Experiment Name: Write a program to execute 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.

Theory:

Image Read: Image reading is a crucial step in image processing pipelines. It involves loading image data from a storage location, such as a folder or directory, into memory for further processing. Python provides various libraries for image reading and processing, with OpenCV being one of the most widely used due to its extensive functionality and efficiency.

Image Resizing: Image resizing is the process of changing the dimensions of an image. This is often done to make images suitable for specific purposes, such as fitting them into a fixed-size container or reducing computational overhead in subsequent processing steps. In this experiment, image resizing is implemented using interpolation techniques such as nearest neighbor, bilinear, or bicubic interpolation.

Color Transform: Color transforms are used to alter the color characteristics of an image. Common color transforms include grayscale conversion, adjusting brightness and contrast, and converting between color spaces such as RGB, HSV, and LAB. These transforms are essential for adjusting the visual appearance of images or preparing them for specific analysis tasks.

Image Normalization: Image normalization is the process of standardizing the pixel values of an image to a predefined range or distribution. This is often done to improve the convergence and stability of machine learning algorithms that operate on images. Normalization techniques include min-max scaling, z-score normalization, and scaling to a fixed range such as [0, 1].

Image Filtering: Image filtering involves applying mathematical operations to modify the pixel values of an image. Common filters include Gaussian blur, median filter, and edge detection filters such as Sobel and Canny. Filtering can be used for tasks such as noise reduction, feature enhancement, and edge detection.

Python Source Code:

1.Read Images

```
import os
import cv2

def read_images_from_folder(folder):
    for filename in os.listdir(folder):
        img_path = os.path.join(folder, filename)
        img = cv2.imread(img_path)
        cv2.imshow(filename, img)
        cv2.waitKey(0)
        cv2.destroyWindow(filename)

# Example usage:
input_images_folder = 'input_images/'
read_images_from_folder(input_images_folder)
```

2.Image Resizing

```
import os
import cv2

def resize_images(input_folder, output_folder, target_size):
    for filename in os.listdir(input_folder):
        img_path = os.path.join(input_folder, filename)
        img = cv2.imread(img_path)
        resized_img = cv2.resize(img, target_size)
        output_path = os.path.join(output_folder, filename)
        cv2.imwrite(output_path, resized_img)

# Example usage:
input_folder = 'input_images/'
output_folder = 'resized_images/'
target_size = (300, 200)
resize_images(input_folder, output_folder, target_size)
```

3. Image Color Transform

```
import os
import cv2

def apply_color_transform(input_folder, output_folder):
    for filename in os.listdir(input_folder):
        img_path = os.path.join(input_folder, filename)
        img = cv2.imread(img_path)
        # Perform color transform (e.g., convert to grayscale)
        gray_img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
        output_path = os.path.join(output_folder, filename)
        cv2.imwrite(output_path, gray_img)

# Example usage:
input_folder = 'input_images/'
output_folder = 'color_transformed_images/'
apply_color_transform(input_folder, output_folder)
```

4.Image Normalization

```
import os
import cv2

def normalize_images(input_folder, output_folder):
    for filename in os.listdir(input_folder):
        img_path = os.path.join(input_folder, filename)
        img = cv2.imread(img_path).astype(float)
        # Perform normalization (e.g., min-max scaling)
        normalized_img = (img - img.min()) / (img.max() - img.min())
        output_path = os.path.join(output_folder, filename)
        cv2.imwrite(output_path, normalized_img * 255)

# Example usage:
input_folder = 'input_images/'
output_folder = 'normalized_images/'
normalize_images(input_folder, output_folder)
```

5.Image Filtering

```
import os
import cv2

def filter_images(input_folder, output_folder):
    for filename in os.listdir(input_folder):
        img_path = os.path.join(input_folder, filename)
        img = cv2.imread(img_path)
        # Apply image filtering (e.g., Gaussian blur)
        filtered_img = cv2.GaussianBlur(img, (5, 5), 0)
        output_path = os.path.join(output_folder, filename)
        cv2.imwrite(output_path, filtered_img)

# Example usage:
input_folder = 'input_images/'
output_folder = 'filtered_images/'
filter_images(input_folder, output_folder)
```

Input:



Output:

Resized Images



Color Transformed Images



Normalized Images



Filtered Images

Expeiment Name: Write a program to execute image segmentation.

- Semantic Segmentation.
- Instance Segmentation.
- Multilevel Thresholding Segmentation.

Theory:
Image segmentation is a crucial task in computer vision that involves partitioning an image into multiple segments or regions. Each segment typically corresponds to objects or regions of interest within the image. Segmentation is essential for various applications, including object detection, image recognition, and medical image analysis.
There are several types of image segmentation techniques, including semantic segmentation, instance segmentation, and multilevel thresholding segmentation:

Introduction to Semantic Segmentation:
Semantic segmentation is a fundamental task in computer vision that aims to assign a class label to each pixel in an image.Unlike classification tasks that assign a single label to an entire image, semantic segmentation provides pixel-level understanding of the scene. Semantic segmentation is widely used in various applications, including autonomous driving, medical imaging, scene understanding, and image editing.

Challenges in Semantic Segmentation:
Semantic segmentation faces several challenges, including variations in object appearance, occlusions, cluttered backgrounds, and ambiguous boundaries between classes.
Addressing these challenges requires the development of advanced algorithms capable of capturing spatial dependencies, handling class imbalance, and achieving accurate segmentation results.

Traditional Methods:
Traditional semantic segmentation methods often rely on handcrafted features and classical machine learning techniques such as support vector machines (SVMs), random forests, or graph-based algorithms.
These methods typically involve preprocessing steps like feature extraction, followed by a classification or clustering step to assign labels to pixels.

Deep Learning Approaches:
Deep learning has revolutionized semantic segmentation by enabling end-to-end learning of feature representations directly from raw pixel data.
Convolutional neural networks (CNNs) have emerged as the primary architecture for semantic segmentation tasks due to their ability to capture spatial dependencies and hierarchical features.
Popular CNN architectures for semantic segmentation include Fully Convolutional Networks (FCNs), U-Net, SegNet, and DeepLab.

U-Net Architecture:
U-Net is a widely used architecture for biomedical image segmentation, particularly in applications like cell segmentation and medical image analysis.
The U-Net architecture consists of a contracting path (encoder) and an expansive path (decoder), connected by skip connections.
Skip connections help preserve spatial information and facilitate the precise localization of objects in the segmentation map.

Training and Evaluation:
Training a semantic segmentation model involves optimizing a loss function (e.g., cross-entropy loss) to minimize the discrepancy between predicted and ground truth segmentation maps.
Common evaluation metrics for semantic segmentation include Intersection over Union (IoU), Pixel Accuracy, Mean Intersection over Union (mIoU), and Dice coefficient.
Models are typically evaluated on a separate validation set or through cross-validation to assess their generalization performance.

Applications:
Semantic segmentation has diverse applications across various domains, including autonomous driving (road and lane segmentation), medical imaging (organ and tumor segmentation), satellite imagery analysis, and augmented reality.

