Experiment Number: 01

Name of the Experiment: To 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.

Theory:

In digital image processing, "image reading" means loading an image from storage into computer memory for analysis or manipulation. It involves decoding the image file, converting it into a numerical format, and possibly preprocessing it for further analysis.

Image Resizing: Image resizing refers to the scaling of images. Scaling comes in handy in many image processing as well as machine learning applications. It helps in reducing the number of pixels from an image and that has several advantages e.g. It can reduce the time of training of a neural network as the more the number of pixels in an image more is the number of input nodes that in turn increases the complexity of the model.

Image Color-Transform: Color transforms are used in digital image processing to convert an image from one color space to another. These transforms can be used for a variety of purposes such as correcting color balance, enhancing color contrast, and improving image compression. Some popular color spaces used in digital image processing include RGB, CMYK, HSV, and Lab. Color transforms in digital image processing can be performed using mathematical operations such as matrix multiplication and color lookup tables.

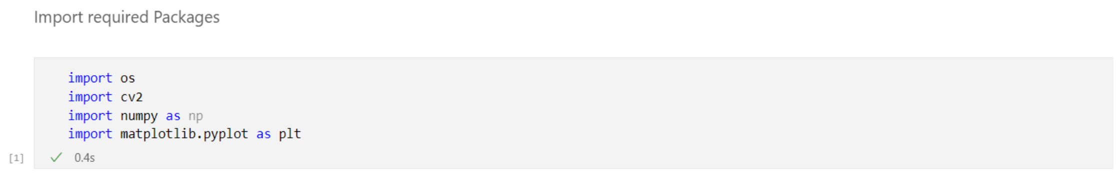
Image Normalization: Image normalization is a typical process in image processing that changes the range of pixel intensity values. Its normal purpose is to convert an input image into a range of pixel values that are more familiar or normal to the senses, hence the term normalization.

In this work, we will perform a function that produces a normalization of an input image (grayscale or RGB). Then, we understand a representation of the range of values of the scale of the image represented between 0 and 255, in this way we get, for example, that very dark images become clearer. The linear normalization of a digital image is performed according to the formula

$$\text{Output channel} = \frac{255 * (\text{Input\_channel} - \text{min})}{(\text{max} - \text{min})}$$

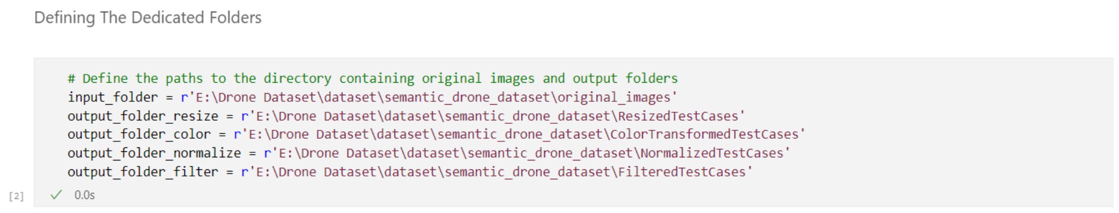
In digital image processing, filters are operations applied to an image to enhance or modify its features. They can be used for tasks like noise reduction, edge detection, smoothing, sharpening, and more. Filters work by modifying pixel values based on their neighboring pixels according to a predefined mathematical operation or kernel. Common types of filters include Gaussian, Sobel, Median, and Laplacian filters.

Code, Input and Output:


Import required Packages

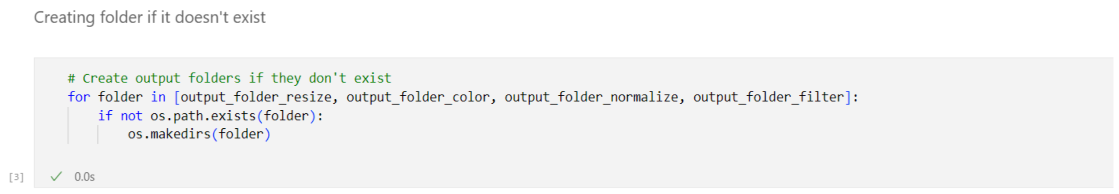
```
import os
import cv2
import numpy as np
import matplotlib.pyplot as plt
```

[1] ✓ 0.4s


Defining The Dedicated Folders

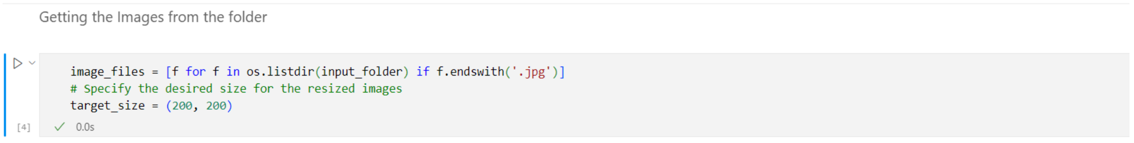
```
# Define the paths to the directory containing original images and output folders
input_folder = r'E:\Drone Dataset\dataset\semantic_drone_dataset\original_images'
output_folder_resize = r'E:\Drone Dataset\dataset\semantic_drone_dataset\ResizedTestCases'
output_folder_color = r'E:\Drone Dataset\dataset\semantic_drone_dataset\ColorTransformedTestCases'
output_folder_normalize = r'E:\Drone Dataset\dataset\semantic_drone_dataset\NormalizedTestCases'
output_folder_filter = r'E:\Drone Dataset\dataset\semantic_drone_dataset\FilteredTestCases'
```

[2] ✓ 0.0s


Creating folder if it doesn't exist

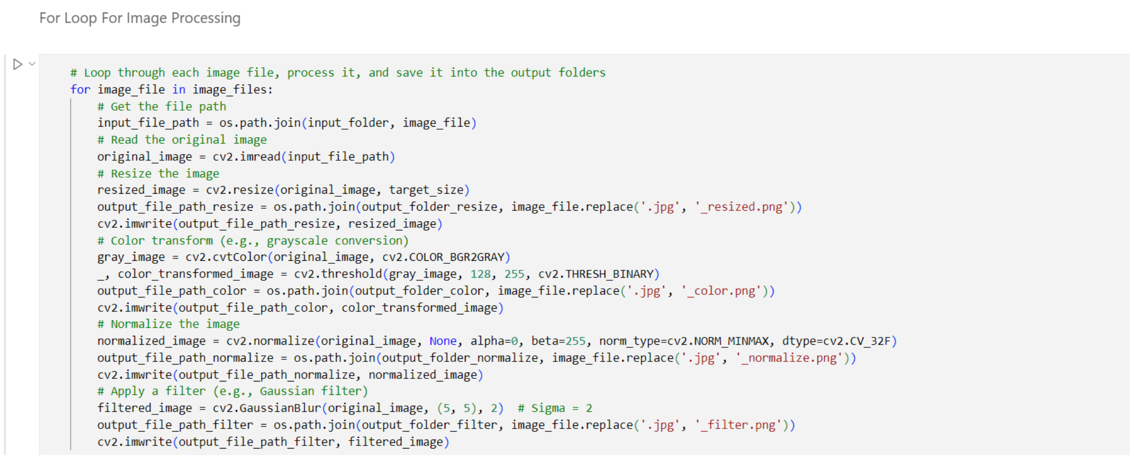
```
# Create output folders if they don't exist
for folder in [output_folder_resize, output_folder_color, output_folder_normalize, output_folder_filter]:
    if not os.path.exists(folder):
        os.makedirs(folder)
```

[3] ✓ 0.0s

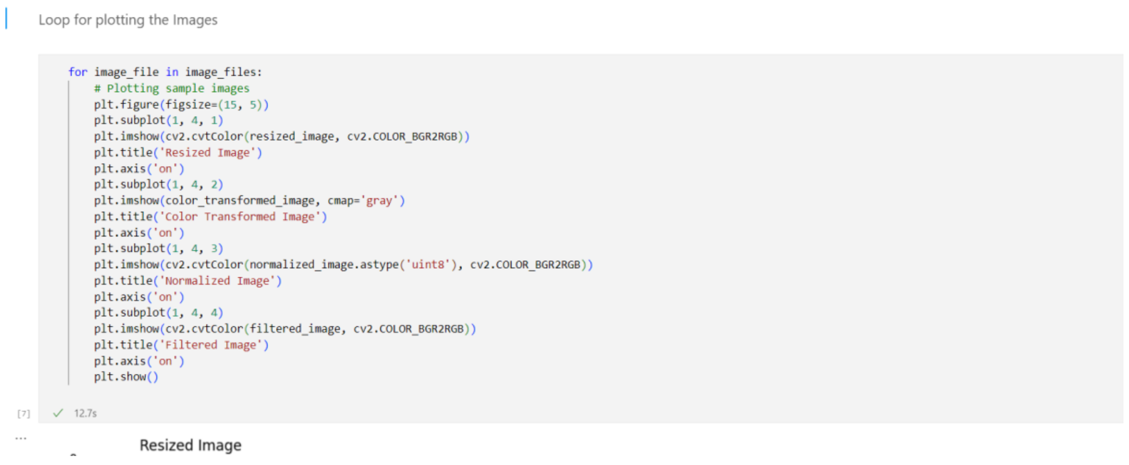

Getting the Images from the folder

```
image_files = [f for f in os.listdir(input_folder) if f.endswith('.jpg')]
# Specify the desired size for the resized images
target_size = (200, 200)
```

