INDEX

SL	Experiment Name	Marks
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.	01
2.	Write a program to execute Semantic Segmentation.	07
3.	Write a program to execute the following problem. • Given an image and a mask, determine the region of the image using the mask, compute the area of the region, then label the region by overlapping the mask over the image.	09
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).	11
5.	Write a program to execute the following edge detections • Canny edge detection • Prewitt edge detection • Sobel edge detection	16
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.	18
7.	Write a program to display the following region of a speech signal. • Voiced region. • Unvoiced region. • Silence region.	21
8.	Write a program to compute zero crossing rate (ZCR) using different window function of a speech signal.	24
9.	Write a program to compute short term auto-correlation of a speech signal.	26
10.	Write a program to estimate pitch of a speech signal.	28

EXPERIMENT NUMBER: 01

EXPERIMENT 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.
- Normalize images and save into a folder.
- Filter images and save into a folder

OBJECTIVES: The primary aim of this laboratory experiment is to acquire practical knowledge in fundamental digital image processing procedures including image retrieval, resizing, and color conversion.

THEORY:

Reading Images from a Folder: Within the realm of digital image processing, images are commonly depicted as arrays containing pixel values.

• The extraction of images from a designated folder involves the utilization of libraries specialized in image processing, such as OpenCV (Open Source Computer Vision Library) in the Python programming language.

• OpenCV offers a multitude of functions designed to read images in various formats like JPEG, PNG, BMP, and more, converting them into arrays or matrices.

• The function imread() within OpenCV is widely employed for the purpose of reading images from files.

Resizing Images: The alteration of image dimensions, while upholding the aspect ratio or applying customized scaling factors, constitutes the process of resizing images.

• Resizing becomes imperative to conform images to specific criteria, like accommodating them within distinct frame sizes or reducing computational intricacy.

• The provision of the resize() function by OpenCV facilitates the resizing of images, allowing the specification of the desired width and height for the resultant image.

Color Transformations: The manipulation of the color space of an image, known as color transformation, serves various objectives including contrast enhancement, brightness adjustment, or extraction of particular color elements.

• Prominent color spaces encompass RGB (Red, Green, Blue), HSV (Hue, Saturation, Value), and Grayscale.

• OpenCV is equipped with functions enabling the conversion of images from one color space to another, exemplified by cvtColor().

Normalizing Images: The normalization process entails the adjustment of pixel values to a predefined range, commonly [0, 1], fostering the standardization of pixel intensities across images for enhanced comparability and processing efficacy. The normalized images are then stored in an assigned folder.

1. High Pass Filtering: The process of high pass filtering in image manipulation serves to accentuate edges and emphasize high-frequency elements within an image while suppressing lower frequency details. This technique involves highlighting variations or rapid shifts in pixel intensities, typically associated with edges and boundaries. Noteworthy high pass filters include the Laplacian filter and the Sobel filter, commonly utilized in tasks like edge detection, image sharpening, and feature extraction.

2. Low Pass Filtering: Low pass filters are instrumental in diminishing noise and blurring images through the smoothing or averaging of pixel intensities across neighboring regions. Low pass filtering is centered around reducing or eliminating high-frequency components within an image while retaining low-frequency details. Examples of low pass filters encompass the Gaussian filter and the Box filter, frequently applied in noise reduction, image smoothing, and preliminary tasks aimed at preparing images for subsequent analysis.

3. Median Filtering: Employed as a non-linear filtering method, the median filter plays a crucial role in noise elimination from images while safeguarding edges and intricate details. Through median filtering, each pixel in the image is substituted with the median value of neighboring pixels within a specified window or kernel. This technique is especially effective in eliminating impulse noise, such as salt-and-pepper noise, without causing blurring of edges or introducing unwanted artifacts. Its widespread application is evident in image denoising scenarios where maintaining image sharpness holds paramount importance.

4. Laplacian Filtering: The Laplacian filter, classified as a high pass filter, finds utility in edge detection and image sharpening endeavors. By computing the second derivative of the image intensity function, this filter reveals areas of abrupt intensity changes, often associated with edges and boundaries. In emphasizing high-frequency components while suppressing low-frequency details, the Laplacian filter demonstrates sensitivity to noise and potential amplification of noise artifacts. Therefore, it is frequently combined with smoothing filters to enhance the precision of edge detection outcomes.

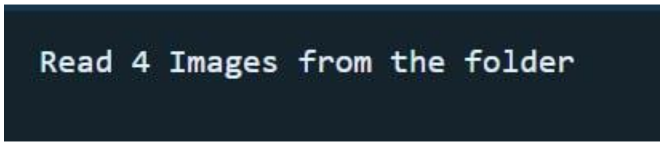
5. Gaussian Filtering: Acting as a low pass filter, the Gaussian filter is pivotal in noise reduction and image smoothing applications. Through the application of a convolution operation utilizing a Gaussian kernel, this filter blurs the image, thereby diminishing noise visibility. Offering isotropic smoothing, wherein uniform smoothing occurs in all directions, the Gaussian filter excels in preserving image details more effectively than box filtering. Its extensive use in image processing domains, particularly in contexts where preserving image sharpness and intricate details is imperative, is evidenced in fields like medical imaging and satellite image processing.

CODE:

Read images from a folder

```
import cv2 as cv
import os
import glob
import matplotlib.pyplot as plt
folder_path = r"D:\Books\6th Semester\image lab\Image Lab\image 4"
image_files = glob.glob(os.path.join(folder_path, '*.jpg'))
images = []
for image_path in image_files:
img = cv.imread(image_path)
if img is not None:
images.append(img)
else:
print(f"Error reading image: {image_path}")
print(f"Read {len(images)} images from the folder.")
```

OUTPUT:



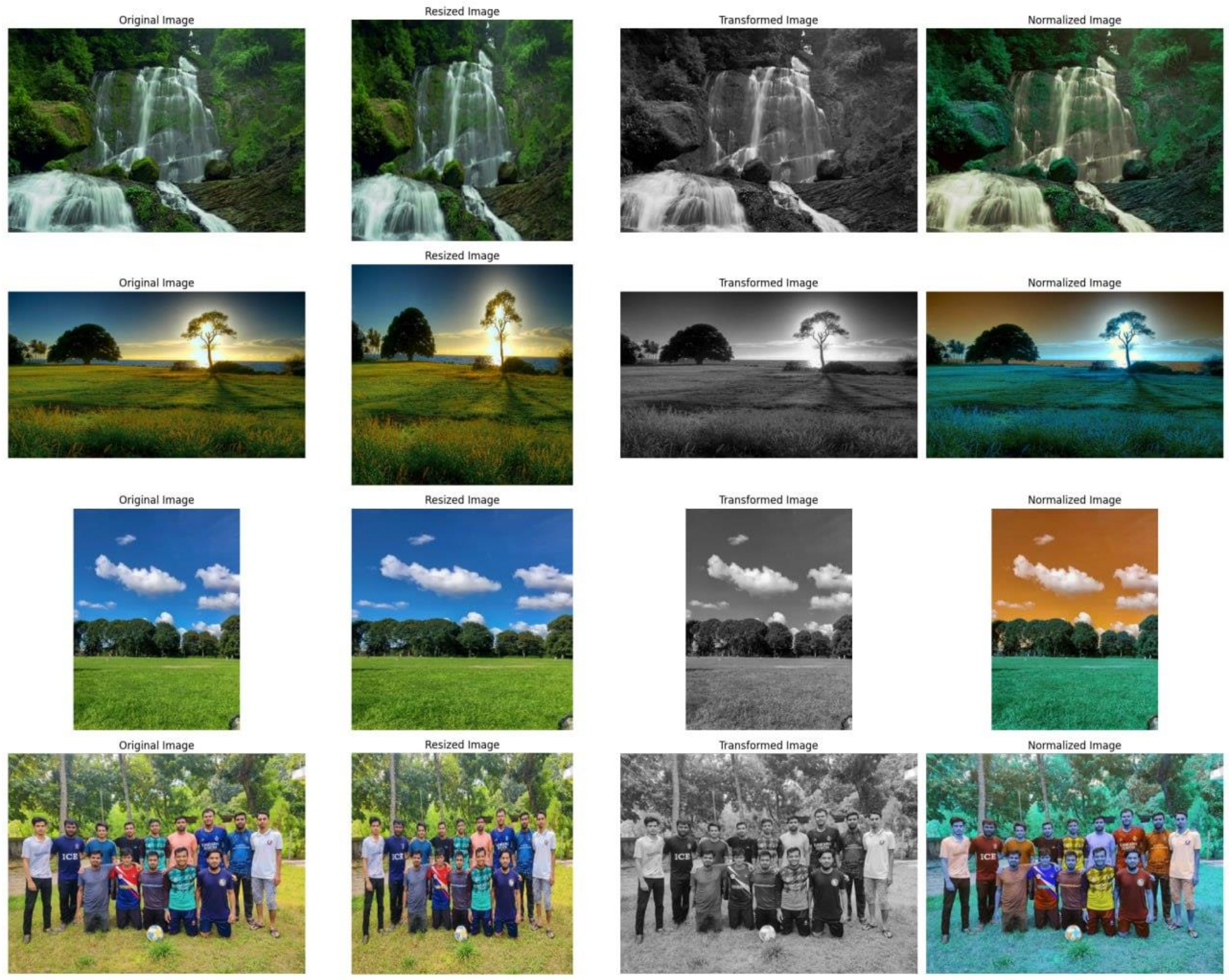
```
Read 4 Images from the folder
```

CODE:

```
import os
import cv2
import glob
import matplotlib.pyplot as plt
def read_images_from_folder(folder_path):
images = []
image_files = []
for filename in os.listdir(folder_path):
img = cv2.imread(os.path.join(folder_path, filename))
if img is not None:
images.append(img)
image_files.append(filename)
return images, image_files
def resize_images(images, new_size=(150, 100)):4
resized_images = []
for img in images:
resized_img = cv2.resize(img, new_size)
resized_images.append(resized_img)
return resized_images
def apply_color_transform(images):
transformed_images = []
for img in images:
transformed_img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
transformed_images.append(transformed_img)
return transformed_images
def normalize_images(images):
normalized_images = []
for img in images:
# Example normalization: Convert to float32 and scale to [0, 1]
normalized_img = img.astype('float32') / 255.0
normalized_images.append((normalized_img * 255).astype('uint8'))
return normalized_images
folder_path = r"D:\Books\6th Semester\image lab\Image Lab\image 4"
output_folder = r"D:\Books\6th Semester\image lab\Image Lab\Processed Image"
# Read images from the folder
images, image_files = read_images_from_folder(folder_path)
# Resize images
resized_images = resize_images(images)
# Apply color transform
transformed_images = apply_color_transform(images)
# Normalize images
normalized_images = normalize_images(images)
# Plot original, resized, transformed, and normalized images
fig, axes = plt.subplots(nrows=len(images), ncols=4, figsize=(20, 4*len(images)))
for i in range(len(images)):
axes[i, 0].imshow(cv2.cvtColor(images[i], cv2.COLOR_BGR2RGB))
axes[i, 0].set_title('Original Image')
axes[i, 0].axis('off')
axes[i, 1].imshow(cv2.cvtColor(resized_images[i], cv2.COLOR_BGR2RGB))
axes[i, 1].set_title('Resized Image')
axes[i, 1].axis('off')
axes[i, 2].imshow(transformed_images[i], cmap='gray')
axes[i, 2].set_title('Transformed Image')
axes[i, 2].axis('off')
axes[i, 3].imshow(normalized_images[i], cmap='gray')
```

```
axes[i, 3].set_title('Normalized Image')
axes[i, 3].axis('off')5
plt.tight_layout()
plt.show()
```

