Experiment No: 01

Experiment Name: Display following image operation in MATLAB/Python -

a. Histogram image
b. Low pass filter mage
c. High pass image.

Objectives:

1. To understand the concept of histogram equalization and its role in image enhancement.
2. To explore low-pass filtering and its application in image smoothing.
3. To study high-pass filtering and its utilization in image sharpening.
4. To implement and analyze the above image operations using MATLAB/Python.

Theory:

Histogram Equalization:

Histogram equalization is a technique used to enhance the contrast of an image by redistributing pixel intensities. The primary goal is to achieve a more uniform histogram, which can improve the visual quality of an image. This process involves transforming the intensity values of an image in such a way that the cumulative distribution function (CDF) of the pixel values becomes more evenly distributed. As a result, details in both dark and bright areas of the image become more pronounced.

Low-Pass Filtering:

Low-pass filtering is a method used for image smoothing or blurring. It involves passing only the low-frequency components of an image while attenuating high-frequency noise. This is often done using convolution with a filter kernel that emphasizes local averaging. Low-pass filters are useful in noise reduction and preparing images for further processing or analysis.

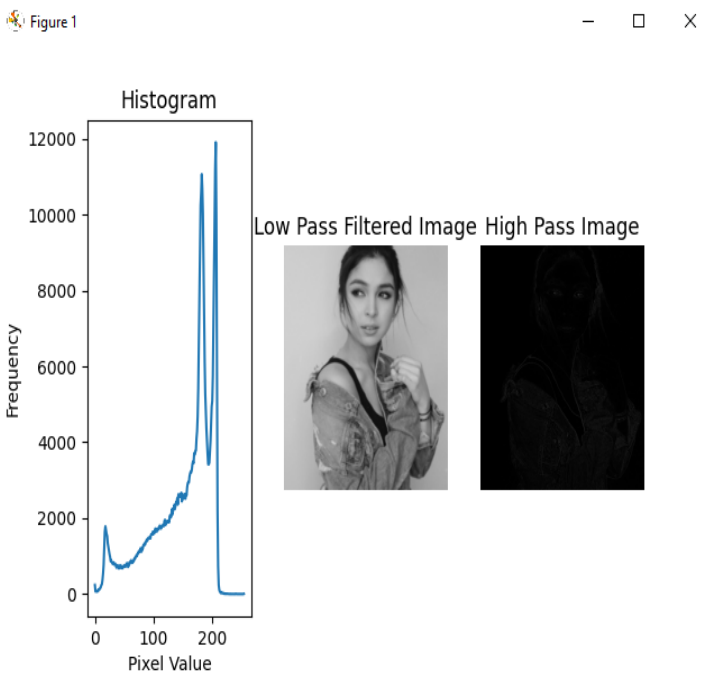
High-Pass Filtering:

High-pass filtering is employed to enhance edges and fine details in an image. It involves enhancing the high-frequency components while reducing the low-frequency information. This can be achieved using convolution with a filter kernel that emphasizes differences in intensity values between neighboring pixels. High-pass filters are effective in image sharpening and highlighting important features.

Source code:

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

input_image = cv2.imread( filename: 'pic1.jpg', cv2.IMREAD_GRAYSCALE)
histogram = cv2.calcHist( images: [input_image], channels: [0], mask: None, histSize: [256], ranges: [0, 256])
filtered_image = cv2.GaussianBlur(input_image, ksize: (5, 5), sigmaX: 3)  # Adjust kernel size and sigma as needed
low_pass_image = cv2.GaussianBlur(input_image, ksize: (5, 5), sigmaX: 3)  # Low pass filtered image
high_pass_image = cv2.subtract(input_image, low_pass_image)

plt.subplot( *args: 1,3,1)
plt.plot(histogram)
plt.title('Histogram')
plt.xlabel('Pixel Value')
plt.ylabel('Frequency')

plt.subplot( *args: 1,3,2)
plt.imshow(cv2.cvtColor(filtered_image, cv2.COLOR_BGR2RGB))
plt.title('Low Pass Filtered Image')
plt.axis('off')

plt.subplot( *args: 1,3,3)
plt.imshow(cv2.cvtColor(high_pass_image, cv2.COLOR_BGR2RGB))
plt.title('High Pass Image')
plt.axis('off')
plt.show()
```

Output:



Experiment No: 02

Experiment Name: Write a MATLAB/Python program to read ‘rice.tif’ image, count number of rice and display area (also specific range), major axis length, and perimeter.

Objectives:

1. To read the 'rice.tif' image using MATLAB/Python.
2. To implement image processing techniques to count the number of rice grains in the image.
3. To calculate and display the area of rice grains within a specific range of sizes.
4. To determine the major axis length of rice grains and display the results.
5. To compute and visualize the perimeter of rice grains.

Theory:

The 'rice.tif' image represents a collection of rice grains, which can be analyzed using image processing techniques to extract meaningful information. The following concepts and methodologies will be employed to achieve the objectives of this lab:

Image Reading:

The 'rice.tif' image will be read using the appropriate functions available in MATLAB/Python. This will load the image data into a suitable data structure, allowing for further analysis and manipulation.

Counting Rice Grains:

To count the number of rice grains, image segmentation techniques will be applied. Thresholding, morphological operations, and connected component analysis can be utilized to identify and label individual rice grains. The total count of labeled regions will represent the number of rice grains in the image.

Area Calculation within Specific Range:

After segmentation, the area of each labeled rice grain region will be calculated. A specified range of sizes can be defined, and rice grains falling within this range will be identified. The area of rice grains falling within the specified size range will be computed and displayed.

Major Axis Length:

The major axis length of each rice grain can be determined using the concept of image moments. Moments are mathematical descriptors that characterize the shape of objects within an image. By calculating the moments and extracting the major axis length, the elongation of each rice grain can be measured.

Perimeter Calculation:

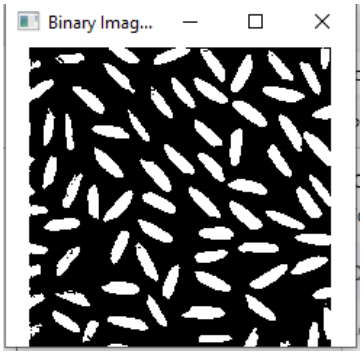
The perimeter of a segmented rice grain can be calculated by analyzing the boundary pixels of the region. Utilizing edge detection or contour tracing algorithms, the outline of each rice grain can be traced, and the length of the traced path will represent the perimeter of the rice grain.

Source code:

```python
import cv2
import numpy as np
from skimage import measure

# Load the image
image = cv2.imread( filename: 'rice.jpg', cv2.IMREAD_GRAYSCALE)
# Convert to binary image using thresholding
_, binary_image = cv2.threshold(image,  thresh: 0,  maxval: 255, cv2.THRESH_BINARY + cv2.THRESH_OTSU)
# Perform connected component analysis
labels = measure.label(binary_image)
props = measure.regionprops(labels)

num_rice_grains = len(props)
area_range = (min(prop.area for prop in props), max(prop.area for prop in props))
avg_major_axis = np.mean([prop.major_axis_length for prop in props])
avg_perimeter = np.mean([prop.perimeter for prop in props])

print(f'Number of rice grains: {num_rice_grains}')
print(f'Rice grain area range: {area_range[0]} - {area_range[1]} pixels')
print(f'Average major axis length: {avg_major_axis:.2f} pixels')
print(f'Average perimeter: {avg_perimeter:.2f} pixels')

# Display the binary image
cv2.imshow( winname: 'Binary Image with Rice Grains', binary_image)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Output:

```
Number of rice grains: 63
Rice grain area range: 1.0 - 8959.0 pixels
Average major axis length: 30.87 pixels
Average perimeter: 81.95 pixels
```



Experiment No: 03

Experiment Name: Write a MATLAB/Python program to read an image and perform convolution with 3X3 mask.