[4] ✓ 0.0s


For Loop For Image Processing

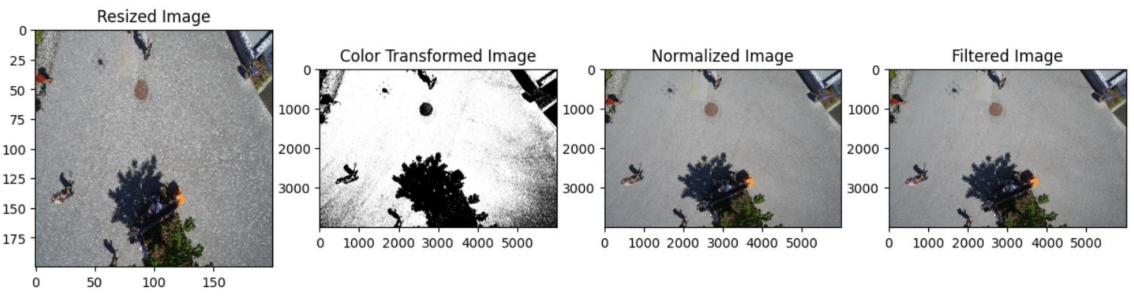
```
# Loop through each image file, process it, and save it into the output folders
for image_file in image_files:
    # Get the file path
    input_file_path = os.path.join(input_folder, image_file)
    # Read the original image
    original_image = cv2.imread(input_file_path)
    # Resize the image
    resized_image = cv2.resize(original_image, target_size)
    output_file_path_resize = os.path.join(output_folder_resize, image_file.replace('.jpg', '_resized.png'))
    cv2.imwrite(output_file_path_resize, resized_image)
    # Color transform (e.g., grayscale conversion)
    gray_image = cv2.cvtColor(original_image, cv2.COLOR_BGR2GRAY)
    _, color_transformed_image = cv2.threshold(gray_image, 128, 255, cv2.THRESH_BINARY)
    output_file_path_color = os.path.join(output_folder_color, image_file.replace('.jpg', '_color.png'))
    cv2.imwrite(output_file_path_color, color_transformed_image)
    # Normalize the image
    normalized_image = cv2.normalize(original_image, None, alpha=0, beta=255, norm_type=cv2.NORM_MINMAX, dtype=cv2.CV_32F)
    output_file_path_normalize = os.path.join(output_folder_normalize, image_file.replace('.jpg', '_normalize.png'))
    cv2.imwrite(output_file_path_normalize, normalized_image)
    # Apply a filter (e.g., Gaussian filter)
    filtered_image = cv2.GaussianBlur(original_image, (5, 5), 2)  # Sigma = 2
    output_file_path_filter = os.path.join(output_folder_filter, image_file.replace('.jpg', '_filter.png'))
    cv2.imwrite(output_file_path_filter, filtered_image)
```

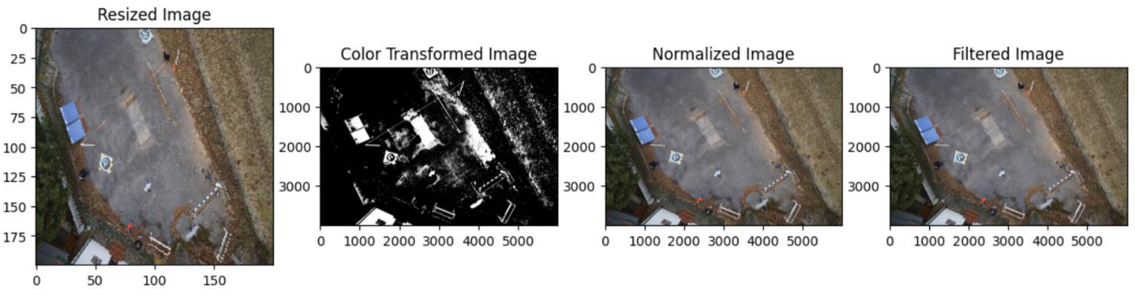

Loop for plotting the Images

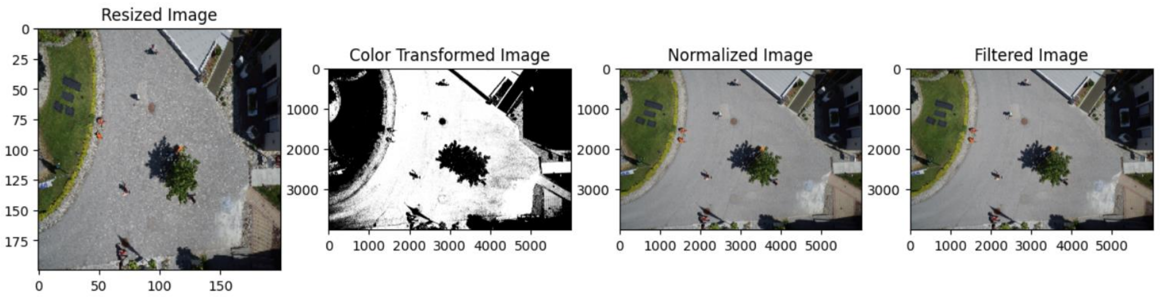
```
for image_file in image_files:
    # Plotting sample images
    plt.figure(figsize=(15, 5))
    plt.subplot(1, 4, 1)
    plt.imshow(cv2.cvtColor(resized_image, cv2.COLOR_BGR2RGB))
    plt.title('Resized Image')
    plt.axis('on')
    plt.subplot(1, 4, 2)
    plt.imshow(color_transformed_image, cmap='gray')
    plt.title('Color Transformed Image')
    plt.axis('on')
    plt.subplot(1, 4, 3)
    plt.imshow(cv2.cvtColor(normalized_image.astype('uint8'), cv2.COLOR_BGR2RGB))
    plt.title('Normalized Image')
    plt.axis('on')
    plt.subplot(1, 4, 4)
    plt.imshow(cv2.cvtColor(filtered_image, cv2.COLOR_BGR2RGB))
    plt.title('Filtered Image')
    plt.axis('on')
    plt.show()
```

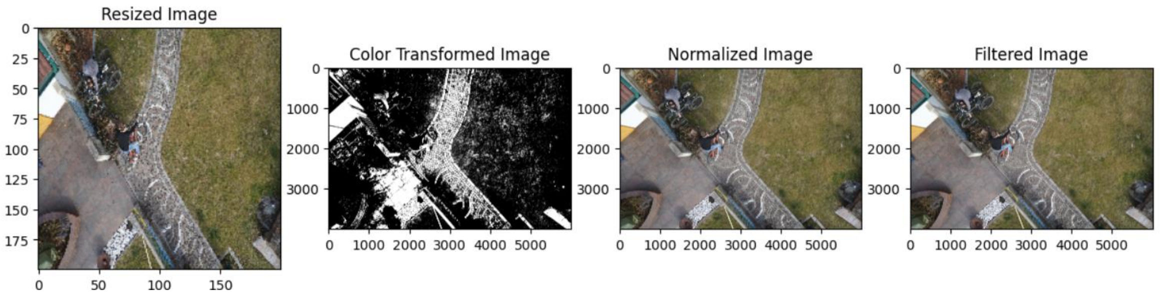
[7] ✓ 12.7s

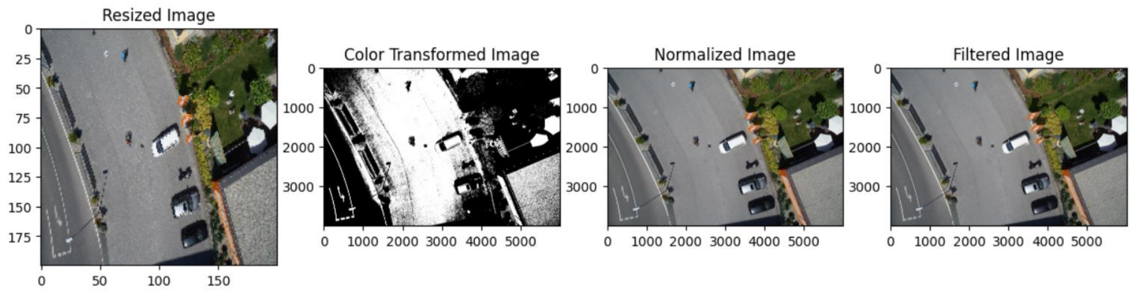
Resized Image


Resized Image
Color Transformed Image
Normalized Image
Filtered Image


Resized Image
Color Transformed Image
Normalized Image
Filtered Image


Resized Image
Color Transformed Image
Normalized Image
Filtered Image


Resized Image
Color Transformed Image
Normalized Image
Filtered Image


Resized Image
Color Transformed Image
Normalized Image
Filtered Image

Experiment Number: 02

Name of the Experiment: To write a program to execute Semantic Segmentation.

Objectives: To know the basic concept about image segmentation and knowing that how semantic segmentation is performed and how it differs from instance segmentation and panoptic segmentation. And also knowing that what models we should use for semantic segmentation to get higher accuracy.

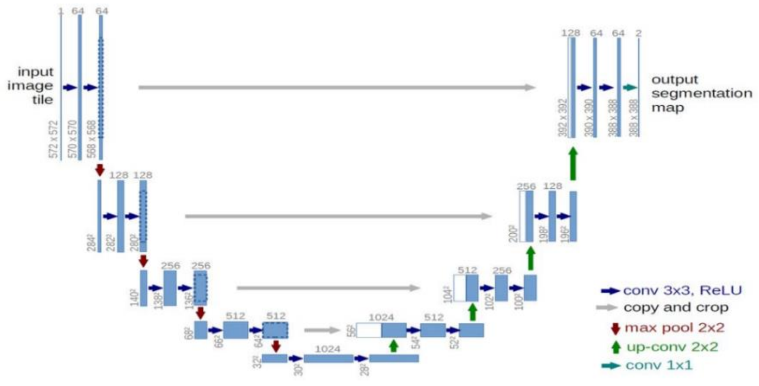
Theory:

Image segmentation is the process of partitioning an image into multiple segments or regions to simplify its representation and make it easier to analyze. The goal of segmentation is to divide an image into meaningful parts based on certain characteristics such as color, intensity, texture, or boundaries. Each segment typically corresponds to a distinct object or region of interest within the image. Image segmentation is a fundamental task in computer vision and is used in various applications such as object detection, medical imaging, scene understanding, and image editing.

Image segmentation can further be divided into the following categories — instance segmentation, semantic segmentation, and panoptic segmentation.

During a semantic segmentation task, segmentation masks represent fully labeled images. It means that all pixels in the image should belong to some category, whether they belong to the same instance or not. However, in this case, all pixels with the same category are represented as a single segment. If two pixels are categorized as "people" then segmentation mask pixel values will be the same for both of them. The same image, as was shown for instance segmentation, is below presented as a semantic segmentation mask. Now all persons in the image have the same(orange) color label and all glasses/cups have the same(green) color label, but there is also the grey color that is labeled as background.