OUTPUT:



Code:

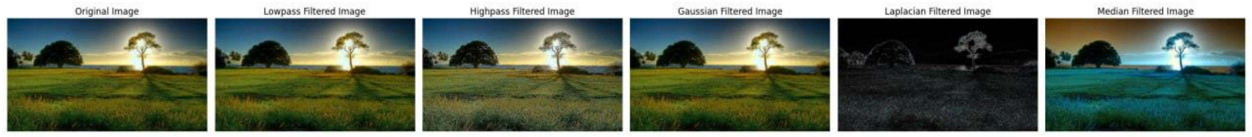
Applying different types of filter's on images

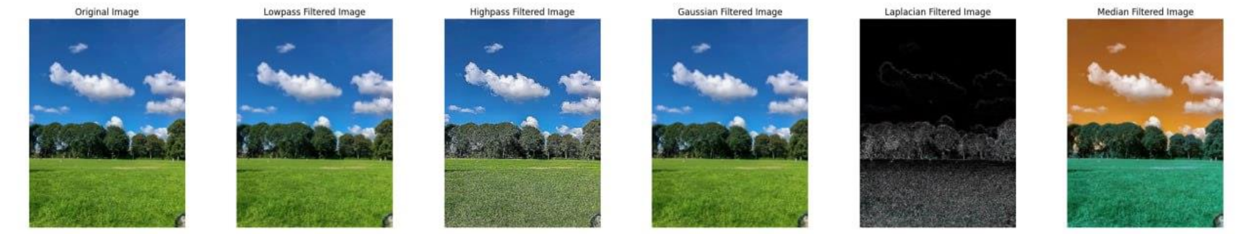
```
import os
import cv2
import numpy as np
import matplotlib.pyplot as plt
def read_images_from_folder(folder_path):
images = []
image_files = []
for filename in os.listdir(folder_path):
img = cv2.imread(os.path.join(folder_path, filename))
if img is not None:
images.append(img)
image_files.append(filename)
```

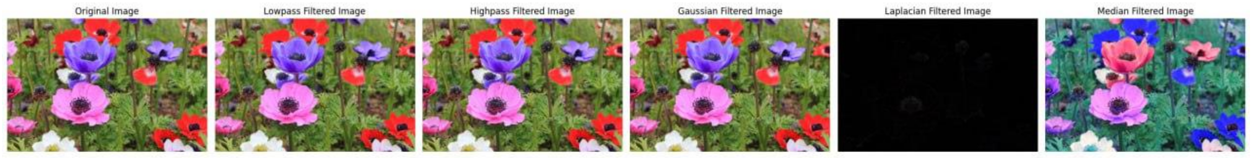
```
return images, image_files
def save_image(image, output_folder, filename):
output_path = os.path.join(output_folder, filename)
cv2.imwrite(output_path, image)
def lowpass_filter(images, output_folder):
lowpass_images = []6
for i, img in enumerate(images):
lowPass = cv2.GaussianBlur(img, (3, 3), 0)
save_image(lowPass, output_folder, f"lowpass_{i}.jpg")
lowpass_images.append(lowPass)
return lowpass_images
def highpass_filter(images, output_folder):
highpass_images = []
for i, img in enumerate(images):
kernal = np.array([[-1, -1, -1], [-1, 9, -1], [-1, -1, -1]])
highPass = cv2.filter2D(img, -1, kernal)
save_image(highPass, output_folder, f"highpass_{i}.jpg")
highpass_images.append(highPass)
return highpass_images
def gaussian_filter(images, output_folder):
gaussian_images = []
for i, img in enumerate(images):
gaussian = cv2.GaussianBlur(img, (3, 3), 0)
save_image(gaussian, output_folder, f"gaussian_{i}.jpg")
gaussian_images.append(gaussian)
return gaussian_images
def laplacian_filter(images, output_folder):
laplacian_images = []
for i, img in enumerate(images):
laplacian = cv2.Laplacian(img, cv2.CV_64F)
laplacian = np.uint8(np.absolute(laplacian))
save_image(laplacian, output_folder, f"laplacian_{i}.jpg")
laplacian_images.append(laplacian)
return laplacian_images
def median_filter(images, output_folder):
median_images = []
for i, img in enumerate(images):
median = cv2.medianBlur(img, 3)
save_image(median, output_folder, f"median_{i}.jpg")
median_images.append(median)
return median_images
def plot_filtered_images(images, lowpass_images, highpass_images, gaussian_images,
laplacian_images,
median_images):
fig, axes = plt.subplots(nrows=len(images), ncols=6, figsize=(24, 4*len(images)))
for i in range(len(images)):
axes[i, 0].imshow(cv2.cvtColor(images[i], cv2.COLOR_BGR2RGB))
axes[i, 0].set_title('Original Image')
axes[i, 0].axis('off')
```

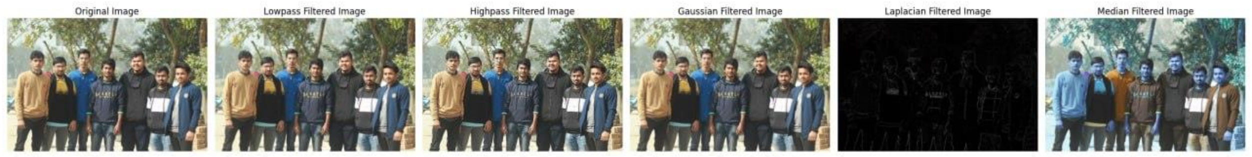
```
axes[i, 1].imshow(cv2.cvtColor(lowpass_images[i], cv2.COLOR_BGR2RGB))
axes[i, 1].set_title('Lowpass Filtered Image')
axes[i, 1].axis('off')
axes[i, 2].imshow(cv2.cvtColor(highpass_images[i], cv2.COLOR_BGR2RGB))
axes[i, 2].set_title('Highpass Filtered Image')
axes[i, 2].axis('off')
axes[i, 3].imshow(cv2.cvtColor(gaussian_images[i], cv2.COLOR_BGR2RGB))
axes[i, 3].set_title('Gaussian Filtered Image')
axes[i, 3].axis('off')
axes[i, 4].imshow(laplacian_images[i], cmap='gray')
axes[i, 4].set_title('Laplacian Filtered Image')
axes[i, 4].axis('off')
axes[i, 5].imshow(median_images[i], cmap='gray')
axes[i, 5].set_title('Median Filtered Image')7
axes[i, 5].axis('off')
plt.tight_layout()
plt.show()
# Path to the folder containing images
folder_path = r"D:\Books\6th Semester\image lab\Image Lab\image 4"
# Output folders for filtered images
output_folder_lowpass = "output/lowpass_images"
output_folder_highpass = "output/highpass_images"
output_folder_gaussian = "output/gaussian_images"
output_folder_laplacian = "output/laplacian_images"
output_folder_median = "output/median_images"
# Read images from the folder
images, image_files = read_images_from_folder(folder_path)
# Apply filters to images and save filtered images
lowpass_images = lowpass_filter(images, output_folder_lowpass)
highpass_images = highpass_filter(images, output_folder_highpass)
gaussian_images = gaussian_filter(images, output_folder_gaussian)
laplacian_images = laplacian_filter(images, output_folder_laplacian)
median_images = median_filter(images, output_folder_median)
# Plot filtered images
plot_filtered_images(images, lowpass_images, highpass_images, gaussian_images,
laplacian_images,
median_images)
```

Output:









EXPERIMENT NUMBER: 02
EXPERIMENT NAME: Write a program to execute Semantic Segmentation.
OBJECTIVES:

To Perform semantic segmentation on image to accurately classify each pixel into predefined categories or classes.

Theory:

Image segmentation is an essential task within the field of computer vision, involving the division of an image into distinct regions or segments based on specific characteristics or criteria. The primary objective of image segmentation is to streamline and transform the visual representation of an image into a more meaningful format that is easier to interpret.

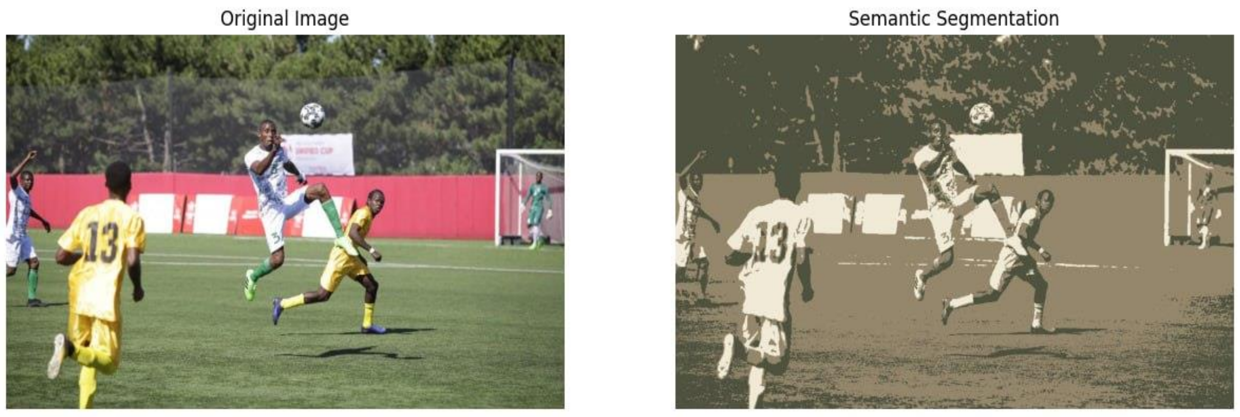
Semantic segmentation stands out as a sophisticated technique for image segmentation, as it assigns semantic labels to individual pixels in an image, classifying them into significant classes or categories. In contrast to conventional image segmentation approaches that rely on low-level features such as color or texture to segment an image, semantic segmentation strives to comprehend the image's content by recognizing and annotating objects or regions with shared semantic meanings. This method empowers computers to interpret and comprehend the visual environment at a granular pixel level, thereby enhancing various computer vision applications.

