set the quality variable at
    # your desired level, The more
    # the value of quality variable
    # and lesser the compression
    picture.save("Compressed_"+file,
                 "JPEG",
                 optimize = True,
                 quality = 10)
    return
  # Define a main function
def main():
        verbose = False
         # checks for verbose flag
    if (len(sys.argv)>1):
                if (sys.argv[1].lower()=="-v"):
            verbose = True
    # finds current working dir
    cwd = os.getcwd()
      formats = ('.jpg', '.jpeg')
         # looping through all the files
    # in a current directory
    for file in os.listdir(cwd):
                # If the file format is JPG or JPEG
        if os.path.splitext(file)[1].lower() in formats:
            print('compressing', file)
            compressMe(file, verbose)
print("Done")
  # Driver code
if __name__ == "__main__":
    main()
```

Folder Before Compression:



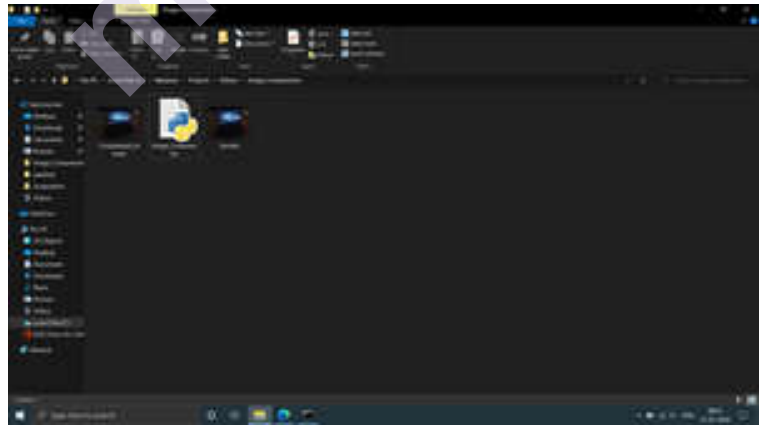
Folder before running file

Command Line for executing Code:

PS: Please run code after getting into the directory.

Command Line for executing Code

Folder after execution of Code:



Folder after running code

You can clearly see the compressed file.

❖❖❖❖

Experiment VI

6

MORPHOLOGICAL OPERATION

Aim: Morphological operational: Dilation, Erosion, Opening, Closing.

Theory:

EROSION AND DILATION IN MORPHOLOGICAL PROCESSING.

These operations are fundamental to morphological processing.

Erosion:

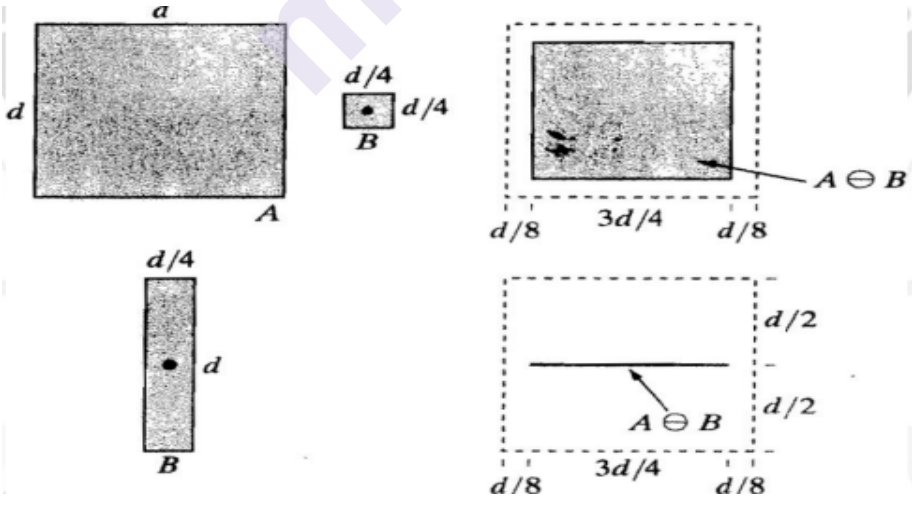
With A and B as sets in Z2 , the erosion of A by B, denoted A B, is defined as

$$A \ominus B = \{z | (B)_z \subseteq A\}$$

In words, this equation indicates that the erosion of A by B is the set of all points z such that B, translated by z, is contained in A. In the following discussion, set B is assumed to be a structuring element. The statement that B has to be contained in A is equivalent to B not sharing any common elements with the background; we can express erosion in the following equivalent form:

$$A \ominus B = \{z | (B)_z \cap A^c = \emptyset\}$$

where, A c is the complement of A and Ø is the empty set.