U-Net was introduced in the paper, U-Net: Convolutional Networks for Biomedical Image Segmentation. The model architecture is fairly simple: an encoder (for down sampling) and a decoder (for up sampling) with skip connections. As Figure 1 shows, it shapes like the letter U hence the name U-Net. The gray arrows indicate the skip connections that concatenate the encoder feature map with the decoder, which helps the backward flow of gradients for improved training.



Experiment Number: 03

Name of the Experiment: To 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 know the concept of calculating the area covered by the mask of an image and also generate a new image which labels the original image according to the mask.

Theory: The theory behind this task involves image segmentation using a binary mask, area computation, and overlaying the mask onto the original image. here's a more detailed explanation of the theory behind the task:

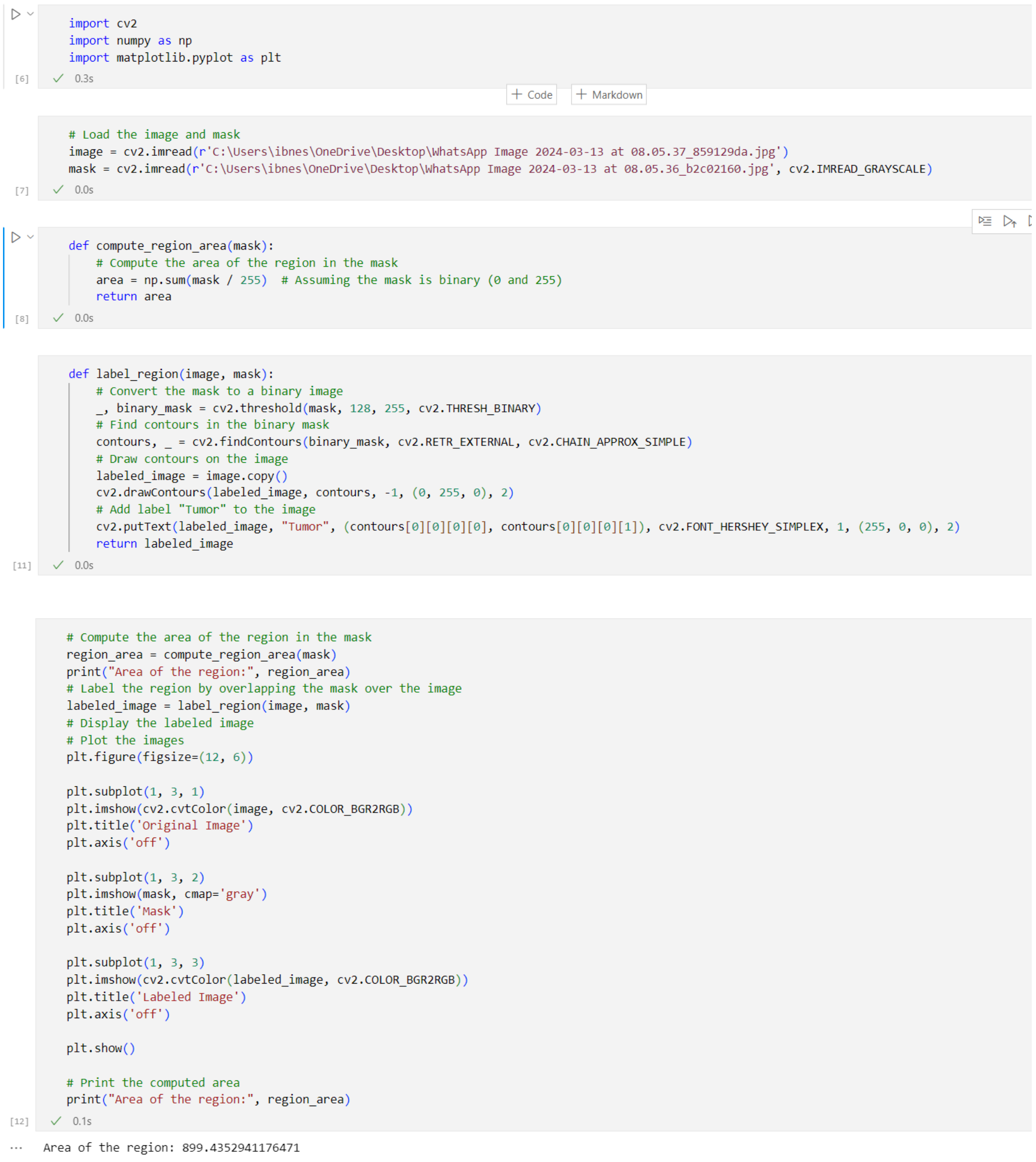
Image Segmentation: Image segmentation is the process of partitioning an image into multiple segments or regions based on certain characteristics. In this task, we're provided with a binary mask, which serves as a segmentation map. The mask indicates the region of interest in the image.

Area Computation: Once we have the binary mask, we need to compute the area of the region it represents in the original image. The area can be calculated by counting the number of pixels in the mask that belong to the region of interest. Since the mask is binary (containing only black and white pixels), we count the number of white pixels (usually represented by pixel values of 255 in an 8-bit grayscale image).

Overlaying the Mask: After computing the area, we want to visualize the identified region in the original image. This is done by overlaying the binary mask onto the original image. We can achieve this by blending the two images together using a specified opacity level. The overlay allows us to visually identify the region of interest within the context of the original image.

Overall, the theory involves segmenting the image using the provided mask, computing the area of the segmented region, and then visually labeling and highlighting this region by overlaying the mask onto the original image.

Code, Input and Output:



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

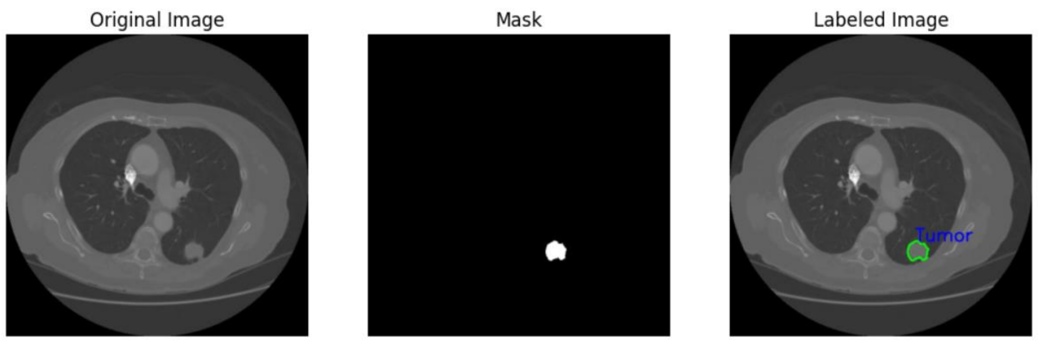
```python
# Load the image and mask
image = cv2.imread(r'C:\Users\ibnes\OneDrive\Desktop\WhatsApp Image 2024-03-13 at 08.05.37_859129da.jpg')
mask = cv2.imread(r'C:\Users\ibnes\OneDrive\Desktop\WhatsApp Image 2024-03-13 at 08.05.36_b2c02160.jpg', cv2.IMREAD_GRAYSCALE)
```

```python
def compute_region_area(mask):
    # Compute the area of the region in the mask
    area = np.sum(mask / 255)  # Assuming the mask is binary (0 and 255)
    return area
```

```python
def label_region(image, mask):
    # Convert the mask to a binary image
    _, binary_mask = cv2.threshold(mask, 128, 255, cv2.THRESH_BINARY)
    # Find contours in the binary mask
    contours, _ = cv2.findContours(binary_mask, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
    # Draw contours on the image
    labeled_image = image.copy()
    cv2.drawContours(labeled_image, contours, -1, (0, 255, 0), 2)
    # Add label "Tumor" to the image
    cv2.putText(labeled_image, "Tumor", (contours[0][0][0][0], contours[0][0][0][1]), cv2.FONT_HERSHEY_SIMPLEX, 1, (255, 0, 0), 2)
    return labeled_image
```

```python
# Compute the area of the region in the mask
region_area = compute_region_area(mask)
print("Area of the region:", region_area)
# Label the region by overlapping the mask over the image
labeled_image = label_region(image, mask)
# Display the labeled image
# Plot the images
plt.figure(figsize=(12, 6))

plt.subplot(1, 3, 1)
plt.imshow(cv2.cvtColor(image, cv2.COLOR_BGR2RGB))
plt.title('Original Image')
plt.axis('off')

plt.subplot(1, 3, 2)
plt.imshow(mask, cmap='gray')
plt.title('Mask')
plt.axis('off')

plt.subplot(1, 3, 3)
plt.imshow(cv2.cvtColor(labeled_image, cv2.COLOR_BGR2RGB))
plt.title('Labeled Image')
plt.axis('off')

plt.show()

# Print the computed area
print("Area of the region:", region_area)
```

```
Area of the region: 899.4352941176471
```



Experiment Number: 04

Name of the Experiment: To 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).

Objectives: The objective of these techniques is to manipulate pixel intensities or filter images to enhance features or extract relevant information.

Theory:

Basic Intensity Transformation: Intensity transformations are fundamental operations in image processing used to alter the pixel intensity values to achieve desired effects. These transformations include:

- Negation: Inverts the intensity values of an image, resulting in a negative-like image where dark areas become bright and vice versa.
- Log transformation: Adjusts the dynamic range of pixel intensities by taking the logarithm of each pixel value. It enhances details in dark regions while compressing the intensity values in bright regions.
- Power-law transformation: Also known as gamma correction, it raises the intensity values of an image to a power function to adjust brightness and contrast. It is commonly used for nonlinear adjustments in image enhancement.
- Piece-wise linear transformation: Divides the intensity range of an image into segments and applies different linear functions to each segment. This allows for more precise control over contrast and brightness adjustments.

Convolution: Convolution is a fundamental operation in image processing that involves applying a kernel (a small matrix) to an image to produce a modified version of the original image. Different types of convolution filters are used for various purposes:

- High-pass filter: Enhances high-frequency components in an image, such as edges and details, while suppressing low-frequency components, such as smooth areas and noise.
- Low-pass filter: Suppresses high-frequency components and retains low-frequency components, resulting in a smoother image with reduced noise.
- Laplacian filter: Detects edges and other features by highlighting regions of rapid intensity change in an image. It is commonly used for edge detection and image sharpening.

These techniques are essential tools in image processing for tasks such as image enhancement, noise reduction, edge detection, and feature extraction.