Python Source Code:

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

# Read the input image
input_image = cv2.imread("download3.jpg")  # Change this to the path of your input image

# Convert image to grayscale
gray = cv2.cvtColor(input_image, cv2.COLOR_BGR2GRAY)

# Apply thresholding
_, thresh = cv2.threshold(gray, 0, 255, cv2.THRESH_BINARY_INV + cv2.THRESH_OTSU)

# Find contours
contours, _ = cv2.findContours(thresh, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)

# Create a blank canvas for drawing contours
segmented_image = np.zeros_like(input_image)

# Draw contours on the blank canvas
cv2.drawContours(segmented_image, contours, -1, (255, 255, 255), thickness=cv2.FILLED)

# Convert segmented image to grayscale
segmented_image_gray = cv2.cvtColor(segmented_image, cv2.COLOR_BGR2GRAY)
```

```
# Display input and segmented images side by side
plt.figure(figsize=(10, 5))

# Display input image
plt.subplot(1, 2, 1)
plt.imshow(cv2.cvtColor(input_image, cv2.COLOR_BGR2RGB))
plt.title('Input Image')
plt.axis('off')

# Display segmented image in black and white
plt.subplot(1, 2, 2)
plt.imshow(segmented_image_gray, cmap='gray')
plt.title('Segmented Image (Black and White)')
plt.axis('off')

plt.show()
```

Input and Output:

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.

Theory:

Image Masking: An image mask is a binary image that defines a region of interest (ROI) within an image.

It consists of binary values (0 or 255) where 0 represents the background and 255 represents the foreground (or the region of interest).

Region of Interest (ROI): The region of interest is the part of the image defined by the mask. It is obtained by applying the mask to the original image, retaining only the pixels corresponding to the foreground in the mask.

Area Calculation: Once the region of interest is determined, the area can be calculated by counting the number of non-zero pixels in the mask. If the image has a known physical size, the area can be converted to a real-world unit (e.g., square centimeters) based on the pixel size.

Labeling: Labeling involves overlaying the mask onto the original image to visually identify the region of interest.

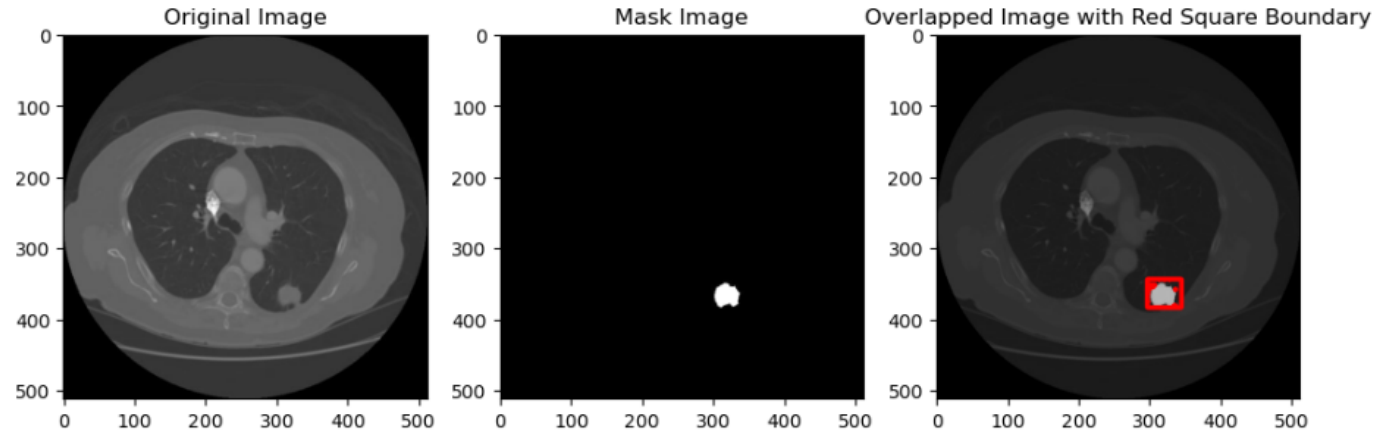
This is often done by blending the original image and the mask using transparency or a weighted addition.

Python Source Code:

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

# Load the image and the mask
image = cv2.imread('main.jpg')
mask = cv2.imread('mask.jpg', cv2.IMREAD_GRAYSCALE)

# Verify if images are loaded correctly
print("Image shape:", image.shape)
print("Mask shape:", mask.shape)

# Define pixel size of the image (e.g., pixel_size = 0.01 for 1 cm x 1 cm pixel)
pixel_size = 0.01  # Example: Assuming each pixel corresponds to 1 square centimeter

# Determine the region of interest in the image using the mask
region_of_interest = cv2.bitwise_and(image, image, mask=mask)

# Compute the area of the region
pixel_area = np.sum(mask != 0)  # Count non-zero pixels in the mask
area = pixel_area * pixel_size
print("Area of the region:", area, "square units")

# Label the region by overlapping the mask over the image
labeled_image = cv2.addWeighted(image, 0.5, cv2.cvtColor(mask, cv2.COLOR_GRAY2BGR), 0.5, 0)

# Plotting
fig, axes = plt.subplots(1, 3, figsize=(12, 4))

# Display the original image
axes[0].imshow(cv2.cvtColor(image, cv2.COLOR_BGR2RGB))
axes[0].set_title('Original Image')

# Display the mask image
axes[1].imshow(mask, cmap='gray')
axes[1].set_title('Mask Image')

# Display the overlapped image
axes[2].imshow(cv2.cvtColor(labeled_image, cv2.COLOR_BGR2RGB))
axes[2].set_title('Overlapped Image')

plt.show()
```

Input and Output:

```
Image shape: (512, 512, 3)
Mask shape: (512, 512)
Area of the region: 11.86 square units
```



Experiment Name: 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)

Theory:

Basic Intensity Transformations:

Basic intensity transformations are fundamental operations in image processing that alter the pixel intensities of an image to achieve desired enhancements. These transformations are typically applied to grayscale images, where each pixel represents the brightness level at that location.

1. Negation Transformation:

The negation transformation is a simple operation that involves inverting the pixel intensities of an image. In a grayscale image, each pixel's intensity value is subtracted from the maximum intensity value (e.g., 255 for 8-bit images). This process effectively creates a photographic negative, where bright regions become dark, and vice versa.

2. Log Transformation:

The log transformation is a non-linear operation that enhances the contrast of images with low-intensity values while compressing the high-intensity values. It is particularly useful for expanding the dynamic range of images captured under low-light conditions. The formula for the log transformation is:

$$s = c * \log(1 + r)$$

Where,

s is the output pixel intensity.
r is the input pixel intensity.
c is a constant for scaling.

By taking the logarithm of the pixel intensity values, low-intensity regions are stretched, while high-intensity regions are compressed, resulting in an image with improved contrast.

3. Power Law Transformation:

The power law transformation, also known as gamma correction, adjusts the brightness and contrast of an image by raising each pixel intensity to a power γ. This operation is expressed as:

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

Where,

s is the output pixel intensity.
r is the input pixel intensity.
c is a constant for scaling.
γ is the gamma value.

By varying the gamma value, different levels of contrast enhancement can be achieved. A gamma value less than 1 will brighten the image, while a gamma value greater than 1 will darken it.