a) Set (b) Square structuring element, (c) Erosion of by shown shaded. (d) Elongated structuring element. (e) Erosion of by using this element. The dotted border in (c) and (e) is the boundary of set A, shown only for reference.

The elements of A and B are shown shaded and the background is white. The solid boundary in Fig. (c) is the limit beyond which further displacements of the origin of B would cause the structuring element to cease being completely contained in A. Thus, the locus of points (locations of the origin of B) within (and including) this boundary, constitutes the erosion of A by B. We show the erosion shaded in Fig. (c).The boundary of set A is shown dashed in Figs. (c) and (e) only as a reference; it is not part of the erosion operation. Figure (d) shows an elongated structuring element, and Fig. (e) shows the erosion of A by this element. Note that the original set was eroded to a line. However, these equations have the distinct advantage over other formulations in that they are more intuitive when the structuring element B is viewed as a spatial mask.

Thus, erosion shrinks or thins objects in a binary image. In fact, we can view erosion as a morphological filtering operation in which image details smaller than the structuring element are filtered (re-moved) from the image

(i) Dilation

However, the preceding definitions have a distinct advantage over other formulations in that they are more intuitive when the structuring element B is viewed as a convolution mask. The basic process of flipping (rotating) B about its origin and then successively displacing it so that it slides over set (image) A is analogous to spatial convolution. Keep in mind, however, that dilation is based on set operations and therefore is a nonlinear operation, whereas convolution is a linear operation. Unlike erosion, which is a shrinking or thinning operation, dilation "grows" or "thickens" objects in a binary image. The specific manner and extent of this thickening is controlled by the shape of the structuring element used. In the following Figure (b) shows a structuring element (in this case B = B because the SE is symmetric about its origin). The dashed line in Fig. (c) shows the original set for reference, and the solid line shows the limit beyond which any further displacements of the origin of B by z would cause the intersection of B and A to be empty. Therefore, all points on and inside this boundary constitute the dilation of A by B. Figure (d) shows a structuring element designed to achieve more dilation vertically than horizontally, and Fig. (e) shows the dilation achieved with this element

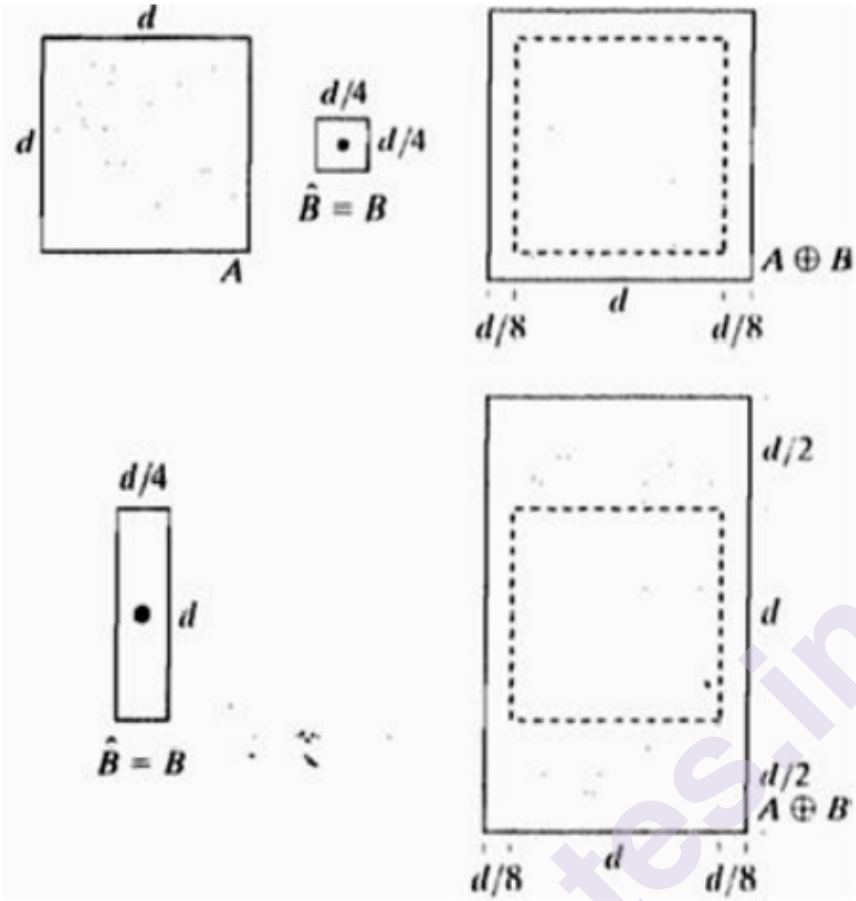


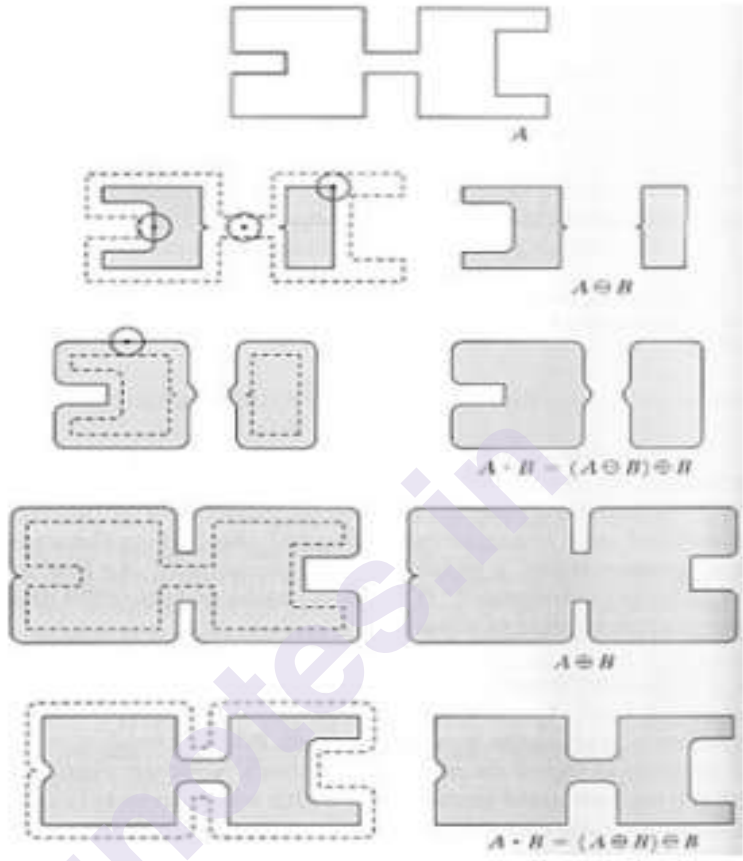
FIG:4.1.5 (a) Set (b) Square structuring element (the dot denotes the origin). (c) Dilation of by shown shaded. (d) Elongated structuring element. (e) Dilation of using this element. The dotted border in (c) and (e) is the boundary of set shown only for reference.