Experiment Number: 05

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

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

Objectives: The objective of edge detection is to identify and localize significant changes in intensity or color within an image.

Theory:

An "edge" in the context of image processing and computer vision refers to a boundary or transition area within an image where there is a significant change in intensity, color, or texture. It represents the visual separation between distinct regions or objects in the image. Edge detection algorithms aim to identify and highlight these areas of abrupt intensity changes, which are often indicative of object boundaries or features of interest within the image.

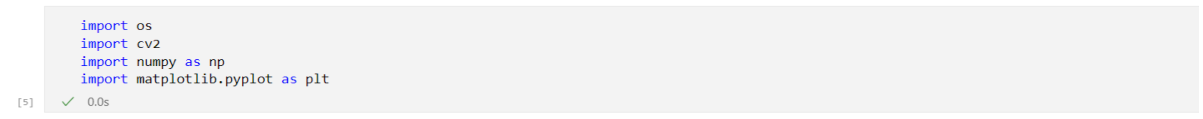
Edge detection is a process in image processing and computer vision that involves identifying the boundaries or edges within an image where there are significant changes in intensity or color.

Canny Edge Detection: The Canny edge detection algorithm, developed by John F. Canny in 1986, is a multi-stage method widely used in image processing and computer vision. It begins by smoothing the image to reduce noise, then calculates gradients to identify areas of rapid intensity change. Subsequent steps involve non-maximum suppression to thin edges and hysteresis thresholding to determine true edges. Canny edge detection is preferred for its accuracy, ability to handle noise, and capability to detect a wide range of edges while minimizing false positives.

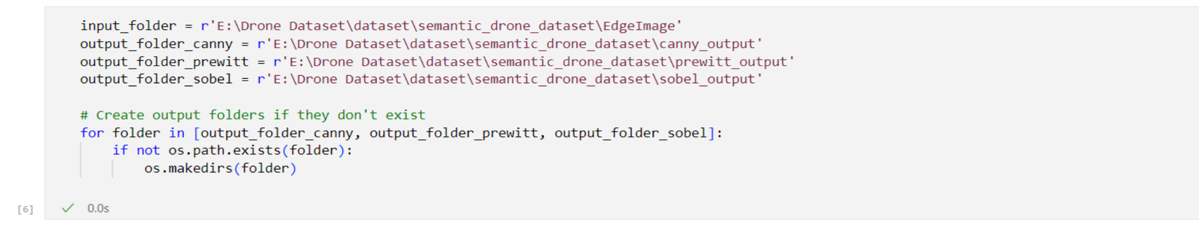
Prewitt Edge Detection: The Prewitt edge detection algorithm is a simple and computationally efficient method for detecting edges in digital images. It involves convolving the image with a pair of 3x3 convolution kernels, one estimating gradients in the horizontal direction and the other in the vertical direction. These kernels highlight areas of intensity change along the horizontal and vertical axes, respectively. After convolving the image with these kernels, the gradient magnitude and direction can be calculated. Prewitt edge detection is sensitive to noise and may produce less accurate results compared to more sophisticated algorithms like Canny.

Sobel Edge Detection: The Sobel edge detection algorithm is another popular method for detecting edges in images. Similar to Prewitt, Sobel employs two 3x3 convolution kernels to estimate gradients in the horizontal and vertical directions. However, Sobel gives higher weight to pixels in the central part of the kernel, making it more selective in detecting edges. After convolving the image with these kernels, the gradient magnitude and direction can be computed. Sobel edge detection is often preferred for its simplicity and effectiveness in many applications.

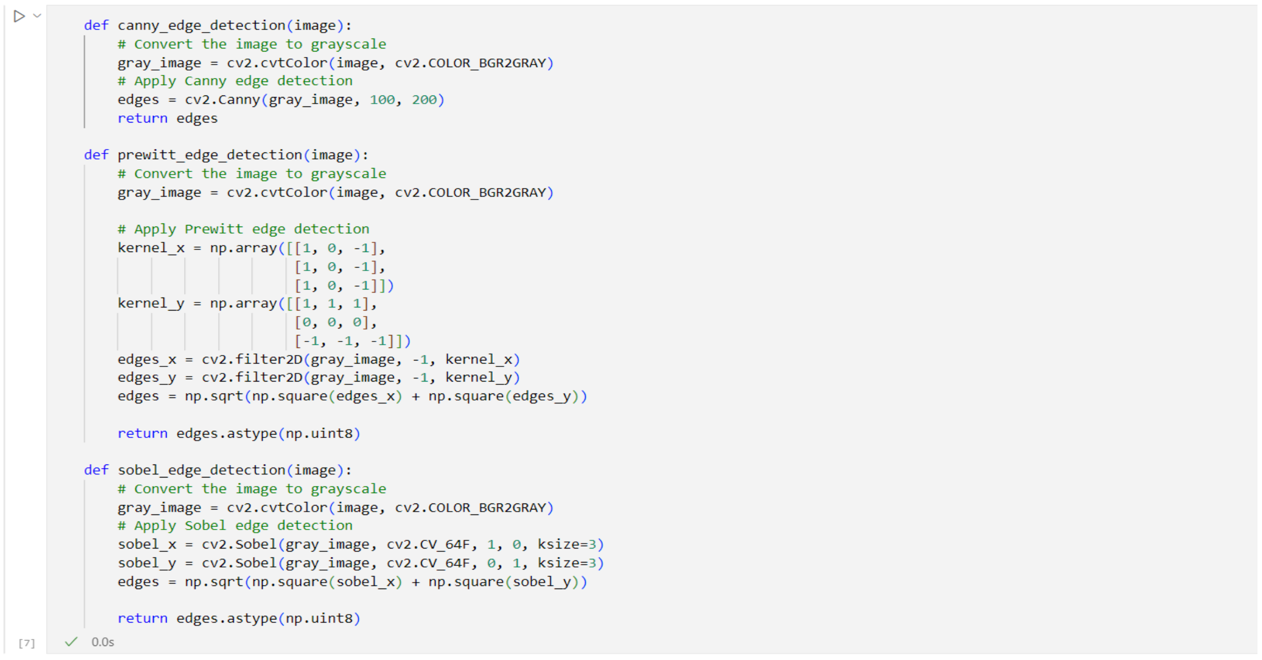
Code, Input and Output:



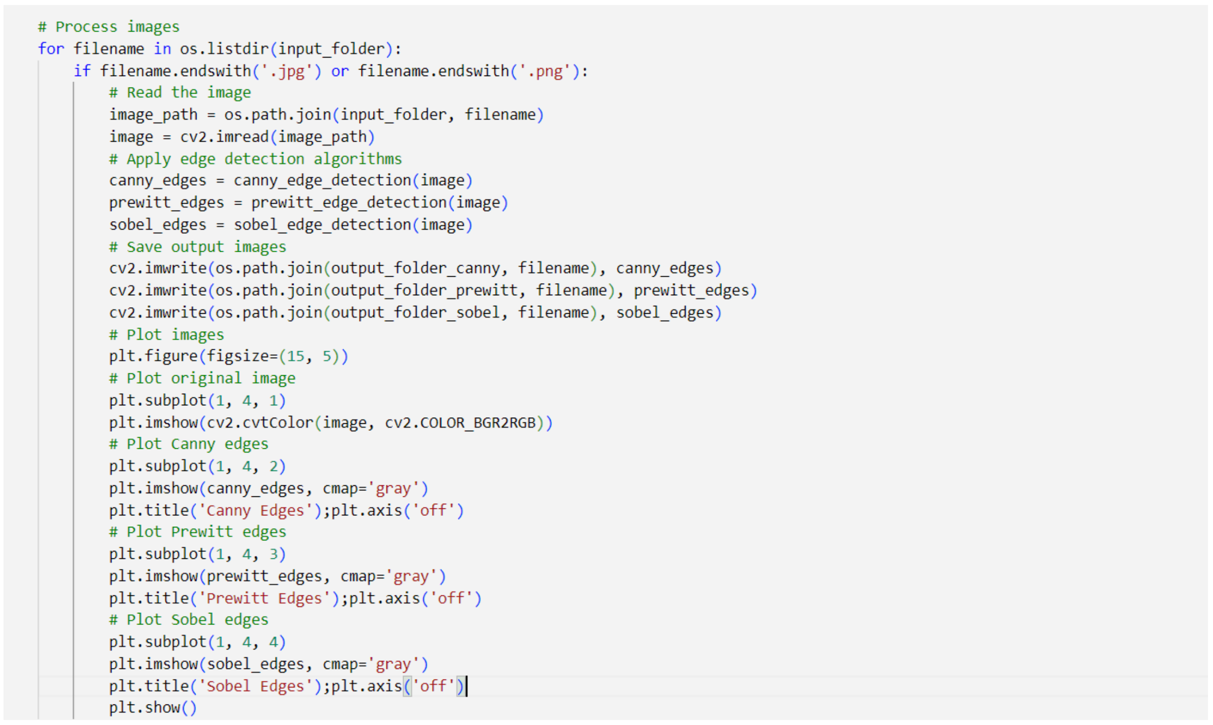
```python
import os
import cv2
import numpy as np
import matplotlib.pyplot as plt
```



```python
input_folder = r'E:\Drone Dataset\dataset\semantic_drone_dataset\EdgeImage'
output_folder_canny = r'E:\Drone Dataset\dataset\semantic_drone_dataset\canny_output'
output_folder_prewitt = r'E:\Drone Dataset\dataset\semantic_drone_dataset\prewitt_output'
output_folder_sobel = r'E:\Drone Dataset\dataset\semantic_drone_dataset\sobel_output'

# Create output folders if they don't exist
for folder in [output_folder_canny, output_folder_prewitt, output_folder_sobel]:
    if not os.path.exists(folder):
        os.makedirs(folder)
```