4. Piece-wise Linear Transformation:

The piece-wise linear transformation allows for more fine-grained control over the enhancement process by applying different linear functions to different intensity ranges of the image. This technique involves defining multiple linear functions, each with its own slope and intercept parameters, and specifying the intensity ranges over which each function should be applied.

By dividing the intensity range of the image into segments and applying different linear functions to each segment, specific regions of the image can be enhanced or suppressed as needed. Piece-wise linear transformations are commonly used for contrast adjustments and histogram equalization.

Python Source Code:

Negation:

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

# Load the image
image = cv2.imread('download.jpg', cv2.IMREAD_GRAYSCALE)

# Negation transformation
negation_image = 255 - image

# Plotting
plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.imshow(image, cmap='gray')
plt.title('Original Image')
plt.subplot(1, 2, 2)
plt.imshow(negation_image, cmap='gray')
plt.title('Negation Transformation')
plt.show()
```

Log Transformation:

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

# Load the image
image = cv2.imread('original_image.jpg', cv2.IMREAD_GRAYSCALE)

# Log transformation
c = 255 / np.log(1 + np.max(image))
log_image = c * (np.log(image + 1))

# Normalize the image
log_image = np.uint8(log_image)

# Plotting
plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.imshow(image, cmap='gray')
plt.title('Original Image')
plt.subplot(1, 2, 2)
plt.imshow(log_image, cmap='gray')
plt.title('Log Transformation')
plt.show()
```

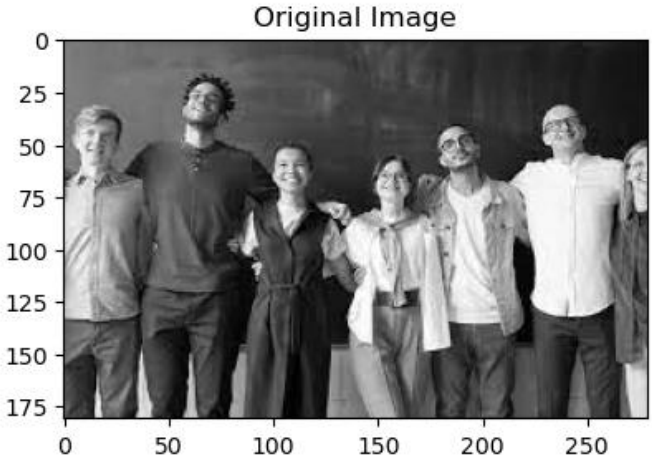
Power Law Transformation:

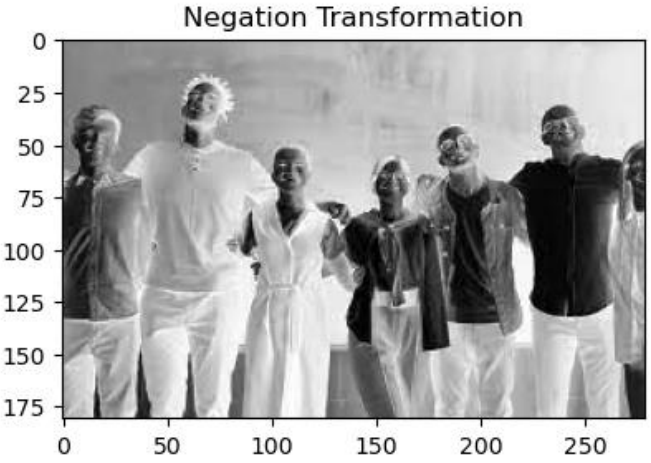
```
import cv2
import numpy as np
import matplotlib.pyplot as plt

# Load the image
image = cv2.imread('original_image.jpg',
cv2.IMREAD_GRAYSCALE)

# Power law transformation
gamma = 1.5
power_law_image = np.power(image, gamma)

# Normalize the image
power_law_image = np.uint8(power_law_image)

# Plotting
plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.imshow(image, cmap='gray')
plt.title('Original Image')
plt.subplot(1, 2, 2)
plt.imshow(power_law_image, cmap='gray')
plt.title('Power Law Transformation')
plt.show()
```

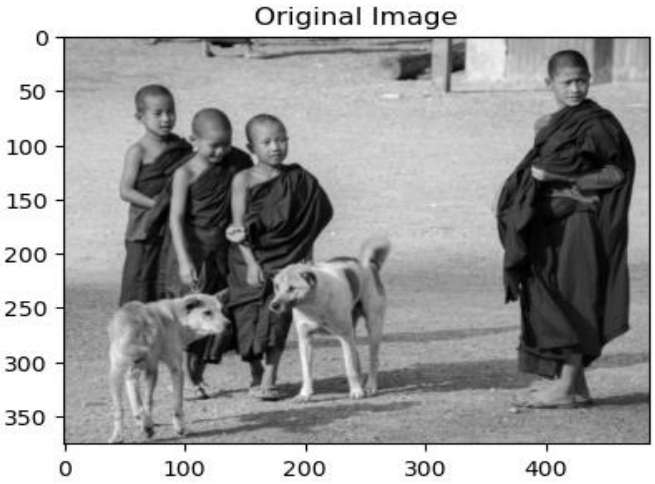
Piece-wise Linear Transformation:

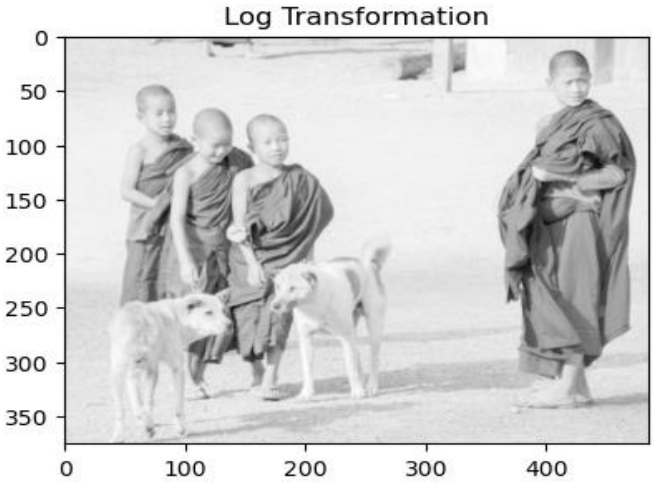
```
import cv2
import numpy as np
import matplotlib.pyplot as plt

# Load the image
image = cv2.imread('original_image.jpg',
cv2.IMREAD_GRAYSCALE)

# Piece-wise linear transformation
piecewise_image = np.zeros_like(image)
piecewise_image = cv2.addWeighted(image, 0.5,
piecewise_image, 0, 0)
piecewise_image = cv2.addWeighted(image, 1.5,
piecewise_image, 0, 0)

# Plotting
plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.imshow(image, cmap='gray')
plt.title('Original Image')
plt.subplot(1, 2, 2)
plt.imshow(piecewise_image, cmap='gray')
plt.title('Piece-wise Linear Transformation')
plt.show()
```