Code:

```
import cv2 as cv
import os
import glob
import numpy as np
import matplotlib.pyplot as plt
```

```
def load_images(folder_path):
"""Load images from a folder."""
image_files = glob.glob(os.path.join(folder_path, '*.jpg'))
images = [cv.imread(image_path) for image_path in image_files]
return images
def semantic_segmentation(images, num_clusters=3):
"""Perform semantic segmentation using K-means clustering."""
segmented_images = []
criteria = (cv.TERM_CRITERIA_EPS + cv.TERM_CRITERIA_MAX_ITER, 100, 0.2)
for img in images:
pixels = img.reshape((-1, 3))
pixels = np.float32(pixels)
_, labels, centers = cv.kmeans(pixels, num_clusters, None, criteria, 10,
cv.KMEANS_RANDOM_CENTERS)
centers = np.uint8(centers)
segmented_img = centers[labels.flatten()].reshape(img.shape)
segmented_images.append(segmented_img)
return segmented_images
# Load images
folder_path = "/content/drive/MyDrive/image 4"
images = load_images(folder_path)
# Check if images were loaded correctly
if len(images) == 0:
print("No images found in the specified folder.")
exit()
# Perform segmentation
semantic_segmented_images = semantic_segmentation(images)9
# Display results
for i in range(min(3, len(images))): # Ensure we don't go out of range if less than 3 images were loaded
plt.figure(figsize=(15, 5))
plt.subplot(1, 2, 1)
plt.imshow(cv.cvtColor(images[i], cv.COLOR_BGR2RGB))
plt.title('Original Image')
plt.axis('off')
plt.subplot(1, 2, 2)
plt.imshow(cv.cvtColor(semantic_segmented_images[i], cv.COLOR_BGR2RGB))
plt.title('Semantic Segmentation')
plt.axis('off')
plt.show()
```

Output:



EXPERIMENT NUMBER: 03
EXPERIMENT NAME: Write a program to execute the following problem.
• Given an image and a mask, determine the region of the image using the mask, compute the area of the region, then label the region by overlapping the mask over the image.
OBJECTIVES: To explore medical image analysis and visualization for tumor detection using OpenCV and Matplotlib.

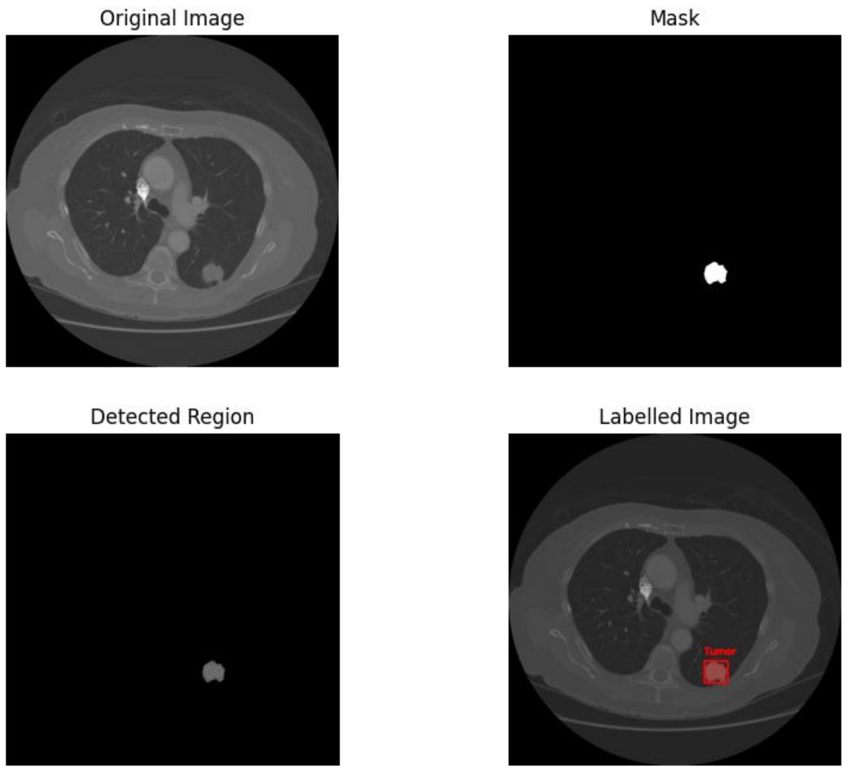
Theory:

This study aims to examine medical images, comprising an initial image and a corresponding mask that identifies a specific area of interest, such as a tumor. The process commences by loading the images and confirming their successful retrieval. Subsequently, the mask is utilized on the original image through bitwise operations, thereby isolating the region of interest. The area of this particular region is determined by tallying non-zero pixels in the mask. Following this, the program transforms the mask into a color image and superimposes it onto the original image, accentuating the recognized area. Moreover, it outlines a bounding box around the region and annotates it as "Tumor" for clarity. Lastly, the manipulated images are displayed using Matplotlib, and the area of the identified region is presented. In essence, the program provides a thorough method for scrutinizing medical images, facilitating the identification and presentation of significant anatomical characteristics or irregularities.

Code:

```
import cv2
import numpy as np
from matplotlib import pyplot as plt
# Load the image and mask
image_path = r"D:\Books\6th Semester\image lab\Image Lab\image 4\Image.png"
mask_path = r"D:\Books\6th Semester\image lab\Image Lab\image 4\Mask.png"
image = cv2.imread(image_path)
mask = cv2.imread(mask_path, cv2.IMREAD_GRAYSCALE)
# Check if images are loaded properly
if image is None:
raise FileNotFoundError(f"Unable to load image from path: {image_path}")
if mask is None:
raise FileNotFoundError(f"Unable to load mask from path: {mask_path}")
# Determine the region using the mask
region = cv2.bitwise_and(image, image, mask=mask)
# Compute the area of the determined region
area = np.sum(mask != 0)
# Create a copy of the mask with red color
red_mask = cv2.cvtColor(mask, cv2.COLOR_GRAY2BGR)
red_mask[mask != 0] = [0, 0, 255] # Set non-zero mask pixels to red color (BGR format)
# Overlay the red mask onto the original image
labelled_image = cv2.addWeighted(image, 0.7, red_mask, 0.3, 0)
# Add a rectangle outside the determined region
x, y, w, h = cv2.boundingRect(mask) # Get bounding box coordinates
cv2.rectangle(labelled_image, (x, y), (x + w, y + h), (0, 0, 255), 2) # Draw rectangle
# Add label "tumor" to the rectangle
cv2.putText(labelled_image, 'Tumor', (x, y - 10), cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 0, 255),
2)11
# Display the results
plt.figure(figsize=(10, 8))
# Original Image
plt.subplot(2, 2, 1)
plt.imshow(cv2.cvtColor(image, cv2.COLOR_BGR2RGB))
plt.title("Original Image")
plt.axis("off")
# Mask
plt.subplot(2, 2, 2)
plt.imshow(mask, cmap='gray')
plt.title("Mask")
plt.axis("off")
# Region
plt.subplot(2, 2, 3)
plt.imshow(cv2.cvtColor(region, cv2.COLOR_BGR2RGB))
plt.title("Detected Region")
plt.axis("off")
# Labelled Image
plt.subplot(2, 2, 4)
plt.imshow(cv2.cvtColor(labelled_image, cv2.COLOR_BGR2RGB))
plt.title("Labelled Image")
plt.axis("off")
plt.show()
# Output area
print("Area of the determined region:", area)
```

Output:



EXPERIMENT NUMBER: 04
EXPERIMENT NAME: Write a program to execute the following image enhancement.
• Basic Intensity Transformation (Negation, Log transformation, Power law transformation, Piecewise-Linear Transformation).
• Convolution, High/low pass/Laplacian filter.

Objective:

✓ Delve into fundamental intensity transformations for the purpose of improving the quality of images.

✓ Examination of the impact of convolution on images.

✓ Execution of edge detection techniques such as Canny, Prewitt, and Sobel operators to emphasize object boundaries and characteristics within images.

Theory:

Image enhancement represents a fundamental task within the field of image processing, with the primary objective of enhancing the visual quality or appearance of an image to facilitate improved analysis, interpretation, or presentation. This process encompasses a variety of techniques designed to alter the characteristics of an image in order to render it more suitable for particular applications or to emphasize specific features.

Basic intensity transformations constitute uncomplicated operations that are implemented on individual pixels within an image to modify their intensity values. Several commonly utilized intensity transformations include the following:

• Image Negations: Image negation is a straightforward process involving the inversion of pixel values within an image. In the case of a typical grayscale image, where pixel values range from 0 to 255 (assuming an 8-bit depth), negating the image entails subtracting each pixel value from the maximum value (255 in this scenario), effectively generating a photographic negative effect.

• Log Transformations: Log transformations can be defined mathematically as s = clog(1+r), where s represents the output intensity, r≥0 denotes the input intensity of the pixel, and c is a scaling constant. The value of c is determined by 255/(log (1 + m)), with m representing the maximum pixel value in the image. This adjustment is made to prevent the final pixel value from exceeding (L-1) or 255, thereby enabling log transformation to map a narrow range of low-intensity input values to a broader spectrum of output values.

• Power-Law (Gamma) Transformations: Power-law (gamma) transformations are expressed mathematically as $S = cr\gamma$. Gamma correction plays a critical role in ensuring the accurate display of images on screens, thereby averting the issues of bleaching or darkening that may arise when images are viewed across varying types of monitors with distinct display configurations.

• Piecewise-Linear Transformation Functions: A piecewise linear transformation function is a mathematical function that facilitates the mapping of input values from one range to output values within another range by employing linear segments between specific points. This particular type of function finds widespread application in image processing for diverse objectives such as contrast stretching, intensity transformations, and histogram equalization.

• Convolution Operation: The process of convolution entails the movement of a filter, also known as a kernel or mask, across the image. A weighted sum of the pixel values within the filter window is computed. The filter, typically a small matrix with numeric coefficients, dictates the weight given to neighboring pixels. Each position involves aligning the filter's center with the pixel under processing, followed by calculating the weighted sum through element-wise multiplication of the filter coefficients and corresponding pixel values, culminating in summation.

• Kernel: The kernel selection determines the nature of the operation executed, whether it be blurring, sharpening, edge detection, and so forth. For instance, a Gaussian kernel is commonly applied for blurring to achieve a smoothing effect, while a Sobel or Prewitt kernel is utilized for edge detection.

• Border Handling: The convolution process at image borders presents challenges due to inadequate neighboring pixels for all kernel positions. Techniques like zero-padding, mirror-padding, or wrap-around methods are employed to effectively manage border pixels.

• Output Size and Stride: The dimensions of the output image post-convolution are contingent on factors like the input image size, kernel dimensions, and convolution stride (step size). The stride dictates the filter's movement between successive positions during convolution.

• Mathematical Representation: Convolution is mathematically denoted as the integral of the product of two functions, where one function is reversed and shifted across the other. In the discrete realm of images, this integral simplifies to a summation of the element-wise product of the image and kernel.

• Feature Extraction: Convolution serves as a mechanism for extracting features, accentuating specific patterns or structures within the image. Through the utilization of suitable kernels, convolution can highlight edges, textures, or other image attributes pertinent to subsequent processing endeavors.

• Implementation: Efficient implementation of convolution is achieved through methodologies such as Fast Fourier Transform (FFT) or Direct Convolution. Popular libraries like OpenCV or TensorFlow furnish optimized convolution functions for image processing tasks. Overall, convolution assumes a vital role in diverse image processing activities, offering a versatile and potent means of extracting meaningful insights from images.