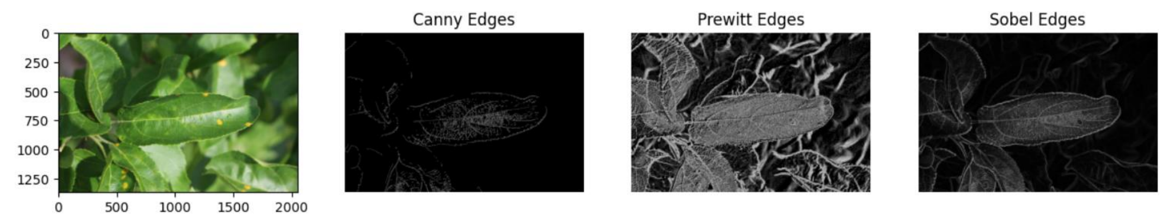


```python
def canny_edge_detection(image):
    # Convert the image to grayscale
    gray_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    # Apply Canny edge detection
    edges = cv2.Canny(gray_image, 100, 200)
    return edges

def prewitt_edge_detection(image):
    # Convert the image to grayscale
    gray_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

    # Apply Prewitt edge detection
    kernel_x = np.array([[1, 0, -1],
                         [1, 0, -1],
                         [1, 0, -1]])
    kernel_y = np.array([[1, 1, 1],
                         [0, 0, 0],
                         [-1, -1, -1]])
    edges_x = cv2.filter2D(gray_image, -1, kernel_x)
    edges_y = cv2.filter2D(gray_image, -1, kernel_y)
    edges = np.sqrt(np.square(edges_x) + np.square(edges_y))

    return edges.astype(np.uint8)

def sobel_edge_detection(image):
    # Convert the image to grayscale
    gray_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    # Apply Sobel edge detection
    sobel_x = cv2.Sobel(gray_image, cv2.CV_64F, 1, 0, ksize=3)
    sobel_y = cv2.Sobel(gray_image, cv2.CV_64F, 0, 1, ksize=3)
    edges = np.sqrt(np.square(sobel_x) + np.square(sobel_y))

    return edges.astype(np.uint8)
```

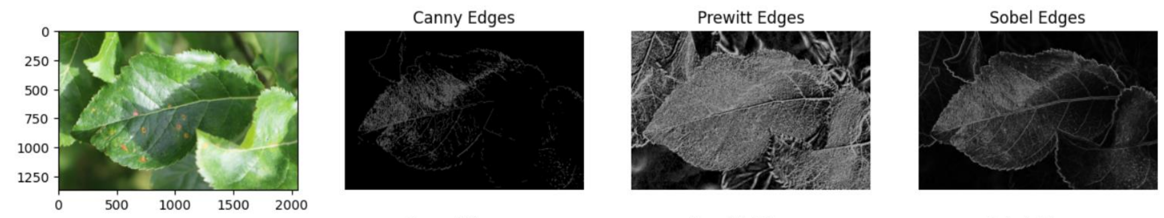


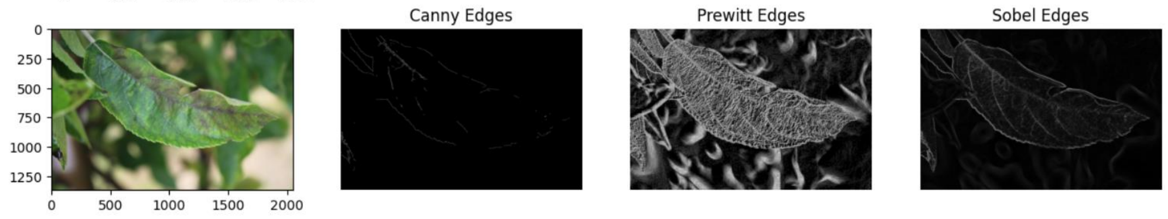
```python
# Process images
for filename in os.listdir(input_folder):
    if filename.endswith('.jpg') or filename.endswith('.png'):
        # Read the image
        image_path = os.path.join(input_folder, filename)
        image = cv2.imread(image_path)
        # Apply edge detection algorithms
        canny_edges = canny_edge_detection(image)
        prewitt_edges = prewitt_edge_detection(image)
        sobel_edges = sobel_edge_detection(image)
        # Save output images
        cv2.imwrite(os.path.join(output_folder_canny, filename), canny_edges)
        cv2.imwrite(os.path.join(output_folder_prewitt, filename), prewitt_edges)
        cv2.imwrite(os.path.join(output_folder_sobel, filename), sobel_edges)
        # Plot images
        plt.figure(figsize=(15, 5))
        # Plot original image
        plt.subplot(1, 4, 1)
        plt.imshow(cv2.cvtColor(image, cv2.COLOR_BGR2RGB))
        # Plot Canny edges
        plt.subplot(1, 4, 2)
        plt.imshow(canny_edges, cmap='gray')
        plt.title('Canny Edges');plt.axis('off')
        # Plot Prewitt edges
        plt.subplot(1, 4, 3)
        plt.imshow(prewitt_edges, cmap='gray')
        plt.title('Prewitt Edges');plt.axis('off')
        # Plot Sobel edges
        plt.subplot(1, 4, 4)
        plt.imshow(sobel_edges, cmap='gray')
        plt.title('Sobel Edges');plt.axis('off')
        plt.show()
```

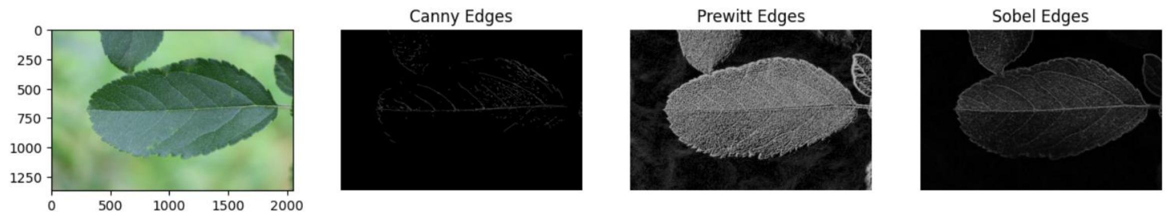
...

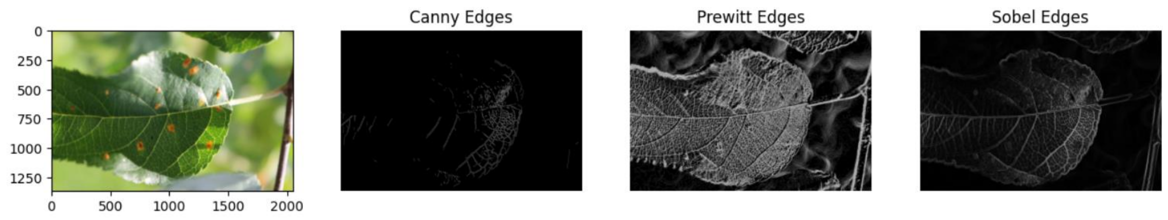

Canny Edges
Prewitt Edges
Sobel Edges
0
250
500
750
1000
1250
0
500
1000
1500
2000

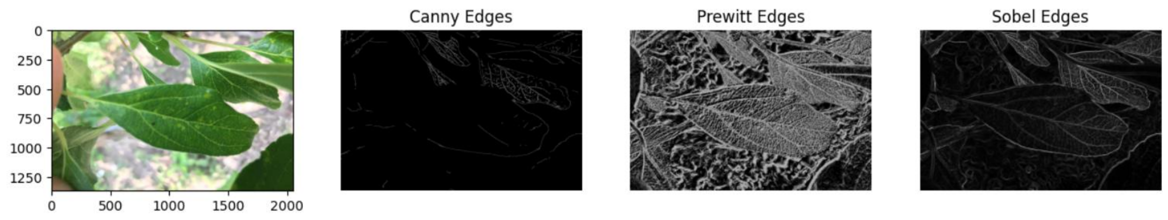
...


Canny Edges
Prewitt Edges
Sobel Edges
0
250
500
750
1000
1250
0
500
1000
1500
2000


Canny Edges
Prewitt Edges
Sobel Edges
0
250
500
750
1000
1250
0
500
1000
1500
2000


Canny Edges
Prewitt Edges
Sobel Edges
0
250
500
750
1000
1250
0
500
1000
1500
2000


Canny Edges
Prewitt Edges
Sobel Edges
0
250
500
750
1000
1250
0
500
1000
1500
2000


Canny Edges
Prewitt Edges
Sobel Edges
0
250
500
750
1000
1250
0
500
1000
1500
2000

Experiment Number: 06

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

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

Objectives: Down-sample speech signal, reducing sampling frequency while preserving quality, then save, maintaining intelligibility for efficient storage/transmission.

Theory: To identify the sampling frequency of a speech signal, we would typically refer to the metadata or specifications provided with the signal file or recording. The sampling frequency, represents the number of samples captured per second during the recording process. Common sampling frequencies for speech signals include 8 kHz, 16 kHz, 44.1 kHz, and 48 kHz. The sampling frequency determines the highest frequency that can be accurately represented in the signal, following the Nyquist theorem, which states that the sampling frequency must be at least twice the highest frequency component of the signal to prevent aliasing.

The bit resolution, often referred to as the bit depth, indicates the number of bits used to represent each sample of the speech signal. Common bit depths include 8-bit, 16-bit, and 24-bit. A higher bit depth allows for more precise representation of the amplitude of the signal, resulting in higher fidelity audio.

To perform down-sampling of the speech signal, we would reduce the sampling frequency by resampling the signal at a lower rate. This process involves removing samples from the original signal while preserving its essential characteristics. Down-sampling can be achieved through various signal processing techniques such as decimation or interpolation.

After down-sampling the speech signal, we can save the modified signal to a file in a suitable audio format such as WAV or MP3. It's important to note that down-sampling may result in some loss of audio quality, particularly if the new sampling frequency is significantly lower than the original. However, down-sampling can be useful for reducing file size or processing requirements while still maintaining acceptable audio quality for certain applications.