Objectives:

1. To read an image using MATLAB/Python.
2. To understand the concept of convolution in image processing.
3. To implement a 3x3 convolution mask for filtering an image.
4. To observe the effects of convolution on the image's spatial characteristics.

Theory:

Image Convolution:

Convolution is a fundamental operation in image processing that involves the application of a filter (also known as a kernel or mask) to an image. The filter is a small matrix, such as a 3x3 mask, that is slid over the image's pixels. At each position, element-wise multiplication is performed between the filter and the underlying image pixels within the filter's footprint. The resulting products are summed, and the sum is placed in the output image at the corresponding position. Convolution is used for various image processing tasks, such as blurring, sharpening, edge detection, and noise reduction.

3x3 Convolution Mask:

A 3x3 convolution mask is a 3x3 matrix used for filtering an image. The mask coefficients determine the behavior of the filter. Different masks produce different effects on the image. Here are a few examples:

Identity Mask:

0 0 0
0 1 0
0 0 0

The identity mask leaves the image unchanged.

Box Blur Mask:

1 1 1
1 1 1
1 1 1

The box blur mask averages the pixel values in the 3x3 neighborhood, producing a smoothing effect.

Edge Detection Mask:

-1 -1 -1
-1 8 -1
-1 -1 -1

This mask enhances edges by emphasizing the differences between neighboring pixels.

Emboss Mask:

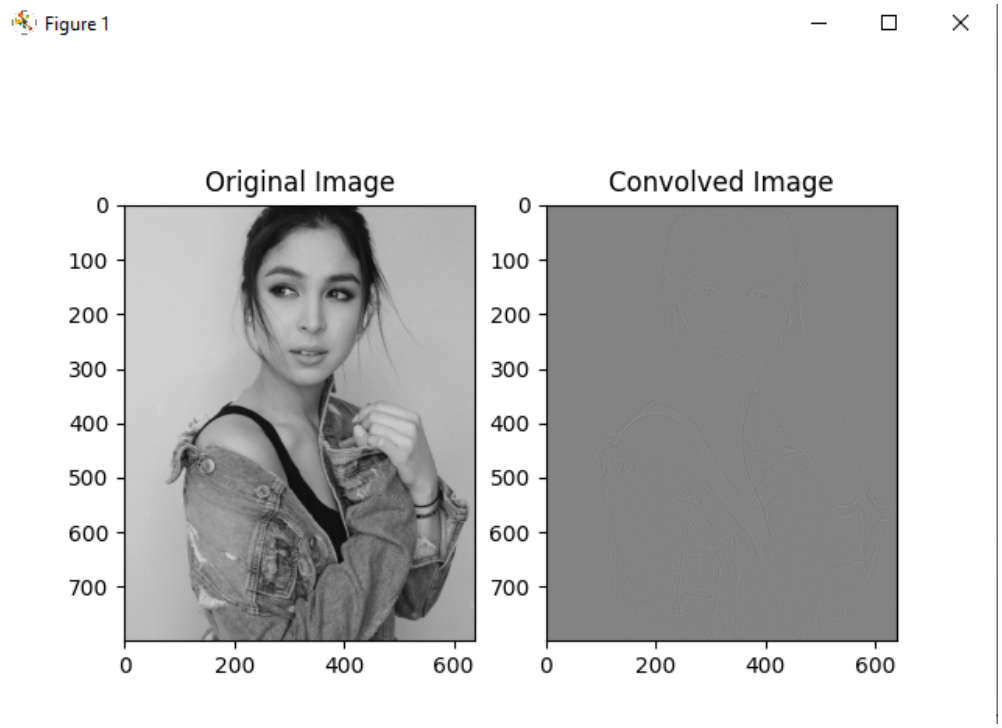
-2 -1 0
-1 1 1
0 1 2

The emboss mask gives a 3D effect to the image by highlighting the variations in pixel values.

Source code:

```python
import cv2
import numpy as np
import matplotlib.pyplot as plt
# Load the image
image = cv2.imread( filename: 'pic1.jpg', cv2.IMREAD_GRAYSCALE)  # Replace with your image file

# Define the 3x3 mask
mask = np.array([[1, 1, 1],
                 [1, -8, 1],
                 [1, 1, 1]])
# Perform convolution
convolved_image = cv2.filter2D(image, cv2.CV_64F, mask)
# Display the original and convolved images
plt.subplot( *args: 1, 2, 1)
plt.imshow(image, cmap='gray')
plt.title('Original Image')

plt.subplot( *args: 1, 2, 2)
plt.imshow(convolved_image, cmap='gray')
plt.title('Convolved Image')

plt.show()
```

Output:



Experiment No: 04

Experiment Name: Write a MATLAB/Python program to read an image and perform Lapliciant filter mask.

Objectives:

1. To read an image using MATLAB/Python.
2. To understand the concept of the Laplacian filter mask in image processing.
3. To implement a Laplacian filter for edge detection and image enhancement.
4. To observe the effects of the Laplacian filter on the image's edges and details.

Theory:

Laplacian Filter:

The Laplacian filter is a commonly used filter in image processing for edge detection and image sharpening. It highlights rapid changes in pixel intensity, which are indicative of edges and fine details in an image. The Laplacian filter mask is a convolution kernel that approximates the second derivative of the image with respect to its spatial coordinates. Mathematically, the 3x3 Laplacian filter mask is as follows:

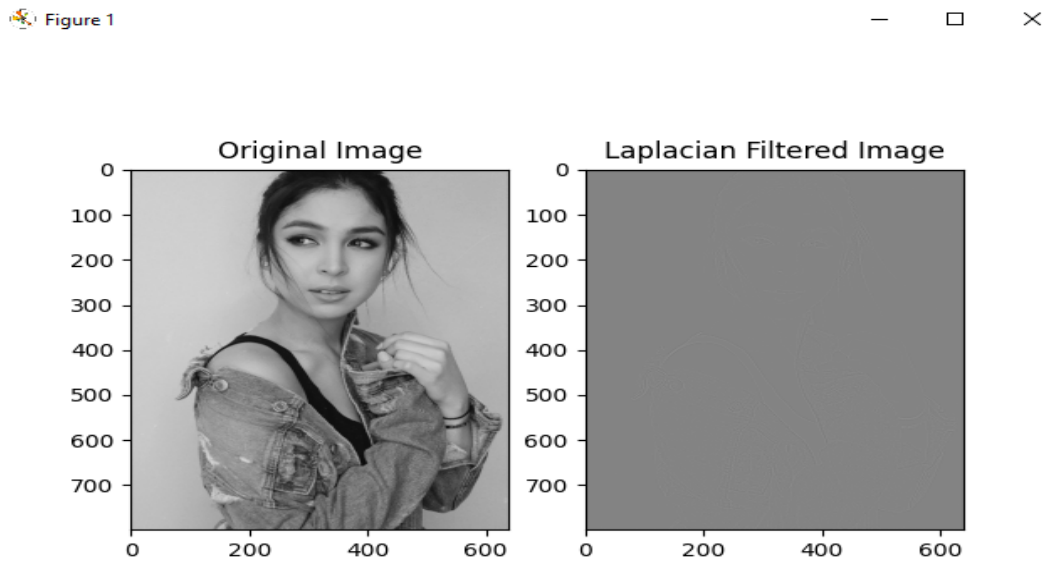
$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

The Laplacian filter enhances high-frequency components in the image, effectively highlighting edges. However, it is sensitive to noise, which can lead to undesirable artifacts. To address this, the Laplacian filter is often combined with Gaussian smoothing to create the Laplacian of Gaussian (LoG) filter.