Code:

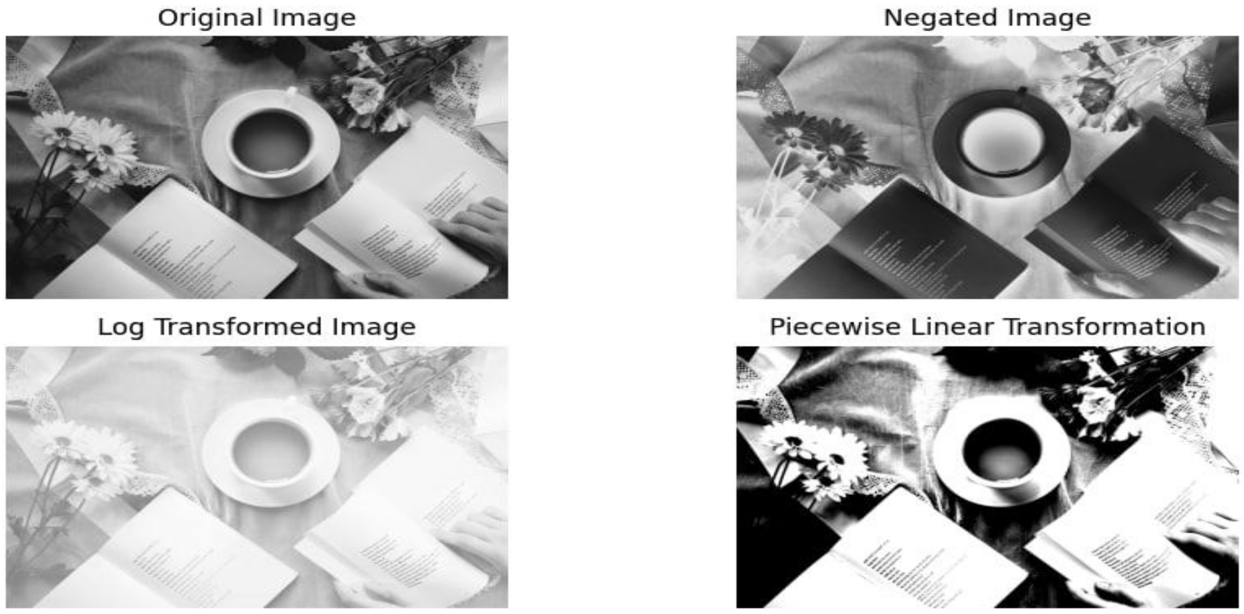
Applying basic intensity transformation functions on image

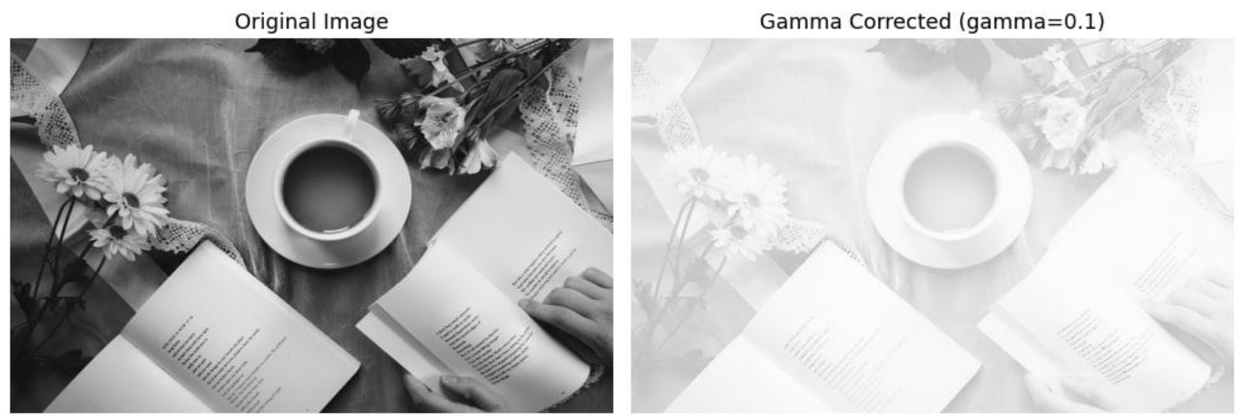
```
import cv2
import numpy as np
import matplotlib.pyplot as plt
import os
# Function for image negation
def image_negation(image):
negated_image = 255 - image
return negated_image
# Function for log transformation
def log_transform(image):
```

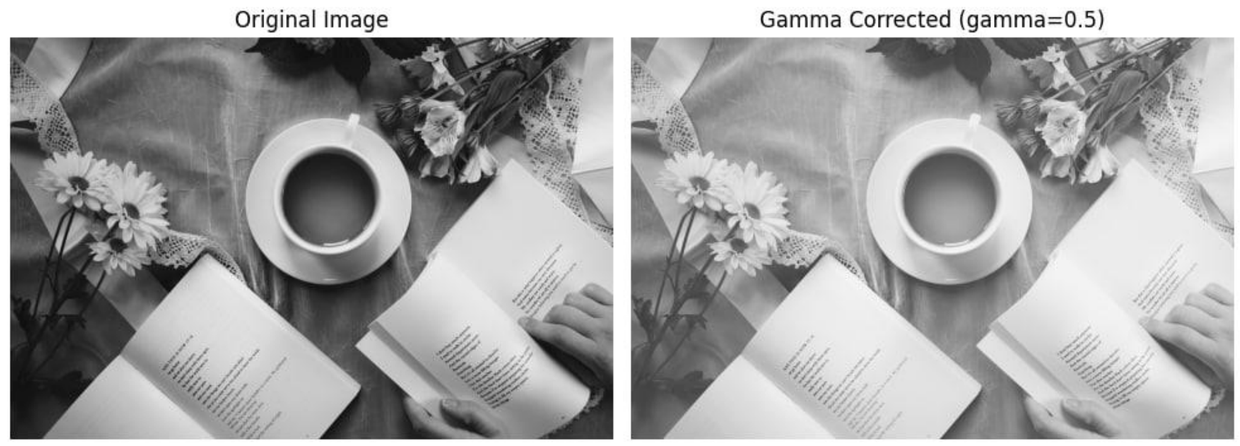
```
c = 255 / (np.log(1 + np.max(image)))
log_transformed = c * np.log(1 + image)
log_transformed = np.array(log_transformed, dtype=np.uint8)
return log_transformed
# Function for piecewise linear transformation
def piecewise_linear_transform(image, r1, s1, r2, s2):
def pixelVal(pix):
if 0 <= pix <= r1:
return (s1 / r1) * pix
elif r1 < pix <= r2:
return ((s2 - s1) / (r2 - r1)) * (pix - r1) + s1
else:
return ((255 - s2) / (255 - r2)) * (pix - r2) + s2
pixelVal_vec = np.vectorize(pixelVal)
transformed_image = pixelVal_vec(image)
return transformed_image
# Function for gamma correction
def gamma_correction(image, gamma):
gamma_corrected = np.array(255*(image / 255) ** gamma, dtype='uint8')
return gamma_corrected14
# Function for plotting original and corrected images
def plot_images(original, corrected, gamma):
fig, axes = plt.subplots(1, 2, figsize=(10, 5))
axes[0].imshow(cv2.cvtColor(original, cv2.COLOR_BGR2RGB))
axes[0].set_title('Original Image')
axes[0].axis('off')
axes[1].imshow(cv2.cvtColor(corrected, cv2.COLOR_BGR2RGB))
axes[1].set_title(f'Gamma Corrected (gamma={gamma})')
axes[1].axis('off')
plt.tight_layout()
plt.show()
# Open the image
image = cv2.imread('photoassignmentsfeat.jpg', cv2.IMREAD_GRAYSCALE)
# Perform image negation
negated_image = image_negation(image)
# Apply log transform
log_transformed = log_transform(image)
# Apply piecewise linear transformation
contrast_stretched = piecewise_linear_transform(image, r1=70, s1=0, r2=140, s2=255)
# Plot the images
fig, axes = plt.subplots(2, 2, figsize=(10, 5))
# Original image
axes[0, 0].imshow(image, cmap='gray')
axes[0, 0].set_title('Original Image')
axes[0, 0].axis('off')
# Negated image
axes[0, 1].imshow(negated_image, cmap='gray')
axes[0, 1].set_title('Negated Image')
axes[0, 1].axis('off')
# Log transformed image
axes[1, 0].imshow(log_transformed, cmap='gray')
axes[1, 0].set_title('Log Transformed Image')
axes[1, 0].axis('off')
```

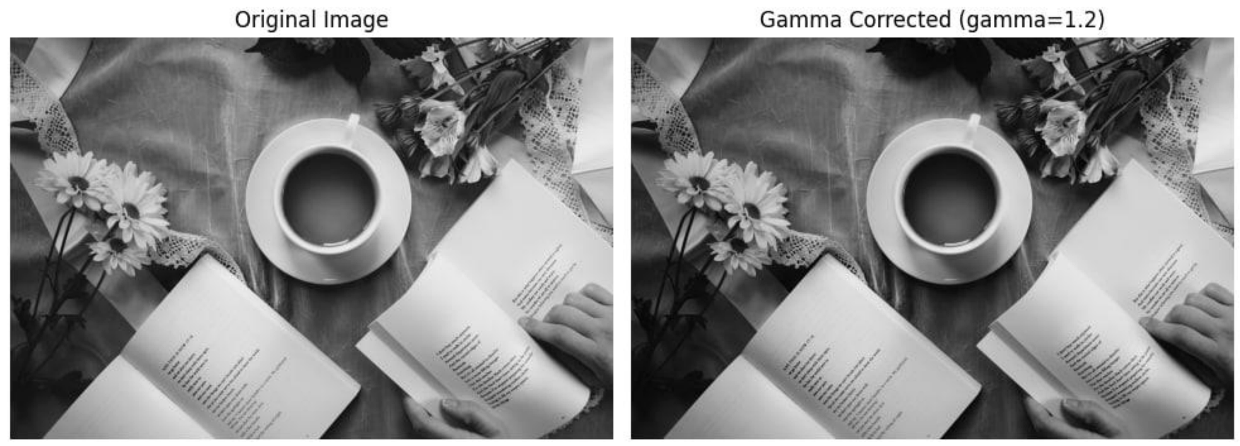
```
# Piecewise linear transformed image
axes[1, 1].imshow(contrast_stretched, cmap='gray')
axes[1, 1].set_title('Piecewise Linear Transformation')
axes[1, 1].axis('off')
plt.tight_layout()
plt.show()
# Save the transformed images
output_folder = 'transformed_images'
os.makedirs(output_folder, exist_ok=True)
cv2.imwrite(os.path.join(output_folder, 'negated_image.jpg'), negated_image)
cv2.imwrite(os.path.join(output_folder, 'log_transformed.jpg'), log_transformed)
cv2.imwrite(os.path.join(output_folder, 'contrast_stretched.jpg'), contrast_stretched)
# Trying 4 gamma values
gamma_values = [0.1, 0.5, 1.2, 2.2]
for gamma in gamma_values:
# Apply gamma correction
gamma_corrected = gamma_correction(image, gamma)
# Save edited images15
output_path = os.path.join(output_folder, f'gamma_transformed_{gamma}.jpg')
cv2.imwrite(output_path, gamma_corrected)
# Plot the original and gamma-corrected images
plot_images(image, gamma_corrected, gamma)
```