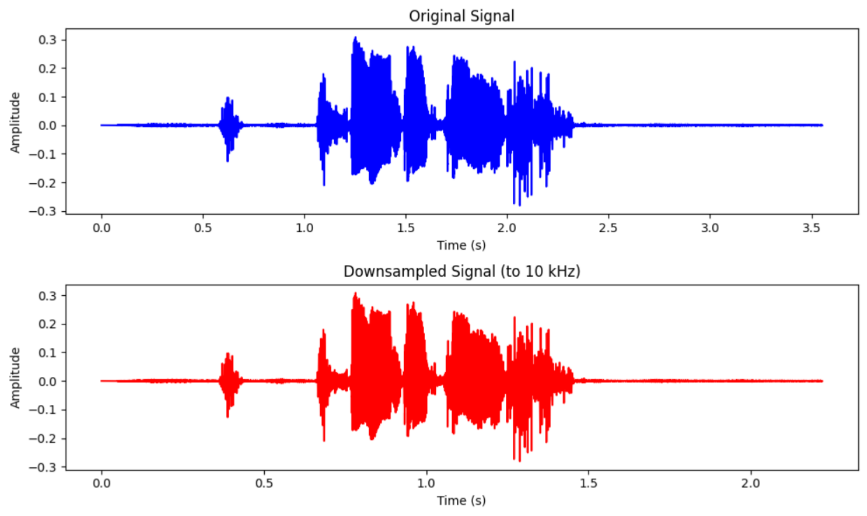
Code, Input and Output:

```
import librosa
import soundfile as sf
import numpy as np
import matplotlib.pyplot as plt
from IPython.display import Audio
```
[1] ✓ 0.5s

```
def speech_preprocessing(audio_file):
    # Load the audio file
    signal, sampling_frequency = librosa.load(audio_file, sr=None, mono=True)

    # Get the bit resolution
    bit_resolution = librosa.get_samplerate(audio_file)

    # Downsample the signal
    downsampled_signal = librosa.resample(signal, orig_sr=sampling_frequency, target_sr=10000)

    return signal, downsampled_signal, sampling_frequency, bit_resolution

def save_audio(signal, output_file, sampling_frequency):
    # Save the audio file
    sf.write(output_file, signal, sampling_frequency)
```
[2] ✓ 0.0s

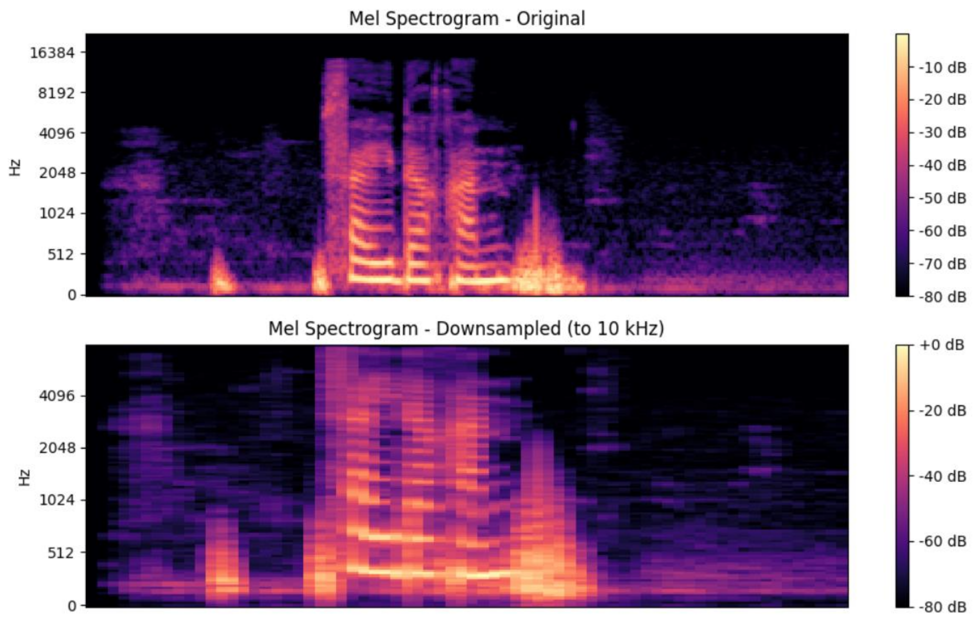
```
input_audio_file = r'E:\sayeeda_khan.wav'  # Path to the input audio file
output_audio_file = r'E:\output.wav' # Path to save the output audio file

# Perform speech preprocessing
original_signal, preprocessed_signal, sampling_frequency, bit_resolution = speech_preprocessing(input_audio_file)

# Plot original and downsampled signals
time_original = np.arange(0, len(original_signal)) / sampling_frequency
time_downsampled = np.arange(0, len(preprocessed_signal)) / 16000
```
[3] ✓ 1.8s

```
plt.figure(figsize=(10, 6))
plt.subplot(2, 1, 1)
plt.plot(time_original, original_signal, color='b')
plt.title('Original Signal')
plt.xlabel('Time (s)')
plt.ylabel('Amplitude')

plt.subplot(2, 1, 2)
plt.plot(time_downsampled, preprocessed_signal, color='r')
plt.title('Downsampled Signal (to 10 kHz)')
plt.xlabel('Time (s)')
plt.ylabel('Amplitude')

plt.tight_layout()
plt.show()
```
[4] ✓ 0.3s



```
# Save the preprocessed audio
save_audio(preprocessed_signal, output_audio_file, 16000)

# Compute and plot Mel spectrograms
mel_original = librosa.feature.melspectrogram(y=original_signal, sr=sampling_frequency)
mel_preprocessed = librosa.feature.melspectrogram(y=preprocessed_signal, sr=16000)

plt.figure(figsize=(10, 6))
plt.subplot(2, 1, 1)
librosa.display.specshow(librosa.power_to_db(mel_original, ref=np.max), y_axis='mel', sr=sampling_frequency)
plt.colorbar(format='%+2.0f dB')
plt.title('Mel Spectrogram - Original')

plt.subplot(2, 1, 2)
librosa.display.specshow(librosa.power_to_db(mel_preprocessed, ref=np.max), y_axis='mel', sr=16000)
plt.colorbar(format='%+2.0f dB')
plt.title('Mel Spectrogram - Downsampled (to 10 kHz)')

plt.tight_layout()
plt.show()

# Playback the audio
Audio(data=preprocessed_signal, rate=16000)
```

[5] ✓ 0.4s



Experiment Number: 07

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

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

Objectives: Segment speech into voiced, unvoiced, silence regions for accurate analysis in speech processing tasks like recognition and synthesis.

Theory: In speech signal processing, speech can be segmented into three main regions: voiced, unvoiced, and silence regions.

Voiced Region:

Voiced regions in speech correspond to periods where the vocal cords vibrate, producing periodic waveform patterns. These regions typically contain harmonically rich frequency components. During voiced regions, the pitch or fundamental frequency of the speech signal remains relatively stable. Examples of voiced sounds include vowels and certain consonants like /m/, /n/, and /l/. In spectrograms or frequency domain representations of speech signals, voiced regions appear as vertical striations or bands, corresponding to the harmonics of the fundamental frequency.

Unvoiced Region:

Unvoiced regions occur when there is no vibration of the vocal cords, resulting in turbulent airflow through the vocal tract. These regions exhibit a lack of periodicity and may contain broad-spectrum noise-like components. Consonants such as /s/, /sh/, and /f/ are typically unvoiced. In spectrograms, unvoiced regions appear as diffuse energy spread across a wide range of frequencies, without distinct harmonic structure.

Silence Region:

Silence regions occur when there is no significant speech activity or when the signal falls below a certain amplitude threshold. These regions represent periods of silence or background noise between speech segments. Silence regions can be identified by detecting periods of low energy or amplitude in the speech signal. In spectrograms, silence regions appear as areas with very low energy or amplitude, often represented by a dark or blank space.

Segmenting speech into these regions is essential for various speech processing tasks such as speech recognition, speaker diarization, and speech synthesis. By accurately detecting and distinguishing between voiced, unvoiced, and silence regions, algorithms can better analyze and understand the content and structure of speech signals.

Experiment Number: 08

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

Objectives: The objective of finding the zerocrossing rate (ZCR) in speech signal processing is to quantify the rate at which the signal's waveform changes direction, which provides information about its periodicity and temporal characteristics.

Theory:

The Zero Crossing Rate (ZCR) is a simple yet effective feature used in speech and audio signal processing. The theory behind ZCR revolves around the concept of detecting the rate at which a signal crosses the zero-amplitude axis.

Here's the theory explained:

Definition: Zero crossing refers to the point in the signal where the value changes its sign, i.e., it crosses the zero-amplitude axis. The Zero Crossing Rate (ZCR) measures how often a signal crosses this axis within a given time frame or window.

Characterization of Signal: ZCR provides insights into the characteristics of the signal, particularly its temporal properties. Signals with higher ZCRs tend to have more high-frequency content or rapid variations, while signals with lower ZCRs are typically smoother or contain lower frequency components.

Applications:

Speech Analysis: In speech processing, ZCR can be used to distinguish between voiced and unvoiced segments. Voiced segments tend to have lower ZCRs due to their relatively periodic nature, while unvoiced segments exhibit higher ZCRs because of their random or noisy characteristics.

Audio Classification: ZCR can also be employed in audio classification tasks. For example, music signals often have lower ZCRs compared to speech signals due to their tonal nature.

Feature Extraction: ZCR serves as a fundamental feature for extracting information about the temporal dynamics of a signal. It is often used in combination with other features for tasks such as speaker recognition, emotion detection, and music genre classification.

Computation: Mathematically, ZCR is computed by counting the number of times the signal changes sign within a given window of time. It can be expressed as:

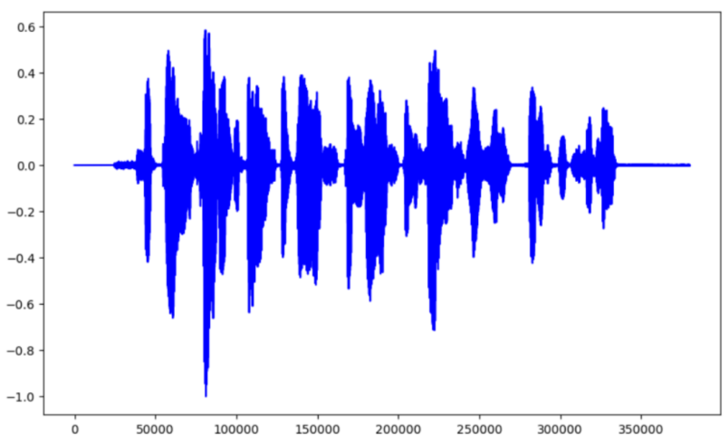
$$ZCR = \frac{1}{N-1}\sum_{n=1}^{N-1} |sgn(x[n]) - sgn(x[n-1])|$$

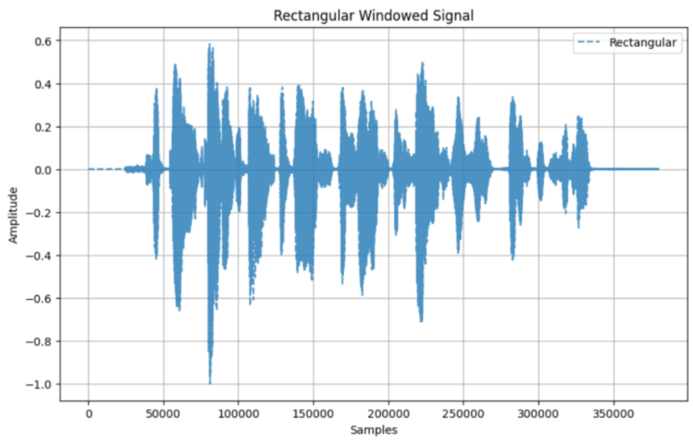
Where:

- N is the number of samples in the window.
- $x[n]$ represents the sample values of the signal.
- $sgn(\cdot)$ is the sign function, returning -1 for negative values, 0 for zero, and 1 for positive values.