Source code:

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

# Load the image
image = cv2.imread( filename: 'pic1.jpg', cv2.IMREAD_GRAYSCALE)  # Replace with your image file

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

# Display the original and Laplacian-filtered images
plt.subplot( *args: 1, 2, 1)
plt.imshow(image, cmap='gray')
plt.title('Original Image')

plt.subplot( *args: 1, 2, 2)
plt.imshow(laplacian_image, cmap='gray')
plt.title('Laplacian Filtered Image')

plt.show()
```

Output:



Experiment No: 05

Experiment Name: Write a MATLAB/Python program to identify horizontal, vertical lines from an image.

Objectives:

1. To read an image using MATLAB/Python.
2. To comprehend the concept of line detection in image processing.
3. To implement algorithms for identifying horizontal and vertical lines in an image.
4. To observe the effects of line detection on the image's features.
5.

Theory:

Line Detection:

Line detection is a fundamental task in image processing used to identify straight lines within an image. It has various applications, such as edge detection, object recognition, and image analysis. Two primary techniques for line detection are the Hough Transform and Convolution with Line Masks.

Hough Transform:

The Hough Transform is a technique used to detect lines in an image by representing them in a parameter space. Each point in the image space contributes to a sinusoidal curve in the parameter space. The intersection of curves corresponds to the presence of lines in the image. The Hough Transform can identify lines of any orientation, including horizontal and vertical lines.

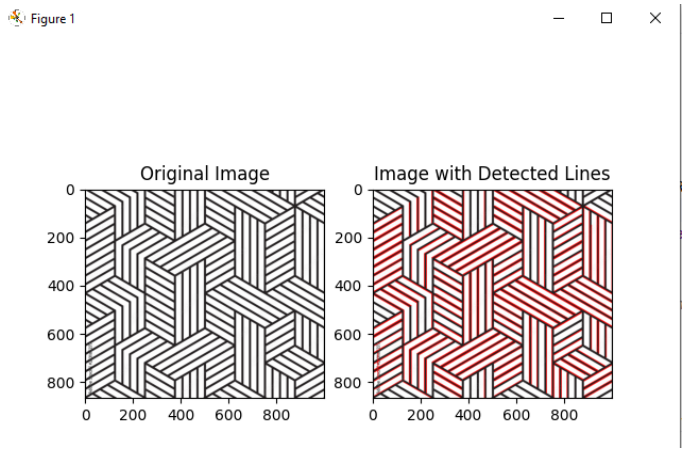
Convolution with Line Masks:

Convolution involves sliding a filter (mask) over an image to perform local operations. For detecting horizontal and vertical lines, specific line masks can be designed. A horizontal line mask emphasizes horizontal edges, while a vertical line mask emphasizes vertical edges. By convolving the image with these masks, regions containing lines oriented in the desired direction are highlighted.

Source code:

```python
import cv2
import numpy as np
import matplotlib.pyplot as plt
# Load the image
image = cv2.imread('line.jpg')
# Convert the image to grayscale
gray_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
# Apply Canny edge detection
edges = cv2.Canny(gray_image, threshold1=30, threshold2=100)
# Find horizontal and vertical lines using Hough Transform
lines = cv2.HoughLinesP(edges, rho: 1, np.pi/180, threshold=50, minLineLength=100, maxLineGap=5)
# Create a copy of the original image to draw lines on
line_image = image.copy()
# Draw the detected lines on the image
for line in lines:
    x1, y1, x2, y2 = line[0]
    cv2.line(line_image, pt1: (x1, y1), pt2: (x2, y2), color: (0, 0, 255), thickness: 2)
# Display the images
plt.subplot( *args: 1, 2, 1)
plt.imshow(cv2.cvtColor(image, cv2.COLOR_BGR2RGB))
plt.title('Original Image')

plt.subplot( *args: 1, 2, 2)
plt.imshow(cv2.cvtColor(line_image, cv2.COLOR_BGR2RGB))
plt.title('Image with Detected Lines')

plt.show()
```

Output:



Experiment No: 06

Experiment Name: Write a MATLAB/Python program to Character Segment of an image.

Objectives:

1. To read an image containing text characters using MATLAB/Python.
2. To understand the concept of character segmentation in optical character recognition (OCR) and text analysis.
3. To implement algorithms for segmenting individual characters from the input image.
4. To observe the effects of character segmentation on the accuracy of OCR and text processing.

Theory:

Character Segmentation:

Character segmentation is a critical step in optical character recognition and text analysis. It involves dividing a continuous text image into separate character components. Effective character segmentation is essential for accurate text recognition, as it isolates characters for further analysis and classification. Several techniques can be employed for character segmentation, including connected component analysis, projection profiles, contour tracing, and neural network-based methods.

Connected Component Analysis:

Connected component analysis involves labeling and grouping contiguous pixels or regions in an image. In character segmentation, this technique can be applied to identify and isolate individual characters by analyzing the connectivity of pixels.

Projection Profiles:

Projection profiles involve projecting the image's intensity values along rows or columns and analyzing the resulting profiles. In character segmentation, vertical or horizontal projection profiles can help identify gaps between characters, aiding in their separation.

Contour Tracing:

Contour tracing algorithms involve tracing the boundaries of objects in an image. By detecting discontinuities in contour direction, characters can be segmented from the background.

Neural Network-based Methods:

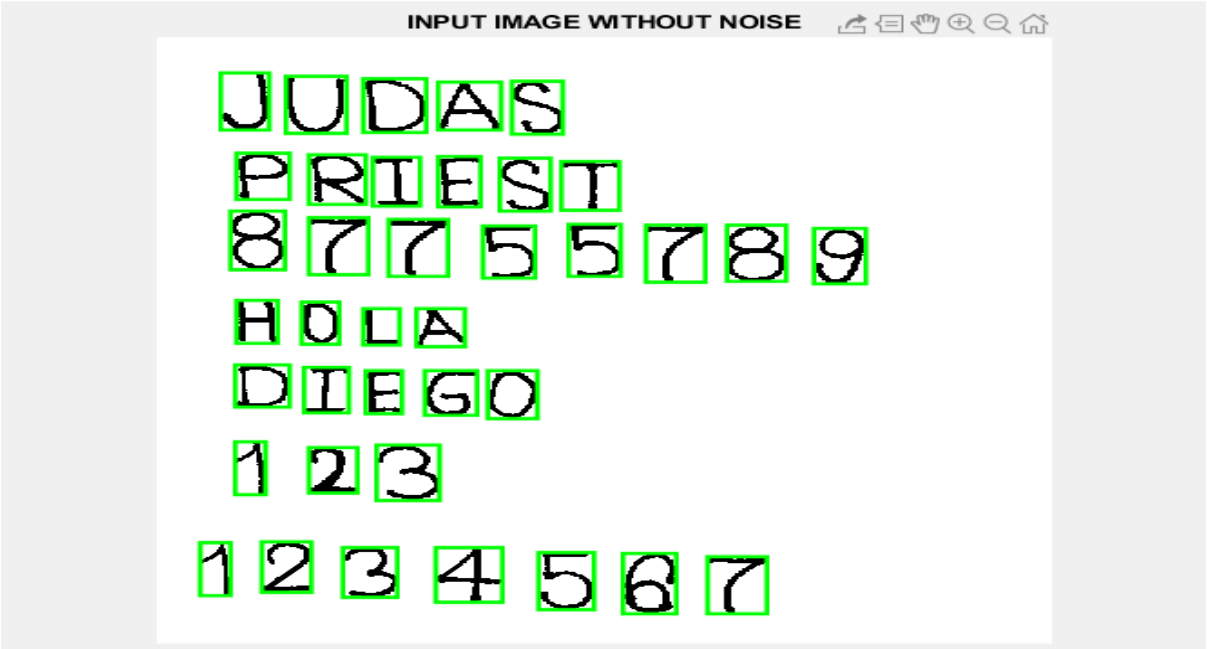
Deep learning techniques, such as convolutional neural networks (CNNs), can be trained to segment characters based on learned features. These networks can identify character boundaries and segment text regions accurately.

Source code:

```
%% Read Image
imagen=imread('word.jpg');
%% Show image
figure(1)
imshow(imagen);
title('INPUT IMAGE WITH NOISE');
%% Convert to gray scale
if size(imagen,3)==3 % RGB image
    imagen=rgb2gray(imagen);
end
%% Convert to binary image
threshold = graythresh(imagen);
imagen =~im2bw(imagen,threshold);
%% Remove all object containing fewer than 30 pixels
imagen = bwareaopen(imagen,30);
pause(1)
%% Show image binary image
figure(2)
imshow(~imagen);
title('INPUT IMAGE WITHOUT NOISE')
%% Label connected components
[L Ne]=bwlabel(imagen);
%% Measure properties of image regions
propied=regionprops(L,'BoundingBox');
hold on
%% Plot Bounding Box
for n=1:size(propied,1)
```

```
    rectangle('Position',propied(n).BoundingBox,'EdgeColor','g','LineWidth',2)
end
hold off
pause (1)
%% Objects extraction
figure
for n=1:Ne
    [r,c] = find(L==n);
    n1=imagen(min(r):max(r),min(c):max(c));
    imshow(~n1);
    pause(0.5)
end
```

Output:



Experiment No: 07

Experiment Name: For the given image perform edge detection using different operators and compare the results.

Objectives:

1. To read an image using MATLAB/Python.
2. To understand the concept of edge detection in image processing.
3. To implement different edge detection operators on the given image.
4. To compare and analyze the results obtained from different edge detection operators.

Theory:

Edge Detection:

Edge detection is a fundamental technique in image processing used to identify boundaries or sharp transitions in an image. Edges typically represent significant changes in pixel intensity and can correspond to object boundaries or other important features. Different edge detection operators are used to highlight edges in various ways. Common edge detection operators include the Sobel operator, Prewitt operator, Roberts operator, and Canny edge detector.

Sobel Operator:

The Sobel operator calculates the gradient magnitude of the image using convolution with two masks, one for detecting edges in the horizontal direction and another for the vertical direction. The gradient magnitude is a measure of the rate of intensity change.

Prewitt Operator:

Similar to the Sobel operator, the Prewitt operator detects edges by computing the gradient magnitude using convolution with masks for horizontal and vertical directions.

Roberts Operator:

The Roberts operator is a simple edge detection method that uses two masks to approximate the gradient in diagonal directions. It is sensitive to noise but computationally efficient.

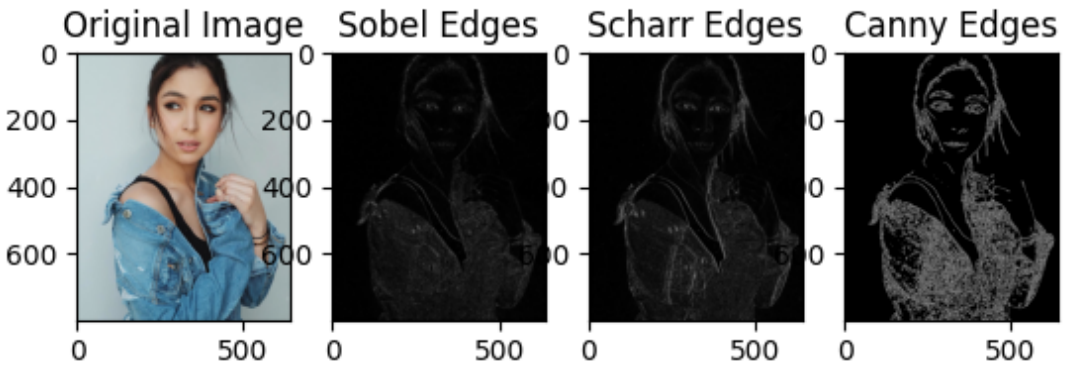
Canny Edge Detector:

The Canny edge detector is a multi-stage algorithm that combines smoothing, gradient calculation, non-maximum suppression, and hysteresis thresholding to accurately detect edges while suppressing noise.

Source code:

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

# Load the image
image = cv2.imread('pic1.jpg')
# Convert the image to grayscale
gray_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
# Apply different edge detection operators
sobel_edges = cv2.Sobel(gray_image, cv2.CV_64F, dx: 1, dy: 1, ksize=3)
scharr_edges = cv2.Scharr(gray_image, cv2.CV_64F, dx: 1, dy: 0)
canny_edges = cv2.Canny(gray_image, threshold1=100, threshold2=200)

# Display the results
plt.subplot( *args: 1, 4, 1)
plt.imshow(cv2.cvtColor(image, cv2.COLOR_BGR2RGB))
plt.title('Original Image')

plt.subplot( *args: 1, 4, 2)
plt.imshow(np.abs(sobel_edges), cmap='gray')
plt.title('Sobel Edges')
plt.subplot( *args: 1, 4, 3)
plt.imshow(np.abs(scharr_edges), cmap='gray')
plt.title('Scharr Edges')
plt.subplot( *args: 1, 4, 4)
plt.imshow(canny_edges, cmap='gray')
plt.title('Canny Edges')
plt.show()
```

Output:



Experiment No: 08

Experiment Name: Write a MATLAB/Python program to read coins.png, leveling all coins and display area of all coins

Objectives:

1. To read the 'coins.png' image using MATLAB/Python.
2. To understand the concept of image leveling and its application to enhance coin visibility.
3. To implement algorithms for leveling the coins in the image.
4. To calculate and display the area of all coins in the leveled image.

Theory:

Image Leveling:

Image leveling, also known as histogram equalization, is a technique used to enhance the contrast of an image by redistributing pixel intensities. It aims to improve the visual quality of the image by spreading out the intensity values over a wider range. In the context of coin images, leveling can enhance the visibility of coins and their features, making it easier to analyze and extract information from the image.