Output:









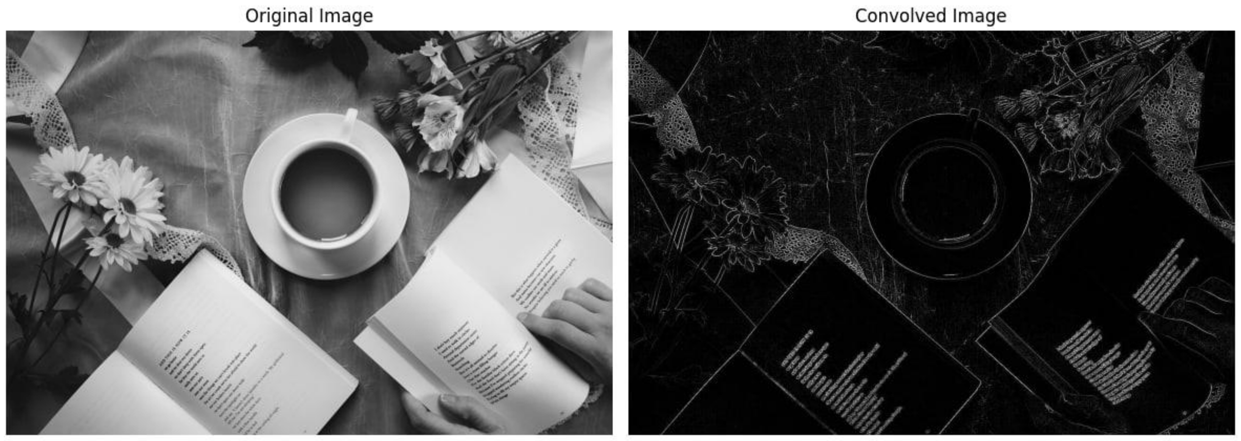
Code:

Convolution with 3*3 mask

```
import cv2
import matplotlib.pyplot as plt
import numpy as np
image_path = 'photoassignmentsfeat.jpg'
image = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)
kernel = np.array([[-1, -1, -1], [-1, 8, -1], [-1, -1, -1]])
convolved_image = cv2.filter2D(image, -1, kernel)
fig, axes = plt.subplots(1, 2, figsize=(12, 10))
```

```
axes[0].imshow(image, cmap='gray')
axes[0].set_title('Original Image')
axes[0].axis('off')
axes[1].imshow(convolved_image, cmap='gray')
axes[1].set_title('Convolved Image')
axes[1].axis('off')
plt.tight_layout()
plt.show()
```

Output:



EXPERIMENT NUMBER: 05
EXPERIMENT NAME: Write a program to execute the following edge detections
• Canny edge detection
• Prewitt edge detection
• Sobel edge detection

OBJECTIVES:
✓ To implement Canny, Prewitt, and Sobel edge detection algorithms to extract edges from images.

Theory:

Edge detection constitutes a fundamental operation within image processing, with the purpose of identifying boundaries in images where notable changes in intensity are observed. The Canny, Prewitt, and Sobel algorithms are extensively utilized techniques for the detection of edges in images. The following elucidates the theoretical framework underpinning each of these methodologies:

• Canny Edge Detection: The Canny edge detection algorithm is a multi-stage process that initiates with the application of a Gaussian filter to the image for noise reduction. Subsequently, the Sobel operator is employed to compute the gradient magnitude and direction. To refine the edges to a single pixel width, a non-maximum suppression technique is implemented. Ultimately, hysteresis thresholding is utilized to identify edges based on designated high and low threshold values, connecting edge pixels above the high threshold and considering those linked to them above the low threshold as edges.

• Prewitt Edge Detection: Prewitt edge detection is a straightforward gradient-based approach for detecting edges. It involves the calculation of gradient magnitude and direction using a pair of 3x3 convolution kernels - one for horizontal alterations and the other for vertical modifications. The magnitude of the gradient signifies the strength of the edge, while the direction indicates the orientation of the edge.

• Sobel Edge Detection: Analogous to Prewitt, the Sobel edge detection technique also determines gradient magnitude and direction through the utilization of convolution kernels. Sobel kernels are specifically crafted to assign greater emphasis to the central pixels, thereby yielding superior outcomes in edge detection. Sobel operators are commonly favored due to their efficacy and ease of implementation.

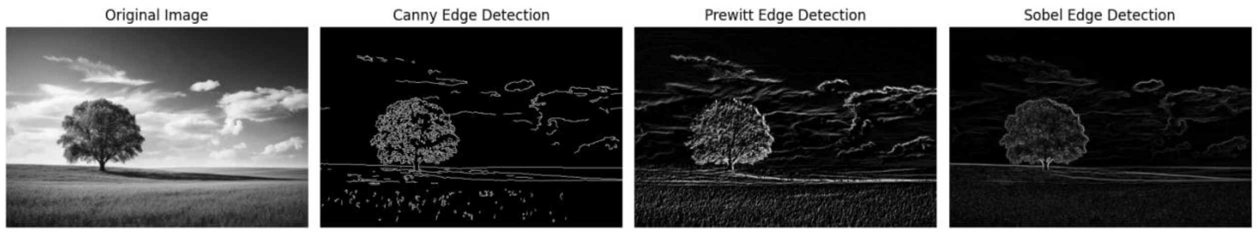
Code:

Applying edge detection using Canny, Prewitt and Sobel Operator

```
import cv2import matplotlib.pyplot as
plt# Read the imageimage =
cv2.imread('photoassignmentsfeat.jpg',
cv2.IMREAD_GRAYSCALE)#
Canny edge detectionedges_canny =
cv2.Canny(image, 100, 200)# Prewitt
edge detectionkernelx_prewitt =
cv2.getDerivKernels(1, 0,
3)kernely_prewitt =
cv2.getDerivKernels(0, 1,
3)edges_prewitt_x =
cv2.filter2D(image, -1,
kernelx_prewitt[0] *
kernely_prewitt[0].T) edges_prewitt_y
= cv2.filter2D(image, -1,
kernely_prewitt[0] *
kernelx_prewitt[0].T) edges_prewitt =
edges_prewitt_x + edges_prewitt_y#
Sobel edge detectionedges_sobel_x =
cv2.Sobel(image, cv2.CV_64F, 1, 0,
ksize=3) edges_sobel_y =
cv2.Sobel(image, cv2.CV_64F, 0, 1,
ksize=3) edges_sobel =
cv2.magnitude(edges_sobel_x,
edges_sobel_y)# Plotting
```

```
fig, axes = plt.subplots(1, 4, figsize=(15,
5))axes[0].imshow(image, cmap='gray')
axes[0].set_title('Original Image')
axes[0].axis('off')axes[1].imshow(edges_canny,
cmap='gray') axes[1].set_title('Canny Edge
Detection')
axes[1].axis('off')axes[2].imshow(edges_prewitt,
cmap='gray') axes[2].set_title('Prewitt Edge
Detection')
axes[2].axis('off')axes[3].imshow(edges_sobel,
cmap='gray') axes[3].set_title('Sobel Edge
Detection') axes[3].axis('off')plt.tight_layout()
plt.show()
```

Output:



EXPERIMENT NUMBER: 06
EXPERIMENT NAME: Write a program to execute the following speech preprocessing
• Identify sampling frequency.
• Identify bit resolution
• Make downsampling frequency then save the speech signal

Objectives :

✓ The aim of this study is to determine the sampling frequency of the speech signal.
✓ This study seeks to ascertain the bit resolution of the speech signal.
✓ The objective is to reduce the sampling frequency of the speech signal to a specific target.
✓ The final goal is to store the preprocessed speech signal.

Theory:

1. The Sampling Frequency (Sampling Rate) pertains to the number of samples acquired from a signal within a second, typically quantified in Hertz (Hz). Within the realm of digital signal processing, encompassing audio processing, the sampling frequency establishes the quantity of data points recorded per time unit. An increased sampling frequency enhances fidelity, yet it necessitates augmented storage capacity and computational capabilities.

2. Bit Resolution denotes the quantity of bits utilized to portray each sample of the signal. It defines the dynamic range and accuracy of the signal depiction. Enhanced bit resolution enables more refined distinctions among signal levels, culminating in superior audio quality, particularly for dynamic ranges.

3. Downsampling is the process of diminishing the sampling frequency of a signal. This technique is frequently utilized to reduce data size or tailor the signal to specific requirements, such as compatibility with designated hardware or communication systems. During downsampling, a low-pass filter is commonly implemented to prevent aliasing, wherein high-frequency components surpassing the new Nyquist frequency fold back into the signal. In this **particular experiment:**

- The speech signal is loaded utilizing the Librosa library.
- The original signal's sampling frequency and bit resolution are ascertained and displayed.
- The waveform and spectrogram of the original signal are graphed for visualization.
- Playback of the original signal is initiated.
- The signal is downsampled to a target sampling frequency (8000 Hz in this instance).
- The downsampled signal's sampling frequency and bit resolution are determined and exhibited.
- The downsampled signal is stored as a WAV file.
- The bit resolution of the downsampled signal is decreased to 16 bits.
- The downsampled signal with reduced bit resolution is saved as a WAV file.
- Graphs illustrating the waveform and spectrogram of the downsampled signal with reduced bit resolution are generated for visualization.
- Playback of the downsampled signal with reduced bit resolution is initiated.

