**Experiment No: 01**

**Name of the Experiment:** 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:**

1. Develop a program to read images from a designated folder.
2. Implement image resizing functionality to adjust images to specified dimensions and save them to an output folder.
3. Apply color transformations to enhance image quality and save transformed images into a separate folder.
4. Normalize images to ensure consistency across the dataset and save normalized images into an output folder.
5. Implement image filtering techniques to remove noise or enhance features, saving filtered images into a designated 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.

**Source Code**:

import os

import cv2

import numpy as np

def resize\_images(input\_folder, output\_folder, target\_size=(300, 300)):

os.makedirs(output\_folder, exist\_ok=True)

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)

print("resized image and save a folder done!")

return output\_folder

def apply\_color\_transform(input\_folder, output\_folder):

os.makedirs(output\_folder, exist\_ok=True)

for filename in os.listdir(input\_folder):

img\_path = os.path.join(input\_folder, filename)

img = cv2.imread(img\_path)

# Example color transformation (convert to grayscale)

transformed\_img = cv2.cvtColor(img, cv2.COLOR\_BGR2GRAY)

output\_path = os.path.join(output\_folder, filename)

cv2.imwrite(output\_path, transformed\_img)

print("apply color\_transform and save a folder done!")

def normalize\_images(input\_folder, output\_folder):

os.makedirs(output\_folder, exist\_ok=True)

for filename in os.listdir(input\_folder):

img\_path = os.path.join(input\_folder, filename)

img = cv2.imread(img\_path).astype(np.float32)

# Example normalization (scaling pixel values to [0, 1])

normalized\_img = img / 255.0

output\_path = os.path.join(output\_folder, filename)

cv2.imwrite(output\_path, normalized\_img \* 255.0)

print("normalize images and save a folder done!")

def filter\_images(input\_folder, output\_folder):

os.makedirs(output\_folder, exist\_ok=True)

for filename in os.listdir(input\_folder):

img\_path = os.path.join(input\_folder, filename)

img = cv2.imread(img\_path)

# Example filtering (Gaussian blur)

filtered\_img = cv2.GaussianBlur(img, (5, 5), 0)

output\_path = os.path.join(output\_folder, filename)

cv2.imwrite(output\_path, filtered\_img)

print("apply filter and save a folder done!")

if \_\_name\_\_ == "\_\_main\_\_":

input\_folder = "cat"

output\_folder\_resized = "cat\_resized"

output\_folder\_transformed = "cat\_resized\_color\_transformed"

output\_folder\_normalized = "cat\_resized\_normalized"

output\_folder\_filtered = "cat\_resized\_filtered"

resized\_folder = resize\_images(input\_folder, output\_folder\_resized)

apply\_color\_transform(resized\_folder, output\_folder\_transformed)

normalize\_images(resized\_folder, output\_folder\_normalized)

filter\_images(resized\_folder, output\_folder\_filtered)

**Output:**

resized image and save a folder done!

apply color\_transform and save a folder done!

normalize images and save a folder done!

apply filter and save a folder done!

**Experiment No: 02**

**Name of the Experiment:** Write a program to execute Semantic segmentation.

**Objective:**

**i)** Develop a program to perform semantic segmentation on digital images.

**ii)**Aim to assign semantic labels to each pixel, effectively segmenting the image into distinct object categories.

**iii)**Utilize deep learning techniques, particularly convolutional neural networks (CNNs), to achieve accurate pixel-level classification of objects within images.

**Theory:**   
Semantic segmentation is a computer vision task that involves assigning semantic labels to each pixel in an image, thus dividing the image into semantically meaningful regions. Unlike classification, which assigns a single label to the entire image, semantic segmentation provides a detailed understanding of the scene by segmenting it into different object classes or categories.

Semantic segmentation models typically employ deep learning architectures, such as convolutional neural networks (CNNs), often with encoder-decoder structures or advanced architectures like U-Net or DeepLab. These models leverage large datasets with pixel-level annotations to learn the complex mapping between input images and their corresponding segmentation maps.