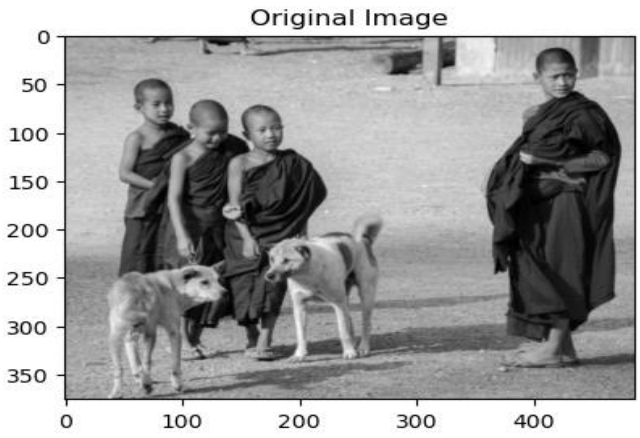
Input and Output:

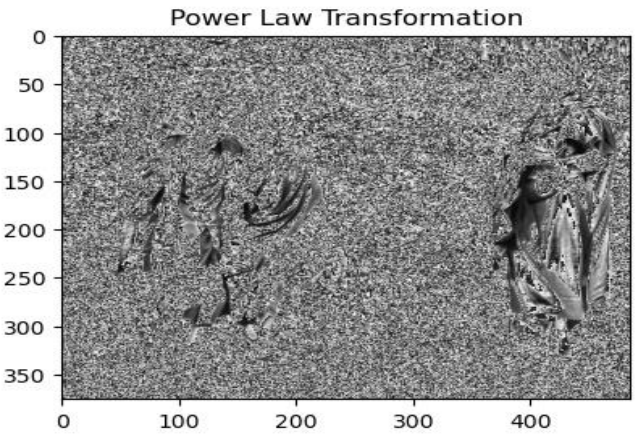


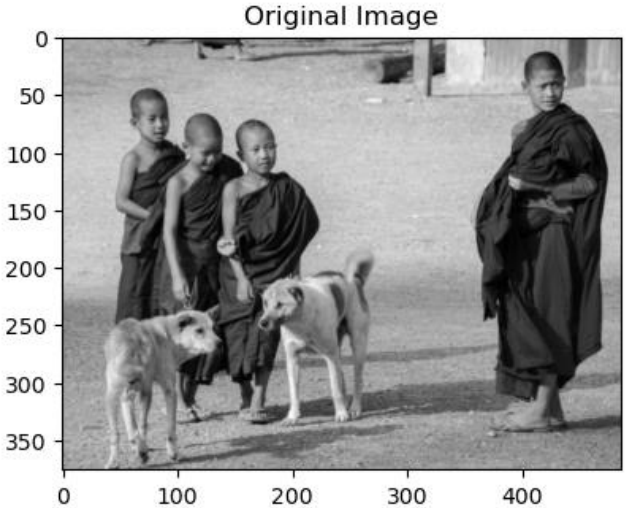


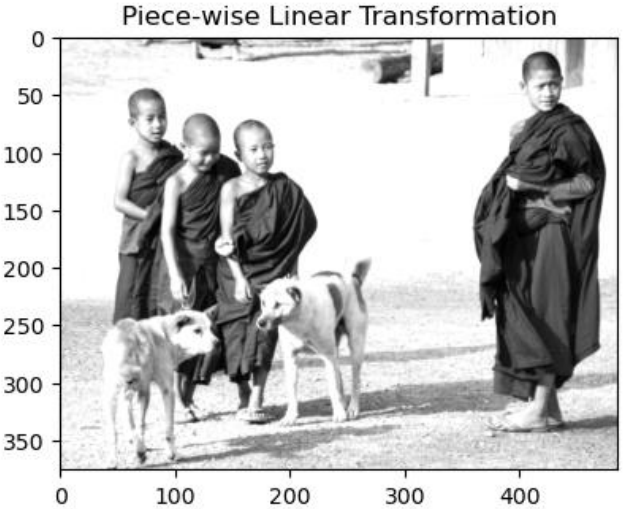












Convolution:
Convolution is a fundamental operation in image processing that involves applying a kernel (also known as a filter or mask) to an image to produce a transformed output. This operation is widely used for various tasks, including smoothing, sharpening, edge detection, and noise reduction.

1. High Pass Filter:
High pass filters are designed to enhance the high-frequency components of an image while suppressing the low-frequency components. They are commonly used for sharpening images by accentuating edges and details. A high pass filter kernel typically has positive values in the center surrounded by negative values, such as:

$$\begin{matrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{matrix}$$

When convolving this kernel with an image, the resulting output will emphasize the differences between adjacent pixel intensities, enhancing the edges and fine details.

2. Low Pass Filter:
Low pass filters, on the other hand, are used to smooth images by attenuating high-frequency components while preserving the low-frequency components. They are effective in reducing noise and blurring images. A common example of a low pass filter is the Gaussian filter, which uses a Gaussian distribution as the kernel. The Gaussian kernel emphasizes pixels closer to the center and gradually attenuates the influence of pixels farther away, resulting in a smooth, blurred effect.

3. Laplacian Filter:
The Laplacian filter is a second-order derivative filter used for edge detection and feature extraction. It highlights regions of rapid intensity change in an image, typically indicating the presence of edges or boundaries. The Laplacian filter kernel is computed as the sum of the second derivatives of the image intensity with respect to spatial coordinates. A commonly used Laplacian kernel is:

$$\begin{matrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{matrix}$$

When convolving this kernel with an image, it detects areas of rapid intensity change, resulting in enhanced edges.

Procedure for Convolution:

- Define a kernel or filter matrix that specifies the weights to be applied to neighboring pixels.
- Place the center of the kernel at each pixel location in the image.
- Multiply the kernel values by the corresponding pixel values in the image.
- Sum up the products to obtain the new pixel value for that location.
- Repeat the process for every pixel in the image to obtain the convolved output.

Python Source Code:

High Pass Filter (Sharpening):

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

# Load the image
image = cv2.imread('original_image.jpg',
cv2.IMREAD_GRAYSCALE)

# Define the high pass filter kernel
kernel = np.array([[0, -1, 0],
                   [-1, 5, -1],
                   [0, -1, 0]])