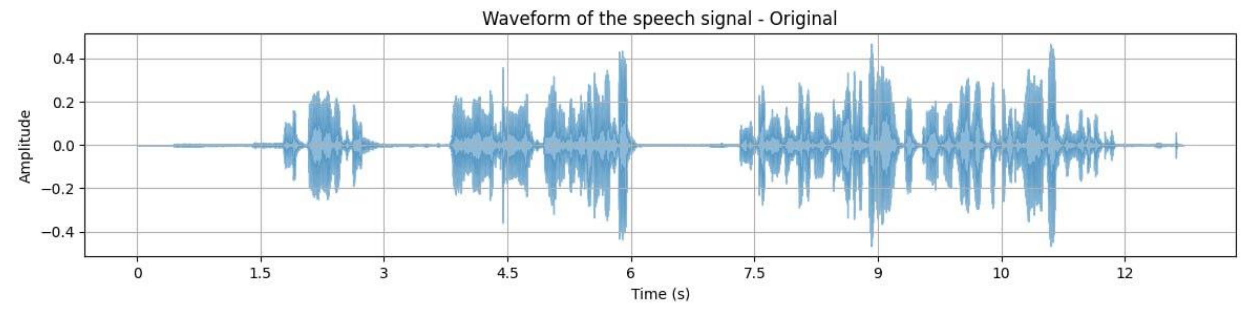
Code:

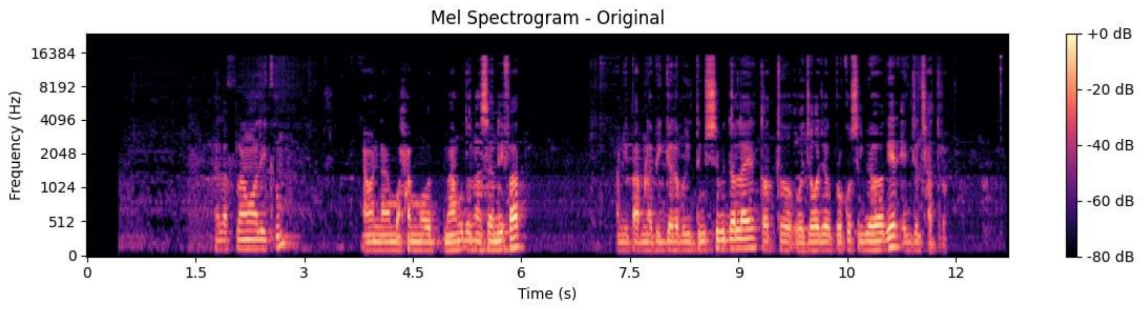
```
import numpy as np
import librosa
import librosa.display
import matplotlib.pyplot as plt
from scipy.io import wavfile
from IPython.display import Audio, display
# Example usage:
file_path = "/content/drive/MyDrive/Digital Image-Signal Processing/harvard.wav"
def plot_waveform_and_spectrogram(signal, sr, title):20
# Plot the waveform
plt.figure(figsize=(12, 6))
plt.subplot(2, 1, 1)
librosa.display.waveshow(signal, sr=sr, alpha=0.5)
plt.title('Waveform of the speech signal - ' + title)
plt.xlabel('Time (s)')
plt.ylabel('Amplitude')
plt.grid()
# Plot the spectrogram
plt.subplot(2, 1, 2)
spectrogram = librosa.feature.melspectrogram(y=signal, sr=sr)
librosa.display.specshow(librosa.power_to_db(spectrogram, ref=np.max), sr=sr, x_axis='time',
y_axis='mel')
plt.colorbar(format='%+2.0f dB')
plt.title('Mel Spectrogram - ' + title)
plt.xlabel('Time (s)')
plt.ylabel('Frequency (Hz)')
plt.tight_layout()
plt.show()
# Load the audio file with Librosa
signal, sr = librosa.load(file_path, sr=None)
# Identify sampling frequency and bit resolution
print('Original Sampling Frequency:', sr)
print('Original Bit Resolution:', signal.dtype)
# Plot waveform and spectrogram of the original signal
plot_waveform_and_spectrogram(signal, sr, 'Original')
# Play the original signal
display(Audio(signal, rate=sr))
# Set sampling frequency to 8000 Hz
sr_8000 = 8000
signal_resampled = librosa.resample(signal, orig_sr=sr, target_sr=sr_8000)
# Identify sampling frequency and bit resolution of the resampled signal
print('Resampled Sampling Frequency:', sr_8000)
print('Resampled Bit Resolution:', signal_resampled.dtype)
# Save the resampled signal
wavfile.write('speech_signal_8000Hz.wav', sr_8000, (signal_resampled * 32767).astype(np.int16))
# Plot waveform and spectrogram of the resampled signal
plot_waveform_and_spectrogram(signal_resampled, sr_8000, 'Resampled (8000 Hz)')
# Play the resampled signal
display(Audio(signal_resampled, rate=sr_8000))
# Reduce bit resolution to 16 bits
signal_resampled_16bit = (signal_resampled * 32767).astype(np.int16)
```

```
# Identify sampling frequency and bit resolution of the resampled signal
print('Resampled Sampling Frequency:', sr_8000)
print('Resampled Bit Resolution:', signal_resampled_16bit.dtype)
# Save the resampled signal with reduced bit resolution
wavfile.write('speech_signal_8000Hz_16bit.wav', sr_8000, signal_resampled_16bit)
# Plot waveform and spectrogram of the resampled signal
plot_waveform_and_spectrogram(signal_resampled, sr_8000, 'Resampled (8000 Hz)')
# Play the resampled signal
display(Audio(signal_resampled, rate=sr_8000))
```

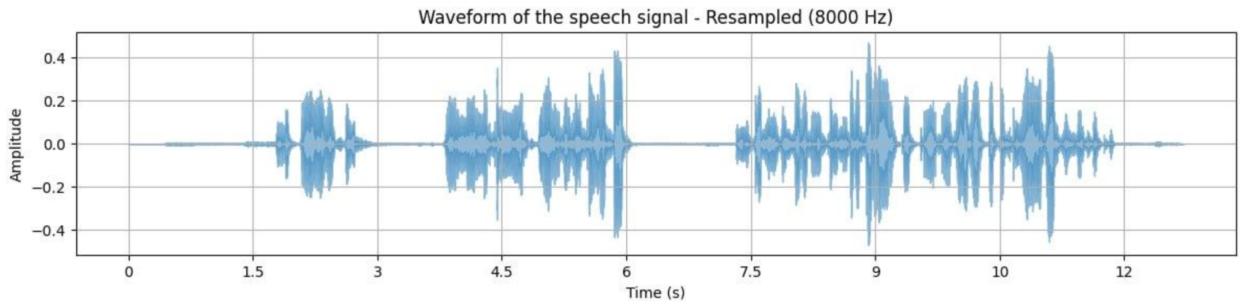
Output:

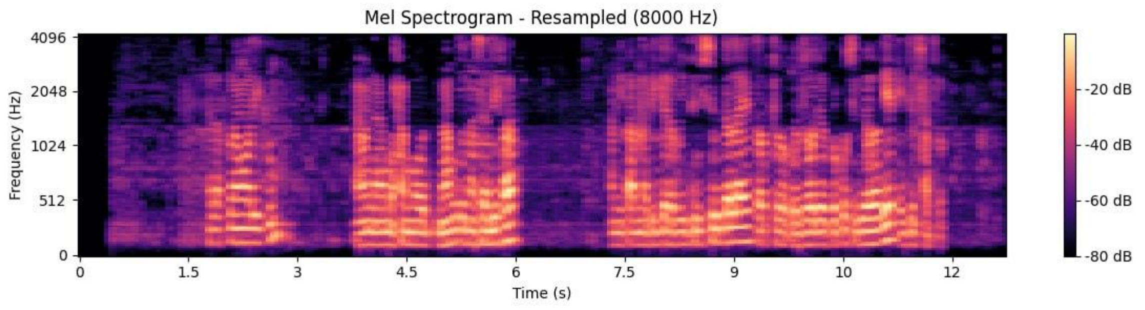
```
Original Sampling Frequency: 44100
Original Bit Resolution: float32
```



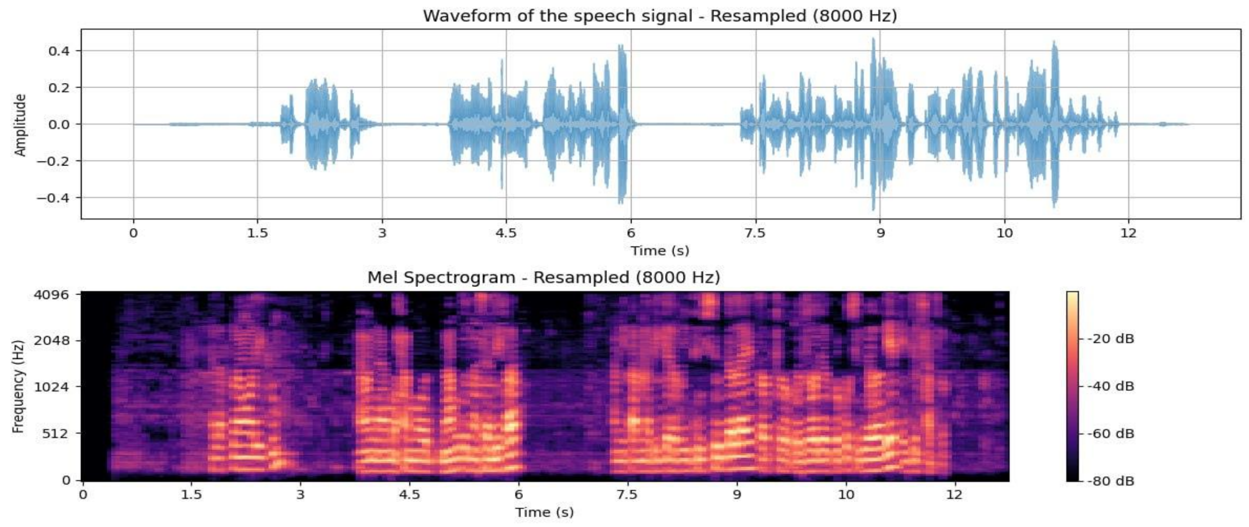


```
Resampled Sampling Frequency: 8000
Original Bit Resolution: float32
```





Resampled Sampling Frequency: 8000
Original Bit Resolution: int16



EXPERIMENT NUMBER: 07
EXPERIMENT NAME: Write a program to display the following region of a speech signal.
• Voiced region.
• Unvoiced region.
• Silenced region.

Objectives:

- ✓ To create a program that uses a speech signal's properties to separate it into voiced, unvoiced, and
- ✓ To display and distinguish the speech signal's voiced, unvoiced, and quiet areas for later processing and analysis.

Theory:

The waveform that the human vocal tract produces during speech is known as a speech signal. It is a complicated audio signal made up of different speech-related frequencies, amplitudes, and durations. Speech recognition, synthesis, and analysis are just a few of the applications that can be made possible by analyzing and processing these signals to extract information such as phonemes, words, or phrases. The voice signal can be divided into three categories, which are as follows:

1. **Voiced Speech:** The sounds made by the larynx's vocal cords, also known as the vocal folds, vibrate during voiced speech. When speaking aloud, air travels through the vocal cords, causing them to vibrate and create a periodic waveform. These noises typically have a lower frequency and a distinctive periodicity.
2. **Unvoiced Speech:** Sounds generated without the vocal chords vibrating are referred to as unvoiced speech. Turbulence caused by airflow over a narrowed vocal tract during unvoiced speaking produces noise-like sounds. Due to the existence of noise components, these sounds have a higher frequency range and lack the regularity of vocal speech.
3. **Speech that is Silenced:** Speech that is silenced is speech that is produced without or with less vocalization. The vocal cords do not vibrate during silent speaking, and the vocal tract's airflow may be restricted or entirely stopped. Whispered speech and speech generated with closed or occluded vocal tract movements are two examples of hushed speech. The regularity of voiced discourse and the volatility of unvoiced speech are frequently absent from silenced speech.

