



## **FACULTY OF ENGINEERING AND TECHNOLOGY**

**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**

**B.E. (CSE) - Artificial Engineering and Machine Learning**

**IV SEMESTER**

### **AICP308 - IMAGE AND SPEECH PROCESSING LAB MANUAL**

**Dr. N. J. Nalini**  
**Associate Professor**

**Staff-in-charge**

## **List of Exercises**

### **Cycle 1 : Image Processing**

1. Smoothening and Sharpening filters in spatial domain.
2. Implementation of Edge detection methods.
3. Write a program to find the histogram equalization
  - a) For full image.
  - b) For part of the image.
4. Write a program to find the Fourier transform of a given image.
5. Write a program to segment the given image using Kmeans algorithm.
6. Implementation of morphological dilation and erosion operations for a given image.
7. Write programs to extract SIFT and SURF features for given input image samples.
8. Implementation of Mouse as a paint Brush.

### **Cycle 2 : Speech Processing**

9. Write a program to find pitch, short time energy and zero crossing rate for a given speech signal.
10. Write a program to perform convolution and correlation of speech signals.
11. Write a program to perform simple low pass filtering and high pass filtering of speech signal.
12. Write a program to extract MFCC feature from sample speech signal.

### **Assignment Exercise**

13. Write a program to capture the still Image.
14. Text dependent speaker recognition using Dynamic Time Warping

## Ex. No. 1.

## Smoothing and Sharpening of Image

### a). Smoothing of Image

#### Aim:

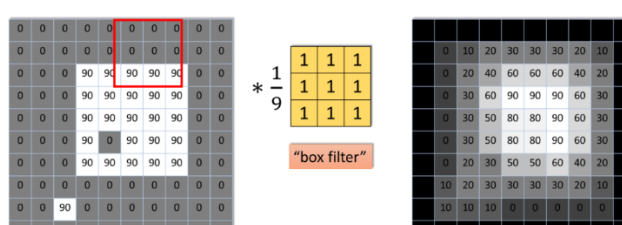
To smoothing the given input gray scale image using filters in spatial domain.

#### Theoretical Concept:

**Smoothing of images :** Image blurring is achieved by convolving the image with a low-pass filter kernel. It is useful for removing noise. It actually removes high frequency content (eg: noise, edges) from the image. So edges are blurred a little bit in this operation (there are also blurring techniques which don't blur the edges).

This is done by convolving an image with a normalized box filter. It simply takes the average of all the pixels under the kernel area and replaces the central element. This is done by the function [`cv.blur\(\)`](#). We should specify the width and height of the kernel. A 3x3 normalized box filter would look like the below:

#### Averaging filter

$$F(x, y) * H(u, v) = G(x, y)$$


$G = F * H$

#### Implementation:

```
import cv2

import numpy as np

from matplotlib import pyplot as plt

img = cv2.imread('opencv.jpg')

blur = cv2.blur(img,(5,5))

plt.subplot(121),plt.imshow(img),plt.title('Original')

plt.xticks([]), plt.yticks([])

plt.subplot(122),plt.imshow(blur),plt.title('Blurred')

plt.xticks([]), plt.yticks([])

plt.show()
```

## Sample Input and Output



The above code can be modified for Gaussian and Bilateral blurring:

```
blur = cv.GaussianBlur(img,(5,5),0)  
blur = cv.bilateralFilter(img,9,75,75)
```

### Result:

Thus, a program has been written to Smoothen the given input image using python.

## b).                      Sharpening of images

### Aim:

To smoothening the given input gray scale image using filters in spatial domain.

### Theoretical Concept:

The process of sharpening is usually used to enhance edges in an image. There are many filters that we can use but one that can sharpen our image is represented in a matrix below.

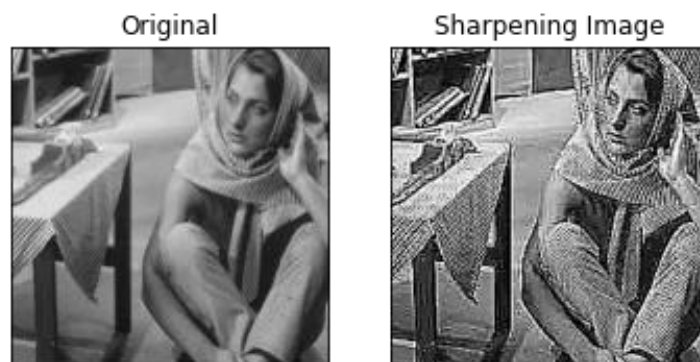
-1	-1	-1
-1	9	-1
-1	-1	-1

This filter has a positive 9 in a center, whereas it has -1 at all other places. For this particular filter we don't have an implemented OpenCV function. Therefore, we will use a function **cv2.filter2D()** which will process our image with an arbitrary filter. This filter is often used for sharpening colored images.

### Implementation:

```
import cv2
import numpy as np
from matplotlib import pyplot as plt
img = cv2.imread('opencv.jpg',1)
# Creating our sharpening filter
filter = np.array([[ -1, -1, -1], [-1, 9, -1], [-1, -1, -1]])
# Applying cv2.filter2D function on our Cybertruck image
sharpen_img=cv2.filter2D(img,-1,filter)
plt.subplot(121),plt.imshow(img),plt.title('Original')
plt.xticks([], plt.yticks([]))
plt.subplot(122),plt.imshow(sharpen_img),plt.title('Sharpening Image')
plt.xticks([], plt.yticks([]))
plt.show()
```

### Sample input and Output



### Result:

Thus, a program has been written to Sharpen the given input image using python.

**Ex. No. 2.****Implementation of Edge Detection Methods****Aim:**

To Implement Edge Detection for the given input image using Sobel edge detection Method.

**Theoretical Concept:**

**Edge detection** is one of the fundamental operations when we perform image processing. It helps us reduce the amount of data (pixels) to process and maintains the structural aspect of the image.

Sobel edge detector is a gradient based method based on the first order derivatives. It calculates the first derivatives of the image separately for the X and Y axes. The operator uses two 3X3 kernels which are convolved with the original image to calculate approximations of the derivatives - one for horizontal changes, and one for vertical. The matrix below shows Sobel Kernels in x-dir and y-dir:

$$\begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix}$$

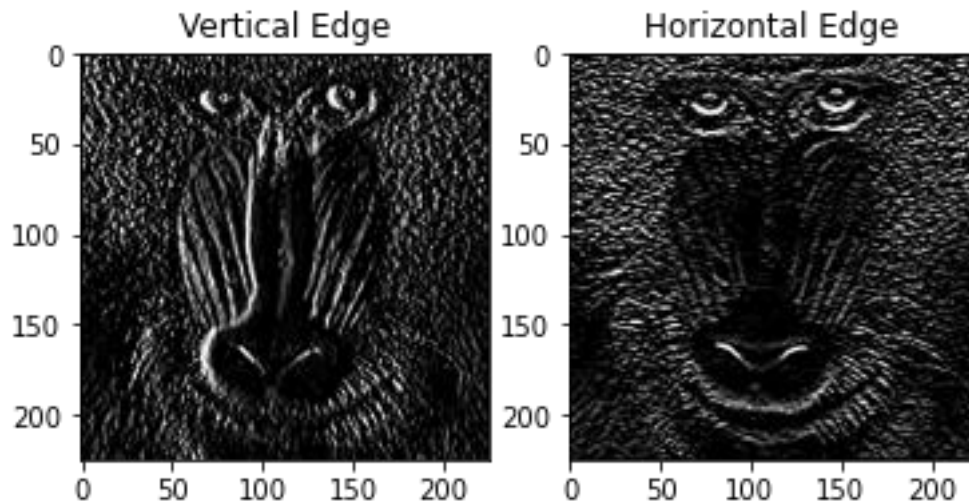
$$\begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

**Implementation**

```
import numpy as np
import cv2
import matplotlib.pyplot as plt
filter = np.array([[ -1, 0, 1], [-2, 0, 2], [-1, 0, 1]])
image = cv2.imread("baboon.jpg")
#Filter the image using filter2D, which has inputs: (grayscale image, bit-depth, kernel)
sobel_verti_image = cv2.filter2D(image,-1,filter)
sobel_hori_image = cv2.filter2D(image,-1,np.flip(filter.T, axis=0))
plt.subplot(121)
plt.imshow(sobel_verti_image, cmap='gray')
plt.title("Vertical Edge")
plt.subplot(122)
plt.imshow(sobel_hori_image, cmap='gray')
```

```
plt.title("Horizontal Edge")
plt.show()
```

### Sample Input and Output



#### Result:

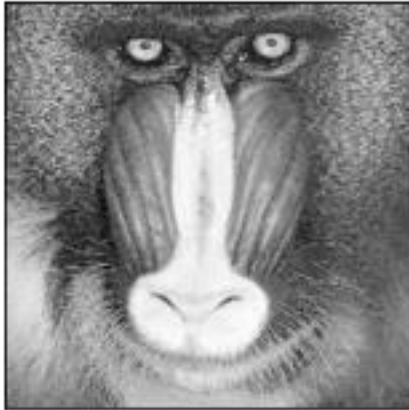
Thus, a program has been written to Detect the Edges of the given input image using Sobel Edge Detection Algorithm.