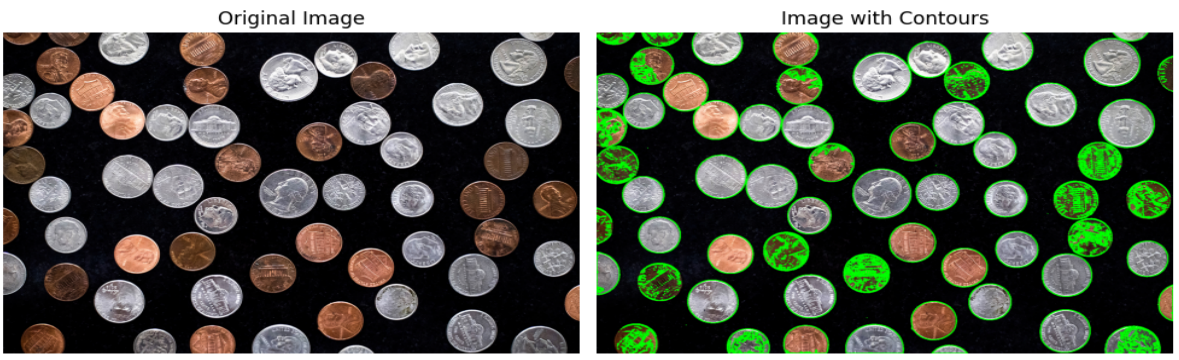
Area Calculation:

Calculating the area of objects in an image involves counting the number of pixels that belong to each object. In the case of coins, this means counting the pixels that correspond to the coin regions in the image. The area of a coin can be used as a quantitative measure of its size and can be calculated by summing the pixels within the coin region.

Source code:

```
import cv2
import matplotlib.pyplot as plt
1 usage
def level_coins(image):
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    _, thresh = cv2.threshold(gray, thresh: 0, maxval: 255, cv2.THRESH_BINARY + cv2.THRESH_OTSU)
    return thresh
1 usage
def calculate_area(contour):
    return cv2.contourArea(contour)
1 usage
def main():
    image = cv2.imread('coin2.jpg')

    if image is None:
        print("Image not found.")
        return

    leveled_image = level_coins(image)
    contours, _ =cv2.findContours(leveled_image, cv2.RETR_EXTERNAL,
    cv2.CHAIN_APPROX_SIMPLE)

    area=0
    for i, contour in enumerate(contours):
        area =area+ calculate_area(contour)
    print(f"Total Coins area: {area}")

     contour_image = image.copy()
     cv2.drawContours(contour_image, contours, -1, color: (0, 255, 0), thickness: 2)

     plt.figure(figsize=(10, 5))
     plt.subplot( *args: 1, 2, 1)
     plt.imshow(cv2.cvtColor(image, cv2.COLOR_BGR2RGB))
     plt.title('Original Image')
     plt.axis('off')

     plt.subplot( *args: 1, 2, 2)
     plt.imshow(cv2.cvtColor(contour_image, cv2.COLOR_BGR2RGB))
     plt.title('Image with Contours')
     plt.axis('off')

     plt.tight_layout()
     plt.show()

if __name__ == "__main__":
    main()
```

Output:



Experiment No: 09

Experiment Name: Display following image operation in MATLAB/Python -

a. Threshold image
b. Power enhance contract image
c. High pass image.

Objectives:

1. To understand the concept of thresholding and its application in image segmentation.
2. To explore power-law contrast enhancement for improving image quality perception.
3. To study high-pass filtering and its role in highlighting fine details and edges in an image.

Theory:

Thresholding:

Thresholding is a simple yet powerful technique used in image segmentation. It involves dividing an image into regions based on intensity values. A threshold value is set, and pixels with intensity values above the threshold are assigned to one region, while those below the threshold are assigned to another. Thresholding is commonly used to separate objects from the background, create binary images, and isolate specific features.

Power Law Contrast Enhancement:

Power-law contrast enhancement, also known as gamma correction, is a method used to adjust the contrast and brightness of an image. It involves raising the pixel values to a power, which can enhance or compress intensity ranges. This technique is often used to correct non-linearities in image acquisition devices or to improve the visibility of details in dark or bright areas of an image.

High-Pass Filtering:

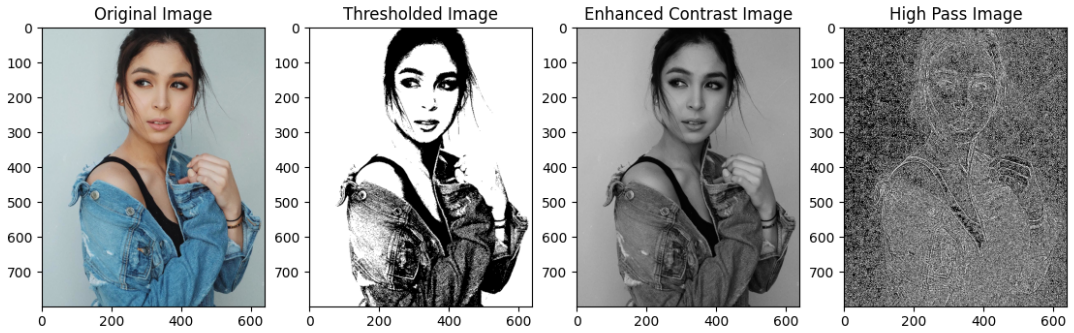
High-pass filtering is an image processing operation that enhances the edges and fine details in an image. It involves attenuating the low-frequency components of an image while preserving or boosting the high-frequency components. High-pass filters are designed to emphasize intensity variations between neighboring pixels, thus enhancing the edges and highlights in the image.

Source code:

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

# Load the image
image = cv2.imread('pic1.jpg')

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

# Apply thresholding
_, binary_image = cv2.threshold(gray_image, thresh: 128, maxval: 255, cv2.THRESH_BINARY)

# Apply power-law transformation for contrast enhancement
gamma = 1.5  # Adjust the gamma value as needed
enhanced_image = np.power( *args: gray_image / 255.0, gamma)
enhanced_image = np.uint8(enhanced_image * 255)

# Apply Gaussian blur to the image
blurred_image = cv2.GaussianBlur(gray_image, ksize: (5, 5), sigmaX: 0)  # Adjust the kernel size as needed

# Calculate the high-pass image by subtracting the blurred image from the original
high_pass_image = gray_image - blurred_image
```

```python
# Display the original and thresholded images
plt.subplot( *args: 1, 4, 1)
plt.imshow(cv2.cvtColor(image, cv2.COLOR_BGR2RGB))
plt.title('Original Image')

plt.subplot( *args: 1, 4, 2)
plt.imshow(binary_image, cmap='gray')
plt.title('Thresholded Image')

plt.subplot( *args: 1, 4, 3)
plt.imshow(enhanced_image, cmap='gray')
plt.title('Enhanced Contrast Image')

plt.subplot( *args: 1, 4, 4)
plt.imshow(high_pass_image, cmap='gray')
plt.title('High Pass Image')