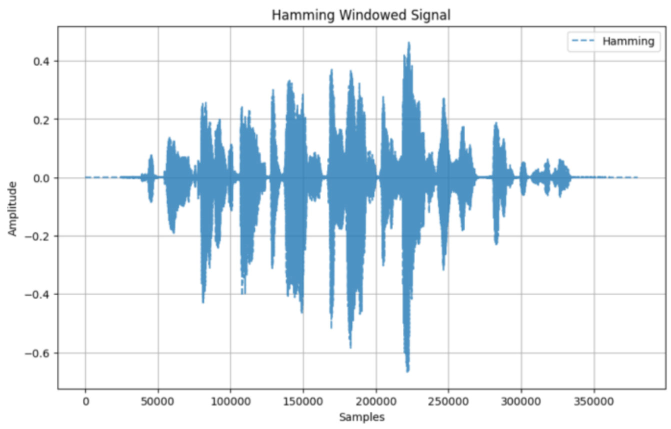
Code, Input and Output:

```
import numpy as np
import matplotlib.pyplot as plt
import soundfile as sf  # For loading audio files
# Function to compute zero crossing rate
def zero_crossing_rate(signal):
    zcr = np.mean(np.abs(np.diff(np.sign(signal))) / 2)
    return zcr
# Function to apply window function
def apply_window(signal, window_type):
    window = np.hamming(len(signal))
    if window_type == 'rectangular':
        window = np.ones(len(signal))
    elif window_type == 'hamming':
        window = np.hamming(len(signal))
    elif window_type == 'hanning':
        window = np.hanning(len(signal))
    return signal * window
# Load speech signal
file_path = r'C:\Users\ibnes\OneDrive\Documents\Sound Recordings\Voice.wav'
# Change this to your audio file path
signal, sampling_rate = sf.read(file_path)
# Convert to mono if stereo
if len(signal.shape) > 1:
    signal = signal.mean(axis=1)
# Normalize the signal
signal = signal / np.max(np.abs(signal))
# Compute ZCR for the original signal
zcr_original = zero_crossing_rate(signal)
# Apply different window functions and compute ZCR
window_types = ['rectangular', 'hamming', 'hanning']
zcr_values = []
plt.figure(figsize=(10, 6))
plt.plot(signal, label='Original Signal', color='blue')
for window_type in window_types:
    windowed_signal = apply_window(signal, window_type)
    zcr = zero_crossing_rate(windowed_signal)
    zcr_values.append(zcr)
    plt.figure(figsize=(10, 6))
    plt.plot(windowed_signal, label=window_type.capitalize(), linestyle='--',
alpha=0.8)
    plt.title(f'{window_type.capitalize()} Windowed Signal')
    plt.xlabel('Samples')
    plt.ylabel('Amplitude')plt.legend()
    plt.grid(True)plt.show()
    print(f"{window_type.capitalize()} Window ZCR:", zcr)
# Print original ZCR
print("Original ZCR:", zcr_original)
```

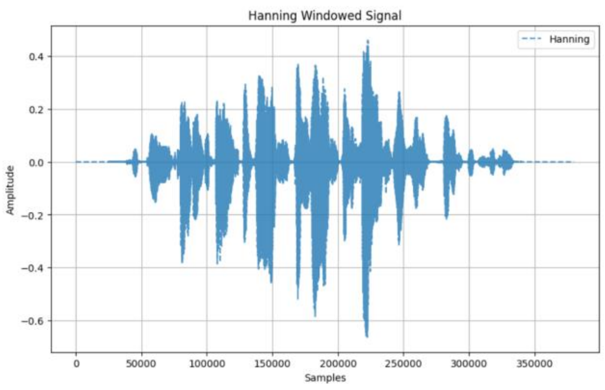




```
Rectangular Window ZCR: 0.05563067032478516
```



```
Hamming Window ZCR: 0.05563067032478516
```



```
Hanning Window ZCR: 0.05563198556393509
Original ZCR: 0.05563067032478516
```

Experiment Number: 9
Name of the Experiment: To Write a program to compute short term auto-correlation of a speech signal.

Objectives: The objective of short-term autocorrelation analysis in speech is to detect pitch, classify voiced/unvoiced segments, and analyze formants.

Theory:

The theory of short-term autocorrelation of a speech signal involves examining the correlation between adjacent segments of the signal within a short time window. Autocorrelation measures the similarity between a signal and a delayed version of itself. In short-term autocorrelation, this is done within smaller time windows or frames.

Mathematically, the short-term autocorrelation function is computed as the sum of products of signal samples within the window:

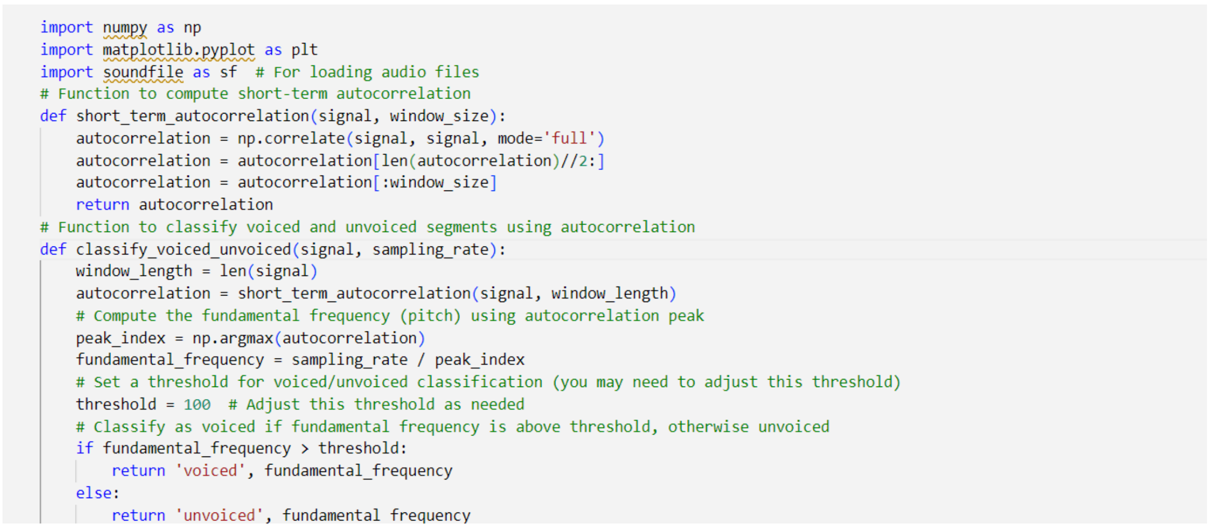
$$R(k) = \sum_{n=0}^{N-1} x[n] \cdot x[n-k]$$

where $x[n]$ represents the speech signal samples and N is the length of the window.

Interpretation of short-term autocorrelation involves understanding its peaks and patterns. Peaks in the autocorrelation function correspond to periodicity or repeating patterns within the signal. These peaks indicate the presence of voiced segments in the speech signal. Unvoiced segments, characterized by random or noisy behavior, may result in a relatively flat autocorrelation function without distinct peaks.

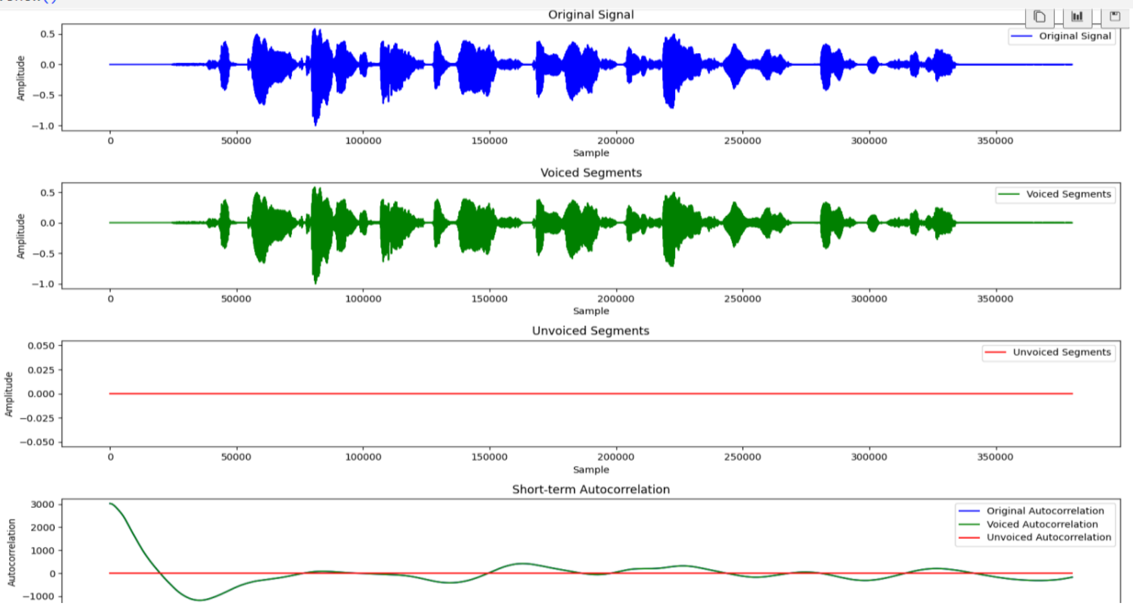
Short-term autocorrelation analysis is often used in speech processing tasks such as pitch detection, where the periodicity of the signal is essential for determining the fundamental frequency (pitch) of the voiced segments. Overall, short-term autocorrelation provides valuable insights into the temporal characteristics, periodicity, and structure of speech signals, making it a crucial tool in speech processing applications.