**The Edge detection can be done with Canny edge Detection method. Try with following code.**

```
import numpy as np
import cv2 as cv
from matplotlib import pyplot as plt
img = cv.imread('baboon.jpg',0)
edges = cv.Canny(img,100,200)
plt.subplot(121),plt.imshow(img,cmap = 'gray')
plt.title('Original Image'), plt.xticks([]), plt.yticks([])
plt.subplot(122),plt.imshow(edges,cmap = 'gray')
plt.title('Edge Image'), plt.xticks([]), plt.yticks([])
plt.show()
```

## Sample Input and Output

Original Image



Edge Image





### Ex. No. 3

### Histogram Equalization of an Image

#### Aim

To adjust the contrast of an image, histogram equalization technique to be applied for full image and Part of an image.

#### Theoretical Concept:

Histogram as a graph or plot, which gives you an overall idea about the intensity distribution of an image. It is a plot with pixel values (ranging from 0 to 255, not always) in X-axis and corresponding number of pixels in the image on Y-axis.

Consider an image whose pixel values are confined to some specific range of values only. For eg, brighter image will have all pixels confined to high values. But a good image will have pixels from all regions of the image. So you need to stretch this histogram to either ends (as given in below image, from wikipedia) and that is what Histogram Equalization does (in simple words). This normally improves the contrast of the image.

Numpy provides a function, **np.histogram()** for creating histogram

```
histogram, bin_edges = np.histogram(image, bins=256, range=(0, 1))
```

The parameter **bins** determines the histogram size, or the number of “bins” to use for the histogram. Bins= 256 because the pixel count for each of the 256 possible values in the grayscale image. The bins method of calculation is: 0-0.99, 1-1.99, 2-2.99, etc. So the last range is 255-255.99.

The parameter **range** is the range of values each of the pixels in the image can have. 0 and 1, which is the value range of our input image after transforming it to grayscale. 0 to 256 for color image.

The first output of the **np.histogram** function is a one-dimensional NumPy array, with 256 rows and one column, representing the number of pixels with the color value corresponding to the index. I.e., the first number in the array is the number of pixels found with color value 0, and the final number in the array is the number of pixels found with color value 255. The second output of np.histogram is an array with the bin edges and one column and 257 rows (one more than the histogram itself). There are no gaps between the bins, which means that the end of the first bin, is the start of the second and so on. For the last bin, the array also has to contain the stop, so it has one more element, than the histogram.

OpenCV has a function to do this, **cv.equalizeHist()**. Its input is color image and output is our histogram equalized image.

## Implementation

### a. Histogram equalization for full image

```
import numpy as np
import cv2 as cv
from matplotlib import pyplot as plt
img = cv.imread('butterfly.jpg',0)
#subplot with 2 rows and 2 columns
plt.subplot(2,2,1)
#To display original image
plt.title('Original Image')
plt.imshow(img)

# get image histogram
hist,bins = np.histogram(img.flatten(),256,[0,256])
# cumulative distribution function
cdf = hist.cumsum()
# normalize
cdf_normalized = cdf * float(hist.max()) / cdf.max()
##subplot with 2 rows and 2 columns
plt.subplot(2,2,2)
#plotting cdf and histogram
plt.plot(cdf_normalized, color = 'b')
plt.hist(img.flatten(),256,[0,256], color = 'r')
plt.xlim([0,256])
plt.legend(('cdf','histogram'), loc = 'upper left')
plt.xlabel('Bins')
cdf_m = np.ma.masked_equal(cdf,0)
cdf_m = (cdf_m - cdf_m.min())*255/(cdf_m.max()-cdf_m.min())
cdf = np.ma.filled(cdf_m,0).astype('uint8')
# normalized image
```

```

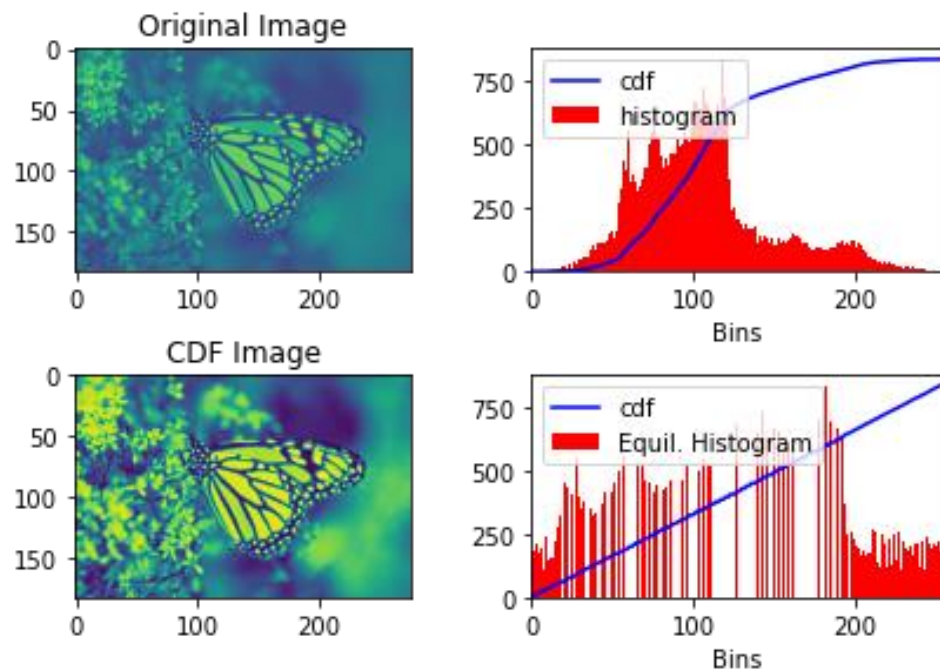
img2 = cdf[img]
plt.subplot(2,2,3)
plt.title('CDF Image')
#To display Cdf image
plt.imshow(img2)
#histogram Equilization
hist,bins = np.histogram(img2.flatten(),256,[0,256])
cdf = hist.cumsum()
cdf_normalized = cdf * float(hist.max()) / cdf.max()
plt.subplot(2,2,4)
#plotting cdf and H.Equilization
plt.plot(cdf_normalized, color = 'b')
plt.hist(img2.flatten(),256,[0,256], color = 'r')
plt.xlim([0,256])
plt.legend(('cdf','Equil. Histogram'), loc = 'upper left')
plt.tight_layout()
plt.xlabel('Bins')
plt.show()

#OpenCv Implementation
img = cv.imread('butterfly.jpg',0)
equ = cv.equalizeHist(img)
res = np.hstack((img,equ)) #stacking images side-by-side
cv.imwrite('res.png',res)
img = cv.imread('res.png',0)
plt.imshow(img)
plt.show()

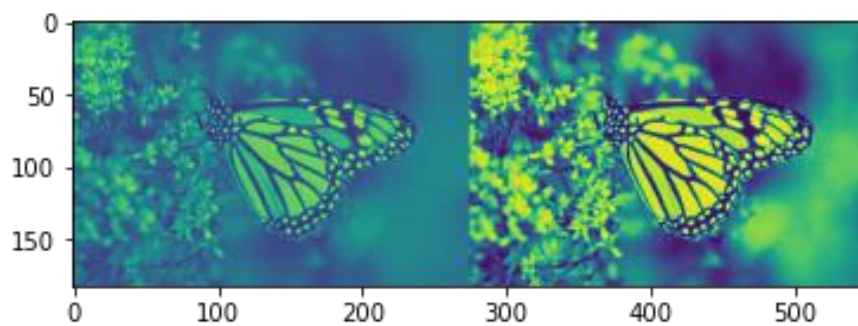
```