Opening and Closing

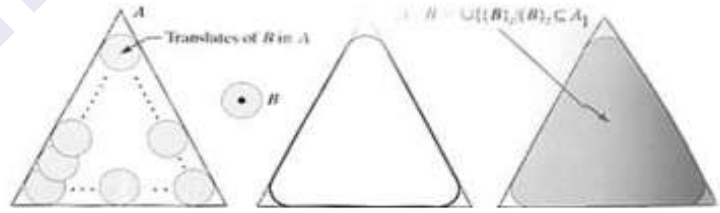
Opening generally smoothes the contour object, breaks narrow isthmuses, and eliminates thin protrusions. Closing also tends to smooth sections of contours but, ass opposed to opening, it generally fuses narrow breaks and long thin gulfs, eliminates small holes, and fills gaps in the contour

Opening: $A \circ B = (A \ominus B) \oplus B.$

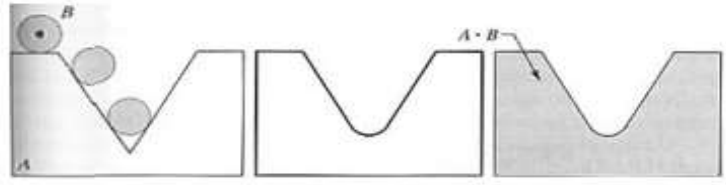
Closing: $A \bullet B = (A \oplus B) \ominus B.$



Opening: roll B around the *inside* of A.



Closing: roll B around the *outside* of A.



Morphological Operations in Image Processing (Opening)

Morphological operations are used to extract image components that are useful in the representation and description of region shape. Morphological operations are some basic tasks dependent on the picture shape. It is typically performed on binary images. It needs two data sources, one is the **input image**, the second one is called **structuring component**. Morphological operators take an input image and a structuring component as input and these elements are then combines using the set operators. The objects in the input image are processed depending on attributes of the shape of the image, which are encoded in the structuring component.

Opening is similar to erosion as it tends to remove the bright foreground pixels from the edges of regions of foreground pixels. The impact of the operator is to safeguard foreground region that has similarity with the structuring component, or that can totally contain the structuring component while taking out every single other area of foreground pixels. Opening operation is used for removing internal noise in an image. **Opening is erosion operation followed by dilation operation.**

$$A \circ B = (A \ominus B) \oplus B$$

Syntax: *cv2.morphology Ex(image, cv2.MORPH_OPEN, kernel)*
Parameters:
*-> **image**: Input Image array.*
*-> **cv2.MORPH_OPEN**: Applying the Morphological Opening operation.*
*-> **kernel**: Structuring element.*
Below is the Python code explaining Opening Morphological Operation –

```
# Python program to illustrate
# Opening morphological operation
# on an image
# organizing imports
import cv2
import numpy as np
# return video from the first webcam on your computer.
screenRead = cv2.VideoCapture(0)

# loop runs if capturing has been initialized.
while(1):
    # reads frames from a camera
```

```
    _, image = screenRead.read()
    # Converts to HSV color space, OCV reads colors as BGR
    # frame is converted to hsv
    hsv = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)
        # defining the range of masking
    blue1 = np.array([110, 50, 50])
    blue2 = np.array([130, 255, 255])
        # initializing the mask to be
    # convoluted over input image
    mask = cv2.inRange(hsv, blue1, blue2)
     # passing the bitwise_and over
    # each pixel convoluted
    res = cv2.bitwise_and(image, image, mask = mask)
        # defining the kernel i.e. Structuring element
    kernel = np.ones((5, 5), np.uint8)
        # defining the opening function
    # over the image and structuring element
    opening = cv2.morphologyEx(mask, cv2.MORPH_OPEN, kernel)
       # The mask and opening operation
    # is shown in the window
    cv2.imshow('Mask', mask)
    cv2.imshow('Opening', opening)
        # Wait for 'a' key to stop the program
    if cv2.waitKey(1) & 0xFF == ord('a'):
       break
# De-allocate any associated memory usage
cv2.destroyAllWindows()
 # Close the window / Release webcam
screenRead.release()
```

Input Frame:

Mask:

Output Frame:

The system recognizes the defined blue book as the input as removes and simplifies the internal noise in the region of interest with the help of the Opening function.