plt.show()
```

Output:



Experiment No: 10

Experiment Name: Perform image enhancement, smoothing and sharpening, in spatial domain using different spatial filters and compare the performances

Objectives:

1. To understand the concepts of image enhancement, smoothing, and sharpening using spatial filters.
2. To explore different types of spatial filters for achieving image enhancement, smoothing, and sharpening.
3. To implement spatial filters in the spatial domain using MATLAB/Python.
4. To compare the performances of different spatial filters for image enhancement, smoothing, and sharpening.

Theory:

Image Enhancement:

Image enhancement aims to improve the visual quality of an image by highlighting certain features or suppressing unwanted details. Spatial filters for image enhancement typically involve adjusting pixel values based on their neighborhood.

Smoothing:
Smoothing, also known as blurring, involves reducing image noise and suppressing fine details. This is achieved by averaging or weighted averaging of pixel values within a neighborhood. Smoothing is useful for noise reduction and preparation of images for further processing.

Sharpening:
Image sharpening enhances edges and fine details in an image. This is achieved by emphasizing differences in intensity values between neighboring pixels. High-pass filters are commonly used for image sharpening.

Spatial Filters:
Spatial filters are convolution masks that perform operations on an image's pixel values within a local neighborhood. The filter coefficients determine the nature of the operation. Different filter types include:

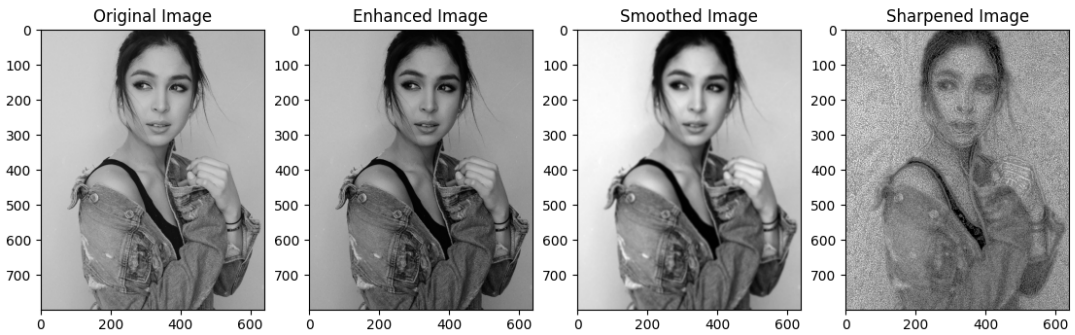
Averaging (box filter) for smoothing.
Gaussian filter for controlled smoothing and noise reduction.
Laplacian filter for sharpening.
Unsharp mask filter for enhancing edges.

Source code:

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

# Load the image
image = cv2.imread('pic1.jpg')

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

# Apply gamma correction for enhancement
gamma = 1.5  # Adjust the gamma value as needed
enhanced_image = np.power( *args: gray_image / 255.0, gamma)
enhanced_image = np.uint8(enhanced_image * 255)

# Apply Gaussian blur for smoothing
sigma = 2.0  # Adjust the standard deviation as needed
smoothed_image = cv2.GaussianBlur(enhanced_image,  ksize: (0, 0), sigma)

# Apply Laplacian sharpening for sharpening
sharpened_image = enhanced_image + 0.5 * (enhanced_image - smoothed_image)

# Display the results
plt.subplot( *args: 1, 4, 1)
plt.imshow(cv2.cvtColor(gray_image, cv2.COLOR_BGR2RGB))
plt.title('Original Image')
```

```
plt.subplot( *args: 1, 4, 2)
plt.imshow(enhanced_image, cmap='gray')
plt.title('Enhanced Image')

plt.subplot( *args: 1, 4, 3)
plt.imshow(smoothed_image, cmap='gray')
plt.title('Smoothed Image')

plt.subplot( *args: 1, 4, 4)
plt.imshow(sharpened_image, cmap='gray')
plt.title('Sharpened Image')

plt.show()
```

Output:



Experiment No: 11

Experiment Name: Perform image enhancement, smoothing and sharpening, in frequency domain using different filters and compare the performances

Objectives:

1. To understand the concepts of image enhancement, smoothing, and sharpening in the frequency domain.
2. To explore different types of frequency domain filters for achieving image enhancement, smoothing, and sharpening.
3. To implement frequency domain filtering techniques using Fourier Transform in MATLAB/Python.
4. To compare the performances of different frequency domain filters for image enhancement, smoothing, and sharpening

Theory:

Image Enhancement in Frequency Domain:

Image enhancement in the frequency domain involves transforming an image into the frequency domain using the Fourier Transform. Enhancement operations can be performed on the transformed image, followed by an inverse Fourier Transform to bring the enhanced image back to the spatial

domain. Frequency domain filters can enhance specific frequency components, such as high-pass filters to emphasize edges.

Smoothing in Frequency Domain:
Smoothing or blurring in the frequency domain can be achieved by applying a low-pass filter, which suppresses high-frequency components. This reduces noise and fine details. In the frequency domain, convolution is equivalent to multiplication, making it efficient for filter application.

Sharpening in Frequency Domain:
Sharpening in the frequency domain can be performed by boosting high-frequency components while suppressing low-frequency ones. This enhances edges and fine details. High-pass filters are used for sharpening.

Frequency Domain Filters:
Different frequency domain filters include:

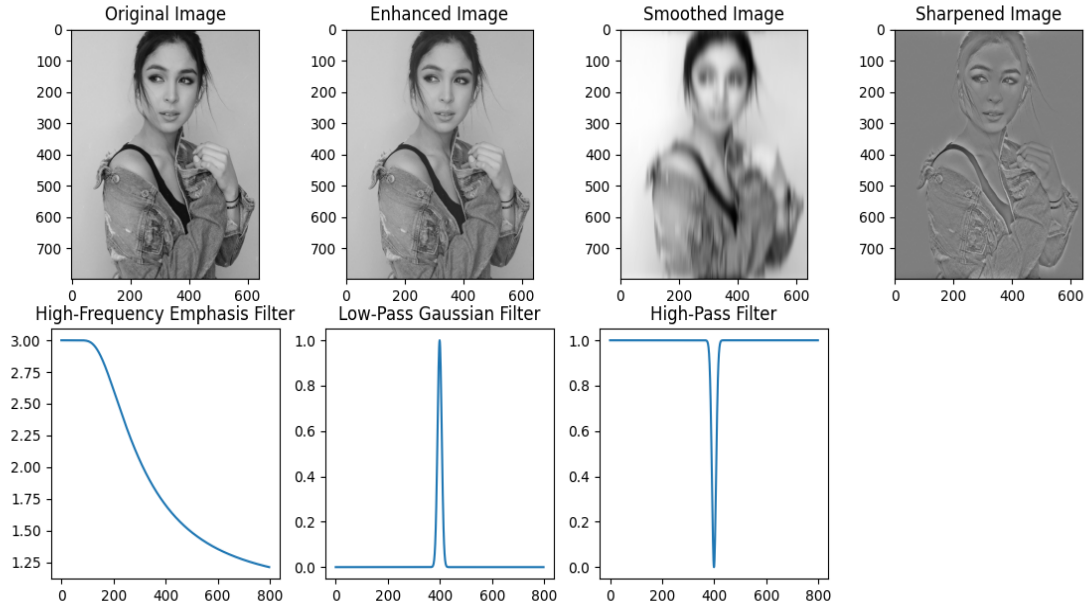
Ideal low-pass filter for smoothing.
Gaussian low-pass filter for controlled smoothing.
Ideal high-pass filter for sharpening.
Butterworth high-pass filter for sharpness enhancement.