## Sample Input and Output

### Numpy Implementation



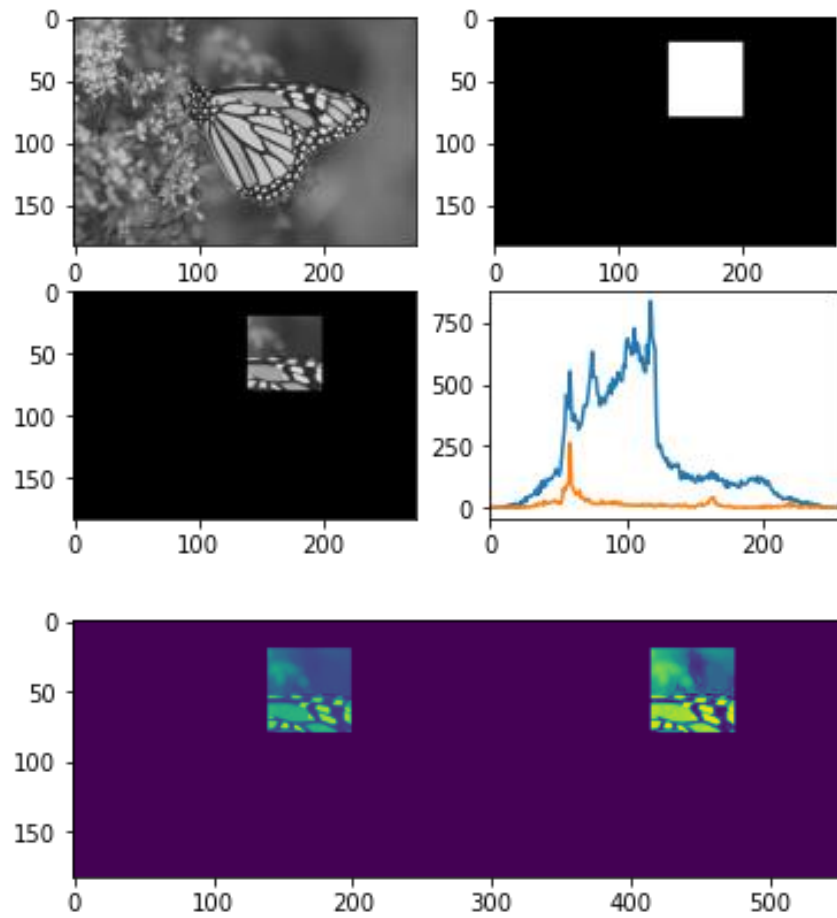
### Opencv Implementation



## **b. Histogram equalization for part of an Image**

```
import numpy as np
import cv2 as cv
from matplotlib import pyplot as plt
img = cv.imread('home.jpg',0)
# create a mask
mask = np.zeros(img.shape[:2], np.uint8)
mask[20:80, 140:200] = 255
masked_img = cv.bitwise_and(img,img,mask = mask)
# Calculate histogram with mask and without mask
# Check third argument for mask
hist_full = cv.calcHist([img],[0],None,[256],[0,256])
hist_mask = cv.calcHist([img],[0],mask,[256],[0,256])
plt.subplot(221), plt.imshow(img, 'gray')
plt.subplot(222), plt.imshow(mask,'gray')
plt.subplot(223), plt.imshow(masked_img, 'gray')
plt.subplot(224), plt.plot(hist_full), plt.plot(hist_mask)
plt.xlim([0,256])
plt.show()
equ = cv.equalizeHist(masked_img)
res = np.hstack((masked_img,equ)) #stacking images side-by-side
cv.imwrite('result.png',res)
img = cv.imread('result.png',0)
plt.imshow(img)
plt.show()
```

## Sample Input and Output



### Result:

Thus, a program has been written to adjust the contrast of the given input image using Histogram Equalization technique.

**Ex. No. 4.****Fast Fourier Transform****Aim**

To write a program that converts an image from spatial domain to frequency domain.

**Theoretical Concept:**

Fast Fourier Transform is applied to convert an image from the image (spatial) domain to the frequency domain. Applying filters to images in frequency domain is computationally faster than to do the same in the image domain. Once the image is transformed into the frequency domain, filters can be applied to the image by convolutions. FFT turns the complicated convolution operations into simple multiplications. An inverse transform is then applied in the frequency domain to get the result of the convolution.

**Step 1: Compute the 2-dimensional Fast Fourier Transform**

The result from FFT process is a complex number array which is very difficult to visualize directly. Therefore, we have to transform it into 2-dimension space. FFT can be visualized in terms of Spectrum. The white area in the spectrum image shows the high power of frequency. The corners in the spectrum image represent low frequencies. Therefore, combining two points, the white area on the corner indicates that there is high energy in low/zero frequencies which is a very normal situation for most images.

**Step 2: Shift the zero-frequency component to the center of the spectrum.**

2-D FFT has translation and rotation properties, so we can shift frequency without losing any piece of information. zero-frequency component to the center of the spectrum which makes the spectrum image more visible for human. Moreover, this translation could help us implement high/low-pass filter easily

**Step 3 :Shift the zero-frequency component back to original location****Step 4 :Compute the 2-dimensional inverse Fast Fourier Transform**

The processes of step 3 and step 4 are converting the information from spectrum back to gray scale image. It could be done by applying inverse shifting and inverse FFT operation.

**Functions**

`fft.fft2( )` :Compute the 2-dimensional discrete Fourier Transform by means of the Fast Fourier Transform (FFT).

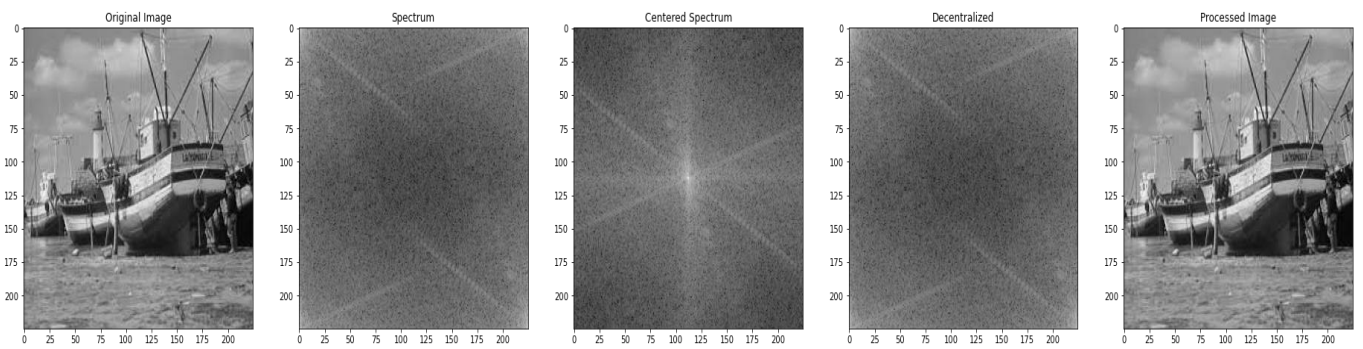
`fft.fftshift( )` :Shift the zero-frequency component to the center of the spectrum.

## Implementation

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
img_c1 = cv2.imread("ship.jpg", 0)
img_c2 = np.fft.fft2(img_c1)
img_c3 = np.fft.fftshift(img_c2)
img_c4 = np.fft.ifftshift(img_c3)
img_c5 = np.fft.ifft2(img_c4)
plt.figure(figsize=(6.4*5, 4.8*5), constrained_layout=False)
plt.subplot(151)
plt.imshow(img_c1, "gray")
plt.title("Original Image")
plt.subplot(152)
plt.imshow(np.log(1+np.abs(img_c2)), "gray")
plt.title("Spectrum")
plt.subplot(153)
plt.imshow(np.log(1+np.abs(img_c3)), "gray")
plt.title("Centered Spectrum")
plt.subplot(154)
plt.imshow(np.log(1+np.abs(img_c4)), "gray")
plt.title("Decentralized")
plt.subplot(155)
plt.imshow(np.abs(img_c5), "gray")
plt.title("Processed Image")
plt.show()
```



## Sample Input and Output



## Result:

Thus, a program has been written to convert an image from spatial domain to frequency domain using Fast Fourier Transform.

**Ex. No. 5.****Image Segmentation****Aim**

To segment the given input image using Cluster based segmentation algorithm.

**Theoretical Concept:**

Image segmentation is a branch of digital image processing which focuses on partitioning an image into different parts according to their features and properties. The primary goal of image segmentation is to simplify the image for easier analysis. In image segmentation, you divide an image into various parts that have similar attributes. The parts in which you divide the image are called Image Objects.

Clustering and similar machine learning algorithms use this method to detect unknown features and attributes. K-means is a simple unsupervised machine learning algorithm. It classifies an image through a specific number of clusters. It starts the process by dividing the image space into k pixels that represent k group centroids. Then they assign each object to the group based on the distance between them and the centroid. When the algorithm has assigned all pixels to all the clusters, it can move and reassign the centroids.

**Implementation**

```
import numpy as np
import matplotlib.pyplot as plt
import cv2
%matplotlib inline

# Read in the image
image = plt.imread('butterfly.jpg')

# Change color to RGB (from BGR)
print(image.shape)
plt.subplot(121)
plt.imshow(image)

plt.title("Original Image")
# Reshaping the image into a 2D array of pixels and 3 color values (RGB)
pixel_vals = image.reshape((-1,3))

# Convert to float type
pixel_vals = np.float32(pixel_vals)
#the below line of code defines the criteria for the algorithm to stop running, #which will happen
is 100 iterations are run or the epsilon (which is the required accuracy) #becomes 85%

criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 100, 0.85)

# then perform k-means clustering with number of clusters defined as 3
```

#also random centres are initially chosed for k-means clustering k = 3

```
retval, labels, centers = cv2.kmeans(pixel_vals, k, None, criteria, 10,  
cv2.KMEANS_RANDOM_CENTERS)
```

# convert data into 8-bit values

```
centers = np.uint8(centers)
```

```
segmented_data = centers[labels.flatten()]
```

# reshape data into the original image dimensions

```
segmented_image = segmented_data.reshape((image.shape))
```

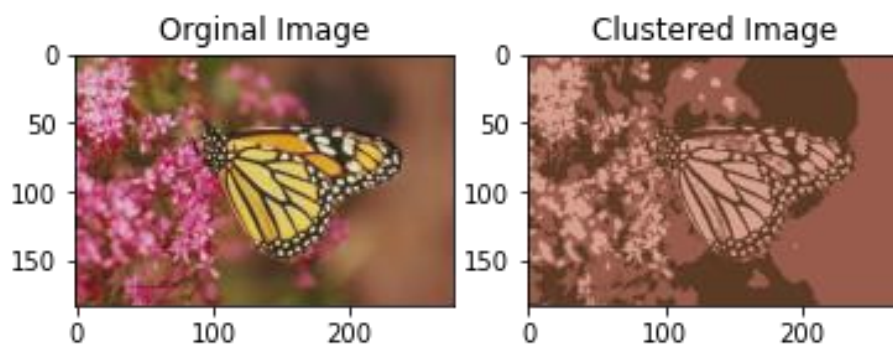
```
print(segmented_image.shape)
```

