Index

SL	Experiment Name
1.	Write a program to execute the following image pre-processing. • Read images from a folder. • Resize images and save to a folder. • Apply color transform on images and save to a folder. • Normalize images and save into a folder. • Filter images and save into a folder.
2.	Write a program to execute Semantic Segmentation.
3.	Given an image and a mask, determine the region of the image using the mask,write a program to compute the area of the region, then label the region by overlapping the mask over the image.
4.	Write a program to execute the following image enhancement. • Basic Intensity Transformation (Negation, Log transformation, Power low transformation and Piece-wise linear transformation). • Convolution (High pass, Low pass and Laplacian filter).
5.	Write a program to execute the following edge detections • Canny edge detection • Prewitt edge detection • Sobel edge detection
6.	Write a program to execute the following speech preprocessing • Identify sampling frequency • Identify bit resolution • Make down sampling frequency then save the speech signal.
7.	Write a program to display the following region of a speech signal. • Voiced region. • Unvoiced region. • Silence region.
8.	Write a program to compute zero crossing rate (ZCR) using different window function of a speech signal.
9.	Write a program to compute short term auto-correlation of a speech signal.
10.	Write a program to estimate pitch of a speech signal.

Problem No: 01

Problem Name:

Write a program to execute the following image pre-processing.

- Read images from a folder.
- Resize images and save to a folder.
- Apply color transform on images and save to a folder.
- Filter images and save into a folder.
- Normalize images and save into a folder.

Objectives:

To read some images from a folder and show the output.

To resize the images, save to a folder and show the output.

To apply color transform on images, save to a folder and show the output.

To filter images, save into a folder and show the output.

To normalize images, save into a folder and show the output.

Theory:

Reading Images from a Folder and Showing the Output:

The code starts by defining the directory containing the original images (original_folder_path). It then lists all files in the folder using os.listdir(). The number of images is calculated, and if there are any images found, they are displayed using Matplotlib's plt.imshow() function.

Resizing Images, Saving to a Folder, and Showing the Output:

After displaying the original images, the code proceeds to resize each image to dimensions of 300x400 pixels using PIL's resize() function. Resized images are saved to the folder specified by resized_folder_path. The number of resized images is then calculated, and if there are any, they are displayed similarly to the original images.

Applying Color Transform on Images, Saving to a Folder, and Showing the Output:

Next, the code applies a color transformation to the resized images. Specifically, it converts the images to grayscale using PIL's convert() function. The grayscale images are saved to the folder specified by color_transform_folder_path. After saving, the transformed images are displayed using Matplotlib.

Filtering Images, Saving into a Folder, and Showing the Output:

The code then applies a mean filter (box blur) to the grayscale images using PIL's filter() function with the ImageFilter.BoxBlur filter. Filtered images are saved to the folder specified by filtered_folder_path and displayed using Matplotlib.

Normalizing Images, Saving into a Folder, and Showing the Output:

Finally, the code normalizes the color-transformed images by converting them to NumPy arrays, dividing by 255 to scale pixel values to the range [0, 1], and then converting back to images using PIL's fromarray() function. Normalized images are saved to the folder specified by normalized_folder_path and displayed using Matplotlib.

Code:

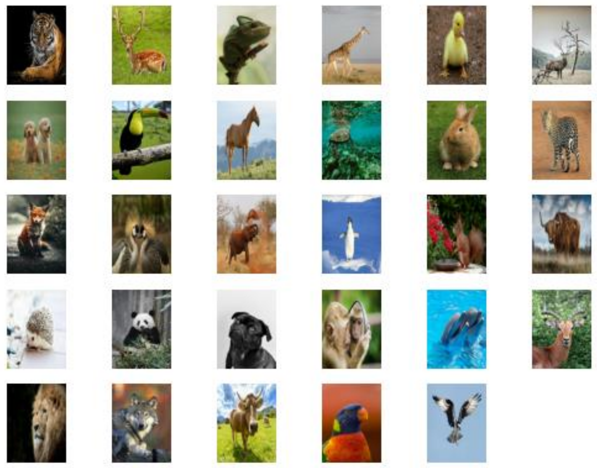
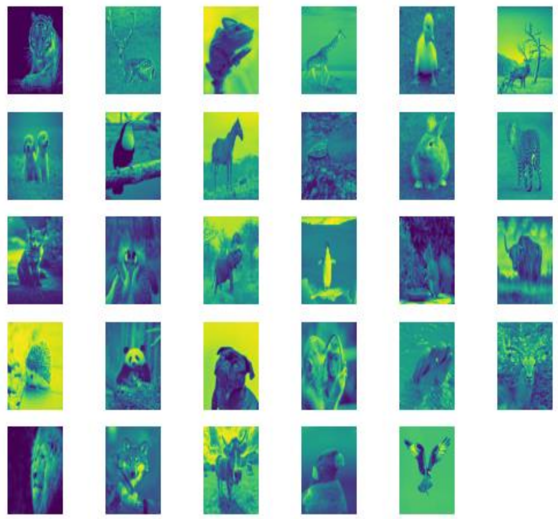
```
import os
from PIL import Image, ImageFilter
import matplotlib.pyplot as plt
import numpy as np
original_folder_path = "/content/drive/
            MyDrive/animal"

# Define the directory to save the resized
images
resized_folder_path=
"/content/drive/MyDrive/resized_images"

# Create the resized folder if it doesn't exist
os.makedirs(resized_folder_path,
exist_ok=True)

# List all files in the original folder
files = os.listdir(original_folder_path)

# Display original images
num_images = len(files)
if num_images == 0:
   print("No images found.")
else:
   num_rows = (num_images - 1) // 6 + 1
   for i in range(num_images):
      plt.subplot(num_rows, 6, i+1)
      plt.imshow(Image.open(os.path.join(
original_folder_path, files[i])))
      plt.axis('off')
   plt.show()
```

```
for file in files:
   file_path = os.path.join(original_folder_path, file)
   if file.lower().endswith(('.png', '.jpg', '.jpeg', '.bmp', '.gif')):
      img = Image.open(file_path)
      img_resized = img.resize((300, 400))
      resized_file_path = os.path.join(resized_folder_path,
file)
      img_resized.save(resized_file_path)
print("This is the resized image (300*400)")
num_resized_images = len(os.listdir(resized_folder_path))
if num_resized_images == 0:
   print("No images found.")
else:
   num_rows = (num_resized_images - 1) // 6 + 1
   for i in range(num_resized_images):
      plt.subplot(num_rows, 6, i+1)

plt.imshow(Image.open(os.path.join(resized_folder_path,
os.listdir(resized_folder_path)[i])))
      plt.axis('off')
   plt.show()
color_transform_folder_path                                  =
"/content/drive/MyDrive/color_transform_images"
os.makedirs(color_transform_folder_path, exist_ok=True)

for file in files:
   file_path = os.path.join(resized_folder_path, file)
   if file.lower().endswith(('.png', '.jpg', '.jpeg', '.bmp', '.gif')):
      img_resized = Image.open(file_path)
      img_color_transformed = img_resized.convert("L")
      color_transformed_file_path                            =
os.path.join(color_transform_folder_path, file)

img_color_transformed.save(color_transformed_file_path)
print("All the Color transformation images")
```

```
# Display all the Color transformation
images
num_images =
len(os.listdir(color_transform_folder_path))
if num_images == 0:
    print("No images found.")
else:
    num_rows = (num_images - 1) // 6 + 1
    for i in range(num_images):
        plt.subplot(num_rows, 6, i+1)
plt.imshow(Image.open(os.path.join(
color_transform_folder_path, files[i])))
        plt.axis('off')
    plt.show()
print("Color transformation and saving
completed.")
filtered_folder_path =
"/content/drive/MyDrive/filtered_images"
os.makedirs(filtered_folder_path,
exist_ok=True)
# Filter and save all images
for file in files:
    file_path = os.path.join(
color_transform_folder_path, file)
    if file.lower().endswith(
  ('.png', '.jpg', '.jpeg', '.bmp', '.gif')):
        img_ct = Image.open(file_path)
        img_filtered =
img_ct.filter(ImageFilter.BoxBlur(radius=2))
        filtered_file_path =
os.path.join(filtered_folder_path, file)
        img_filtered.save(filtered_file_path)
print("The filtered output image")
num_images = len(files)
if num_images == 0:
    print("No images found.")
else:
    num_rows = (num_images - 1) // 6 + 1
    for i in range(num_images):
        plt.subplot(num_rows, 6, i+1)
        plt.imshow(Image.open(os.path.join
(filtered_folder_path, files[i])))
        plt.axis('off')
    plt.show()
print("Filtering and saving completed.")
```

```

# Define the directory to save the normalized images
normalized_folder_path =
"/content/drive/MyDrive/normalized_color_images"
os.makedirs(normalized_folder_path, exist_ok=True)

# List all files in the color-transform folder
files = os.listdir(color_transform_folder_path)

# Normalize and save all images
for file in files:
    file_path = os.path.join(color_transform_folder_path, file)
    if file.lower().endswith(('.png', '.jpg', '.jpeg', '.bmp', '.gif')):
        img_ct = Image.open(file_path)
        img_array = np.array(img_ct)
        img_normalized = img_array / 255.0
        img_normalized = Image.fromarray((img_normalized *
255).astype(np.uint8))
        normalized_file_path =
os.path.join(normalized_folder_path, file)
        img_normalized.save(normalized_file_path)

# Get the list of normalized image files
normalized_files = os.listdir(normalized_folder_path)

# Display the normalized images
num_images = len(normalized_files)
if num_images == 0:
    print("No images found.")
else:
    num_rows = (num_images - 1) // 6 + 1
    for i in range(num_images):
        plt.subplot(num_rows, 6, i+1)

plt.imshow(Image.open(os.path.join(normalized_folder_path,
normalized_files[i])))
        plt.axis('off')
    plt.show()
print("Normalization and saving completed.")
```

Output:

Original images
`This is the resized image (300*400)`

	`All the Color transformation images`
	
	`Color transformation and saving completed.`

`The filtered output image`
 	
`Filtering and saving completed.`
	
	`Normalization and saving completed.`

Problem No: 02

Problem Name:

Write a program to execute Semantic Segmentation.

Objectives:

To understand the semantic segmentation & classify the objects.