Code, Input and Output:



```python
import numpy as np
import matplotlib.pyplot as plt
import soundfile as sf  # For loading audio files
# Function to compute short-term autocorrelation
def short_term_autocorrelation(signal, window_size):
    autocorrelation = np.correlate(signal, signal, mode='full')
    autocorrelation = autocorrelation[len(autocorrelation)//2:]
    autocorrelation = autocorrelation[:window_size]
    return autocorrelation
# Function to classify voiced and unvoiced segments using autocorrelation
def classify_voiced_unvoiced(signal, sampling_rate):
    window_length = len(signal)
    autocorrelation = short_term_autocorrelation(signal, window_length)
    # Compute the fundamental frequency (pitch) using autocorrelation peak
    peak_index = np.argmax(autocorrelation)
    fundamental_frequency = sampling_rate / peak_index
    # Set a threshold for voiced/unvoiced classification (you may need to adjust this threshold)
    threshold = 100  # Adjust this threshold as needed
    # Classify as voiced if fundamental frequency is above threshold, otherwise unvoiced
    if fundamental_frequency > threshold:
        return 'voiced', fundamental_frequency
    else:
        return 'unvoiced', fundamental_frequency
```

```python
# Load speech signal
file_path = r'C:\Users\ibnes\OneDrive\Documents\Sound Recordings\Voice.wav'  # Change this to your audio file path
signal, sampling_rate = sf.read(file_path)
# Convert to mono if stereo
if len(signal.shape) > 1:
    signal = signal.mean(axis=1)
# Normalize the signal
signal = signal / np.max(np.abs(signal))
# Set window size for short-term autocorrelation
window_size = 500
# Classify voiced and unvoiced segments
voiced_segments = []
unvoiced_segments = []
segment_lengths = []
i = 0
while i < len(signal):
    segment = signal[i:i+window_size]
    segment_type, _ = classify_voiced_unvoiced(segment, sampling_rate)
    if segment_type == 'voiced':
        voiced_segments.append(segment)
    else:
        unvoiced_segments.append(segment)
    segment_lengths.append(len(segment))
    i += window_size

# Flatten the segments into single arrays
voiced_signal = np.concatenate(voiced_segments)
unvoiced_signal = np.zeros(len(signal)) if len(unvoiced_segments) == 0 else np.concatenate(unvoiced_segments)
# Compute short-term autocorrelation for the original, voiced, and unvoiced signals
autocorrelation_original = short_term_autocorrelation(signal, window_size)
autocorrelation_voiced = short_term_autocorrelation(voiced_signal, window_size)
autocorrelation_unvoiced = short_term_autocorrelation(unvoiced_signal, window_size)
# Plotting
plt.figure(figsize=(16, 10))
# Plot original signal
plt.subplot(4, 1, 1)
plt.plot(signal, label='Original Signal', color='blue')
plt.title('Original Signal')
plt.xlabel('Sample')
plt.ylabel('Amplitude')
plt.legend()
# Plot voiced segments
plt.subplot(4, 1, 2)
plt.plot(voiced_signal, label='Voiced Segments', color='green')
plt.title('Voiced Segments')
plt.xlabel('Sample')
plt.ylabel('Amplitude')
plt.legend()
# Plot unvoiced segments
plt.subplot(4, 1, 3)
plt.plot(unvoiced_signal, label='Unvoiced Segments', color='red')
plt.title('Unvoiced Segments')
plt.xlabel('Sample')
plt.ylabel('Amplitude')
plt.legend()
# Plot short-term autocorrelation
plt.subplot(4, 1, 4)
plt.plot(autocorrelation_original, label='Original Autocorrelation', color='blue')
plt.plot(autocorrelation_voiced, label='Voiced Autocorrelation', color='green')
plt.plot(autocorrelation_unvoiced, label='Unvoiced Autocorrelation', color='red')
plt.title('Short-term Autocorrelation')
plt.xlabel('Lag')
plt.ylabel('Autocorrelation')
plt.legend()
plt.tight_layout()
plt.show()
```



Experiment Number: 10

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

Objectives: Estimating pitch in speech signals aims to determine fundamental frequency for speech processing, speaker characterization, synthesis, linguistic analysis, and quality assessment, enhancing understanding and applications of speech technology.

Theory: Pitch estimation in speech signals involves determining the fundamental frequency (F0), which corresponds to the perceived pitch of the sound. The fundamental frequency is closely related to the rate at which the vocal cords vibrate during voiced segments of speech.

Several techniques are commonly used for estimating pitch:

Autocorrelation Method: This method involves computing the autocorrelation function of the speech signal. Peaks in the autocorrelation function at multiples of the fundamental period indicate the pitch period, allowing estimation of pitch frequency.

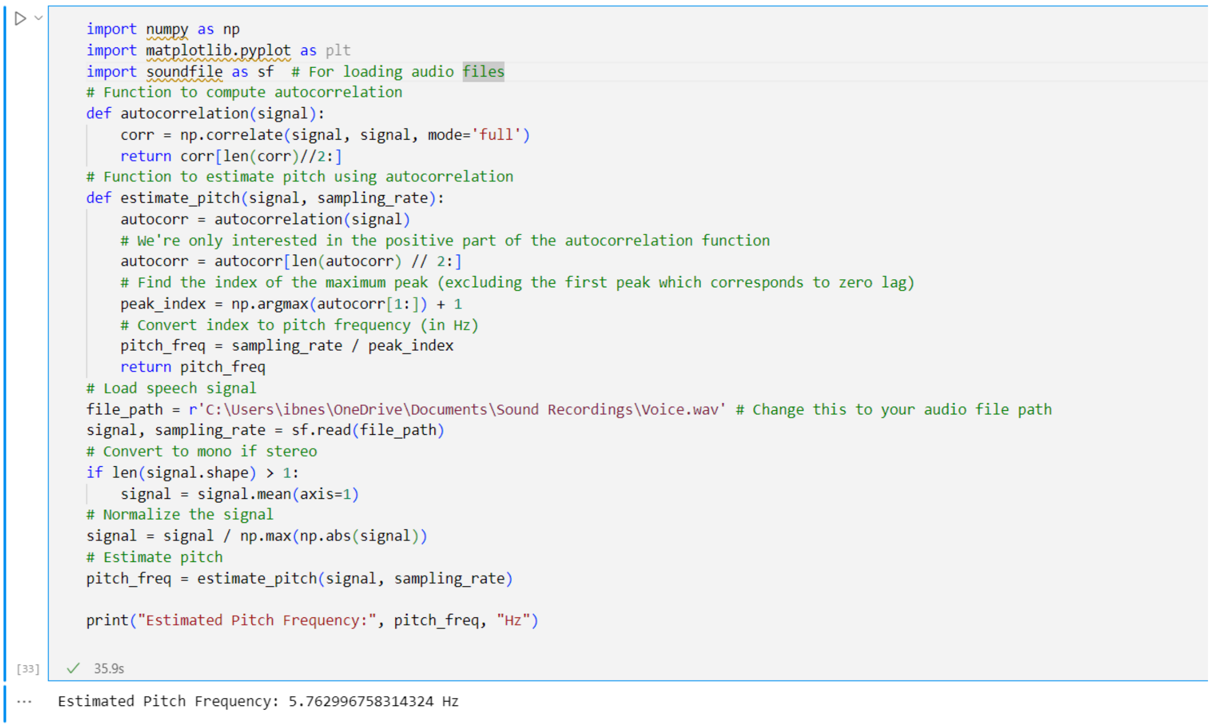
Cepstral Analysis: Cepstral analysis separates pitch-related information from other spectral components in the speech signal. By analyzing the cepstrum, which represents the envelope of the spectrum, pitch can be estimated based on the periodicity of the signal.

Harmonic Summation: This approach involves summing the amplitudes of harmonics in the frequency spectrum of the speech signal. The frequency of the highest peak in the harmonic summation corresponds to the fundamental frequency or pitch.

Time-Domain Methods: Techniques such as zero-crossing rate and peak picking in the waveform directly analyze the speech signal's waveform to identify periodicity and estimate pitch.

Challenges in pitch estimation include background noise, variations in speech production across speakers, and the presence of non-periodic or noisy segments in the signal. Despite these challenges, accurate pitch estimation is crucial for various speech processing tasks, including speech synthesis, voice conversion, and speaker recognition, as well as for music analysis and processing, such as melody extraction and pitch tracking.

Code, Input and Output:



```
[33]
import numpy as np
import matplotlib.pyplot as plt
import soundfile as sf  # For loading audio files
# Function to compute autocorrelation
def autocorrelation(signal):
    corr = np.correlate(signal, signal, mode='full')
    return corr[len(corr)//2:]
# Function to estimate pitch using autocorrelation
def estimate_pitch(signal, sampling_rate):
    autocorr = autocorrelation(signal)
    # We're only interested in the positive part of the autocorrelation function
    autocorr = autocorr[len(autocorr) // 2:]
    # Find the index of the maximum peak (excluding the first peak which corresponds to zero lag)
    peak_index = np.argmax(autocorr[1:]) + 1
    # Convert index to pitch frequency (in Hz)
    pitch_freq = sampling_rate / peak_index
    return pitch_freq
# Load speech signal
file_path = r'C:\Users\ibnes\OneDrive\Documents\Sound Recordings\Voice.wav' # Change this to your audio file path
signal, sampling_rate = sf.read(file_path)
# Convert to mono if stereo
if len(signal.shape) > 1:
    signal = signal.mean(axis=1)
# Normalize the signal
signal = signal / np.max(np.abs(signal))
# Estimate pitch
pitch_freq = estimate_pitch(signal, sampling_rate)

print("Estimated Pitch Frequency:", pitch_freq, "Hz")
```

✓ 35.9s

```
Estimated Pitch Frequency: 5.762996758314324 Hz
```