```
plt.subplot(122)
```

```
plt.imshow(segmented_image)
```

```
plt.title("Clustered Image")
```

### Sample Input and Output



### Result:

Thus, a program has been written to segment an image using K-means algorithm.

**Ex. No. 6.****Morphological Operations****Aim**

Implementation of morphological dilation and erosion operations for a given input image.

**Theoretical Concept:**

**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.

**Dilation :** Increases the object area

Used to accentuate features

**Working of dilation:**

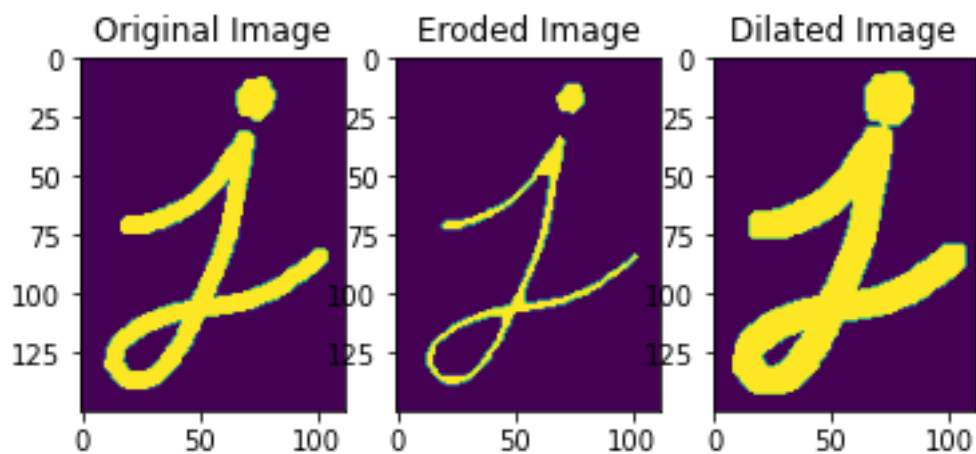
1. A kernel(a matrix of odd size(3,5,7) is convolved with the image
2. A pixel element in the original image is '1' if at least one pixel under the kernel is '1'.
3. It increases the white region in the image or the size of the foreground object increases

**Implementation**

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
img = cv2.imread('j.png',0)
# use a 5x5 kernel with full of ones
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.
erosion = cv2.erode(img,kernel,iterations = 1)
dilation = cv2.dilate(img,kernel,iterations = 1)
plt.subplot(131)
```

```
plt.imshow(img)
plt.title("Original Image")
plt.subplot(132)
plt.imshow(erosion)
plt.title("Eroded Image")
plt.subplot(133)
plt.imshow(dilation)
plt.title("Dilated Image")
```

### Sample Input and Output



### Result

Thus a program has been written to implement morphological dilation and erosion operations for a given input image.

**Ex. No 7.****Feature Extraction Techniques****Aim**

To write the programs to extract SIFT and SURF features for given input image samples

**Theoretical Concept:**

**SIFT** : The scale-invariant feature transform (SIFT) is a feature detection algorithm in computer vision to detect and describe local features in images. SIFT keypoints of objects are first extracted from a set of reference images and stored in a database.

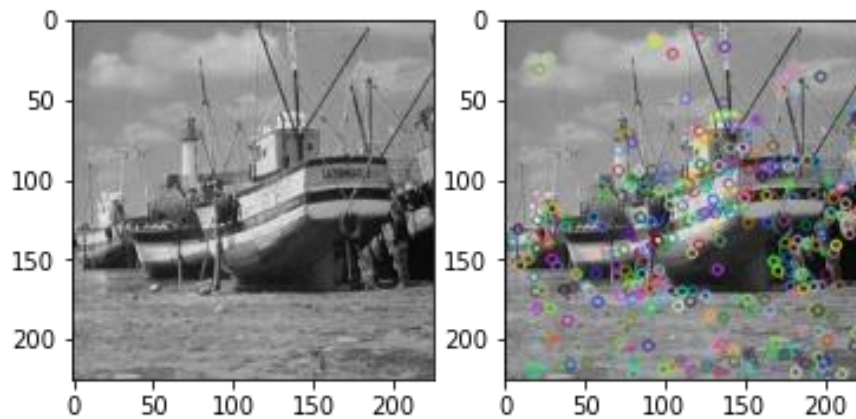
**SURF** : The **SURF** method (Speeded Up Robust Features) is a fast and robust algorithm for local, similarity invariant representation and comparison of images. Similarly to many other local descriptor-based approaches, interest points of a given image are defined as salient features from a scale-invariant representation.

**Implementation****a. Scale-Invariant Feature Transform (SIFT)**

```
import numpy as np
import cv2 as cv
from matplotlib import pyplot as plt
img = cv.imread('ship.jpg')
#plt.subplot(1, 2, 1)
plt.subplot(121)
plt.imshow(img)
# convert to greyscale
gray= cv.cvtColor(img,cv.COLOR_BGR2GRAY)
# create SIFT feature extractor
sift = cv.SIFT_create()
# detect features from the image
kp = sift.detect(gray,None)
# draw the detected key points
img=cv.drawKeypoints(gray,kp,img)
plt.subplot(122)
plt.imshow(img)
```

```
plt.show()
```

### Sample Input and Output



#### b. Speeded Up Robust Features (SURF)

```
import numpy as np
import cv2 as cv
from matplotlib import pyplot as plt
img = cv.imread('home.jpg',0)
# Create SURF object. You can specify params here or later.
# Here I set Hessian Threshold to 400
surf = cv.xfeatures2d.SURF_create(200)
# Find keypoints and descriptors directly
kp, des = surf.detectAndCompute(img,None)
len(kp)
# Check present Hessian threshold
print( surf.getHessianThreshold() )
# We set it to some 50000. Remember, it is just for representing in picture.
# In actual cases, it is better to have a value 300-500
surf.setHessianThreshold(50000)
# Again compute keypoints and check its number.
kp, des = surf.detectAndCompute(img,None)
print( len(kp) )
```





**Ex. No. 8.****Mouse as a Paint Brush****Aim**

To implement a program to use the mouse as a paint brush using double-click.

**Theoretical Concept:**

OpenCV provides a facility to use the mouse as a paint brush or a drawing tool. Whenever any mouse event occurs on the window screen, it can draw anything. Mouse events can be left-button down, left-button up, double-click, etc. It gives us the coordinates (x,y) for every mouse event. By using these coordinates, we can draw whatever we want.

**To Draw Circle**

To draw a circle on the window screen, we first need to create a mouse callback function by using the cv2.setMouseCallback() function. The mouse callback function is facilitated by drawing a circle using double-click.

**Implementation**

```
import cv2

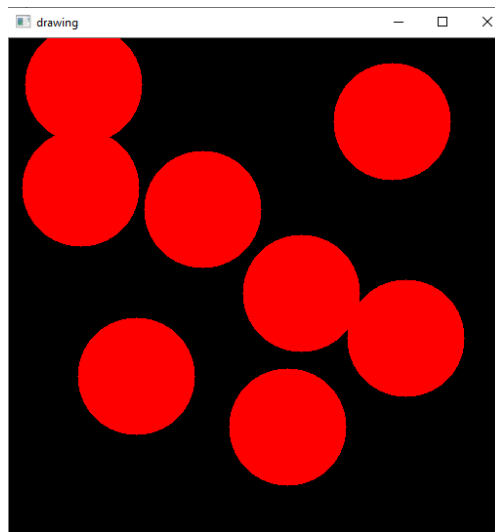
import numpy as np

# Creating mouse callback function
def draw_circle(event,x,y,flags,param):
    if(event == cv2.EVENT_LBUTTONDBLCLK):
        cv2.circle(img,(x,y),100,(255,255, 0),-1)

# Creating a black image, a window and bind the function to window
img = np.zeros((512,512,3), np.uint8)
cv2.namedWindow('image')
cv2.setMouseCallback('image',draw_circle)

while(1):
    cv2.imshow('image',img)
    if cv2.waitKey(20) & 0xFF == 27:
        break:
cv2.destroyAllWindows()
```

## Sample Input and Output



## Sample Input and Output:

The program has been written to use the mouse as a paint brush using double-click.

## SPEECH PROCESSING

### LIBROSA :

- A Python package for audio and music signal processing.
- Audio processing includes digital signal processing, machine learning, information retrieval, and musicology.
- Provides implementations of a variety of common functions used in the field of audio signal processing.

### Reading and plotting of audio files and samples

One option to read audio is to use LIBROSA's function **librosa.load**.