Theory:

Image segmentation is a fundamental task in computer vision that involves partitioning an image into multiple segments or regions to simplify its representation and make it more meaningful for further analysis. There are various types of image segmentation techniques, two of which are region-based segmentation and semantic segmentation:

Semantic Segmentation:

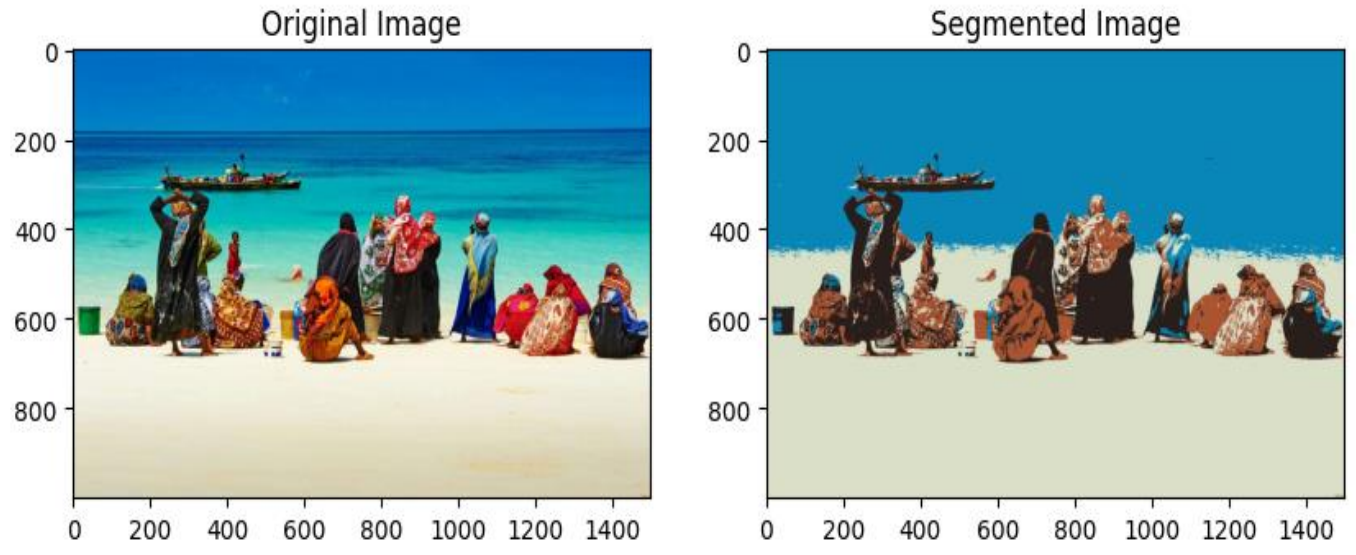
Semantic segmentation is a more advanced form of image segmentation where the goal is to assign a semantic label to each pixel in the image.

Unlike traditional segmentation, where pixels are grouped based on low-level features, semantic segmentation aims to understand the content of the image at a higher level by assigning semantic labels such as "person," "car," "tree," etc., to each pixel.Semantic segmentation is typically performed using deep learning-based approaches, particularly convolutional neural networks (CNNs), which have demonstrated state-of-the-art performance in various segmentation tasks. Fully Convolutional Networks (FCNs), U-Net, and DeepLab are popular architectures used for semantic segmentation tasks. Semantic segmentation finds applications in autonomous driving, scene understanding, medical image analysis, and object detection.

Code:

```
kmeans = KMeans(n_clusters=4)
kmeans.fit(pixels)
# Get the cluster centers and labels
centers = np.uint8(kmeans.cluster_centers_)
labels = kmeans.labels_
# Assign each pixel to its corresponding cluster center
segmented_image = centers[labels]
# Reshape the segmented image to its original shape
segmented_image = segmented_image.reshape(image.shape)
# Display the original image and segmented image
plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.imshow(image)
plt.title('Original Image')
plt.subplot(1, 2, 2)
plt.imshow(segmented_image)
plt.title('Segmented Image')
plt.show()
```

output:



Problem No: 03

Problem Name:

Given an image and a mask, determine the region of the image using the mask, write a program to compute the area of the region, then label the region by overlapping the mask over the image.

Objectives:

To compute the area of the region, then label the region by overlapping the mask over the image and plot the images.

Theory:

Here the segment leverages the OpenCV library for image processing tasks related to medical imaging, specifically aimed at tumor detection and visualization. Initially, the code loads both the original medical image and its corresponding mask, which delineates the tumor area, using the cv2.imread() function. The mask image is loaded in grayscale mode to facilitate subsequent operations. By applying the mask to the original image using bitwise operations, the code isolates the tumor region from the rest of the image. Contours within the mask image are then identified utilizing the cv2.findContours() function, providing a means to delineate the boundary of the tumor. The code calculates the total area encompassed by the mask on the original image by iterating through the detected contours and summing their areas using cv2.contourArea(). The determined tumor region is extracted from the original image based on the mask. Next, the code computes the area of the tumor region by counting the non-zero pixels in the mask. To enhance visualization, the code creates a red-colored mask, overlaying it onto the original image to highlight the tumor area. A bounding rectangle is drawn around the tumor region, and a label "Tumor" is added above the rectangle for clarity. Finally, the results are displayed using matplotlib, with various subplots showcasing the original image, mask, image with mask overlay, detected tumor region, and labeled image. Additionally, the code outputs the area of the determined tumor region, providing quantitative information about its size. Overall, this code exemplifies a basic methodology for tumor detection and visualization in medical images, serving as a foundational component for more advanced image analysis techniques

Code:

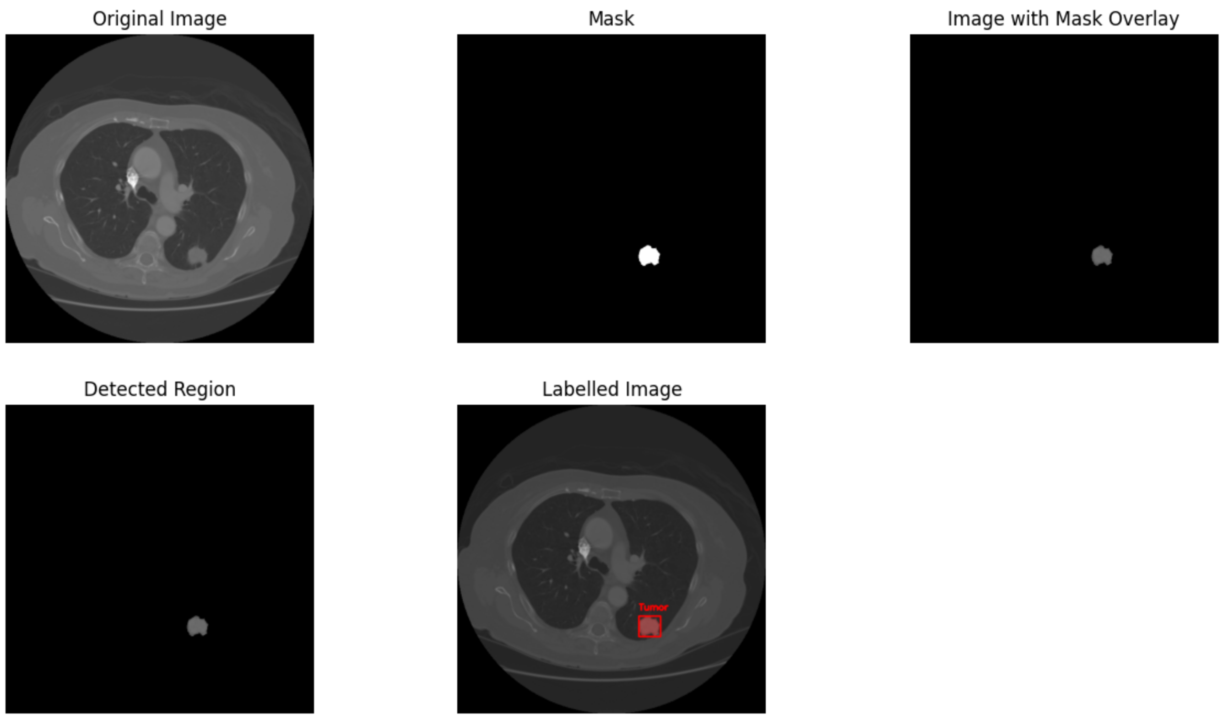
```
import cv2
import numpy as np
import matplotlib.pyplot as plt
image = cv2.imread
("/content/drive/MyDrive/Images.png")
mask = cv2.imread
("/content/drive/MyDrive/Mask.png",
cv2.IMREAD_GRAYSCALE)
masked_image = cv2.bitwise_and
(image, image, mask=mask)
```

```
contours, _ = cv2.findContours(mask,
cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)
mask_area = 0
for contour in contours:
    mask_area += cv2.contourArea(contour)
region = cv2.bitwise_and
(image, image, mask=mask)
area = np.sum(mask != 0)
```

```
red_mask = cv2.cvtColor
(mask, c v2.COLOR_GRAY2BGR)
red_mask[mask != 0] = [0, 0, 255]
labelled_image = cv2.addWeighted
(image, 0.7, red_mask, 0.3, 0)
x, y, w, h = cv2.boundingRect(mask)
cv2.rectangle(labelled_image, (x, y), plt.subplot(2, 3, 3)
plt.imshow(cv2.cvtColor(masked_image,
cv2.COLOR_BGR2RGB))
plt.title("Image with Mask Overlay")
plt.axis("off")
plt.subplot(2, 3, 4)
plt.imshow(cv2.cvtColor(region,
cv2.COLOR_BGR2RGB))
plt.title("Detected Region")
plt.axis("off")
plt.subplot(2, 3, 5)
plt.imshow(cv2.cvtColor(labelled_image,
cv2.COLOR_BGR2RGB))
plt.title("Labelled Image")
plt.axis("off")
```

```

plt.show()

print("Area of the determined region:", area)
(x + w, y + h), (0, 0, 255), 2)
cv2.putText(labelled_image, 'Tumor', (x, y - 10),
cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 0, 255),
2)
plt.figure(figsize=(15, 8))
plt.subplot(2, 3, 1)
plt.imshow(cv2.cvtColor(image,
cv2.COLOR_BGR2RGB))
plt.title("Original Image")
plt.axis("off")

plt.subplot(2, 3, 2)
plt.imshow(mask, cmap='gray')
plt.title("Mask")
plt.axis("off")
```

Output:



```
Area of the determined region: 890
```

Problem No: 04

Problem Name: Write a program to execute the following image enhancement.

- Basic Intensity Transformation (Negation, Log transformation, Power low transformation and Piece-wise linear transformation).
- Image Convolution (High pass, Low pass and Laplacian filter).

Objectives:

To transform and study the intensity of an image using negation, log transformation, power low transformation and piece-wise linear transformation.

To convolution of an image using filters high pass, low pass and laplacian filter

Theory:

Basic intensity transformations are essential tools in image processing that enable the adjustment of pixel intensity values in images. These operations serve as the building blocks for enhancing and manipulating the visual appearance of various types of images, including grayscale and color ones.

Negative Transformation

The negative transformation is achieved by taking the complement of the original pixel intensity values. If the original pixel intensity is denoted by “r,” the corresponding negative intensity, denoted by “s,” can be calculated as:

$$s = L - 1 - r$$

Here, “L” represents the maximum intensity level in the image, often 255 in the case of 8-bit images. This equation subtracts the original intensity from the maximum possible intensity, effectively inverting the pixel values. For example, if the original intensity is 100 in an 8-bit image (L = 255), the negative intensity will be (255–1–100) = 154.