# Apply the high pass filter
high_pass_image = cv2.filter2D(image, -1, kernel)

# Plotting
plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.imshow(image, cmap='gray')
plt.title('Original Image')
plt.subplot(1, 2, 2)
plt.imshow(high_pass_image, cmap='gray')
plt.title('High Pass Filter (Sharpening)')
plt.show()
```

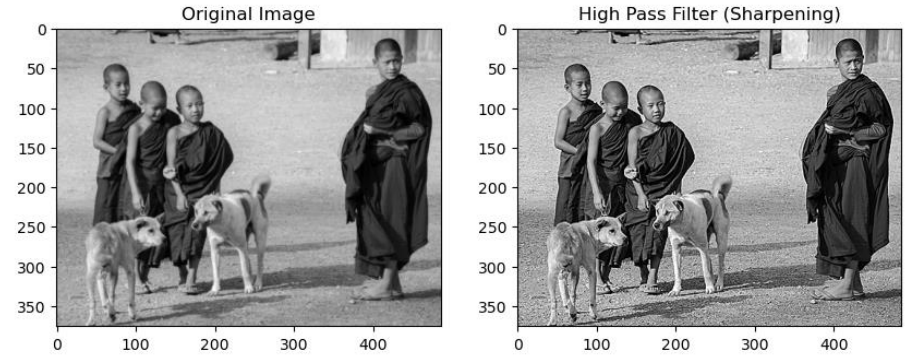
Low Pass Filter (Smoothing):

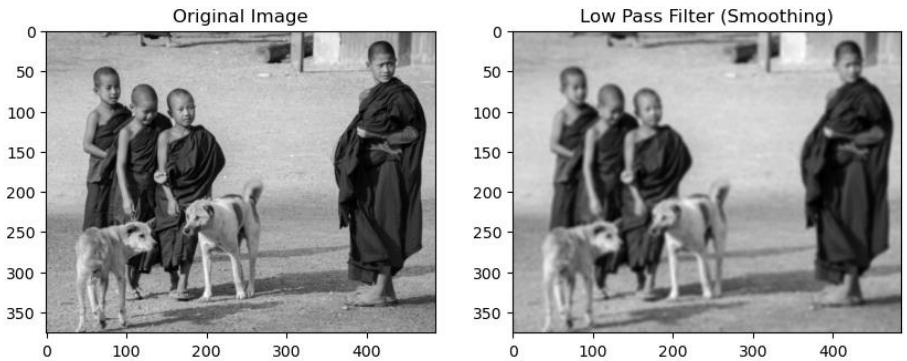
```
import cv2
import numpy as np
import matplotlib.pyplot as plt

# Load the image
image = cv2.imread('original_image.jpg',
cv2.IMREAD_GRAYSCALE)

# Define the low pass filter kernel
kernel = np.ones((5, 5), np.float32) / 25

# Apply the low pass filter
low_pass_image = cv2.filter2D(image, -1, kernel)

# Plotting
plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.imshow(image, cmap='gray')
plt.title('Original Image')
plt.subplot(1, 2, 2)
plt.imshow(low_pass_image, cmap='gray')
plt.title('Low Pass Filter (Smoothing)')
plt.show()
```

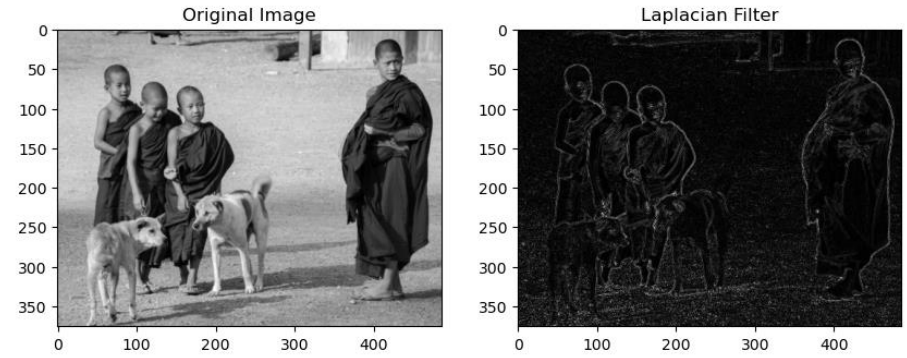
Laplacian Filter:

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

# Load the image
image = cv2.imread('original_image.jpg',
cv2.IMREAD_GRAYSCALE)

# Apply the Laplacian filter
laplacian_image = cv2.Laplacian(image,
cv2.CV_64F)

# Normalize the image
laplacian_image =
cv2.convertScaleAbs(laplacian_image)

# Plotting
plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.imshow(image, cmap='gray')
plt.title('Original Image')
plt.subplot(1, 2, 2)
plt.imshow(laplacian_image, cmap='gray')
plt.title('Laplacian Filter')
plt.show()
```

Input and Output:







Experiment Name: Write a program to execute the following edge detections

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

Theory:

1. Canny Edge Detection:

Canny edge detection is a multi-stage algorithm used to detect a wide range of edges in images. The key stages involved in Canny edge detection are:

Noise Reduction: The image is smoothed using a Gaussian filter to reduce noise and artifacts.

Gradient Calculation: The gradient magnitude and direction are calculated using Sobel operators to identify regions of rapid intensity change.

Non-maximum Suppression: Only local maxima in the gradient direction are retained as potential edge pixels.

Edge Tracking by Hysteresis: Final edge pixels are determined by thresholding and connectivity analysis, ensuring only strong edge pixels are selected.

2. Prewitt Edge Detection:

Prewitt edge detection is a gradient-based method that computes the gradient approximation in both the horizontal and vertical directions. The Prewitt operator kernels are applied to the image to compute the gradient magnitudes. Afterward, the magnitude values are thresholded to detect edges.

3. Sobel Edge Detection:

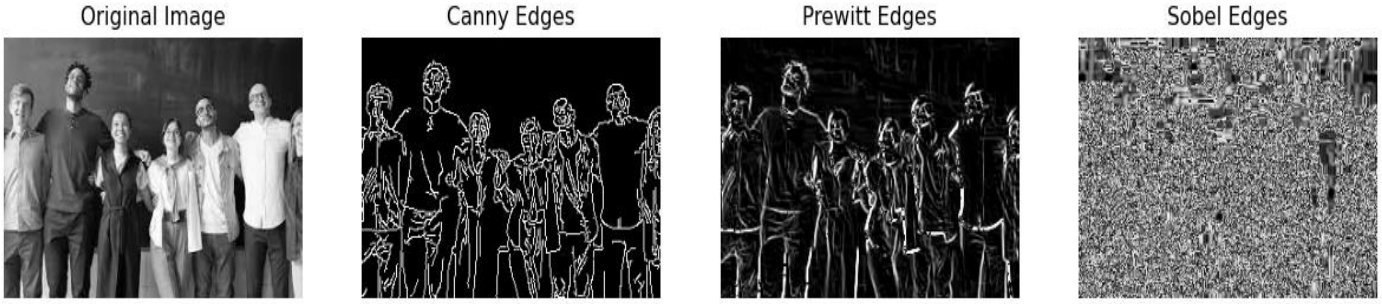
Sobel edge detection is another gradient-based method used to detect edges in images. It computes the gradient approximation using convolution with Sobel operator kernels in both horizontal and vertical directions. The gradient magnitudes are then calculated, and thresholding is applied to identify edges.