- Per default, librosa.load resamples the audio to 22050 Hz . Setting sr=None keeps the native sampling rate.
- The loaded audio is converted to a float with amplitude values lying in the range of  $[-1,1]$  .
- librosa.load is essentially a wrapper that uses either PySoundFile or audioread.
- When reading audio, librosa.load first tries to use PySoundFile. This works for many formats, such as WAV, FLAC, and OGG. However, MP3 is not supported. When PySoundFile fails to read the audio file (e.g., for MP3), a warning is issued, and librosa.load falls back to another library called audioread. When ffmpeg is available, this library can read MP3 files.

### # Playing audio file

```
import librosa
```

```
import IPython
```

```
import matplotlib.pyplot as plt
```

```
#path of the audio file
```

```
audio_data = 'F:\speech.wav'
```

```
#This returns an audio time series as a numpy array with a default sampling rate(sr) of 22KHZ
```

```
x = librosa.load(audio_data, sr=None)
```

```
#We can change this behavior by resampling at sr=44.1KHz.
```

```
x = librosa.load(audio_data, sr=44000)
```

```
IPython.display.Audio(audio_data)
```

### **#plotting audio file**

```
from scipy.io.wavfile import read
import matplotlib.pyplot as plt
import librosa as lr
import IPython
import numpy as np

audio='speech'

#This returns an audio time series as a numpy array with a default sampling rate(sr) of 22KHZ
y, sr = lr.load('./{ }.wav'.format(audio))

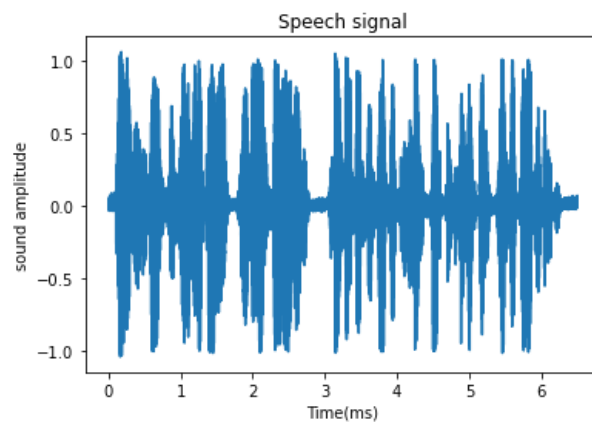
time = np.arange(0,len(y))/sr

fig, ax = plt.subplots() #returns single array of objects, Create just a figure and only one subplot
ax.plot(time,y)

plt.title("Speech signal")

ax.set(xlabel='Time(ms)',ylabel='sound amplitude')

plt.show()
```



### **Reading and plotting audio samples**

```
from scipy.io.wavfile import read
import matplotlib.pyplot as plt

# read audio samples
input_data = read("F:\speech.wav")
audio = input_data[1]

# plot the first 1024 samples
```

```
plt.plot(audio[0:1024])

# label the axes

plt.ylabel("Amplitude")

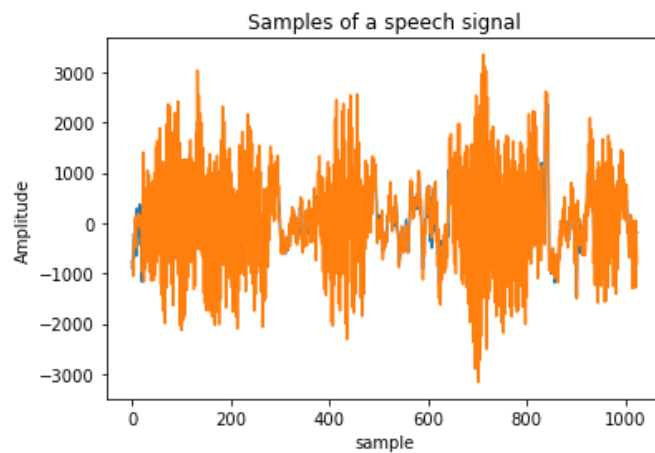
plt.xlabel("sample")

# set the title

plt.title("Samples of a speech signal")

# display the plot

plt.show()
```



### **Resampling at sr=44.1KHz.**

```
x = lr.load(audio_data, sr=44000)

print(x)

IPython.display.Audio(audio_data)
```

Output:

```
(array([-0.02462068, -0.02384715, -0.02957482, ...,  0.07869781,
        0.06410387,  0.04024108], dtype=float32), 44000)
```

## Ex. No. 9 Pitch, Energy and Zero Crossing Rate of speech signal

### Aim:

To find the pitch, Energy and Zero crossing rate of given input speech signal.

### Theoretical Concepts

#### Pitch :

Pitch, in speech, the relative highness or lowness of a tone as perceived by the ear, which depends on the number of vibrations per second produced by the vocal cords. Pitch is the main acoustic correlate of tone and intonation

#### Energy :

Sum of squares of all the samples in the frame

$$E_l = \sum_{n=0}^{N-1} \tilde{x}_l^2(n)$$

#### Zero crossing :

Zero crossing is said to occur if successive samples have different algebraic signs. Simple measure of the frequency content of a signal.

$$Z_l = \frac{1}{N} \sum_{n=0}^{N-1} \frac{1}{2} |\text{sgn}(\tilde{x}_l(n)) - \text{sgn}(\tilde{x}_l(n+1))|$$

where

$$\text{sgn}(\tilde{x}_l(n)) = \begin{cases} 1, & \text{if } x(n) \geq 0 \\ -1, & \text{if } x(n) < 0 \end{cases}$$

#### Implementation :

```
import wave
import numpy as np
import matplotlib.pyplot as plt
from scipy.io.wavfile import read, write
import math

# Calculate the energy of each frame 256 samples are one frame
def calEnergy(wave_data):
    energy = []
    sum = 0
    for i in range(len(wave_data)):
        sum = sum + (int(wave_data[i]) * int(wave_data[i]))
```

```

        if (i + 1) % 256 == 0 :
            energy.append(sum)
            sum = 0
        elif i == len(wave_data) - 1 :
            energy.append(sum)
    return energy

#####read the wav file and parameters #####
f = wave.open("sample1.wav" ,"rb")

# getparams() returns the format information of all WAV files at once
params = f.getparams()

# nframes Number of sampling points
nchannels, sampwidth, framerate, nframes = params[:4]

# readframes() Read data according to the sampling point
Str_data = f.readframes(nframes) # str_data is a binary string

# gets the frame rate
Str_data = np.frombuffer(Str_data, dtype="int16")
f_rate = f.getframerate()

# Convert to a two-byte array form (two bytes per sample point)
wave_data = np.fromstring(Str_data, dtype = np.short)

print( "Number of sample points:" + str(len(wave_data))) #output should be the number of
sample point

# to Plot the x-axis in seconds

# you need get the frame rate

# and divide by size of your signal

# to create a Time Vector

# spaced linearly with the size

# of the audio file

#####

time = np.linspace(0, # start
                    len(Str_data) / f_rate,
                    num = len(Str_data)

```

```

    )
plt.figure(1)
plt.title("Speech Wave")
plt.xlabel("Time")
plt.plot(time, Str_data)
plt.show()

```

```

energy = calEnergy(wave_data)

```

```

plt.figure()
plt.plot(energy)
plt.axis('tight')
plt.xlabel('Frames')
plt.ylabel('Energy')
plt.title('energy calculation')
plt.show()

```

```

#####Pitch Calculation#####

```

```

FRAME_SIZE = 1024

```

```

def ProcessFrame(frame, Fs):

```

```

    freq = max(frame)

```

```

    return freq

```

```

Fs, data = read('sample1.wav')

```

```

numFrames = int(len(data) / FRAME_SIZE)

```

```

frequencies = np.zeros(numFrames)

```

```

for i in range(numFrames):

```

```

    frame = data[i * FRAME_SIZE : (i + 1) * FRAME_SIZE]

```

```

    frequencies[i] = ProcessFrame(frame.astype(float), Fs)

```

```

plt.figure()

```

```

plt.plot(frequencies)

```

```

plt.axis('tight')

```

```

plt.xlabel('Frames')