Log Transformation (Logarithm Function)

The log transformation involves applying the logarithm function to each pixel value in an image. This operation spreads out the darker pixel values, making fine details in shadowed regions more visible. The transformation equation is given by:

$$S = C \times \log(1 + R)$$

S represents the transformed pixel value.

R is the original pixel value.

C is a constant that adjusts the degree of enhancement.

Power-law transformations

Power-law transformations, also known as gamma correction, are a class of mathematical transformations used to adjust the tonal and brightness characteristics of an image. These transformations are particularly useful in image processing and computer vision to enhance the visual quality of images or correct for issues related to illumination and contrast.

The basic idea behind power-law transformations is to raise the pixel values of an image to a certain power (exponent) in order to adjust the image's overall brightness and contrast. The general form of a power-law transformation is $O = kI^{\gamma}$

Where:

O is the output pixel value (transformed value). I the input pixel value (original value).

γ is the exponent, which controls the degree of transformation.

k is a constant that scales the result to fit within the desired intensity range.

Image convolution is a fundamental operation in image processing, used for tasks like blurring, sharpening, edge detection, and more. The Laplacian filter is commonly used for edge detection and identifying regions of rapid intensity change in an image.

Low-Pass Filters (LPF): These filters allow low-frequency components of an image to pass through while attenuating higher-frequency components. This process results in the blurring and smoothing of the image. LPFs are particularly useful in reducing noise and removing fine details. The simplest convolution kernel or filter is of size 3x3 with equal weights and is represented as follows:

$$h(n) = \frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

High-pass filtering: High pass filtering, which means the higher frequency components are allowed to pass while low-frequency components are discarded from the original image. The simplest convolution kernel or filter is of size 3x3 with equal weights and is represented as follows:

$$h(n) = \begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

The Laplacian filter: The Laplacian filter is a second-order differential operator, which means it measures the rate of change of the rate of change, or curvature, of the image intensity. It can be represented by a 3x3 kernel:

$$h(n) = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

Code:

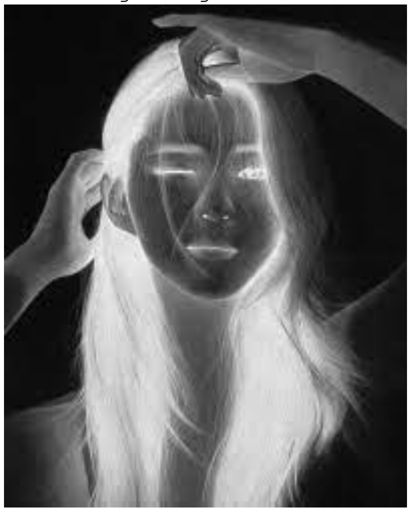
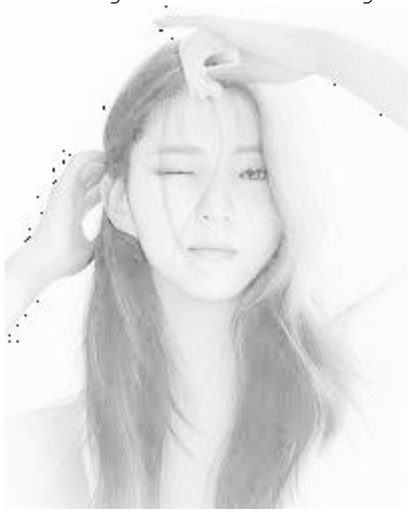
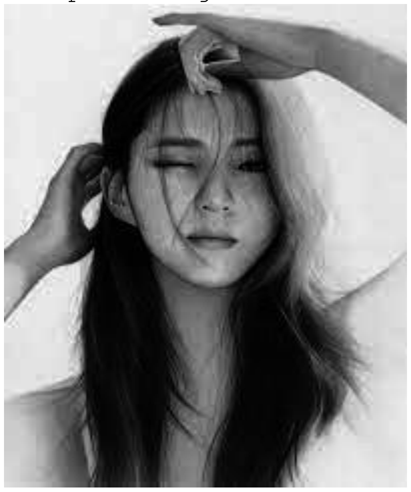
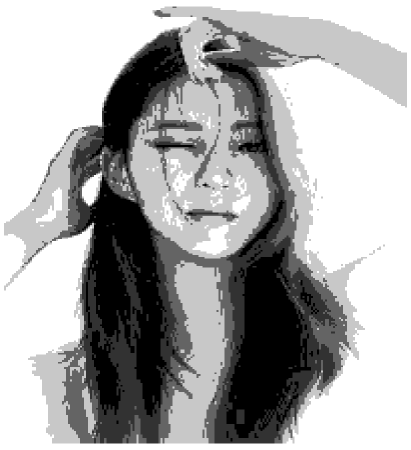
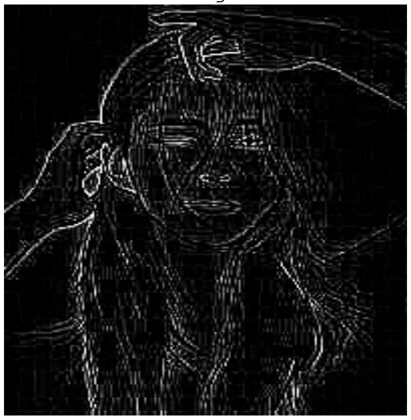
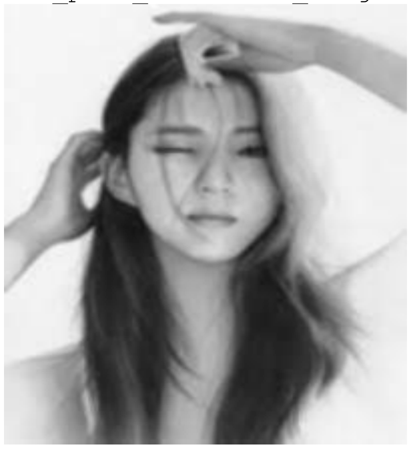
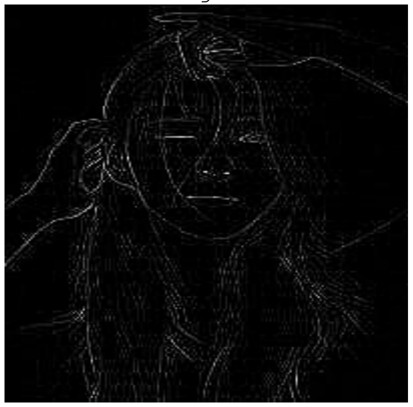
(i) Intensity Transformation Operations on Images

```
import cv2
from google.colab.patches import cv2_imshow
import numpy as np
# Read the image in grayscale
image_gray = cv2.imread("/content/drive/MyDrive/images.jpg", cv2.IMREAD_GRAYSCALE)
# Check if the image is loaded successfully
if image_gray is None:
   print("Error: Could not read the image.")
else:
   # Negate the grayscale image
   negated_image_gray = 255 - image_gray
   # Display the original and negated grayscale images
   print("The orginal image")
   cv2_imshow(image_gray)
   print("The image negation")
   cv2_imshow(negated_image_gray)
   # Apply log transformation
   c = 255 / np.log(1 + np.max(image_gray) + 1e-6)
               # Add a small constant to avoid division by zero
   log_transformed = c * np.log(1 + image_gray)
   log_transformed = np.array(log_transformed, dtype=np.uint8)
   print("The log transform image")
   cv2_imshow(log_transformed)
   # Apply power-law transformation with gamma
   gamma = 2
   adjusted_image = np.uint8(np.power(image_gray / 255.0, gamma) * 255)
   print("The power log transform")
   cv2_imshow(adjusted_image)
   # Define the breakpoints and corresponding mapping values
   breakpoints = [50, 100, 150, 200]
   mapping_values = [0, 50, 100, 200, 255]
   # Apply piecewise-linear transformation
   piecewise = np.piecewise(image_gray, [image_gray < breakpoints[0],
        (image_gray >= breakpoints[0]) & (image_gray < breakpoints[1]),
        (image_gray >= breakpoints[1]) & (image_gray < breakpoints[2]),
        (image_gray >= breakpoints[2]) & (image_gray < breakpoints[3]),
         image_gray >= breakpoints[3]],
    mapping_values)
   print("The piecewise-linear transformation")
   cv2_imshow(piecewise)
```

(ii) Image Convolution (High pass, Low pass and Laplacian filter). import cv2

```
import numpy as np
from matplotlib import pyplot as plt
# Read the image
image = cv2.imread('/content/drive/MyDrive/images.jpg', cv2.IMREAD_GRAYSCALE)
high_pass_kernel = np.array([[-1, -1, -1],
                    [-1,  8, -1],
                    [-1, -1, -1]])
convolved_image = cv2.filter2D(image, -1, high_pass_kernel)
print("Original Image")
cv2_imshow(image)
print("High-pass Filter Convolved Image")
cv2_imshow(convolved_image)
low_pass_kernel = np.ones((3, 3), dtype=np.float32) / 9
low_pass_convolved_image = cv2.filter2D(image, -1, low_pass_kernel)
print("low_pass_convolved_image")
cv2_imshow(low_pass_convolved_image)
laplacian_kernel = np.array([[0, 1, 0],
                    [1, -4, 1],
                    [0, 1, 0]])
lap_filtered_image = cv2.filter2D(image, -1, laplacian_kernel)
print("Display the laplacian filtered images")
cv2_imshow(lap_filtered_image)
```

Output:

The orginal image	The image negation 	The log transform image 
The power log transform 	The piecewise-linear transformation 	
High-pass Filter Convolved Image 	low_pass_convolved_image 	Display the laplacian filtered images 

Problem No: 05

Problem Name:

Write a program to execute the following edge detections

- Canny edge detection
- Prewitt edge detection
- Sobel edge detection

Objectives:

- To study and detect the edge of an image using canny edge detection technique.
- To study and detect the edge of an image using Prewitt edge detection technique.
- To study and detect the edge of an image using Sobel edge detection technique.

Theory:

Edge detection is an image processing technique for finding the boundaries of objects within images. It works by detecting discontinuities in brightness. Edge detection is used for image segmentation and data extraction in areas such as image processing, computer vision, and machine vision.

Here are some of the masks for edge detection.

- Prewitt Operator for Prewitt edge detection technique
- Sobel Operator for Sobel edge detection technique
- Canny edge detection for Canny's edge detection technique

The Prewitt Operator for Prewitt edge detection technique

Prewitt Operator: Prewitt operator is used for edge detection in an image. By using Prewitt operator.we can detects only horizontal and vertical edges. For vertical Edges

Mask for detection of vertical edges $\begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix}$

Above mask will find the edges in vertical direction and it is because the zeros column in the vertical direction. When you will convolve this mask on an image, it will give you the vertical edges in an

image. Horizontal Edges direction

Mask for detection of horizontal edges $\begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix}$

Above mask will find edges in horizontal direction and it is because that zeros column is in horizontal direction. When you will convolve this mask onto an image it would prominent horizontal edges in the image.