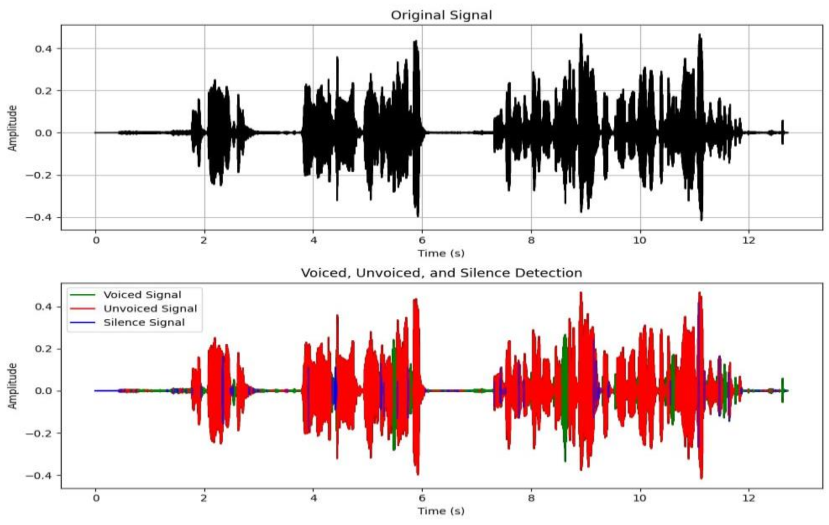
In this trial
1. A predetermined overlap separates the voice signal into fixed-size frames. The energy and zero-crossing rate of each frame are determined separately.
2. The total of the squared magnitudes of the samples in each frame is used to calculate its energy. The distinction between spoken, unvoiced, and quiet areas is aided by this energy meter.
3. The number of times the signal changes sign divided by the frame length yields the zero-crossing rate for each frame. To differentiate between voiced and unvoiced regions, ZCR offers information on the rate of signal variations.
4. To categorize each frame as voiced, unvoiced, or silent, thresholds for energy and zero-crossing rate are established.

Code:

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.lines import Line2D
# Read the audio file
y, fs = librosa.load('/content/drive/MyDrive/Digital Image-Signal Processing/harvard.wav', sr=None)
# Define frame size and overlap (in samples)
frame_size = 256
overlap = 128
# Calculate number of frames
num_frames = (len(y) - frame_size) // (frame_size - overlap) + 1
# Initialize variables
voiced_frames = []
unvoiced_frames = []
silence_frames = []
# Iterate through each frame
for i in range(num_frames):
# Extract current frame
start_idx = i * (frame_size - overlap)
end_idx = start_idx + frame_size
frame = y[start_idx:end_idx]
# Calculate energy of the frame
energy = np.sum(np.abs(frame)**2)
# Calculate zero-crossing rate (ZCR)
zcr = np.sum(np.diff(np.sign(frame)) != 0)
# Thresholds for voiced, unvoiced, and silence detection
voiced_threshold = 0.01 * np.max(energy) # adjust threshold based on your audio
unvoiced_threshold = 0.001 * np.max(energy) # adjust threshold based on your audio
silence_threshold = 0.0001 * np.max(energy) # adjust threshold based on your audio
# Identify frame type based on energy and ZCR
if energy > voiced_threshold and zcr > 10: # adjust values for voiced detection
voiced_frames.append(i)
elif energy > unvoiced_threshold and zcr < 10: # adjust values for unvoiced detection
unvoiced_frames.append(i)
else:
silence_frames.append(i)
# Calculate time axis for plotting
time_axis = np.arange(len(y)) / fs
# Plot the original signal and the segmented parts
plt.figure(figsize=(10, 8))
```

```
# Original Signal
plt.subplot(2, 1, 1)
plt.plot(time_axis, y, 'k')
plt.xlabel('Time (s)')
plt.ylabel('Amplitude')
plt.title('Original Signal')
plt.grid(True)25
# Voiced, Unvoiced, and Silence Detection
plt.subplot(2, 1, 2)
plt.plot(time_axis, y, 'k')
# Plot voiced segments
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', alpha=0.5)
# Plot unvoiced segments
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', alpha=0.5)
# Plot silence segments
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', alpha=0.5)
# Create custom legend entries
legend_elements = [
Line2D([0], [0], color='g', label='Voiced Signal'),
Line2D([0], [0], color='r', label='Unvoiced Signal'),
Line2D([0], [0], color='b', label='Silence Signal')
]
plt.xlabel('Time (s)')
plt.ylabel('Amplitude')
plt.title('Voiced, Unvoiced, and Silence Detection')
plt.legend(handles=legend_elements)
plt.tight_layout()
plt.show()
```

Output:



EXPERIMENT NUMBER: 08
EXPERIMENT NAME: Write a program to compute zero crossing rate (ZCR) using different window function of a speech signal.
OBJECTIVES:
✓ To compute the zero crossing rate (ZCR) of a speech signal using different window functions.

THEORY: The zero crossing rate (ZCR) is a fundamental feature of speech signals that quantifies the rate at which the signal changes its sign. It is a crucial parameter used in various speech processing applications, including speech recognition, speaker identification, and emotion detection.
Zero Crossing Rate (ZCR): The ZCR measures the frequency at which a speech signal changes polarity, indicating the rate of waveform fluctuations. In voiced speech, the ZCR tends to be lower due to the periodic vibration of the vocal cords, resulting in fewer zero crossings. Unvoiced speech and noise tend to have higher ZCR values due to their noisy and turbulent nature. This relation can be modified for non-stationary signals like speech and termed as short term ZCR. It is defined as $(n) = 1\ 2N \sum (m).\ (n - m)\ {}_{N\text{-}1\ m=0}$

Windowing: Windowing involves dividing the speech signal into short segments, or frames, to analyze local characteristics over time. Different window functions can be applied to shape these frames, with common choices including rectangular, Hamming, Hanning, and Blackman windows. Windowing helps reduce spectral leakage and improve the accuracy of ZCR estimation by focusing on local segments of the signal
Computing ZCR: To compute the ZCR, each frame of the speech signal is analyzed individually. Within each frame, the number of times the signal crosses the zero axis is counted. The ZCR is then calculated as the ratio of the number of zero crossings to the total number of samples in the frame. Different window functions may affect the accuracy and sensitivity of ZCR estimation, as they alter the shape and characteristics of the analyzed frames.
In this experiment:

- The speech signal is loaded from an audio file.
- A portion of the speech signal is extracted, typically lasting for 30 milliseconds.
- The zero crossing rate (ZCR) of the extracted portion is computed using different window functions,
- namely rectangular, Hann, and Hamming windows.
- Each window function is applied to the signal portion, and the resulting ZCR is plotted against time for
- visualization.

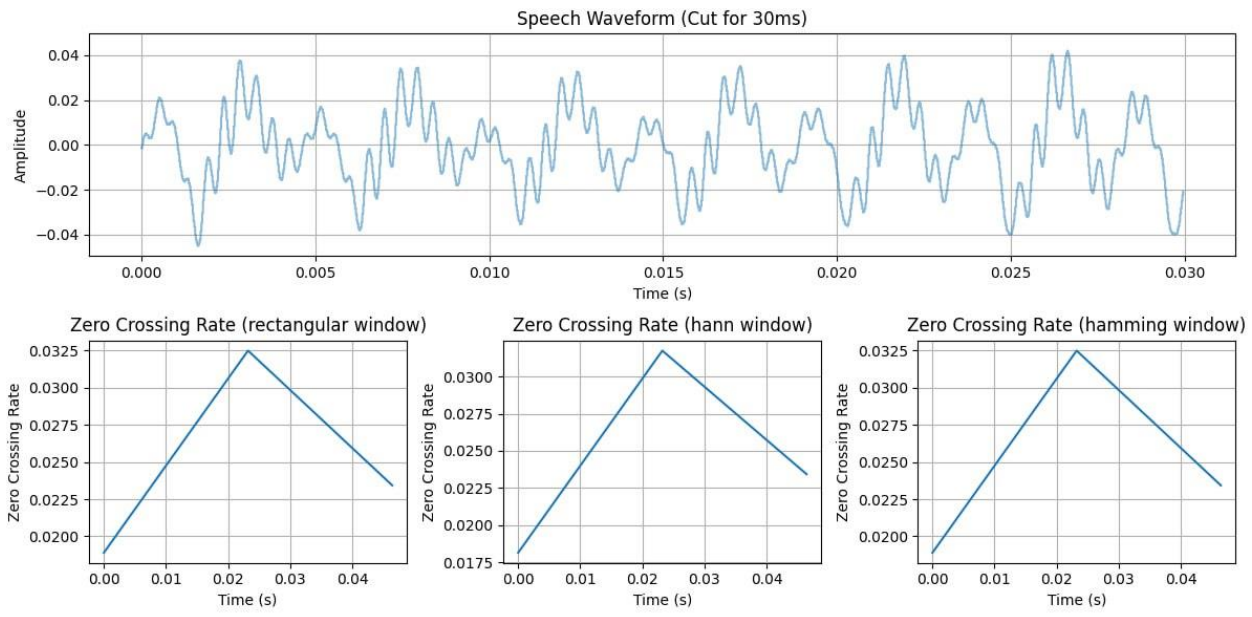
This experiment helps in understanding the influence of window functions on the computation of the zero crossing rate, providing insights into the selection of appropriate window functions for specific applications in speech and audio processing.

Code:

```
import numpy as np
import librosa
import librosa.display
import matplotlib.pyplot as plt
from IPython.display import Audio, display
# Load the audio file
audio_file = "/content/jackhammer.wav"
signal, sr = librosa.load(audio_file, sr=None)
# Play the audio
display(Audio(signal, rate=sr))
# Cut a portion of the speech signal (for example, for 30 ms)
cut_signal = signal[int(sr * 2.5):int(sr * 2.53)]27
# Parameters
frame_length = int(sr * 0.03) # 30 ms frame length
hop_length = 512
# Compute zero-crossing rate (ZCR) for the cut portion with different window functions
window_functions = ['rectangular', 'hann', 'hamming']
plt.figure(figsize=(12, 6))
# Plot speech waveform
```