```



```

plt.ylabel('frequency(Hz)')
plt.title('Pitch')
plt.show()

##### Zero crossing#####

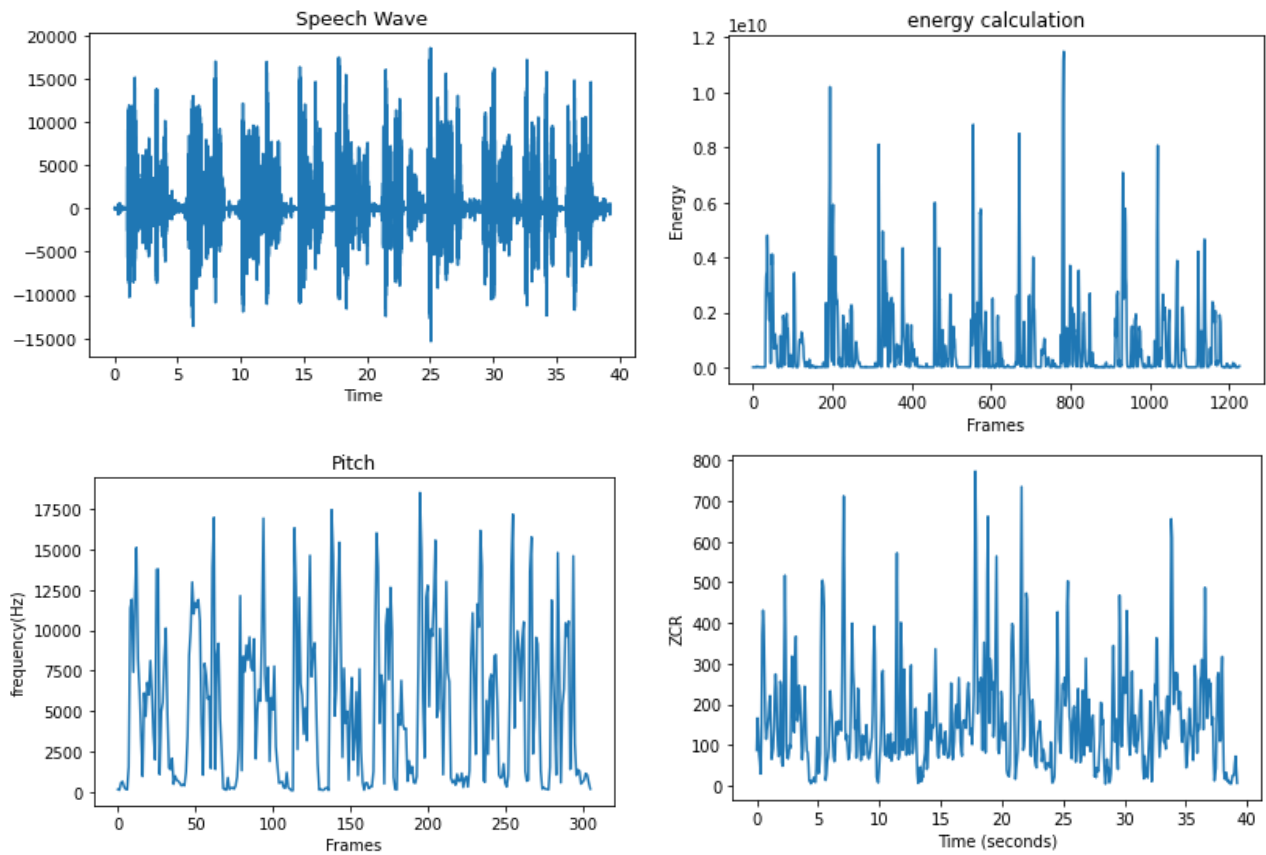
def ZeroCR(waveData,frameSize,overlap):
    wlen = len(waveData)
    step = frameSize - overlap
    frameNum = math.ceil(wlen/step)
    zcr = np.zeros((frameNum,1))
    for i in range(frameNum):
        curFrame = waveData[np.arange(i*step,min(i*step+frameSize,wlen))]
        #To avoid DC bias, usually we need to perform mean subtraction on each frame
        curFrame = curFrame - np.mean(curFrame) # zero-justified
        zcr[i] = sum(curFrame[0:-1]*curFrame[1::]<=0)
    return zcr

overlap = 512

#Read wav file and parameters from energy programm
wave_data.shape = -1, 1
zcr = ZeroCR(wave_data,FRAME_SIZE,overlap)
time2 = np.arange(0, len(zcr)) * (len(wave_data)/len(zcr) / f_rate)
plt.plot(time2, zcr)
plt.ylabel('ZCR')
plt.xlabel('Time (seconds)')
plt.show()
f.close()

```

## Sample Input and Output



## Result :

Thus the program has been written to find pitch, energy and zero crossing rate from the given speech signal.

**Ex. No. 10****Convolution and Correlation of speech signals****(a) Convolution of speech signals****Aim:**

Write a program to perform Convolution of two different speech signals.

**Theoretical Concept:**

Convolution is a mathematical way of combining two signals to form a third signal. Convolution is a formal mathematical operation, just as multiplication and addition.

The convolution is the same operation as multiplying the polynomials whose coefficients are the elements of two signals. The convolution of two signals  $u(n)$  and  $v(n)$  is given by

$$w(k) = \sum_{j=-\infty}^{\infty} u(j)v(k+1-j)$$

The output sample is simply a sum of products involving simple arithmetic operations such as additions, multiplications and delays. But practically  $u(n)$  and  $v(n)$  are finite in length. If the lengths of the two sequences being convolved are  $m$  and  $n$ , then the resulting sequence after convolution is of length  $m+n-1$  and is given by

$$w(k) = \sum_{j=\max(1, k+1-n)}^{\min(k, m)} u(j)v(k+1-j)$$

When  $m = n$ , this gives

$$w(1) = u(1)v(1)$$

$$w(2) = u(1)v(2) + u(2)v(1)$$

$$w(3) = u(1)v(3) + u(2)v(2) + u(3)v(1)$$

...

$$w(n) = u(1)v(n) + u(2)v(n-1) + \dots + u(n)v(1)$$

...

$$w(2n-1) = u(n)v(n)$$

**Implementation**

```
#convolution of two audio signals
```

```
from scipy import signal
```

```
import numpy as np
```

```
import librosa
```

```
import matplotlib.pyplot as plt
```

```
d_file = "set1.wav"
```

```
d1_file="set2.wav"
```

```
#load audio files with librosa
```

```

d, sr = librosa.load(d_file,8000)
#print(sr)
d1, sr = librosa.load(d1_file,8000)

plt.subplot(2,2,1)
t=np.linspace(0,len(d)/sr,len(d))
plt.plot(t,d, lw=1)
#plt.xlabel('Time(s) ')
#plt.ylabel('Amplitude')
#Convolve two N-dimensional arrays.Convolve in1 and in2.discrete, linear convolution

y=signal.convolve(d,d1)
#plotting subplot nrows,ncols,subplot
#plt.plot(d,'r')
plt.grid(True)

#Naming the x-axis, y-axis
plt.xlabel('Time(s)')
plt.ylabel('Amplitude')

#plotting subplot nrows,ncols,subplot
plt.subplot(2,2,2)
t=np.linspace(0,len(d1)/sr,len(d1))
plt.plot(t,d1, lw=1)
#plt.plot(d1,'g')
plt.grid(True)
#Naming the x-axis, y-axis
plt.xlabel('Time(s)')
plt.ylabel(' Amplitude')

#plotting subplot nrows,ncols,subplot
plt.subplot(2,2,3)

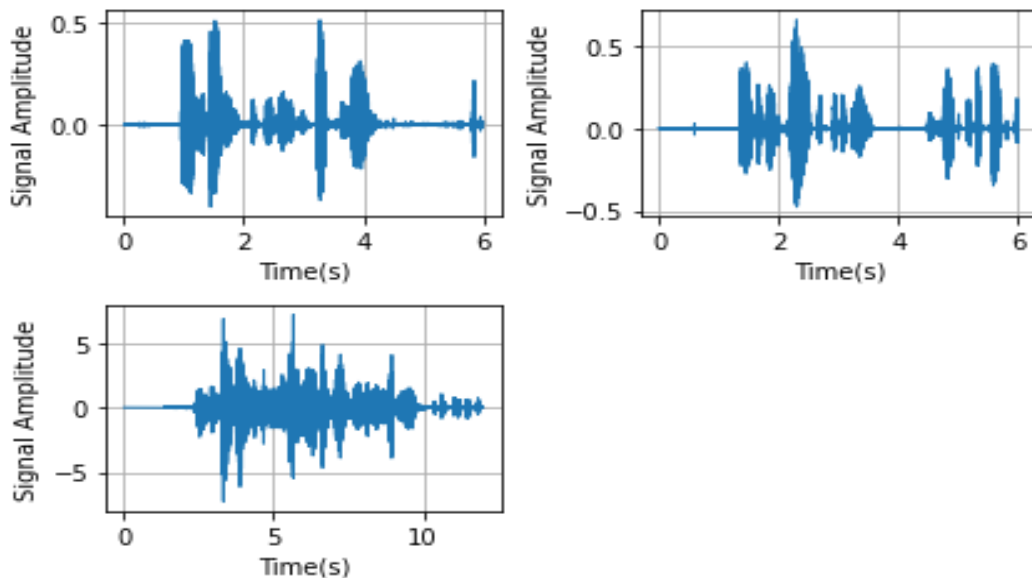
```

```

t=np.linspace(0,len(y)/sr,len(y))
plt.plot(t,y, lw=1)
#plt.plot(y,'b')
plt.grid(True)
#Naming the x-axis, y-axis
plt.title('Covolution of signals')
plt.xlabel('Time(s)')
plt.ylabel(' Amplitude')
#Adds space between different subplots
plt.tight_layout()
plt.grid(True)
#To load the display window
plt.show()
plt.grid(True)
#To load the display window
plt.show()

```

### Sample Input and Output



### Result:

Thus the program has been written to compute the convolution of two input speech signals



## (b) Correlation of Speech signals

### Aim:

Write a program to perform Correlation of speech signals.

### Theoretical Concept :

Correlation is basically used to compare two signals. Correlation measures the similarity between two signals. The correlation process is essentially the convolution of two sequences in which one of the sequence has been reversed. It is classified into two types namely auto correlation and cross correlation.