**Source Code**:

import cv2

import numpy as np

import matplotlib.pyplot as plt

# Read the input image

input\_image = cv2.imread("cat-7.jpg")

# 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()

**Experiment No: 03**

**Name of the Experiment:** 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.

**Objective:**

**i)** Develop a program to extract the region of interest (ROI) from an image using a given mask and compute its area.

**ii)**Implement functionality to overlay the mask onto the image, effectively labeling the region.

**iii)**The objective is to facilitate efficient region-based analysis and labeling, enabling tasks such as object detection, segmentation, and image annotation.

**Theory:**   
The task involves analyzing an image based on a provided mask, aiming to determine the region of interest, compute its area, and visually label it on the original image. the following steps can be followed for this:

1. **Image and Mask Loading**: Load the original image and the corresponding mask into memory. Ensure that the mask is binary, where pixels belonging to the region of interest are represented as white (255) and the background as black (0).
2. **Region Extraction**: Use the mask to extract the region of interest from the original image.
3. **Area Computation**: Calculate the area of the extracted region.
4. **Labeling**: Overlay the mask onto the original image to visually label the region of interest.
5. **Visualization**: Display the original image with the overlaid mask to visualize the labeled region.

**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()

**Experiment No: 04**

**Name of the Experiment:** 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).

**Objective:**

1. Develop a program for image enhancement, incorporating basic intensity transformations (negation, log transformation, power-law transformation, piece-wise linear transformation) to adjust contrast and brightness.
2. Implement convolution operations (high-pass, low-pass, Laplacian filters) to enhance image features such as edges and smoothness.
3. The program aims to provide users with a versatile tool for improving image quality and extracting relevant information through various enhancement techniques.

**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 \*rl

Where,

s is the output pixel intensity.

r is the input pixel intensity.

c is a constant for scaling.

l 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.

**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:

0 -1 0

-1 5 -1

0 -1 0

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:

0 1 0

1 -4 1

0 1 0

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.

**Source Code**:

import cv2

import numpy as np

import matplotlib.pyplot as plt

# Load the image

image = cv2.imread('cat-2.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

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

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

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()

# 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()

# 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()

# 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()

**Experiment No: 05**

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

* Canny edge detection
* Prewitt edge detection
* Sobel edge detection

**Objective:**

**i)** Develop a program to implement Canny, Prewitt, and Sobel edge detection algorithms to identify edges in digital images accurately.

**ii)** Aim to provide users with a versatile tool for detecting edges using different techniques, allowing for comparison and analysis of edge detection results.

**Theory:** Edge detection is a fundamental image processing technique aimed at identifying boundaries between objects in digital images.

**Canny Edge Detection:** Canny's edge detector ensures good noise immunity and at the same time detects true edge points with minimum error.

**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.

**Sobel Edge Detection:** Sobel edge detection computes the gradient approximation using convolution with Sobel masks.It applies weighted averaging to compute gradients, resulting in improved performance in detecting edges, especially in the presence of noise.

**Source Code**:

import cv2

import numpy as np

import matplotlib.pyplot as plt

# Load the image

image = cv2.imread('cat-7.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()

**Experiment No: 06**

**Name of the Experiment:** Write a program to execute the following speech preprocessing

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

**Objective:**

**i)** Develop a program to preprocess speech signals by identifying the sampling frequency and bit resolution to ensure compatibility with further processing.

**ii)** Implement down-sampling of the speech signal to reduce the sampling frequency while maintaining signal quality, facilitating efficient storage and processing.

**Theory:**

**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.

**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.

**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.

**Source 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 = "speech.wav"

# 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)

# 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)

# 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)

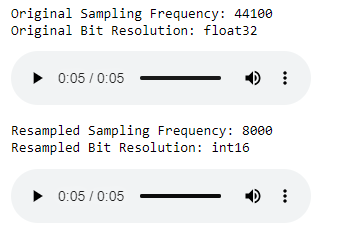
# Save the resampled signal with reduced bit resolution

wavfile.write('speech\_signal\_8000Hz\_16bit.wav', sr\_8000, signal\_resampled\_16bit)

# Play the resampled signal

display(Audio(signal\_resampled, rate=sr\_8000))

**Output:**



**Experiment No: 07**

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

• Voiced region.

• Unvoiced region.

• Silence region.