Python Source Code:

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

# Load the image
image = cv2.imread('download.jpg', cv2.IMREAD_GRAYSCALE)

# Canny edge detection
canny_edges = cv2.Canny(image, 100, 200)

# Prewitt edge detection
kernelx = np.array([[1, 1, 1], [0, 0, 0], [-1, -1, -1]])
kernely = np.array([[-1, 0, 1], [-1, 0, 1], [-1, 0, 1]])
prewitt_edges_x = cv2.filter2D(image, -1, kernelx)
prewitt_edges_y = cv2.filter2D(image, -1, kernely)
prewitt_edges = cv2.magnitude(prewitt_edges_x.astype(np.float64), prewitt_edges_y.astype(np.float64))

# Sobel edge detection
sobel_edges_x = cv2.Sobel(image, cv2.CV_64F, 1, 0, ksize=5)
sobel_edges_y = cv2.Sobel(image, cv2.CV_64F, 0, 1, ksize=5)
```

```
sobel_edges = cv2.magnitude(sobel_edges_x, sobel_edges_y)

# Plotting
plt.figure(figsize=(15, 5))

plt.subplot(1, 4, 1)
plt.imshow(image, cmap='gray')
plt.title('Original Image')
plt.axis('off')

plt.subplot(1, 4, 2)
plt.imshow(canny_edges, cmap='gray')
plt.title('Canny Edges')
plt.axis('off')

plt.subplot(1, 4, 3)
plt.imshow(prewitt_edges.astype(np.uint8), cmap='gray')
plt.title('Prewitt Edges')
plt.axis('off')

plt.subplot(1, 4, 4)
plt.imshow(sobel_edges.astype(np.uint8), cmap='gray')
plt.title('Sobel Edges')
plt.axis('off')

plt.show()
```

Input and Output:



Experiment 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.

Theory:

Speech Preprocessing:

Speech preprocessing is an essential step in speech signal analysis and processing, which involves various techniques to enhance the quality and usability of speech signals. Some common preprocessing tasks include identifying sampling frequency, bit resolution, and downsampling the signal.

1. Identifying Sampling Frequency:

Sampling frequency, also known as the sampling rate, refers to the number of samples per second obtained from a continuous signal to produce a discrete signal. In digital audio, the sampling frequency determines the fidelity of the reconstructed signal. The Nyquist theorem states that the sampling frequency must be at least twice the highest frequency present in the signal to avoid aliasing. Identifying the sampling frequency of a speech signal is crucial for further processing and analysis.

2. Identifying Bit Resolution:

Bit resolution, also known as bit depth, refers to the number of bits used to represent each sample of a digital audio signal. It determines the dynamic range and precision of the signal representation. Common bit resolutions include 8-bit, 16-bit, and 24-bit. Higher bit resolutions result in better signal fidelity and less quantization noise. Identifying the bit resolution of a speech signal helps in understanding its quality and fidelity.

3. Downsampling the Signal:

Downsampling, also known as decimation, is the process of reducing the sampling rate of a signal by removing samples. It is often performed to reduce computational complexity, memory requirements, or to adapt the signal to a specific system or application. Downsampling can be achieved by simply discarding samples or by using anti-aliasing filters to remove high-frequency components before downsampling. Downsampling the speech signal may be necessary to meet system requirements or to reduce file size while preserving essential information.

MATLAB Source Code:

```
% Load the speech signal
[speech_signal, sampling_frequency] = audioread('sample.wav');

% Identify sampling frequency
fprintf('Sampling Frequency: %d Hz\n', sampling_frequency);

% Identify bit resolution
info = audioinfo('sample.wav');
bit_resolution = info.BitsPerSample;
fprintf('Bit Resolution: %d bits\n', bit_resolution);

% Downsampling the frequency
downsampled_signal = downsample(speech_signal, 2); % Downsampling by a factor of 2

% Save the downsampled signal
downsampled_sampling_frequency = sampling_frequency / 2;
audiowrite('downsampled_signal.wav', downsampled_signal, downsampled_sampling_frequency);

disp('Downsampled signal saved as downsampled_signal.wav');
```

Input and Output:
Input: sample.wav
Output:
Sampling Frequency: 44100 Hz
Bit Resolution: 16 bits
Downsampled signal saved as downsampled_signal.wav

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

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

Theory:

1. Voiced Region:

The voiced region in a speech signal corresponds to segments where the vocal cords vibrate, producing periodic waveforms with clear fundamental frequencies. These regions typically contain voiced speech sounds such as vowels and voiced consonants. Voiced speech signals exhibit regular patterns and often have harmonically related frequency components.

To identify the voiced region in a speech signal, various techniques can be employed, such as pitch detection algorithms. These algorithms analyze the periodicity of the signal to estimate the fundamental frequency (pitch) of the voiced segments. Once the fundamental frequency is estimated, regions with consistent pitch values are classified as voiced regions.

2. Unvoiced Region:

The unvoiced region in a speech signal corresponds to segments where the vocal cords do not vibrate, resulting in a turbulent airflow through the vocal tract. These regions typically contain unvoiced speech sounds such as fricatives, plosives, and aspirates. Unvoiced speech signals exhibit random or noisy patterns and lack a clear fundamental frequency.

To identify the unvoiced region in a speech signal, techniques such as energy-based analysis or zero-crossing rate analysis can be used. Unvoiced segments often have high energy levels and a high rate of zero-crossings due to the turbulent nature of the airflow. By analyzing these characteristics, unvoiced regions can be detected.

3. Silence Region:

The silence region in a speech signal corresponds to segments where there is no significant speech activity or background noise dominates the signal. These regions typically contain no discernible speech sounds and are characterized by low energy levels and minimal variation in amplitude.

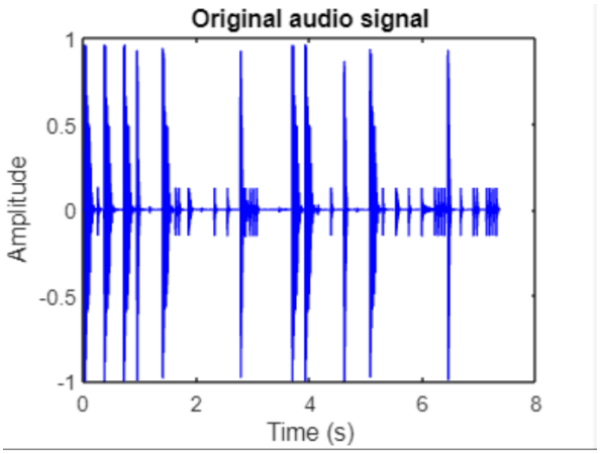
To identify the silence region in a speech signal, energy-based thresholding techniques are commonly used. By setting a threshold on the energy level of the signal, segments with energy levels below the threshold can be classified as silence regions.

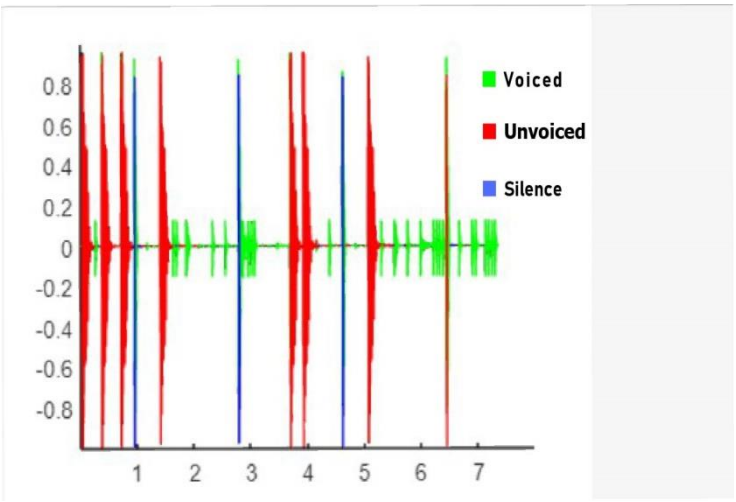
MATLAB Source Code:

```
% Clear the workspace and command window
clc; clear all; close all;

% Read the audio file
[y, fs] = audioread('rashed  .wav');

% Define frame size and overlap (in
samples)
frame_size = 256;
overlap = 128;

% Calculate number of frames
num_frames = floor(length(y) / (frame_size
- overlap));

% Initialize variables
voiced_frames = [];
unvoiced_frames = [];
silence_frames = [];

% Iterate through each frame
for i = 1:num_frames
  % Extract current frame (corrected
indexing)
  frame = y((i-1)*(frame_size-overlap) + 1
: (i-1)*(frame_size-overlap) + frame_size);

  % Calculate energy of the frame
  energy = sum(abs(frame).^2);

  % Calculate zero-crossing rate (ZCR)
  zcr = sum(diff(sign(frame)) ~= 0);

  % Thresholds for voiced, unvoiced, and
silence detection
  voiced_threshold = 0.01 * max(energy); %
adjust threshold based on your audio
  unvoiced_threshold = 0.001 * max(energy);
% adjust threshold based on your audio
  silence_threshold = 0.0001 * max(energy);
% adjust threshold based on your audio

  % Identify frame type based on energy and
ZCR
  if energy > voiced_threshold && zcr > 10
% adjust values for voiced detection
    voiced_frames = [voiced_frames; i];
  elseif energy > unvoiced_threshold && zcr
< 10 % adjust values for unvoiced detection
    unvoiced_frames = [unvoiced_frames; i];
  else
    silence_frames = [silence_frames; i];
  end
end
```

```
% Calculate time axis for plotting
time_axis = (0:length(y)-1) / fs;

% Plot on a single subplot
figure;
hold on;


% Plot original signal
plot(time_axis, y, 'k'); % black for original
signal

% Plot voiced segments
for i = 1:length(voiced_frames)
    start_idx = (voiced_frames(i)-1) * (frame_size
- overlap) + 1;
    end_idx = start_idx + frame_size - 1;
    plot(time_axis(start_idx:end_idx),
y(start_idx:end_idx), 'g');
end

% Plot unvoiced segments
for i = 1:length(unvoiced_frames)
    start_idx = (unvoiced_frames(i)-1) *
(frame_size - overlap) + 1;
    end_idx = start_idx + frame_size - 1;
    plot(time_axis(start_idx:end_idx),
y(start_idx:end_idx), 'r');
end

% Plot silence segments
for i = 1:length(silence_frames)
    start_idx = (silence_frames(i)-1) *
(frame_size - overlap) + 1;
    end_idx = start_idx + frame_size - 1;
    plot(time_axis(start_idx:end_idx),
y(start_idx:end_idx), 'b');
end

hold off;

% Adjust plot layout and title
xlabel('Time (s)');
ylabel('Amplitude');
title('Voiced, Unvoiced, and Silence Detection');
legend('Original Signal', 'Voiced', 'Unvoiced',
'Silence');
```

Input and Output:





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

Theory:

Zero Crossing Rate (ZCR):

The Zero Crossing Rate (ZCR) is a measure of how often a signal crosses the zero axis. It is commonly used in speech and audio processing to characterize the temporal variations in the signal. The ZCR is calculated by counting the number of times the signal changes its sign within a certain time window.

Mathematically, the ZCR can be expressed as:

$$\text{ZCR}=\frac{1}{2\text{T}}\sum_{n=1}^{N}|sgn(x[n])-sgn(x[n-1])|$$

Where,

- x(n) is the speech signal at sample n.
- sgn(x[n]) is the sign function , which returns -1 for negative values , 0 for zero and 1 for positive values
- N is the total number of sample in the signal.
- T is the duration of the analysis window.

MATLAB Source Code:

```
% Load the speech signal
[speech_signal, sampling_frequency] = audioread('sample.wav');

% Define window size and overlap
window_size = 0.02 * sampling_frequency;  % 20 ms window size
overlap = 0.5 * window_size;              % 50% overlap

% Compute the zero crossing rate (ZCR) using different window functions
window_functions = {@rectwin, @hamming, @hann, @blackman};
num_window_functions = length(window_functions);
zcr_values = zeros(num_window_functions, 1);

% Compute ZCR for each window function
for i = 1:num_window_functions
    % Apply the current window function
    window = window_functions{i}(window_size);

    % Compute the ZCR for the entire signal using the current window function
    zcr_values(i) = zerocrossingrate(speech_signal, window, overlap);
end

% Display ZCR values
disp('Zero Crossing Rate (ZCR) using different window functions:');
for i = 1:num_window_functions
    fprintf('%s window function: %.2f\n', func2str(window_functions{i}), zcr_values(i));
end

% Function to compute zero crossing rate (ZCR) for the entire signal
function avg_zcr = zerocrossingrate(signal, window, overlap)
    frame_length = length(window);
    step_size = frame_length - overlap;
    num_frames = floor((length(signal) - overlap) / step_size);
    padded_signal_length = frame_length + (num_frames - 1) * step_size;
    padded_signal = [signal; zeros(padded_signal_length - length(signal), 1)];
    zcr = zeros(num_frames, 1);

    for i = 1:num_frames
        start_index = (i - 1) * step_size + 1;
        end_index = start_index + frame_length - 1;
        frame = padded_signal(start_index:end_index);
        zcr(i) = sum(abs(diff(frame > 0))) / (2 * frame_length);
    end

    % Compute average ZCR across all frames
    avg_zcr = mean(zcr);
end
```

Input: `sample.wav`

Output:

```
Zero Crossing Rate (ZCR) using different window functions:
rectwin window function: 0.03
hamming window function: 0.03
hann window function: 0.03
blackman window function: 0.03
```

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

Theory:

Autocorrelation:

Autocorrelation is a mathematical tool used to measure the similarity between a signal and a time-shifted version of itself. In the context of speech processing, autocorrelation is often used to analyze the periodicity and pitch of speech signals.

Short-Term Autocorrelation:
Short-term autocorrelation involves computing the autocorrelation function for short frames or segments of a signal. This allows us to analyze the temporal characteristics of the signal within localized time windows. Short-term autocorrelation is useful for detecting short-term variations and periodicities in the signal, which can provide insights into speech production mechanisms and characteristics.

Computation:
To compute short-term autocorrelation, we typically divide the speech signal into frames of fixed duration, often overlapping, and calculate the autocorrelation function for each frame. This results in a matrix where each row corresponds to a frame, and each column corresponds to a lag value in the autocorrelation function.

Procedure:
Load the Speech Signal: Begin by loading the speech signal from a WAV file using suitable MATLAB functions like audioread.

Frame Segmentation: Divide the speech signal into short frames of fixed duration, typically ranging from 20 to 30 milliseconds. Ensure overlap between consecutive frames for smooth processing.

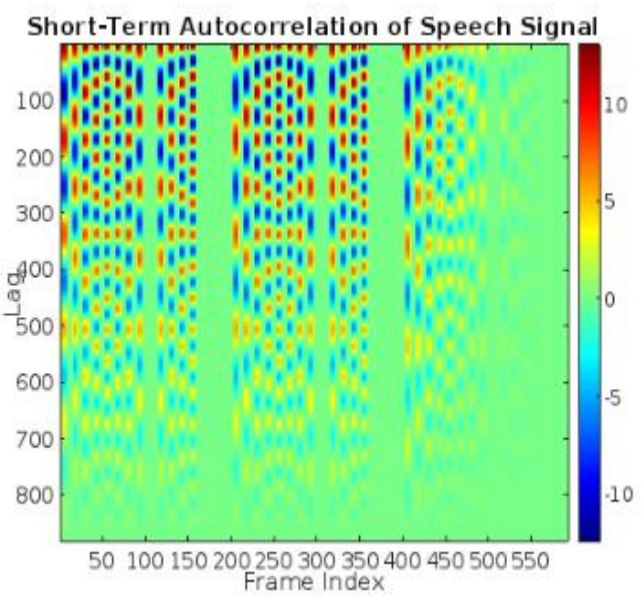
Autocorrelation Calculation: Compute the autocorrelation function for each frame using the xcorr function in MATLAB. Extract the positive lags from the autocorrelation vector, as they represent meaningful time delays.

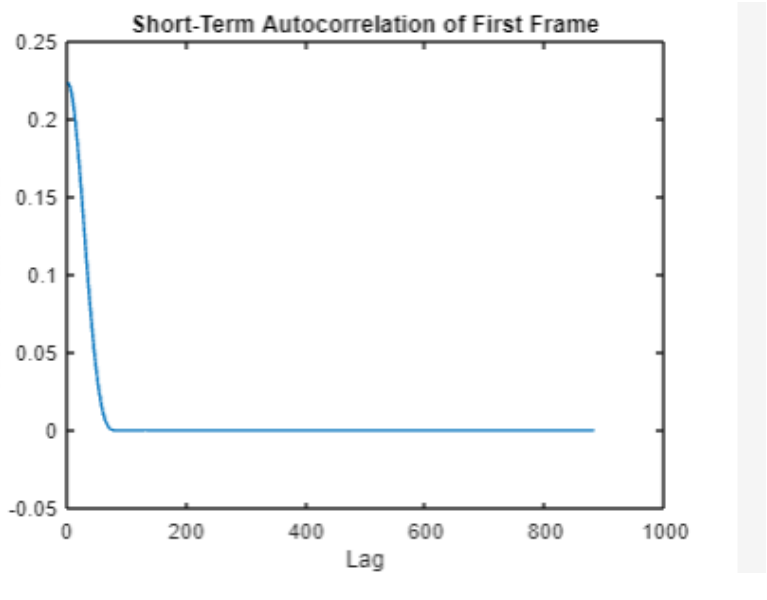
Visualization: Visualize the short-term autocorrelation matrix using an appropriate plot. Use imagesc function to create a heatmap-style plot, where the x-axis represents frame index, the y-axis represents lag, and the color intensity represents the autocorrelation value.

MATLAB Source Code:

```
% Load the speech signal
[speech_signal, sampling_frequency] = audioread('sample.wav');

% Define parameters
frame_length = 0.02 * sampling_frequency; % Frame length (20 ms)
overlap = 0.5 * frame_length;              % Overlap (50%)
num_frames = floor((length(speech_signal) - overlap) / (frame_length - overlap));
max_lag = frame_length - 1;
short_term_autocorr = zeros(num_frames, max_lag + 1);

% Compute short-term autocorrelation
for i = 1:num_frames
    start_index = round((i - 1) * (frame_length - overlap) + 1);
    end_index = start_index + frame_length - 1;
    frame = speech_signal(start_index:end_index);
    autocorr_values = xcorr(frame);
    short_term_autocorr(i, :) = autocorr_values(max_lag + 1:2*max_lag + 1); % Store only positive lags
end

% Plot short-term autocorrelation
figure;

imagesc(short_term_autocorr');
colormap('jet');
colorbar;
xlabel('Frame Index');
ylabel('Lag');
title('Short-Term Autocorrelation of Speech Signal');

% Display short-term autocorrelation values
figure;
plot(short_term_autocorr(1,:)); % Plot the autocorrelation values of the first frame
xlabel('Lag');
ylabel('Autocorrelation Value');
title('Short-Term Autocorrelation of First Frame');
```

Output:





Experiment Name: Write a program to estimate pitch of a speech signal.

Theory:

Pitch Estimation:

Pitch estimation is the process of determining the fundamental frequency (F0) of a speech signal, which corresponds to the perceived pitch of the voice. Pitch is a crucial perceptual attribute of speech that conveys information about the speaker's gender, emotion, and linguistic content.

Fundamental Frequency (F0):

The fundamental frequency represents the rate at which the vocal folds vibrate during phonation. In speech processing, it is often measured in Hertz (Hz) and corresponds to the perceived pitch of the voice. In voiced speech segments, the fundamental frequency corresponds to the repetition rate of glottal pulses.

Pitch Estimation Techniques:

Several techniques exist for estimating pitch from a speech signal. Common methods include:

Autocorrelation Method: This method calculates the autocorrelation function of the speech signal and identifies peaks corresponding to the fundamental frequency.

Zero Crossing Rate (ZCR): The ZCR measures the rate at which the speech signal changes its sign. In voiced speech segments, the ZCR exhibits periodic variations corresponding to the fundamental frequency.

Harmonic Product Spectrum (HPS): HPS computes the product of the spectra of the original signal and its downsampled versions, emphasizing harmonics and facilitating pitch estimation.

Procedure:

Preprocessing: Preprocess the speech signal by filtering out noise, normalizing the

amplitude, and segmenting it into short frames.
Feature Extraction: Extract relevant features from the speech signal, such as autocorrelation, zero crossing rate, or spectral characteristics.

Pitch Estimation: Apply the chosen pitch estimation technique to the extracted features to estimate the fundamental frequency (pitch) of the speech signal.
Visualization: Visualize the estimated pitch contour over time to observe variations in pitch.
Evaluation: Evaluate the accuracy and reliability of the pitch estimation technique by comparing the estimated pitch with ground truth annotations or subjective perceptual judgments.

MATLAB Source Code:

```
% Load the speech signal using audioread (wavread is deprecated)
[signal, fs] = audioread('sample.wav');

% Parameters for pitch estimation
window_size = round(0.03 * fs); % Window size (30 milliseconds)
overlap = round(0.015 * fs); % Overlap (15 milliseconds)

% Calculate autocorrelation
autocorr = xcorr(signal);

% Define a threshold to identify peaks in autocorrelation
threshold = 0.2; % You may need to adjust this threshold depending on your signal

% Find peaks in autocorrelation
[~, locs] = findpeaks(autocorr(window_size:end), 'MinPeakHeight', threshold);

% Calculate pitch (fundamental frequency) based on peak locations
pitch_values = fs ./ locs;

% Take median of pitch values to account for inaccuracies
estimated_pitch = median(pitch_values);

fprintf('Estimated fundamental frequency (pitch): %.2f Hz\n', estimated_pitch);
```

Output:

```
Estimated fundamental frequency (pitch): 0.04 Hz
```