#### Auto correlation

The autocorrelation of a signal describes the similarity of a signal against a time-shifted version of itself. The autocorrelation is useful for finding repeated patterns in a signal. For example, at short lags, the autocorrelation can tell us something about the signal's fundamental frequency. For longer lags, the autocorrelation may tell us something about the tempo of a signal.

$$r(k) = \sum_n x(n)x(n-k)$$

In above equation, k is often called the **lag** parameter. r(k) is maximized at k=0 and is symmetric about k.

#### Cross correlation

Cross correlation between a pair of signals x(n) and y(n) is given by

$$r_{xy}(l) = \sum_{n=-\infty}^{\infty} x(n)y(n-l), \quad l = 0, \pm 1, \pm 2, \dots$$

The parameter l called lag, indicates the time-shift between the pair. The time sequence y(n) is said to be shifted by l samples with respect to the reference sequence x(n) to the right for the positive values of l, and shifted by l samples to the left for negative values of l.

#### #Auto correlation

# Importing the libraries.

```
import numpy as np
```

```
import librosa
```

```
import matplotlib.pyplot as plt
```

```
d_file = "speech.wav"
```

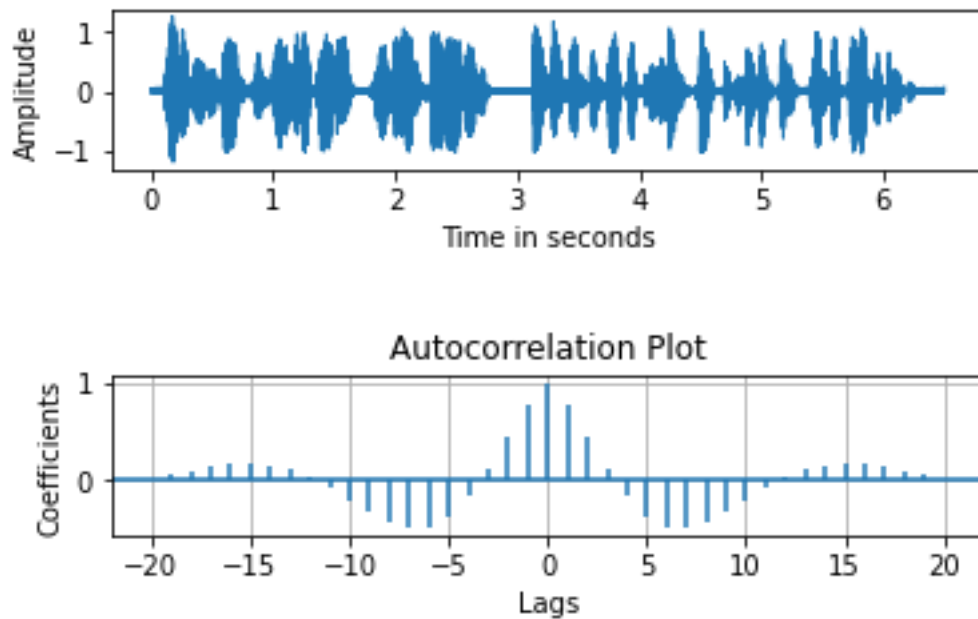
```
d, sr = librosa.load(d_file, 8000)
```

```
plt.subplot(2,1,1)
```

```
t=np.linspace(0,len(d)/sr,len(d))
plt.plot(t,d, lw=1)
plt.xlabel("Time in seconds")
plt.ylabel('Amplitude')
#computing autocorrelation using librosa
r = librosa.autocorrelate(d, max_size=10000)
plt.subplot(2,1,2)
print(r.shape)
# Adding plot title
plt.title("Autocorrelation Plot")
# Providing x-axis name.
plt.xlabel("Lags")
plt.ylabel('Coefficients')
# Plotting the Autocorreleation plot
plt.acorr(r, maxlags = 20)
# Displaying the plot
print("The Autocorrelation plot for the data is:")
plt.grid(True)
plt.tight_layout(pad=3.0)
#plt.tight_layout()
plt.show()
```



## Sample Input and Output



### # crosscorrelation

```
import librosa
import matplotlib.pyplot as plt
from scipy import signal
import numpy as np

d_file = "sample1.wav"
d1_file="sample2.wav"

#load audio files with librosa
d, sr = librosa.load(d_file,8000)
d1, sr = librosa.load(d1_file,8000)
#t=np.linspace(0,len(d)/sr,len(d))
#plt.plot(t,d, lw=1)
#print array values of first .wav file
print(d)
#print array values of second .wav file
print(d1)
```

```

#cross correlation of two arrays of same size i.e. two .wav files of same size
correlation = np.correlate(d, d1, mode="full")
lags = np.arange(-100, 100)
print(correlation)
#plotting subplot nrows,ncols,subplot
plt.subplot(2,2,1)
plt.title('speech file1')
t=np.linspace(0,len(d)/sr,len(d))
plt.plot(t,d, lw=1)
#plotting first .wav file
#plt.plot(d,'g')
plt.grid(True)
plt.xlabel("Time(s)")
plt.ylabel('Amplitude')
#Naming the x-axis, y-axis
#plotting subplot nrows,ncols,subplot
plt.subplot(2,2,2)
plt.title('speech file2')
t=np.linspace(0,len(d1)/sr,len(d1))
plt.plot(t,d1, lw=1)
#plotting second .wav file
#plt.plot(d,'r')
#grid display
plt.grid(True)
plt.xlabel("Time(s)")
plt.ylabel('Amplitude')
#plotting subplot nrows,ncols,subplot
# Plotting the crosscorrelation plot.
plt.subplot(2,2,3)

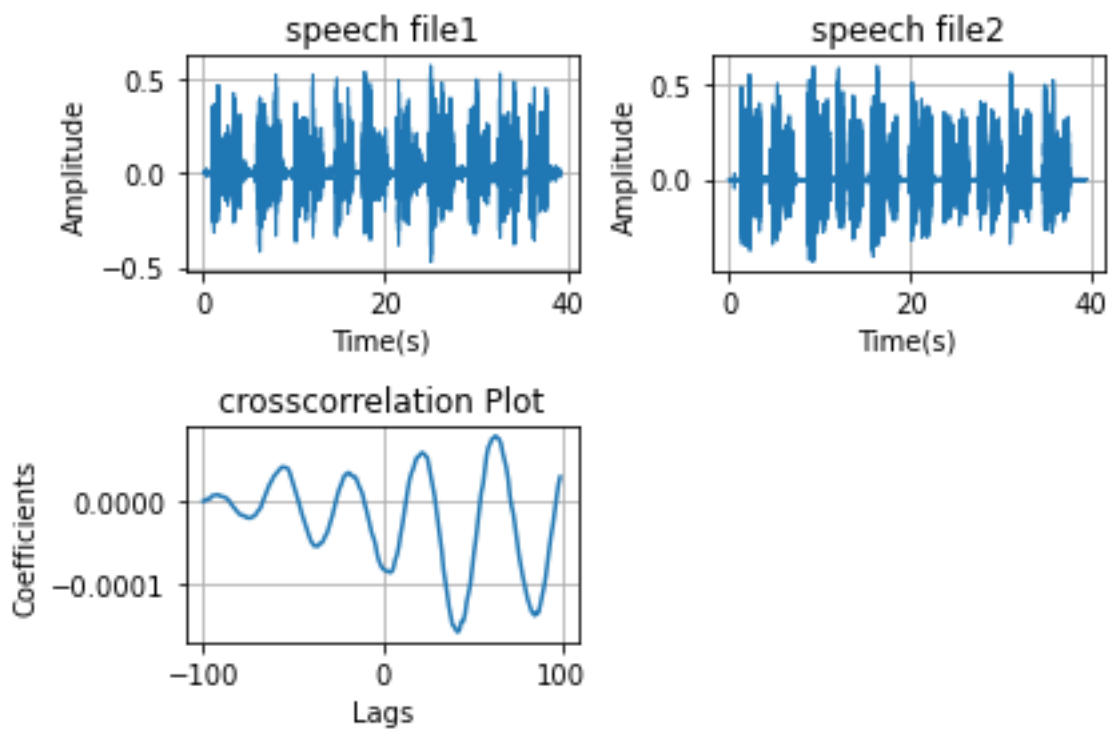
```

```

plt.plot(lags,correlation[0:200])
# Adding plot title.
plt.title("crosscorrelation Plot")
# Providing x-axis name.
plt.xlabel("Lags")
plt.ylabel('Coefficients')
#grid display
plt.grid(True)
#Adds space between different subplots
plt.tight_layout()
plt.show()

```

### Sample Input and Output



### Result :

Thus the program has been written to find the Correlation coefficients of given input speech signal.

**Ex. No. 11****Low pass Filter and High Pass Filter****Aim :**

To apply Low pass filter and High pass filter in the given input speech signal

**Theoretical Concept:****Low pass filter :**

A low-pass filter is a filter that passes signals with a frequency lower than a selected cutoff frequency and attenuates signals with frequencies higher than the cutoff frequency. The exact frequency response of the filter depends on the filter design. A low-pass filter is the complement of a high-pass filter.