Sobel Operator: The sobel operator is very similar to Prewitt operator. It is also a derivate mask and is used for edge detection. Like Prewitt operator sobel operator is also used to detect two kinds of edges in an image:

- Vertical direction
- Horizontal direction

How to differ from Prewitt Operator. The main difference is that in Sobel operator the coefficients of masks are not fixed and they can be adjusted according to our requirement unless they do not violate any property of derivative masks. For vertical Edges

Vertical Mask of Sobel Operator $\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$

This mask works exactly same as the Prewitt operator vertical mask. There is only one difference that is it has “2” and “-2” values in center of first and third column. When applied on an image this mask will highlight the vertical edges. Horizontal Edges direction

Horizontal Mask of Sobel Operator $\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$

Above mask will find edges in horizontal direction and it is because that zeros column is in horizontal direction. When you will convolve this mask onto an image it would prominent horizontal edges in the image. The only difference between it is that it have 2 and -2 as a centre element of first and third row.

Canny Edge Detection is a popular edge detection algorithm. It was developed by John F. Canny in

1. It is a multi-stage algorithm and we will go through each stages.
2. **Noise Reduction**:Since edge detection is susceptible to noise in the image, first step is to remove the noise in the image with a 5x5 Gaussian filter. We have already seen this in previous chapters.
3. **Finding Intensity Gradient of the Image**

   Smoothened image is then filtered with a Sobel kernel in both horizontal and vertical direction to get first derivative in horizontal direction G_x and vertical direction G_y . From these two images, we can find edge gradient and direction for each pixel as follows:

$$Edge\_Gradient(G) = \sqrt{(G_x)^2 + (G_y)^2} \qquad Angle(\theta) = \tan^{-1}\left(\frac{G_y}{G_x}\right)$$

Code:

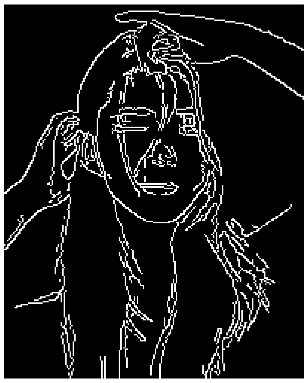
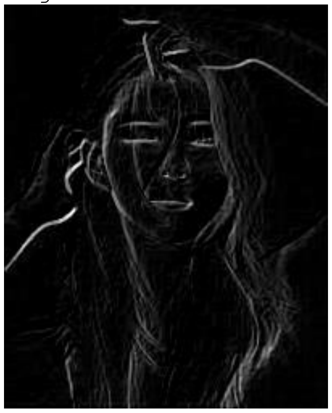
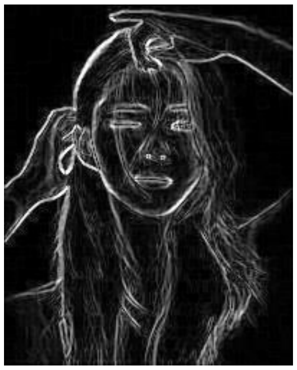
```
import cv2
from google.colab.patches import cv2_imshow
import numpy as np
image_gray = cv2.imread
```

```
edges_x = cv2.filter2D(image_gray, -1, kernel_x)
   edges_y = cv2.filter2D(image_gray, -1,
kernel_y)
```

```
("/content/drive/MyDrive/images.jpg",
cv2.IMREAD_GRAYSCALE)
if image_gray is None:
   print("Error: Could not read the image.")
else:
   edges = cv2.Canny(image_gray, 100, 200)
   cv2_imshow(image_gray)
   cv2_imshow(edges)

   kernel_x = np.array([[-1, 0, 1],
                   [-1, 0, 1],
                   [-1, 0, 1]])

   kernel_y = np.array([[-1, -1, -1],
                   [0, 0, 0],
                   [1, 1, 1]])
```

```
   edges                                    =
cv2.addWeighted(cv2.convertScaleAbs(edges_x),
0.5, cv2.convertScaleAbs(edges_y), 0.5, 0)
   cv2_imshow(edges)

   edges_x = cv2.Sobel(image_gray, cv2.CV_64F,
1, 0, ksize=3)
   edges_y = cv2.Sobel(image_gray, cv2.CV_64F,
0, 1, ksize=3)

   edges_x = cv2.convertScaleAbs(edges_x)
   edges_y = cv2.convertScaleAbs(edges_y)
   edges = cv2.addWeighted(edges_x, 0.5, edges_y,
0.5, 0)

   cv2_imshow(edges)
```

Output:

The orginal image	The canny's edge detection	Display Prewitt edges	The sobel edge detection
			

Problem No: 06

Problem Name:

Write a program to execute the following speech preprocessing

- Identify sampling frequency

- Identify bit resolution
- Make down sampling frequency then save the speech signal.

Objectives:

- To study and plot the original speech signal.
- To identify the sampling frequency and bit resolution of the signal.
- To make down sampling frequency then save the speech signal.

Theory:

Sampling theorem and sampling frequency

The sampling of analog signal is based on sampling theorem. The sampling theorem states that if f_m is the maximum frequency component in the analog signal, then the information present in the signal can be represented by its sampled version provided the number samples taken per second is greater than or equal to twice the maximum frequency component. The number of samples/second is more commonly termed as sampling frequency f_s. According to sampling theorem, f_s should be greater than or equal to 2 f_m

The speech signal has frequency components in the audio frequency range (20 Hz to 20 kHz) of the electromagnetic spectrum. This is the reason for perceiving the information present in the speech signal by human ears. The fundamental question is up to what range of audio frequency, the speech signal has frequency components. We can analyze this experimentally by considering the whole audio range. The standard sampling frequency to sample the entire audio range is 44.1 kHz. This is because, 20 kHz is the maximum frequency component and allowing some guard band, the sampling frequency has been set at 44.1 kHz.

The number of bits used for storing each sample of speech is termed as bit resolution. The number of bits/sample in turn depends on the number of quantization levels used during analog to digital conversion. More the number of quantization levels, finer will be the quantization step and hence better will be the information preserved in the digitized form. However, more will be the requirement of number of bits/sample. Hence it is a trade off between the number of bits and information representation. The effect of bit resolution can be analyzed experimentally. For this experiment the optimal sampling frequency of 16 kHz can be used as proposed earlier.

In signal processing, undersampling or bandpass sampling is a technique where one samples a bandpass-filtered signal at a sample rate below its Nyquist rate (twice the upper cutoff frequency), but is still able to reconstruct the signal.

When one undersamples a bandpass signal, the samples are indistinguishable from the samples of a low-frequency alias of the high-frequency signal. Such sampling is also known as bandpass sampling, harmonic sampling, IF sampling, and direct IF-to-digital conversion

Code:

`import numpy as np`	`plt.show()`

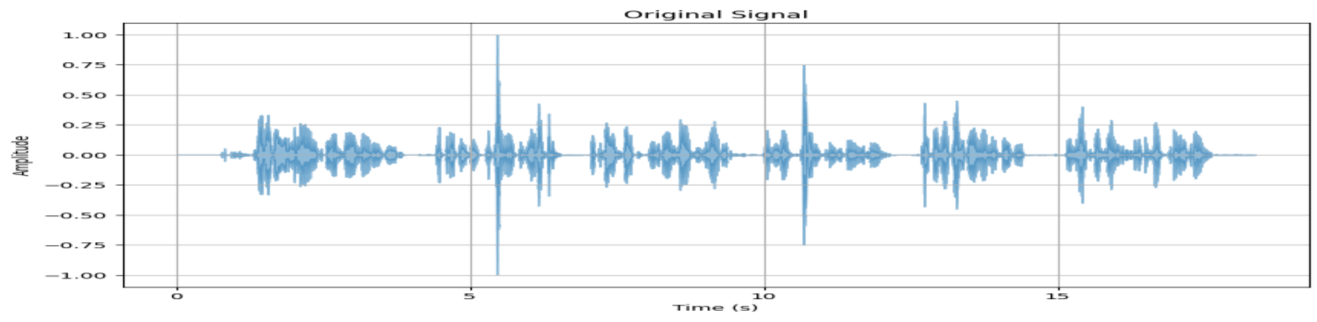
```
import matplotlib.pyplot as plt
import librosa.display
import soundfile as sf
from IPython.display import Audio
signal, sample_rate = librosa.load
('/content/drive/MyDrive/harvard.wav',
sr=None)
print("Original Sampling Frequency
 (Hz):", sample_rate)
dtype = signal.dtype
bit_depth = np.dtype(dtype).itemsize * 8
print("Bit Depth:", bit_depth)
bit_resolution = 2 ** bit_depth
print("Bit Resolution:", bit_resolution)
plt.figure(figsize=(10, 6))
librosa.display.waveshow(signal,
 sr=sample_rate, alpha=0.5)
plt.xlabel('Time (s)')
plt.ylabel('Amplitude')
plt.title('Original Signal')
plt.grid(True)
plt.show()
undersampling_factor = 2
signal_undersampled = signal
[::undersampling_factor]
sample_rate_undersampled = sample_rate
// undersampling_factor
print("Undersampled Sampling Frequency
(Hz):", sample_rate_undersampled)
plt.figure(figsize=(10, 6))
librosa.display.waveshow
(signal_undersampled,
 sr=sample_rate_undersampled, alpha=0.5)
plt.xlabel('Time (s)')
plt.ylabel('Amplitude')
plt.title('Undersampled Signal')
plt.grid(True)
```

```
plt.figure(figsize=(14, 10))
plt.subplot(2, 1, 1)
spectrogram_orig = librosa.feature
.melspectrogram(y=signal,
sr=sample_rate)
librosa.display.specshow
(librosa.power_to_db(spectrogram_orig,
 ref=np.max), sr=sample_rate,
x_axis='time',
 y_axis='mel')
plt.colorbar(format='%+2.0f dB')
plt.title('Original Signal
Spectrogram')
plt.xlabel('Time (s)')
plt.ylabel('Frequency (Hz)')
plt.subplot(2, 1, 2)
spectrogram_undersampled =
librosa.feature.
melspectrogram(y=signal_undersampled,
sr=sample_rate_undersampled)
librosa.display.specshow
(librosa.power_to_db
(spectrogram_undersampled,
ref=np.max),
sr=sample_rate_undersampled,
x_axis='time', y_axis='mel')
plt.colorbar(format='%+2.0f dB')
plt.title('Undersampled Signal
Spectrogram')
plt.xlabel('Time (s)')
plt.ylabel('Frequency (Hz)')
plt.tight_layout()
plt.show()
print("Playing undersampled audio
signal:")
Audio(data=signal_undersampled,
rate=sample_rate_undersampled)
```