Morphological Operations in Image Processing (Closing)

Closing is similar to the opening operation. In closing operation, the basic premise is that the closing is opening performed in reverse. It is defined simply as a **dilation followed by an erosion** using the same structuring element used in the opening operation.

$$A \bullet B = (A \oplus B) \ominus B$$

Syntax: *cv2.morphologyEx(image, cv2.MORPH_CLOSE, kernel)*

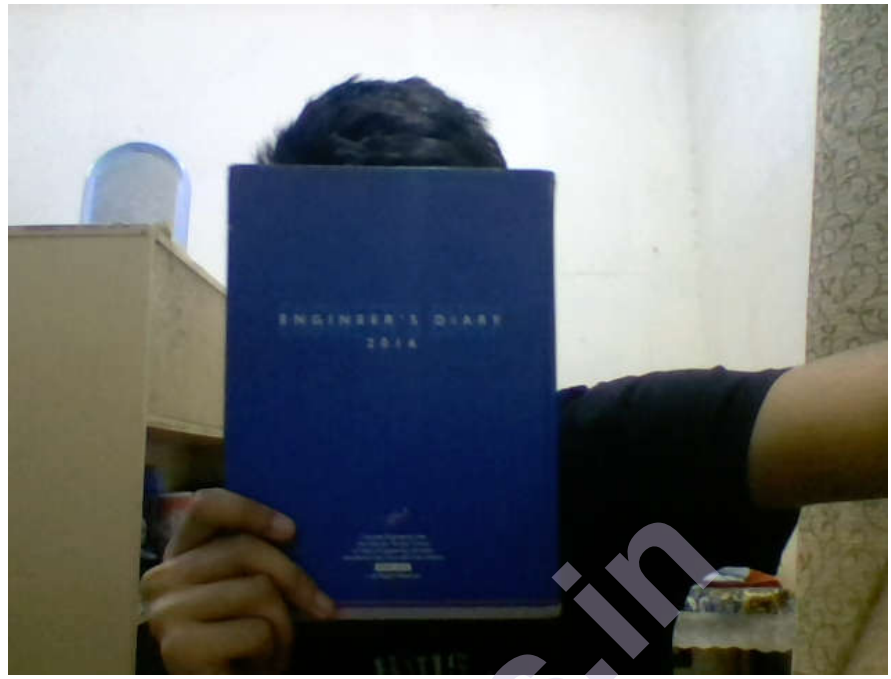
Parameters:
*-> **image**: Input Image array.*
*-> **cv2.MORPH_CLOSE**: Applying the Morphological Closing operation.*
*-> **kernel**: Structuring element.*

Below is the Python code explaining Closing Morphological Operation –

```
# Python program to illustrate
# Closing morphological operation
# on an image
 # organizing imports
import cv2
```