```
plt.subplot(2, 1, 1)
librosa.display.waveshow(cut_signal, sr=sr, alpha=0.5)
plt.xlabel('Time (s)')
plt.ylabel('Amplitude')
plt.title('Speech Waveform (Cut for 30ms)')
plt.grid(True
for i, window_function in enumerate(window_functions, start=1):
# Compute window function
if window_function == 'rectangular':
window = np.ones(frame_length)
else:
window = librosa.filters.get_window(window_function, frame_length, fftbins=True)
# Truncate window to match the length of the signal
window = window[:len(cut_signal)]
# Apply window to the signal
signal_windowed = cut_signal * window
# Compute zero-crossing rate (ZCR)
zcr = librosa.feature.zero_crossing_rate(y=signal_windowed, frame_length=frame_length,
hop_length=hop_length)
plt.subplot(2, len(window_functions), len(window_functions) + i)
plt.plot(librosa.frames_to_time(range(len(zcr[0])), hop_length=hop_length), zcr[0])
plt.xlabel('Time (s)')
plt.ylabel('Zero Crossing Rate')
plt.title(f'Zero Crossing Rate ({window_function} window)')
plt.grid(True)
plt.tight_layout()
plt.show()
```

Output:



EXPERIMENT NUMBER: 09
EXPERIMENT NAME: Write a program to compute short term auto-correlation of a speech signal.
OBJECTIVES:
✓ To compute the short-term autocorrelation of a speech signal.

Theory:
The similarity between a signal and a time-delayed version of itself across brief time intervals is measured by the speech signal's short-term autocorrelation. It offers information about the periodicity, pitch, and other aspects of the speech signal, making it a vital tool in speech processing and analysis. The definition of the equivalent short term autocorrelation for a non-stationary sequence s(n) is:

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

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

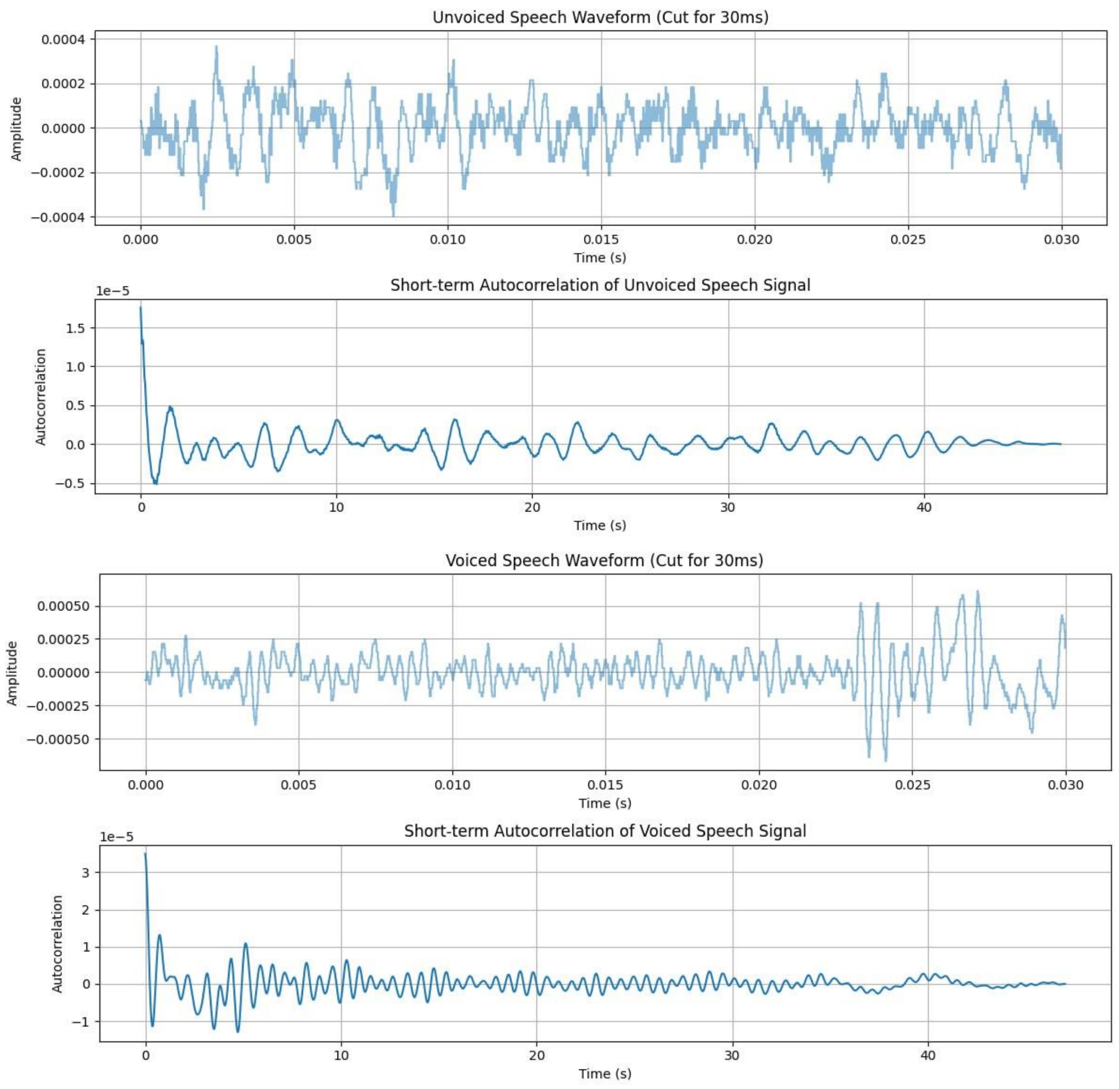
where sw(n) equals s(m).The windowed version of s(n) is w(n-m). The short-term autocorrelation for a given windowed speech segment is therefore a series.
In this trial:

1. An audio file is used to load the speech signal.
2. An extracted segment of the voice signal, usually lasting 30 ms, is received.
3. The librosa.autocorrelate() function is used to calculate the extracted portion's short-term autocorrelation.
4. To see the short-term fluctuations, the computed autocorrelation is then shown versus time.
5. The plots help with the study of voiced and unvoiced regions by revealing information about the periodicity and temporal structure of the speech signal.

We may examine the temporal fluctuations in speech signals by calculating short-term autocorrelation, which is essential for tasks like pitch estimation.

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, type):
# Compute short-term autocorrelation
auto_corr = librosa.autocorrelate(y=cut_signal, max_size=frame_length)
# Plot speech waveform and autocorrelation
plt.figure(figsize=(12, 6))
# Plot speech waveform
plt.subplot(2, 1, 1)
librosa.display.waveshow(cut_signal, sr=sr, alpha=0.5)
plt.xlabel('Time (s)')
plt.ylabel('Amplitude')
plt.title(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 ' + type + ' Speech Signal')29
plt.grid(True)
plt.tight_layout()
plt.show()
# Usage example
audio_file = "/content/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_signal, 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_signal, sr, frame_length, hop_length, 'Unvoiced')
```

Output:



EXPERIMENT NUMBER: 10
EXPERIMENT NAME: Write a program to estimate pitch of a speech signal. **OBJECTIVES:**
✓ To estimate the pitch of a speech signal using autocorrelation.

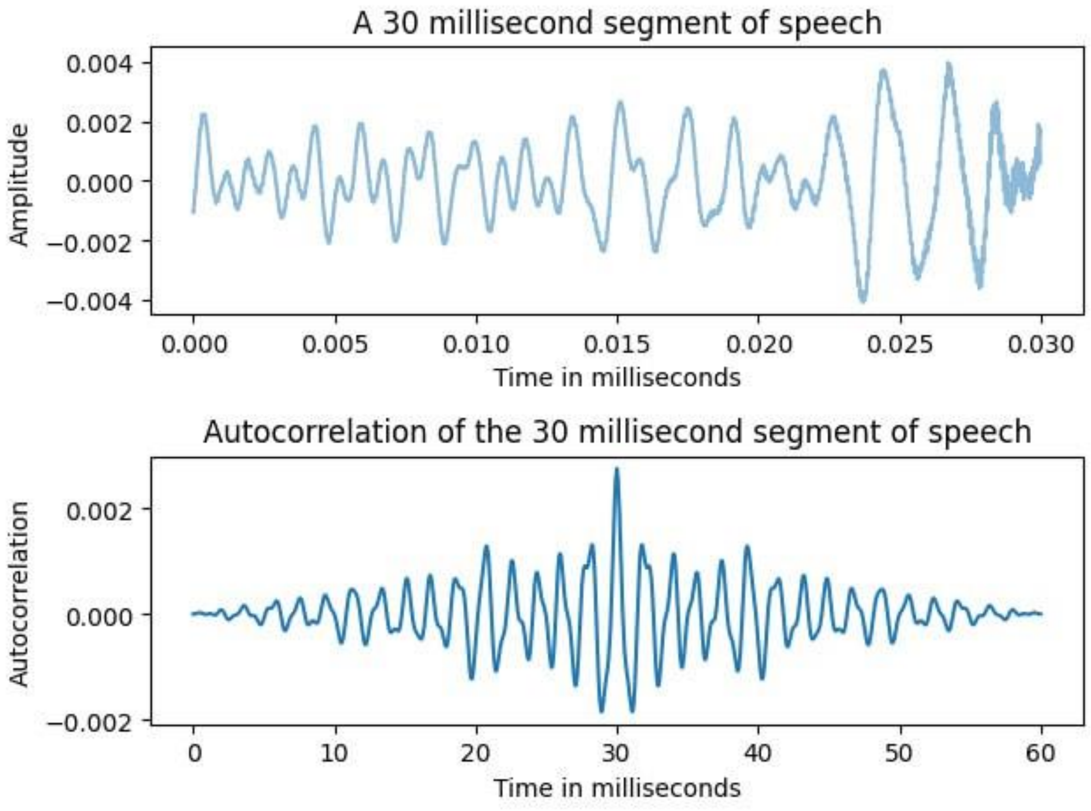
Theory:
Pitch is a characteristic of sound perception that enables us to identify tones as having a greater or lower frequency. Pitch in speech processing is the perceived pitch of the speech stream, which is determined by the fundamental frequency (F0) of the vocal cord vibrations.
A mathematical technique called autocorrelation is used to quantify how similar a signal is to a delayed version of itself. Autocorrelation can be used to determine the speech signal's periodicity, or pitch period, in the context of pitch estimation.

In this evaluation:
1. The Librosa library is used to load the speech segment from a WAV file.
2. For analysis, a segment of the voice signal, usually lasting 30 milliseconds, is taken.
3. The np.correlate() function is used to calculate autocorrelation for the retrieved speech segment.
4. The similarity between the signal and its delayed versions is displayed by the autocorrelation plot that is created.
5. The speech signal's pitch period is represented by the maximum peak in the autocorrelation plot.
6. The location of the greatest peak in the autocorrelation plot is used to calculate the pitch period (To) and pitch frequency (Fo).
7. The estimated pitch of the speech segment is then printed together with the pitch period and pitch frequency.

Speech recognition, speech synthesis, and speaker identification are just a few of the speech processing applications that depend on our ability to estimate the pitch of a spoken signal and determine the fundamental frequency of the vocal cord vibrations.

Code:
```
import numpy as np
import matplotlib.pyplot as plt
import librosa
# Read the speech segment from the WAV file using Librosa
y, Fs = librosa.load('/content/drive/MyDrive/Digital Image-Signal Processing/Pitch_Estimation.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)
# plt.plot(np.arange(len(y)) / Fs * 1000, y)
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')
# Plot autocorrelation
plt.subplot(2, 1, 2)31
plt.plot(kk, autocorrelation)
plt.xlabel('Time in milliseconds')
plt.ylabel('Autocorrelation')
plt.title('Autocorrelation of the 30 millisecond segment of speech')
# Extract relevant part of autocorrelation (21 to 160)
auto = autocorrelation[20:160]
# Find the maximum value and corresponding sample number
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:


A 30 millisecond segment of speech
Amplitude
0.004
0.002
0.000
−0.002
−0.004
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
0.002
0.000
−0.002
0
10
20
30
40
50
60
Time in milliseconds