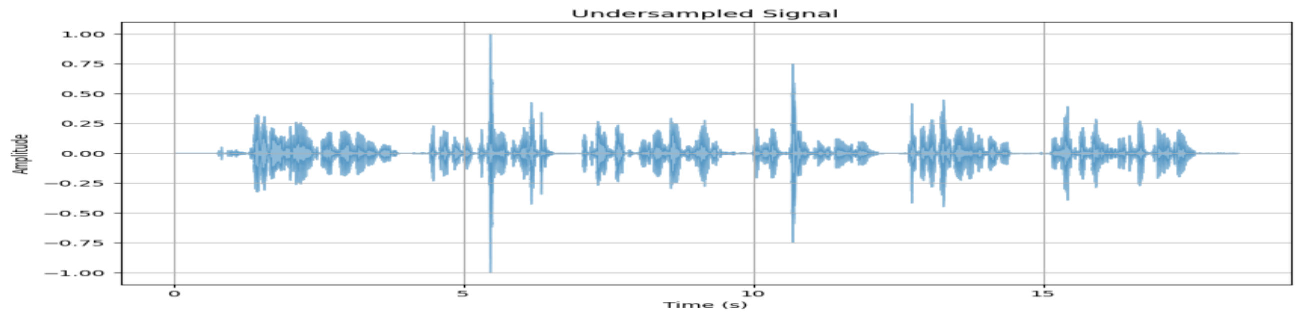
Output:

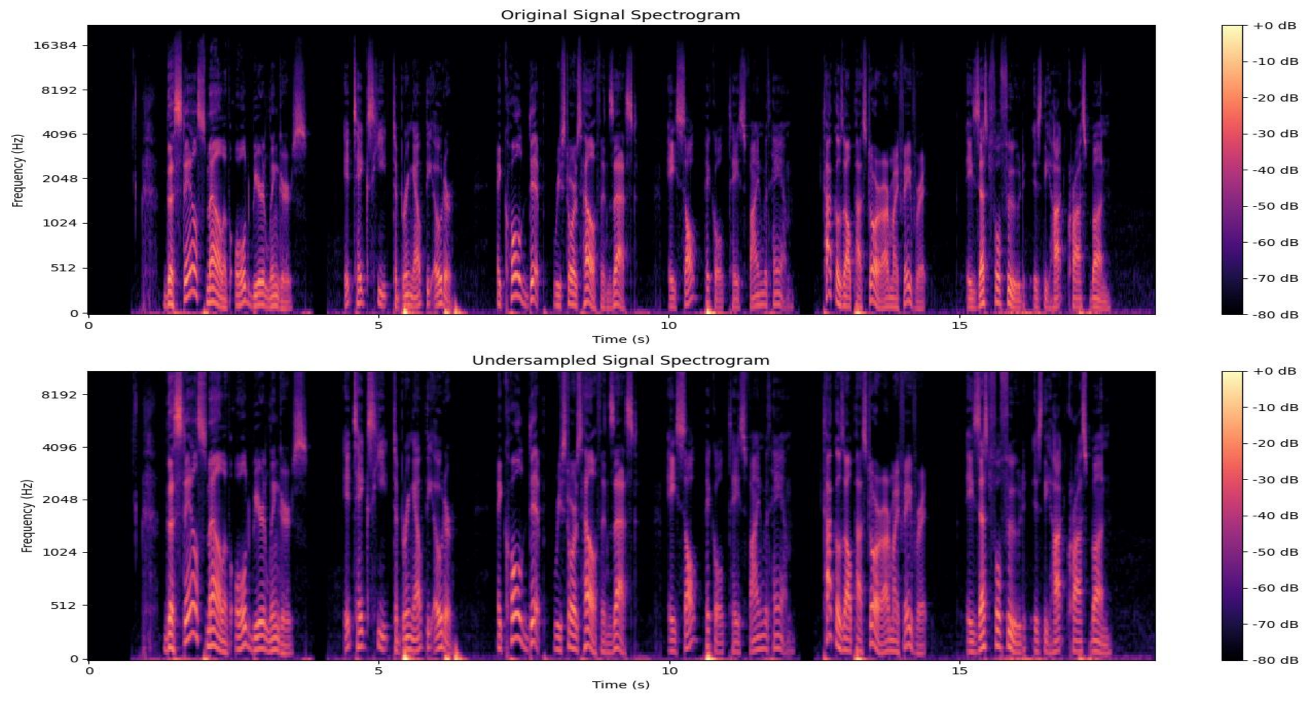
```
Original Sampling Frequency (Hz): 44100
Bit Depth: 32
```

Bit Resolution: 4294967296



```
Undersampled Sampling Frequency (Hz): 22050
```





Problem No: 07

Problem Name:

Write a program to display the following region of a speech signal.

- Voiced region.
- Unvoiced region.
- Silence region.

Objectives:

- To understand about the time and frequency domain characteristics of voiced speech.
- To understand about the time and frequency domain characteristics of unvoiced speech.
- To learn to perform the voiced/unvoiced/silence classification of speech

Theory:

Voiced speech

Voiced speech is characterized by a nearly periodic impulse sequence as its input excitation, resulting in a speech signal that visually appears nearly periodic. During the production of voiced speech, the airflow from the lungs through the trachea is periodically interrupted by the vibrating vocal folds, generating a glottal wave that excites the speech production system and produces the voiced speech.

Unvoiced Speech

Unvoiced speech is characterized by random noise-like excitation, resulting in a speech signal devoid of any periodicity, unlike voiced speech. During the production of unvoiced speech, the airflow from the lungs through the trachea is not interrupted by vibrating vocal folds. However, closure, either total or partial, occurs somewhere along the vocal tract, obstructing airflow completely or narrowly.

Silence Region

The speech production process encompasses the generation of voiced and unvoiced speech segments, interspersed with what is termed the "silence region." During this silence region, no excitation is provided to the vocal tract, resulting in the absence of speech output. Despite its seemingly insignificant amplitude or energy, silence plays a crucial role in speech signals. The presence of silence regions between voiced and unvoiced speech segments is vital for speech intelligibility.

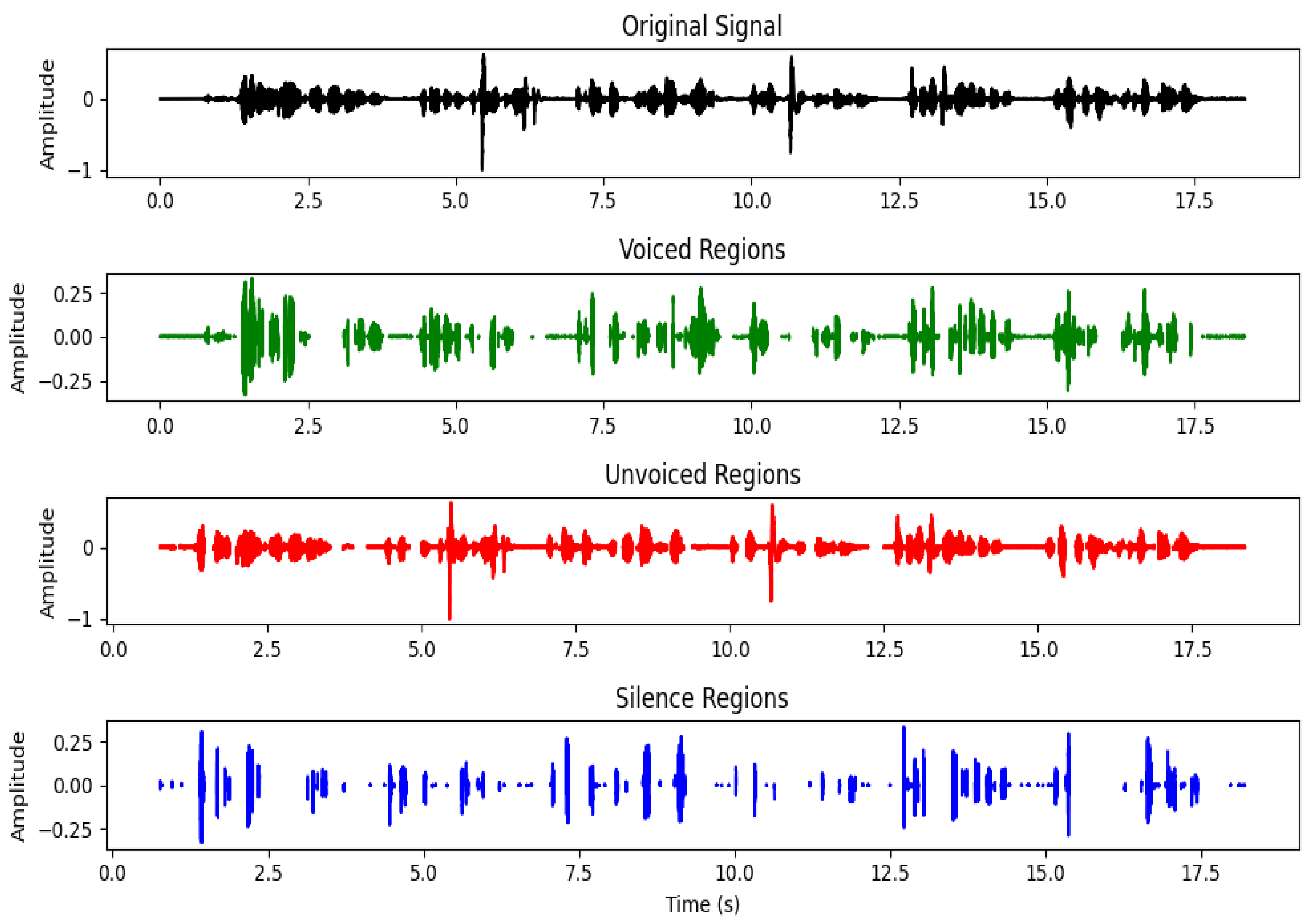
Voiced Speech Identification: Voiced speech exhibits a periodic waveform, indicating regular, repeating patterns in the signal.

Unvoiced/Silence Region Identification: Regions lacking periodicity and having low or negligible energy are classified as either unvoiced speech or silence. Low-amplitude regions are marked as silence, while segments with discernible energy but lacking periodicity are categorized as unvoiced speech.