**High-pass filter**

A high-pass filter (HPF) is an electronic filter that passes signals with a frequency higher than a certain cutoff frequency and attenuates signals with frequencies lower than the cutoff frequency. The amount of attenuation for each frequency depends on the filter design

#####High Pass filter#####

```
import numpy as np
```

```
import scipy.signal as sg
```

```
import matplotlib.pyplot as plt
```

```
import librosa
```

```
from scipy.signal import butter,filtfilt
```

```
d_file = "set2.wav"
```

```
d, sr = librosa.load(d_file,8000)
```

```
t=np.linspace(0,len(d)/sr,len(d))
```

```
plt.plot(t,d, lw=1)
```

```
plt.xlabel('Time(s)')
```

```
plt.ylabel('Amplitude')
```

```
plt.title('Original speech file')
```

```
plt.grid(True)
```

```
plt.show()
```

```
#high-pass filter (2000 Hz cutoff frequency)
```

```
#butter(n,Wn)returns the transfer function coefficients
```

```
#of an nth-order lowpass digital Butterworth filter with normalized cutoff frequency Wn.
```

```
#butter(n,Wn,ftype) designs a lowpass, highpass, bandpass, or bandstop Butterworth filter,
```

#depending on the value of ftype and the number of elements of Wn.

```
b, a = sg.butter(4, 2000. / (sr / 2.), 'high')
```

#performs zero-phase digital filtering by processing the input data, x, in both the forward and reverse directions.

#After filtering the data in the forward direction, filtfilt reverses the filtered sequence and runs it back through the filter.

```
d_fil = sg.filtfilt(b, a, d)
```

#plotting signal

```
fig, ax = plt.subplots(1, 1, figsize=(6, 3))
```

#equal sized arrays

```
t = np.linspace(0., len(d) / sr, len(d))
```

#plotting signal with line width =1

```
ax.plot(t, d, lw=1)
```

#plotting filtered signal with line width=1

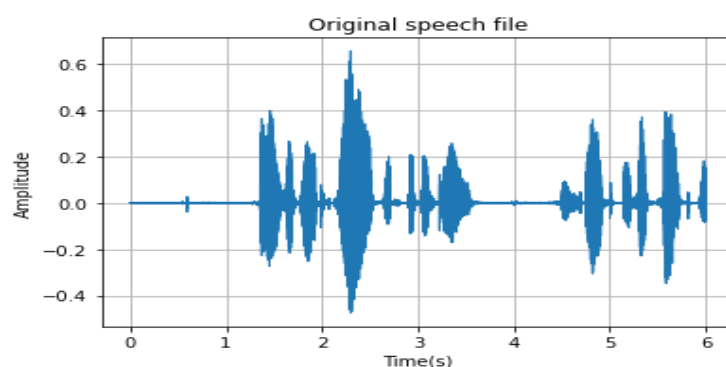
```
ax.plot(t, d_fil, lw=1)
```

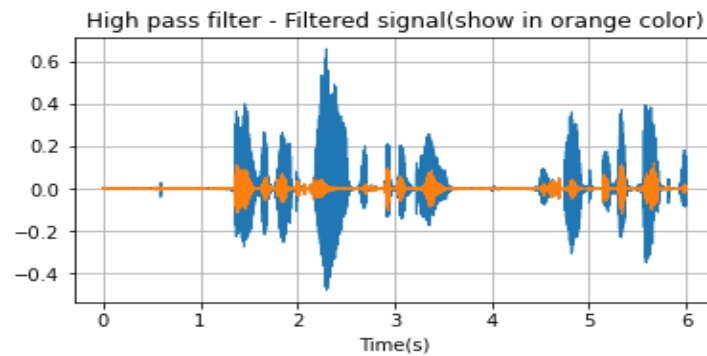
```
plt.xlabel("Time(s)")
```

```
plt.title('High pass filter - Filtered signal(show in orange color)')
```

```
plt.grid(True)
```

```
plt.show()
```





#####Low Pass Filter#####

```
import numpy as np
```

```
import scipy.signal as sg
```

```
import matplotlib.pyplot as plt
```

```
import librosa
```

```
from scipy.signal import butter, filtfilt
```

```
d_file = "F:\speech.wav"
```

```
d, sr = librosa.load(d_file, 8000)
```

```
print(d)
```

```
#Butterworth low-pass filter applied to this sound (500 Hz cutoff frequency)
```

```
#butter(n,Wn)returns the transfer function coefficients
```

```
#of an nth-order lowpass digital Butterworth filter with normalized cutoff frequency Wn.
```

```
#butter(n,Wn,ftype) designs a lowpass, highpass, bandpass, or bandstop Butterworth filter,
```

```
#depending on the value of ftype and the number of elements of Wn.
```

```
b, a = sg.butter(4, 500. / (sr / 2.), 'low')
```

```
#performs zero-phase digital filtering by processing the input data, x, in both the forward and reverse directions.
```

```
#After filtering the data in the forward direction, filtfilt reverses the filtered sequence and runs it back through the filter.
```

```
d_fil = sg.filtfilt(b, a, d)
```

```
#plotting signal
```

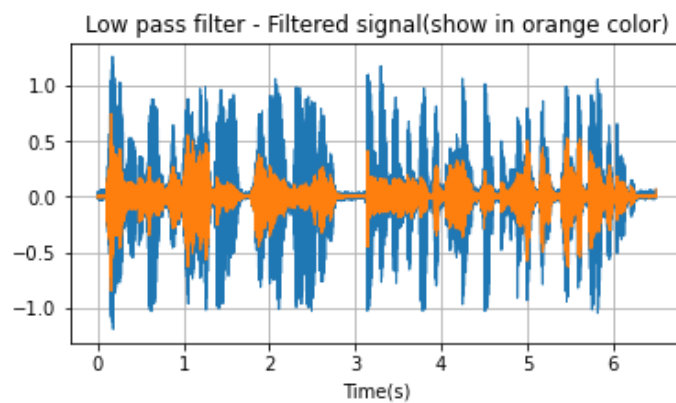
```
fig, ax = plt.subplots(1, 1, figsize=(6, 3))
```

```
#equal sized arrays
```

```
t = np.linspace(0., len(d) / sr, len(d))
```

```
#plotting signal with line width =1
```

```
ax.plot(t, d, lw=1)
#plotting filtered signal with line width=1
ax.plot(t, d_fil, lw=1)
plt.xlabel('Time(s)')
plt.title('Low pass filter - Filtered signal(show in orange color)')
plt.grid(True)
plt.show()
```



### Result:

Thus the program has been implemented to apply low pass filter and high pass filter to the given input speech signal.

**Ex. No. 12****Mel Frequency Cepstral Coefficients****Aim**

To write a program to extract MFCC feature from sample speech signal.

**Theoretical concept:**

Mel-frequency cepstral coefficients (MFCCs) are coefficients that collectively make up an MFC.[1] They are derived from a type of cepstral representation of the audio clip (a nonlinear "spectrum-of-a-spectrum"). The difference between the cepstrum and the mel-frequency cepstrum is that in the MFC, the frequency bands are equally spaced on the mel scale, which approximates the human auditory system's response more closely than the linearly-spaced frequency bands used in the normal spectrum. This frequency warping can allow for better representation of sound, for example, in audio compression.

MFCCs are commonly derived as follows:

1. Take the Fourier transform of (a windowed excerpt of) a signal.
2. Map the powers of the spectrum obtained above onto the mel scale, using triangular overlapping windows or alternatively, cosine overlapping windows.
3. Take the logs of the powers at each of the mel frequencies.
4. Take the discrete cosine transform of the list of mel log powers, as if it were a signal.

The MFCCs are the amplitudes of the resulting spectrum.

```
librosa.feature.mfcc(y=None, sr=22050, S=None, n_mfcc=20, dct_type=2, norm='ortho', lifter=0, **kwargs)
```

y = audio time series

sr = number > 0 [scalar] , sampling rate of y

S = np.ndarray [shape=(d, t)] or None, log-power Mel spectrogram

n\_mfcc = int > 0 [scalar], number of MFCCs to return

dct\_type= { 1, 2, 3 }, Discrete cosine transform (DCT) type. By default, DCT type-2 is used.

Norm = None or 'ortho' :If dct\_type is 2 or 3, setting norm='ortho' uses an ortho-normal DCT basis.

Kwarg = sadditional keyword arguments to melspectrogram, if operating on time series input



## Implentation

```
import librosa
y,sr=librosa.load('speech.wav')
print(y)
print("Sample rate: {0}Hz".format(sr))
print("Audio duration: {0}s".format(len(y) / sr))
print(librosa.feature.mfcc(y=y,sr=sr,n_mfcc=13))
```

## Sample Input and Output

```
[-0.01738697 -0.02810948 -0.00662852 ... 0.07072017 0.07115451 0.06516534]
[[-219.37381 -215.54312 -217.26982 ... -245.7114 -245.99171 -244.60828 ]
 [ 55.12384 58.72249 61.233887 ... 66.5266 65.32631 66.8613 ]
 [-30.448849 -27.89038 -22.451464 ... -19.271458 -23.097046 -18.843634 ]
 ...
 [ -6.342108 -9.058981 -9.702562 ... -11.525118 -9.8483515 -6.803407 ]
 [ -2.7326827 -2.249867 1.7608168 ... 1.4896486 0.93037903 2.8084617 ]
 [-13.274951 -11.341682 -6.6493864 ... -2.445509 -5.0610194 -2.5841856 ]]
```

## Result

Thus the program has been written to extract Mel frequency cepstral coefficients from the given input speech signal.