**Objective:**

**i)** Develop a program to segment speech signals into voiced, unvoiced, and silence regions for detailed analysis.

**ii)** Implement algorithms to accurately classify regions based on pitch, energy, and spectral characteristics.

**Theory:**

**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.

**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.

**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.

**Source Code**:

import numpy as np

import matplotlib.pyplot as plt

from matplotlib.patches import Patch

# Read the audio file

y, fs = librosa.load('speech.wav', sr=None)

# Define frame size and overlap (in samples)

frame\_size = 256

overlap = 128

# Calculate number of frames

num\_frames = int(np.floor(len(y) / (frame\_size - overlap)))

# 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((start\_idx, end\_idx))

elif energy > unvoiced\_threshold and zcr < 10: # adjust values for unvoiced detection

unvoiced\_frames.append((start\_idx, end\_idx))

else:

silence\_frames.append((start\_idx, end\_idx))

# Calculate time axis for plotting

time\_axis = np.arange(len(y)) / fs

# Plot original signal

plt.figure(figsize=(10, 5))

plt.plot(time\_axis, y, 'k') # black for original signal

# Plot voiced, unvoiced, and silence regions

for start\_idx, end\_idx in voiced\_frames:

plt.axvspan(start\_idx / fs, end\_idx / fs, color='green', alpha=0.3)

for start\_idx, end\_idx in unvoiced\_frames:

plt.axvspan(start\_idx / fs, end\_idx / fs, color='red', alpha=0.3)

for start\_idx, end\_idx in silence\_frames:

plt.axvspan(start\_idx / fs, end\_idx / fs, color='blue', alpha=0.3)

# Add legend with color names

legend\_elements = [

Patch(facecolor='green', alpha=0.3, label='Voiced'),

Patch(facecolor='red', alpha=0.3, label='Unvoiced'),

Patch(facecolor='blue', alpha=0.3, label='Silence')

]

plt.legend(handles=legend\_elements, loc='upper right')

plt.xlabel('Time (s)')

plt.ylabel('Amplitude')

plt.title('Original Signal with Voiced, Unvoiced, and Silence Regions')

plt.show()

**Experiment No: 08**

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

**Objective:**

1. Develop a program to compute the zero crossing rate (ZCR) of a speech signal using various window functions.
2. Compare ZCR calculations across different window functions to assess their impact on ZCR estimation accuracy.

**Theory:** ZCR is the rate at which a signal changes its sign, typically measured over a short-time window. To compute ZCR, the speech signal is divided into short-time frames using window functions like rectangular, Hamming, Hanning, or Blackman windows. Within each frame, the number of times the signal crosses the zero amplitude threshold is counted, providing insights into speech temporal characteristics. Different window functions affect ZCR computation due to their spectral properties, with Hamming, Hanning, and Blackman windows typically offering improved accuracy. Understanding these variations is essential for optimizing ZCR-based features in speech processing applications.

**Implementation Steps:**

1. Preprocessing: Remove noise and normalize the amplitude of the speech signal.
2. Frame Segmentation: Divide the signal into short overlapping frames.
3. Windowing: Apply different window functions (e.g., Hamming, Hanning, Rectangular) to each frame.
4. ZCR Calculation: Count the number of zero crossings within each windowed frame.
5. Analysis: Compare ZCR values obtained with different window functions to understand their effects on the ZCR computation and speech analysis.

**Source Code**:

import numpy as np

import scipy.io.wavfile as wavfile

from scipy.signal import get\_window

def zerocrossingrate(signal, window, overlap):

frame\_length = len(window)

step\_size = frame\_length - overlap

num\_frames = int(np.floor((len(signal) - overlap) / step\_size))

padded\_signal\_length = frame\_length + (num\_frames - 1) \* step\_size + overlap # Adjusted calculation

padded\_signal = np.pad(signal, (0, padded\_signal\_length - len(signal)), 'constant')

zcr = np.zeros(num\_frames)

for i in range(num\_frames):

start\_index = i \* step\_size

end\_index = start\_index + frame\_length

frame = padded\_signal[start\_index:end\_index]

zcr[i] = np.sum(np.abs(np.diff(frame > 0))) / (2 \* frame\_length)

avg\_zcr = np.mean(zcr)

return avg\_zcr

# Load the speech signal

sampling\_frequency, speech\_signal = wavfile.read('speech.wav')