Code:

```
import numpy as np
import matplotlib.pyplot as plt
import librosa
y, fs = librosa.load
('/content/drive/MyDrive/harvard.wav',
sr=None)
frame_size = 256
overlap = 128
num_frames = (len(y) - frame_size) //
(frame_size - overlap) + 1
voiced_frames = []
unvoiced_frames = []
silence_frames = []
for i in range(num_frames):
    start_idx = i * (frame_size -
overlap)
    end_idx = start_idx + frame_size
    frame = y[start_idx:end_idx]
    energy = np.sum(np.abs(frame)**2)
    zcr =
np.sum(np.diff(np.sign(frame)) != 0)
    voiced_threshold = 0.01 *
np.max(energy)
    unvoiced_threshold = 0.001 *
np.max(energy)
    silence_threshold = 0.0001 *
np.max(energy)
    if energy > voiced_threshold and zcr
> 10:
        voiced_frames.append(i)
    elif energy > unvoiced_threshold
and zcr < 10:
plt.xlabel('Time (s)')
plt.ylabel('Amplitude')
        unvoiced_frames.append(i)
    else:
        silence_frames.append(i)
time_axis = np.arange(len(y)) / fs
plt.figure(figsize=(10, 6))
plt.subplot(4, 1, 1)
plt.plot(time_axis, y, 'k')
plt.ylabel('Amplitude')
plt.title('Original Signal')
plt.subplot(4, 1, 2)
for i in voiced_frames:
    start_idx = i * (frame_size - overlap)
    end_idx = start_idx + frame_size
    plt.plot(time_axis[start_idx:end_idx],
y[start_idx:end_idx], 'g')
plt.ylabel('Amplitude')
plt.title('Voiced Regions')
plt.subplot(4, 1, 3)
for i in unvoiced_frames:
    start_idx = i * (frame_size - overlap)
    end_idx = start_idx + frame_size
    plt.plot(time_axis[start_idx:end_idx],
y[start_idx:end_idx], 'r')
plt.ylabel('Amplitude')
plt.title('Unvoiced Regions')
plt.subplot(4, 1, 4)
for i in silence_frames:
    start_idx = i * (frame_size - overlap)
    end_idx = start_idx + frame_size
    plt.plot(time_axis[start_idx:end_idx],
y[start_idx:end_idx], 'b')
plt.title('Silence Regions')
plt.tight_layout()
```

Output:



Problem No: 08

Problem Name:

Write a program to compute zero crossing rate (ZCR) using different window function of a speech signal.

Objective:

To compute zero crossing rate (ZCR) using different window function of a speech signal.

Theory:

Zero Crossing Rate gives information about the number of zero-crossings present in a given signal. Intuitively, if the number of zero crossings are more in a given signal, then the signal is changing rapidly and accordingly the signal may contain high frequency information. On the similar lines, if the number of zero crossing are less, hence the signal is changing slowly and accordingly the signal may contain low frequency information. Thus ZCR gives an indirect information about the frequency content of the signal.

The ZCR in case of stationary signal is defined as,

$$z = \sum_{n=-\infty}^{\infty} |sgn(s(n)) - sgn(s(n-1))|$$

$$\text{where } sgn(s(n)) = 1 \text{ if } s(n) \geq 0$$
$$= -1 \text{ if } s(n) < 0$$

This relation can be modified for non-stationary signals like speech and termed as short term ZCR. It is defined as

$$z(n) = \frac{1}{2N} \sum_{m=0}^{N-1} s(m).w(n-m)$$

The factor "2" comes in the denominator to take care of the fact that there will be two zero crossings per cycle of one signal

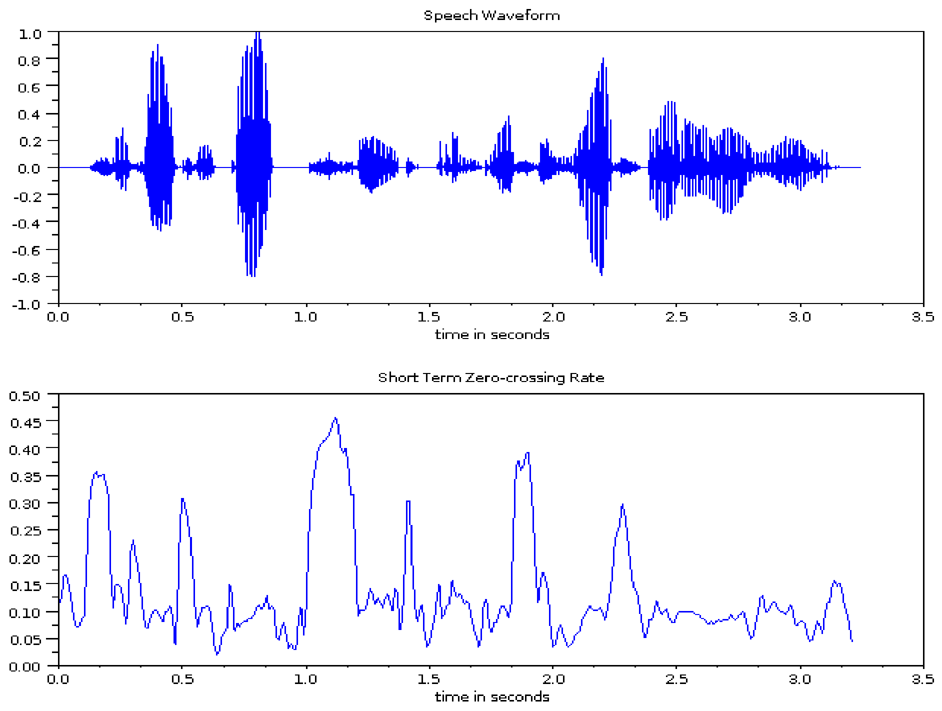


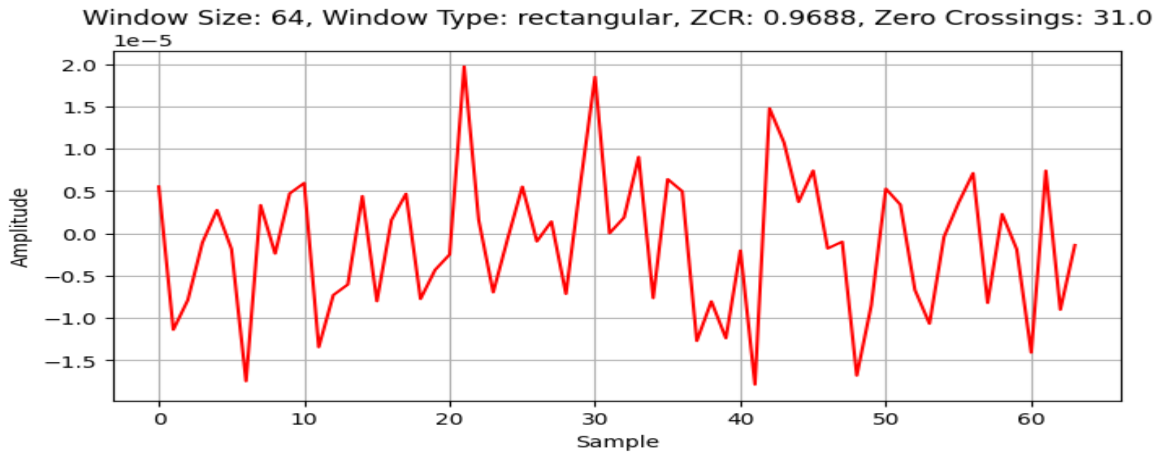
Figure_2: Short term zero crossing rate of a speech signal

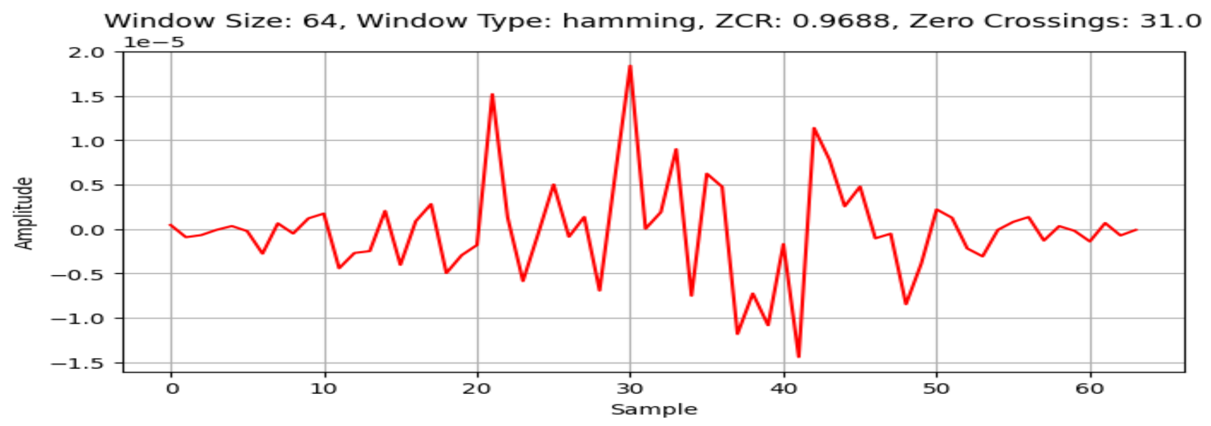
In case of speech the nature of signal changes with time over few msec. For instance, from initial voiced to unvoiced and back to voiced and so on. To have some useful information, ZCR needs to be computed using typical frame size of 10-30 msec with half the frame size as shift.

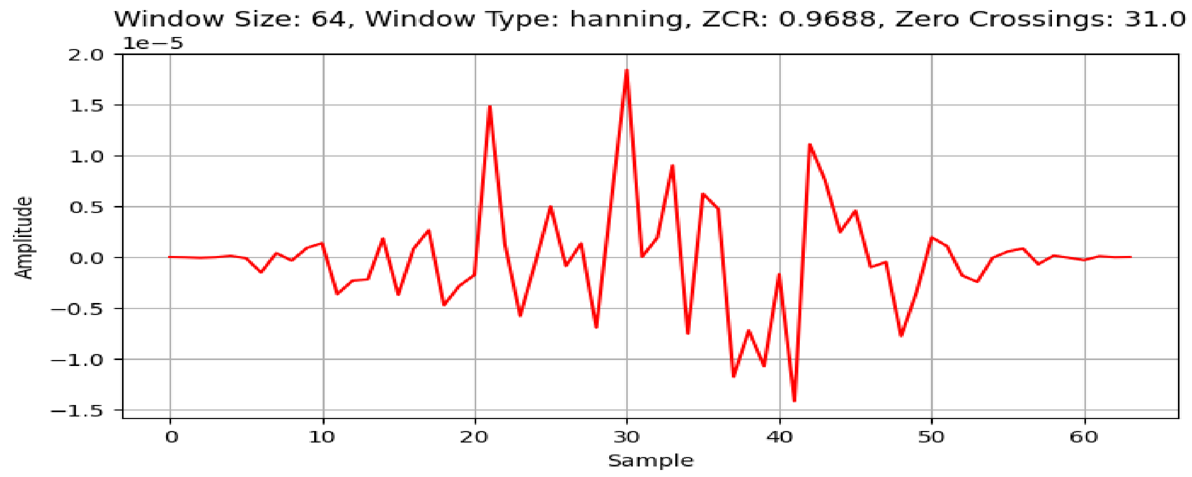
Code:

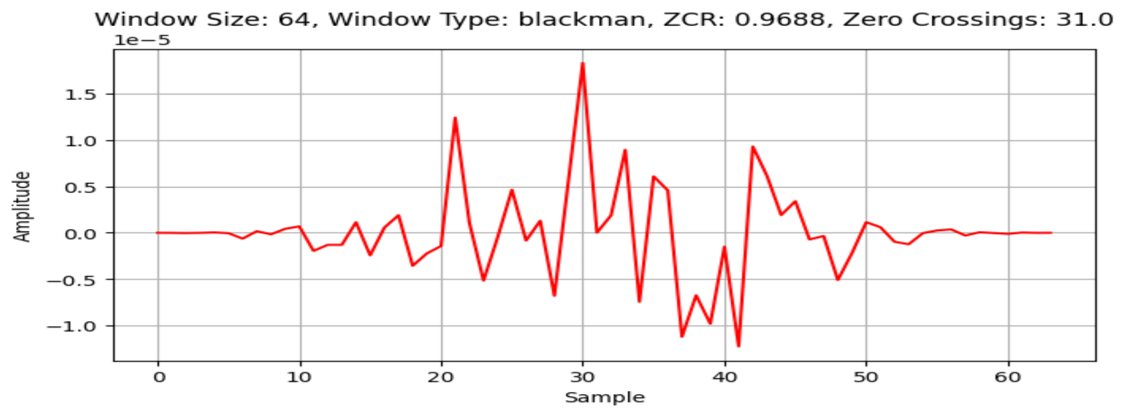
```
import numpy as np
import matplotlib.pyplot as plt
import librosa

# Load the speech signal
y, sr = librosa.load('/content/drive/MyDrive/harvard.wav')

# Define window sizes and types
window_sizes = [64, 128, 256, 512, 1024]
window_types = ['rectangular', 'hamming', 'hanning', 'blackman']

# Compute ZCR and zero crossings for each window size and type
for win_size in window_sizes:
    for win_type in window_types:
        # Generate the window function
        if win_type == 'rectangular':
            window = np.ones(win_size)
        elif win_type == 'hamming':
            window = np.hamming(win_size)
        elif win_type == 'hanning':
            window = np.hanning(win_size)
        elif win_type == 'blackman':
            window = np.blackman(win_size)

        # Apply the window function
        y_windowed = y[:len(window)] * window
        # Compute ZCR
        zcr = np.sum(np.abs(np.diff(np.sign(y_windowed)))) /
                                                    len(y_windowed)

        # Compute zero crossings
        zero_crossings = np.sum(np.abs(np.diff(np.sign(y_windowed)))) / 2

        # Plot the windowed signal
        plt.figure(figsize=(8, 4))
        plt.plot(y_windowed, color='red')
        plt.title(f'Window Size: {win_size}, Window Type: {win_type},
                        ZCR: {zcr:.4f}, Zero Crossings: {zero_crossings}')
        plt.xlabel('Sample')
        plt.ylabel('Amplitude')
        plt.grid(True)
        plt.show()
```

Output:









Other window not shown here.

Problem No: 09

Problem Name:

Write a program to compute short term auto-correlation of a speech signal.

Objective:

To compute short term auto-correlation of a speech signal and plot it.

Theory:

Cross correlation tool from signal processing can be used for finding the similarity among the two sequences and refers to the case of having two different sequences for correlation. Autocorrelation refers to the case of having only one sequence for correlation. In autocorrelation, the interest is in observing how similar the signal characteristics with respect to time. This is achived by providing different time lag for the sequence and computing with the given sequence as reference.

The autocorrelation is a very useful tool in case of speech processing. However due to the non-stationary nature of speech, a short term version of the autocorrelation is needed. The autocorrelation of a stationary sequence $r_{xx}(k)$ is given by

$$r_{xx}(k) = \sum_{m=-\infty}^{\infty} x(m).x(m+k)$$

The corresponding short term autocorrelation of a non-stationary sequence s(n) is defined as

$$r_{ss} = \sum_{m=-\infty}^{\infty} s_w(m).s_w(k+m)$$

$$r_{ss}(n,k) = \sum_{m=-\infty}^{\infty} (s(m)w(n-m).s(k+m).w(n-k+m)))$$

where $s_w(n)$=s(m).w(n-m) is the windowed version of s(n).Thus for a given windowed segment of speech, the short term autocorrelation is a sequence. The nature of short term autocorrelation sequence is primarily different for voiced and unvoiced segments of speech. Hence information from the autocorrelation sequence can be used for discriminating voiced and unvoiced segments. Figure_3 and Figure_4 show segments of voiced and unvoiced speech and the corresponding autocorrelation sequences. The nature of autocorrelation sequence is different for the two cases indicating the difference in case of voiced and unvoiced sequence of speech.

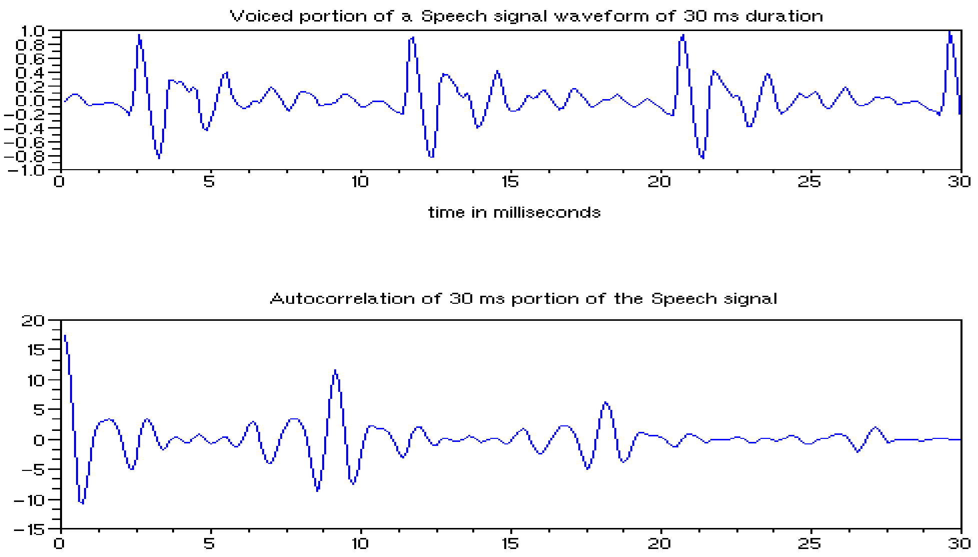


Figure_3**:** voiced segments of speech and Autocorrelation seqence

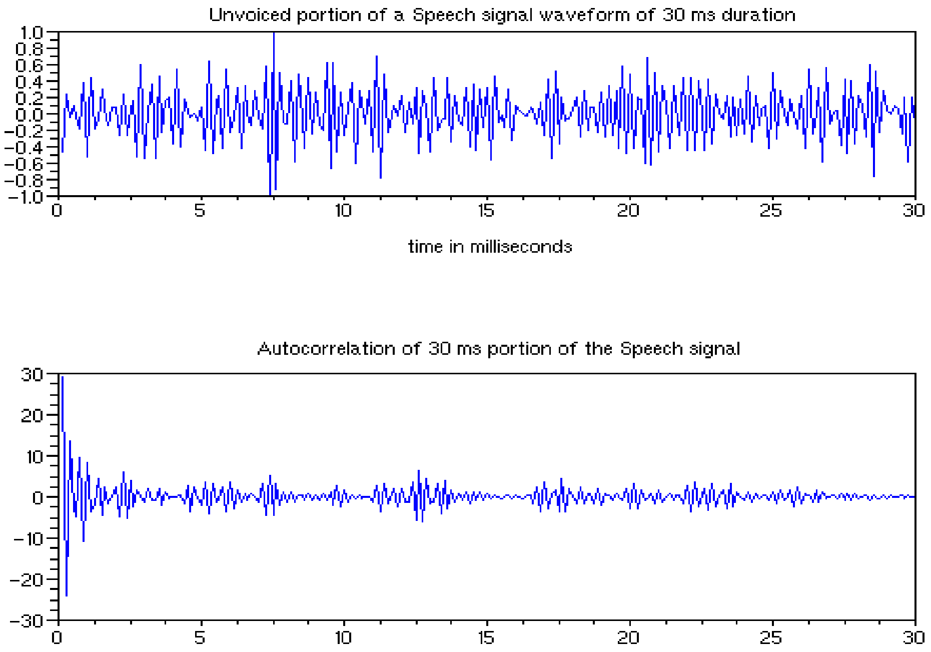


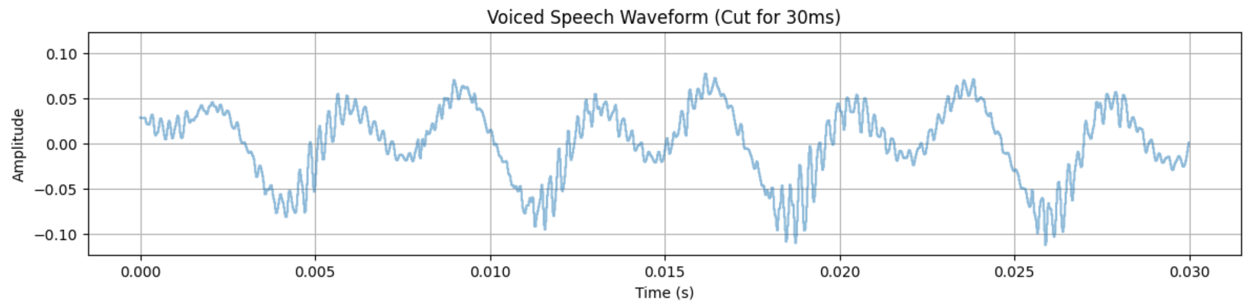
Figure_4: unvoiced segments of speech and Autocorrelation sequence

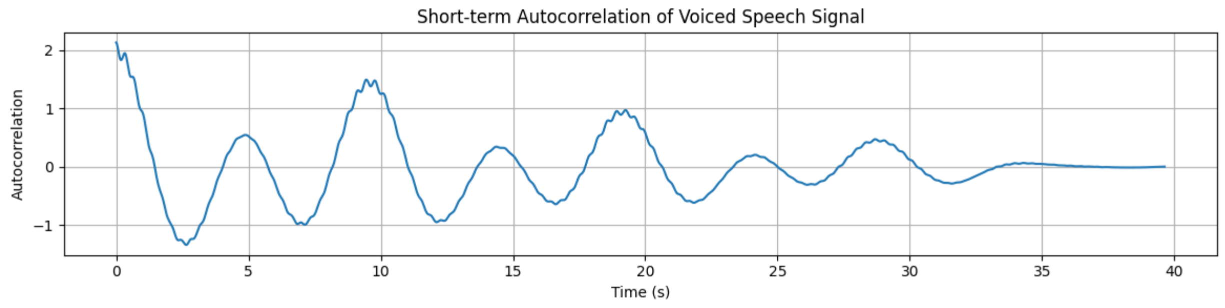
Code:

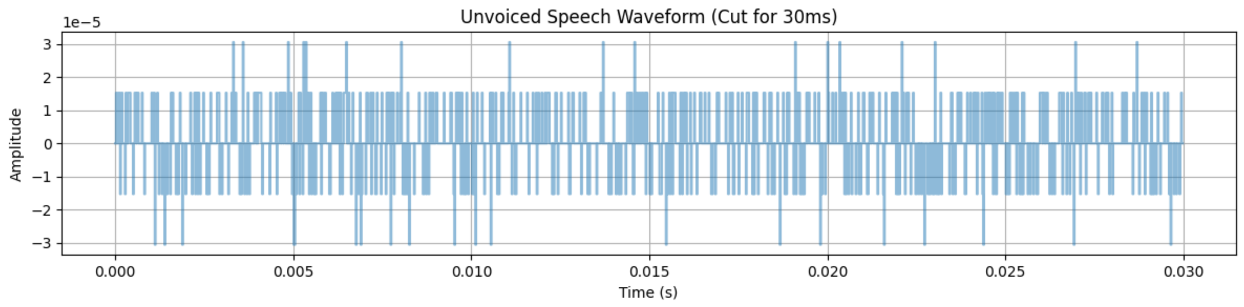
```
import numpy as np
import librosa
import librosa.display
import matplotlib.pyplot as plt
from IPython.display import Audio,
 display

def compute_short_term_
autocorrelation(cut_signal, sr,
 frame_length, hop_length,
signal_type):
    auto_corr = librosa.
autocorrelate(y=cut_signal,
max_size=frame_length)
    plt.figure(figsize=(12, 6))
    plt.subplot(2, 1, 1)
    librosa.display.waveshow
   (cut_signal, sr=sr, alpha=0.5)
    plt.xlabel('Time (s)')
    plt.ylabel('Amplitude')
    plt.title(signal_type + '
Speech Waveform (Cut for 30ms)')
    plt.grid(True)

    # Plot autocorrelation
    plt.subplot(2, 1, 2)
    plt.plot(librosa.frames_to_time
(range(len(auto_corr)),
hop_length=hop_length), auto_corr)
    plt.xlabel('Time (s)')
    plt.ylabel('Autocorrelation')
    plt.title('Short-term
Autocorrelation of ' +
 signal_type + ' Speech Signal')
    plt.grid(True)

    plt.tight_layout()
    plt.show()
```

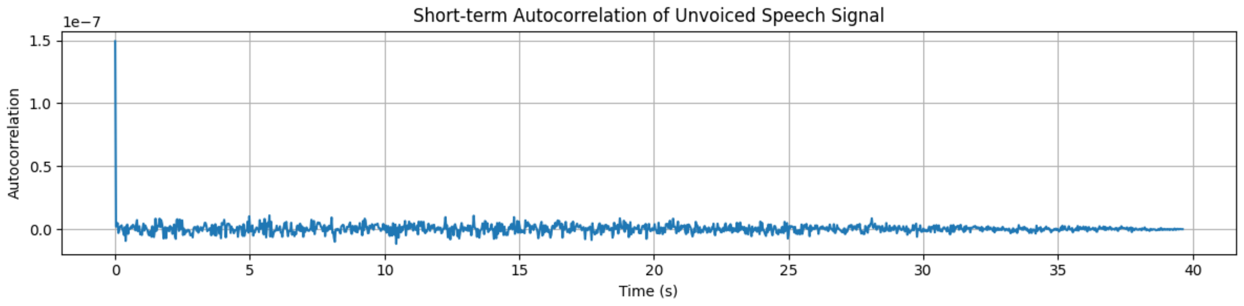
```
# Usage example
audio_file =
r"/content/drive/MyDrive/harvard.wav"

# Load the audio file

signal, sr = librosa.
load(audio_file, sr=None)

# Parameters
frame_duration = 0.03  # 30 ms frame
duration
hop_duration = frame_duration / 2  # Half
of frame duration for 50% overlap

# Convert durations to samples
frame_length = int(sr * frame_duration)
hop_length = int(sr * hop_duration)

# Cut a portion of the speech signal (for
example, for 30 ms)
cut_signal =
signal[int(5*sr):int(5.03*sr)]
compute_short_term_autocorrelation(cut_si
gnal, sr, frame_length, hop_length,
'Voiced')

# Cut a portion of the speech signal (for
example, for 30 ms)
cut_signal =
signal[int(4*sr):int(4.03*sr)]
compute_short_term_autocorrelation(cut_si
gnal, sr, frame_length, hop_length,
'Unvoiced')
```

Output:


Voiced Speech Waveform (Cut for 30ms)
Amplitude
0.10
0.05
0.00
−0.05
−0.10
0.000
0.005
0.010
0.015
0.020
0.025
0.030
Time (s)


Short-term Autocorrelation of Voiced Speech Signal
Autocorrelation
2
1
0
−1
0
5
10
15
20
25
30
35
40
Time (s)


Unvoiced Speech Waveform (Cut for 30ms)
1e−5
Amplitude
3
2
1
0
−1
−2
−3
0.000
0.005
0.010
0.015
0.020
0.025
0.030
Time (s)


Short-term Autocorrelation of Unvoiced Speech Signal
1e−7
Autocorrelation
1.5
1.0
0.5
0.0
0
5
10
15
20
25
30
35
40
Time (s)

Problem No: 10

Problem Name:

Write a program to estimate pitch of a speech signal.

Objective:

To estimate pitch of a speech signal and plot it.

Theory:

Pitch estimation in speech signal processing refers to the process of determining the fundamental frequency (F0) of the voice, which corresponds to the perceived pitch of the speech. The fundamental frequency represents the rate at which the vocal folds vibrate during voiced speech segments.Pitch estimation is crucial in various speech processing tasks, including speech recognition, speaker diarization, prosody analysis, and synthesis. There are several methods for pitch estimation in speech signals, including:

Autocorrelation: Auto-correlation is a mathematical operation used to measure the similarity between a signal and a time-delayed version of itself. In pitch estimation, auto-correlation is applied to speech signals to identify repeating patterns or periodicities, which correspond to the fundamental frequency (F0) or pitch.
Short-Time Fourier Transform (STFT): Analyzes the frequency content of short segments of the speech signal over time. Peaks in the spectrum represent harmonic frequencies related to the fundamental frequency.
Cepstral Analysis: Extracts the pitch-related information from the cepstrum, which is the inverse Fourier transform of the logarithm of the power spectrum. Pitch peaks can be identified in the cepstrum.

Here used the autocorrelation method for speech processing.

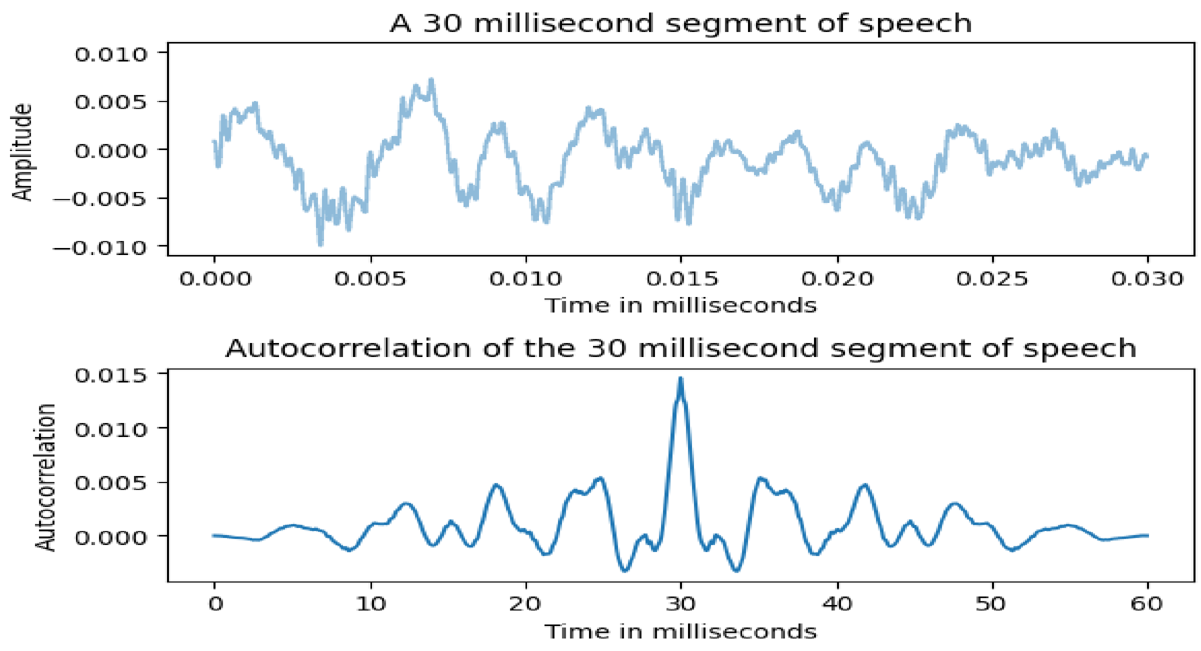
Code:

```
import numpy as np
import matplotlib.pyplot as plt
import librosa
y, Fs = librosa.load(r"/content/drive/MyDrive/harvard.wav", sr=None)
# Cut a portion of the speech signal (for example, for 30 ms)
start_time = 4.515
end_time = 4.545
y = y[int(start_time * Fs):int(end_time * Fs)]
# Compute autocorrelation
autocorrelation = np.correlate(y, y, mode='full')
# Time axis for autocorrelation plot (in milliseconds)
kk = np.arange(0, len(autocorrelation)) / Fs * 1000
```

```
# Plot original signal
plt.subplot(2, 1, 1)
librosa.display.waveshow(y, sr=Fs, alpha=0.5)
plt.xlabel('Time in milliseconds')
plt.ylabel('Amplitude')
plt.title('A 30 millisecond segment of speech')
plt.subplot(2, 1, 2)
plt.plot(kk, autocorrelation)
plt.xlabel('Time in milliseconds')
plt.ylabel('Autocorrelation')
plt.title('Autocorrelation of the 30 millisecond segment of speech')
auto = autocorrelation[20:160]
max_idx = np.argmax(auto)
sample_no = max_idx + 21 # Adjust for the indexing
pitch_period_To = (20 + sample_no) * (1 / Fs)
pitch_freq_Fo = 1 / pitch_period_To
print("Pitch Period (To):", pitch_period_To)
print("Pitch Frequency (Fo):", pitch_freq_Fo)
plt.tight_layout()
plt.show()
```

Output:

```
Pitch Period (To): 0.004081632653061225
Pitch Frequency (Fo): 244.99999999999997
```


A 30 millisecond segment of speech
Amplitude
0.010
0.005
0.000
−0.005
−0.010
0.000
0.005
0.010
0.015
0.020
0.025
0.030
Time in milliseconds
Autocorrelation of the 30 millisecond segment of speech
Autocorrelation
0.015
0.010
0.005
0.000
0
10
20
30
40
50
60
Time in milliseconds