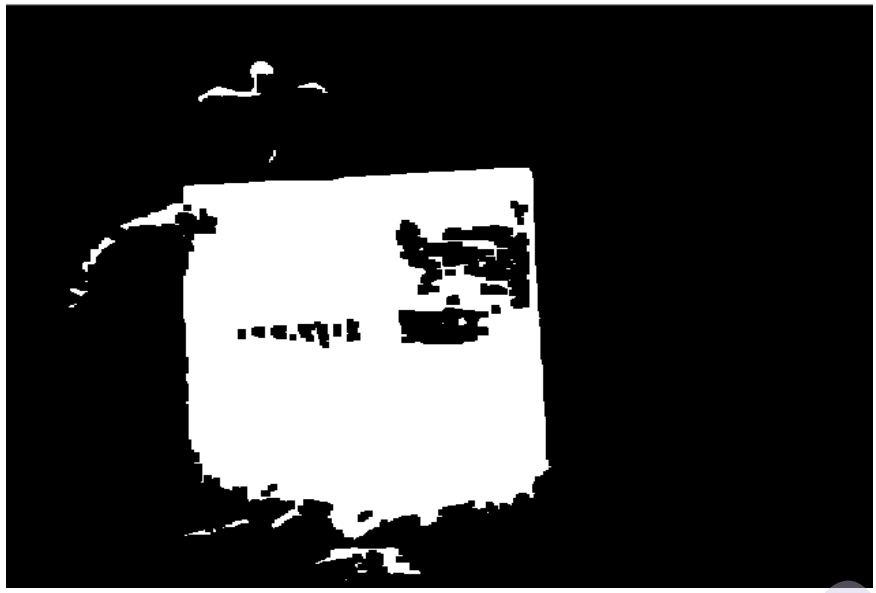
```
import numpy as np
  # return video from the first webcam on your computer.
screenRead = cv2.VideoCapture(0)
  # loop runs if capturing has been initialized.
while(1):
    # reads frames from a camera
    _, image = screenRead.read()
        # Converts to HSV color space, OCV reads colors as BGR
    # frame is converted to hsv
    hsv = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)
        # defining the range of masking
    blue1 = np.array([110, 50, 50])
    blue2 = np.array([130, 255, 255])
        # initializing the mask to be
    # convoluted over input image
    mask = cv2.inRange(hsv, blue1, blue2)
      # passing the bitwise_and over
    # each pixel convoluted
    res = cv2.bitwise_and(image, image, mask = mask)
        # defining the kernel i.e. Structuring element
    kernel = np.ones((5, 5), np.uint8)
        # defining the closing function
    # over the image and structuring element
    closing = cv2.morphologyEx(mask, cv2.MORPH_OPEN, kernel)
        # The mask and closing operation
    # is shown in the window
    cv2.imshow('Mask', mask)
    cv2.imshow('Closing', closing)
# Wait for 'a' key to stop the program
    if cv2.waitKey(1) & 0xFF == ord('a'):
        break
  # De-allocate any associated memory usage
cv2.destroyAllWindows()
  # Close the window / Release webcam
screenRead.release()
```

Input Frame:



Mask:

Output:



Erosion and Dilation of images using OpenCV in python

Morphological operations are a set of operations that process images based on shapes. They apply a structuring element to an input image and generate an output image

. The most basic morphological operations are two: **Erosion and Dilation**

Basics of Erosion:

- Erodes away the boundaries of the foreground object
- Used to diminish the features of an image.

Working of erosion:

1. A kernel(a matrix of odd size(3,5,7) is convolved with the image.
2. A pixel in the original image (either 1 or 0) will be considered 1 only if all the pixels under the kernel are 1, otherwise, it is eroded (made to zero).
3. Thus all the pixels near the boundary will be discarded depending upon the size of the kernel.
4. So the thickness or size of the foreground object decreases or simply the white region decreases in the image.

Basics of dilation:

- Increases the object area
- Used to accentuate features

Working of dilation:

1. A kernel(a matrix of odd size(3,5,7) is convolved with the image
2. A pixel element in the original image is '1' if at least one pixel under the kernel is '1'.

3. It increases the white region in the image or the size of the foreground object increases

```
# Python program to demonstrate erosion and
# dilation of images.
import cv2
import numpy as np
 # Reading the input image
img = cv2.imread('input.png', 0)
 # Taking a matrix of size 5 as the kernel
kernel = np.ones((5,5), np.uint8)
 # The first parameter is the original image,
# kernel is the matrix with which image is
# convolved and third parameter is the number
# of iterations, which will determine how much
# you want to erode/dilate a given image.
img_erosion = cv2.erode(img, kernel, iterations=1)
img_dilation = cv2.dilate(img, kernel, iterations=1)
 cv2.imshow('Input', img)
cv2.imshow('Erosion', img_erosion)
cv2.imshow('Dilation', img_dilation)
 cv2.waitKey(0)
```

Uses of Erosion and Dilation:

1. **Erosion:**
   - It is useful for removing small white noises.
   - Used to detach two connected objects etc.
2. **Dilation:**
   - In cases like noise removal, erosion is followed by dilation. Because, erosion removes white noises, but it also shrinks our object. So we dilate it. Since noise is gone, they won't come back, but our object area increases.
   - It is also useful in joining broken parts of an object.

❖❖❖❖