# Define window size and overlap

window\_size = int(0.02 \* sampling\_frequency) # 20 ms window size

overlap = int(0.5 \* window\_size) # 50% overlap

# Compute the zero crossing rate (ZCR) using different window functions

window\_functions = [np.hamming, np.hanning, np.blackman]

num\_window\_functions = len(window\_functions)

zcr\_values = np.zeros(num\_window\_functions)

# Compute ZCR for each window function

for i, window\_func in enumerate(window\_functions):

# Apply the current window function

window = window\_func(window\_size)

# Compute the ZCR for the entire signal using the current window function

zcr\_values[i] = zerocrossingrate(speech\_signal, window, overlap)

# Display ZCR values

print('Zero Crossing Rate (ZCR) using different window functions:')

for i, window\_func in enumerate(window\_functions):

print(f'{window\_func.\_\_name\_\_} window function: {zcr\_values[i]:.2f}')

**Output:**

Zero Crossing Rate (ZCR) using different window functions:

hamming window function: 0.32

hanning window function: 0.32

blackman window function: 0.32

**Experiment No: 09**

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

**Objective:**

1. Develop a program to compute the short-term auto-correlation of a speech signal, revealing temporal relationships within short-time frames.
2. The objective is to analyze speech signal dynamics by calculating the auto-correlation function for each frame, aiding tasks like speech recognition and speaker identification.

**Theory:** Auto-correlation measures the similarity between a signal and its delayed version across different time lags. For a discrete signal *x*[*n*], the auto-correlation function *Rx*​[*k*] is defined as:

*Rx*​[*k*]=∑ ​*x*[*n*]⋅*x*[*n*−*k*]

where *k* represents the time lag, and *N* is the length of the signal. In the context of speech processing, this formula captures the similarity between the speech signal and its time-delayed version at various temporal offsets.

Short-term auto-correlation involves calculating the auto-correlation function within short segments or frames of the speech signal. This is essential for capturing the dynamic nature of speech, as speech signals exhibit non-stationary characteristics over short time intervals. By computing auto-correlation within short frames, we can analyze the temporal variations and periodicities present in different segments of the speech signal.

The process of computing short-term auto-correlation involves the following steps:

1. Segmentation: The speech signal is divided into short overlapping frames to capture local temporal variations.
2. Windowing: Each frame is multiplied by a window function (e.g., Hamming window) to reduce spectral leakage and minimize artifacts at frame boundaries.
3. Auto-correlation Calculation: For each frame, the auto-correlation function is computed using the formula mentioned earlier, yielding the auto-correlation sequence for that frame.
4. Analysis: The resulting short-term auto-correlation sequences are analyzed to extract temporal features such as pitch periods, formant frequencies, and other speech characteristics.

**Source 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')

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')

**Experiment No: 10**

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

**Objective:**

1. Develop a program to estimate the pitch of a speech signal, providing insights into the fundamental frequency and intonation patterns.
2. Implement algorithms such as autocorrelation, cepstral analysis, or frequency-domain methods to accurately determine the pitch period of the speech signal.

**Theory:** Pitch estimation involves determining the fundamental frequency (F0) of a speech signal. Two common methods are:

i) autocorrelation-based pitch estimation.

ii)cepstrum-based pitch estimation.

Autocorrelation measures the similarity between a signal and its delayed version, identifying peaks that correspond to the pitch period. Cepstrum analysis separates the glottal excitation (pitch-related) from vocal tract characteristics, identifying the dominant peak in the cepstrum as the pitch period.

**Implementation Steps:**

1. Preprocessing: Remove noise and normalize the amplitude of the speech signal.
2. Frame Segmentation: Divide the signal into short overlapping frames.
3. Windowing: Multiply each frame by a window function to reduce spectral leakage.
4. Pitch Estimation: Apply autocorrelation or cepstrum analysis to estimate the pitch period for each frame.
5. Post-processing: Refine pitch estimates for accuracy and smoothness.
6. Pitch Tracking: Optionally track pitch variations across frames for dynamic analysis.

**Source 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(r"C:\Users\Win-10\OneDrive\Desktop\Lab\_3.2\Recording.wav", sr=None)

y, Fs = librosa.load(r"speech.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)

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()