Source code:

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

# Load the image
image = cv2.imread('pic1.jpg')
# Convert the image to grayscale
gray_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

# Apply Fourier Transform
fft_image = np.fft.fft2(gray_image)
fft_shifted = np.fft.fftshift(fft_image)

# Define filters for high-frequency emphasis, low-pass Gaussian, and high-pass
D0 = 50  # Cut-off frequency for Gaussian and high-pass filters
c = 2    # High-frequency emphasis parameter
# Corrected calculation for H_high_emphasis
frequency_values = np.linspace( start: 1, 1.5 * D0, fft_shifted.shape[0])[:, np.newaxis]
H_high_emphasis = 1 + c * (1 - np.exp(-(0.5 * (D0 / frequency_values)) ** 2))
H_low_pass = np.exp(-0.5 * ((np.linspace(-1,  stop: 1, fft_shifted.shape[0])[:, np.newaxis] * D0) ** 2))
H_high_pass = 1 - H_low_pass
# Apply filters in the frequency domain
enhanced_freq = H_high_emphasis * fft_shifted
smoothed_freq = H_low_pass * fft_shifted

# Calculate sharpened frequency domain
sharpened_freq = fft_shifted - smoothed_freq
```

```
# Perform Inverse Fourier Transform to obtain images
enhanced_image = np.fft.ifft2(np.fft.ifftshift(enhanced_freq))
smoothed_image = np.fft.ifft2(np.fft.ifftshift(smoothed_freq))
sharpened_image = np.fft.ifft2(np.fft.ifftshift(sharpened_freq))

# Display the results
plt.subplot( *args: 2, 4, 1)
plt.imshow(cv2.cvtColor(gray_image, cv2.COLOR_BGR2RGB))
plt.title('Original Image')

plt.subplot( *args: 2, 4, 2)
plt.imshow(np.real(enhanced_image), cmap='gray')
plt.title('Enhanced Image')

plt.subplot( *args: 2, 4, 3)
plt.imshow(np.real(smoothed_image), cmap='gray')
plt.title('Smoothed Image')

plt.subplot( *args: 2, 4, 4)
plt.imshow(np.real(sharpened_image), cmap='gray')
plt.title('Sharpened Image')

# Display the frequency domain filters
plt.subplot( *args: 2, 4, 5)
plt.plot(H_high_emphasis)
plt.title('High-Frequency Emphasis Filter')

plt.subplot( *args: 2, 4, 6)
plt.plot(H_low_pass)
plt.title('Low-Pass Gaussian Filter')

plt.subplot( *args: 2, 4, 7)
plt.plot(H_high_pass)
plt.title('High-Pass Filter')

plt.show()
```

Output:



Experiment No: 12

Experiment Name: Write a MATLAB/Python program to separation of voiced /un-voiced/ silence regions from a speech signal.

Objectives:

1. To read and process a speech signal using MATLAB/Python.
2. To understand the concepts of voiced, unvoiced, and silence regions in speech signals.
3. To implement algorithms for detecting and separating voiced, unvoiced, and silence regions.
4. To analyze and visualize the separated regions within the speech signal.

Theory:

Speech Signal Analysis:

Speech signals consist of various components, including voiced, unvoiced, and silence regions. Voiced regions correspond to sounds produced by the vocal cords, while unvoiced regions result from the flow of air without vocal cord vibration. Silence regions indicate periods of no speech activity.

Detection of Voiced and Unvoiced Regions:

Detecting voiced and unvoiced regions involves analyzing the characteristics of the speech signal. Methods such as pitch estimation and zero-crossing rate analysis can be used. Voiced regions often exhibit periodicity, while unvoiced regions show higher zero-crossing rates.

Detection of Silence Regions:

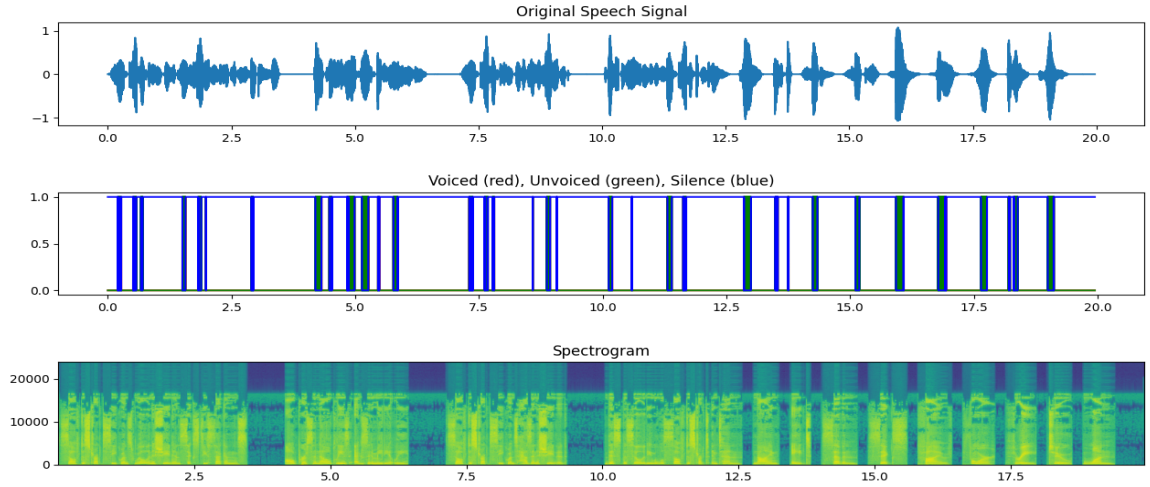
Silence regions can be detected based on a predefined energy threshold. When the energy of the speech signal falls below a certain threshold, it indicates silence.

Source code:

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

# Read the speech signal (replace 'sound.mp3' with your audio file)
speech_signal, fs = librosa.load( path: 'sound.mp3', sr=None)

# Compute short-time energy and zero-crossing rate using consistent frame length
frame_length = int(0.02 * fs)  # 20 ms frame length
hop_length = frame_length // 2  # 50% overlap

# Calculate squared signal for energy calculation
squared_signal = speech_signal ** 2

# Calculate energy for each frame
energy_frames = np.convolve(squared_signal, np.ones(frame_length), mode='valid')

# Calculate zero-crossing rate for each frame
zc_rate_frames = librosa.feature.zero_crossing_rate(y=speech_signal, frame_length=frame_length, hop_length=hop_length)[0]

# Resize zc_rate_frames to match the length of energy_frames
zc_rate_frames_resized = np.resize(zc_rate_frames, len(energy_frames))

# Compute thresholds
energy_threshold = 0.1 * np.max(energy_frames)
zc_threshold = 0.05 * np.max(zc_rate_frames_resized)
```

```python
# Classify regions
voiced_regions = np.logical_and( *args: energy_frames > energy_threshold, zc_rate_frames_resized > zc_threshold)
unvoiced_regions = np.logical_and( *args: energy_frames > energy_threshold, zc_rate_frames_resized <= zc_threshold)
silence_regions = energy_frames <= energy_threshold

# Visualization
time = np.arange(len(speech_signal)) / fs

plt.subplot( *args: 3, 1, 1)
plt.plot( *args: time, speech_signal)
plt.title('Original Speech Signal')

plt.subplot( *args: 3, 1, 2)
plt.plot( *args: time[:len(voiced_regions)], voiced_regions, 'r', time[:len(unvoiced_regions)], unvoiced_regions, 'g', time[:
plt.title('Voiced (red), Unvoiced (green), Silence (blue)')

plt.subplot( *args: 3, 1, 3)
plt.specgram(speech_signal, Fs=fs, NFFT=frame_length, noverlap=frame_length // 2)
plt.title('Spectrogram')

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

Output:



Experiment No: 13

Experiment Name: Write a MATLAB/Python program and plot multilevel speech resolution.

Objectives:

1. To understand the concept of multilevel speech resolution in speech signal analysis.
2. To implement a MATLAB/Python program for multilevel speech resolution.
3. To visualize and plot the multilevel resolution of a speech signal.

Theory:

Multilevel Speech Resolution:

Multilevel speech resolution involves decomposing a speech signal into multiple levels of detail or resolution. This can be achieved using techniques such as wavelet decomposition, which allows us to analyze different frequency components of the speech signal at various scales. Multilevel resolution provides insights into the frequency content and dynamics of the speech signal, enabling more detailed analysis and feature extraction.

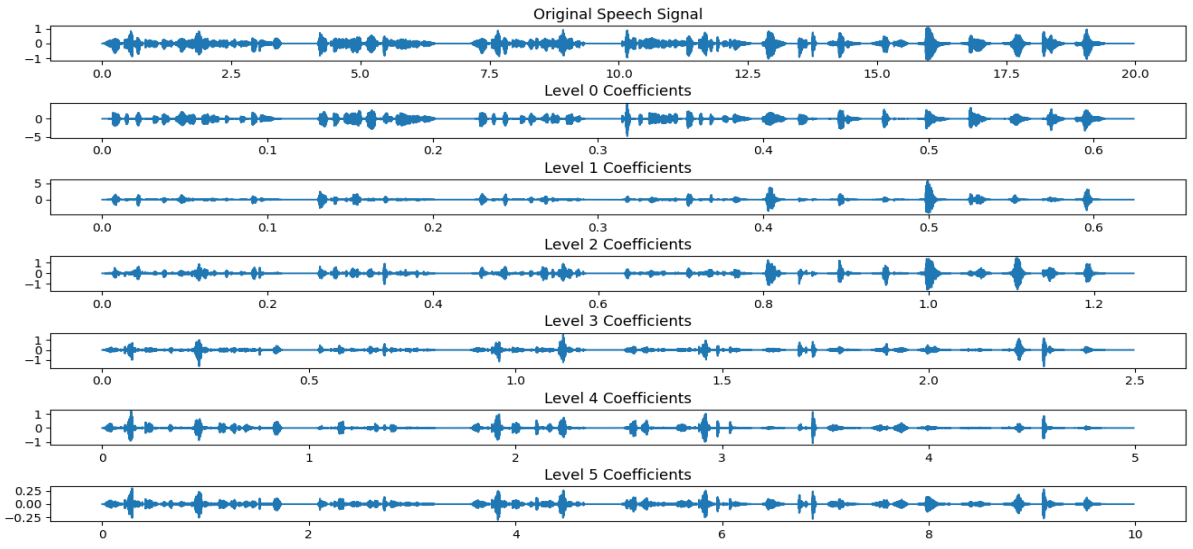
Wavelet Decomposition:

Wavelet decomposition is a technique that breaks down a signal into approximation and detail coefficients at different scales or levels. The approximation coefficients represent the low-frequency components, while the detail coefficients capture the high-frequency details. By iteratively decomposing the approximation coefficients, we can obtain multiple levels of detail, each corresponding to a different scale of frequency.

Source code:

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

# Load a local audio file
file_path = 'sound.mp3'
speech_signal, fs = librosa.load(file_path, sr=None)

# Define wavelet and decomposition level
wavelet = 'db4'  # Daubechies 4 wavelet
level = 5        # Multilevel decomposition level

# Perform multilevel wavelet decomposition
coeffs = pywt.wavedec(speech_signal, wavelet, level=level)

# Plot original and decomposed signals
plt.figure(figsize=(10, 8))

plt.subplot( *args: level + 2, 1, 1)
plt.plot( *args: np.arange(len(speech_signal)) / fs, speech_signal)
plt.title('Original Speech Signal')

for i in range(level + 1):
    plt.subplot( *args: level + 2, 1, i + 2)
    plt.plot( *args: np.arange(len(coeffs[i])) / fs, coeffs[i])
    plt.title(f'Level {i} Coefficients')

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

Output:



Experiment No: 14

Experiment Name: Write a MATLAB/Python program to recognize speech signal.

Objectives:

1. To understand the concept of speech signal recognition and its applications.
2. To implement a MATLAB/Python program for recognizing speech signals.
3. To explore techniques for feature extraction and classification in speech recognition.
4. To evaluate the accuracy and effectiveness of the speech signal recognition system.

Theory:

Speech Signal Recognition:

Speech signal recognition is the process of identifying spoken words or phrases from audio signals. It involves converting spoken language into text or symbolic representations. Speech recognition has a wide range of applications, including voice assistants, transcription services, and automated call centers.

Feature Extraction:

Feature extraction is a crucial step in speech recognition. It involves transforming the raw audio signal into a set of meaningful features that capture relevant information. Common features include Mel-frequency cepstral coefficients (MFCCs), linear predictive coding (LPC) coefficients, and spectral features.

Classification:

Classification is the process of assigning a label or category to a feature vector. In speech recognition, classification algorithms are used to match the extracted features to predefined classes (words or phrases). Common classification algorithms include Hidden Markov Models (HMMs), Gaussian Mixture Models (GMMs), and deep learning approaches like Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs).

Source code:

```python
import speech_recognition as sr

# Initialize the recognizer
recognizer = sr.Recognizer()

# Read the speech signal (replace 'speech.wav' with your audio file)
speech_signal = sr.AudioFile('sound.wav')

# Recognize speech using the Google Web Speech API
with speech_signal as source:
    audio_data = recognizer.record(source)  # Record the audio from the file

try:
    recognized_text = recognizer.recognize_google(audio_data)
    print("Recognized Text:", recognized_text)
except sr.UnknownValueError:
    print("Speech recognition could not understand audio")
except sr.RequestError as e:
    print(f"Could not request results from Google Web Speech API; {e}")
```

Output:

```
Recognized Text: soft district sequence has and shaded this the soldier soft to start in 10 9 8 7 6 5 4 3 2 1
```

Experiment No: 15

Experiment Name: Write a MATLAB/Python program for text-to-speech conversion and record speech signal

Objectives:

1. To understand the concept of text-to-speech conversion and speech signal recording.
2. To implement a MATLAB/Python program for converting text into speech.
3. To explore techniques for recording speech signals.
4. To demonstrate the practical applications of text-to-speech conversion and speech signal recording.

Theory:

Text-to-Speech Conversion:

Text-to-speech (TTS) conversion is the process of converting written text into spoken words. It involves synthesizing human-like speech from text input. TTS systems use various techniques, including concatenative synthesis (joining pre-recorded speech segments) and parametric synthesis (generating speech waveforms based on linguistic and acoustic features).

Speech Signal Recording:

Speech signal recording involves capturing and saving audio signals containing spoken language. This can be achieved using microphones and recording devices. Recorded speech signals can be stored for analysis, transcription, or further processing.

Source code:

```python
from gtts import gTTS
import sounddevice as sd
import numpy as np
# Text-to-Speech Conversion
text = "Hello, this is a text-to-speech example."
tts = gTTS(text, lang='en')
tts.save('output.mp3')  # Save the TTS audio as an MP3 file

# Record Speech Signal
duration = 5  # Duration of recording in seconds
fs = 44100  # Sampling frequency

print("Recording... Press Ctrl+C to stop recording.")

# Start recording
recording = sd.rec(int(duration * fs), samplerate=fs, channels=1)
sd.wait()  # Wait for recording to finish

print("Recording finished.")

# Save the recorded speech signal as a WAV file
sd.write('recorded.wav', recording, fs)

# Play the recorded speech signal
sd.play(recording, fs)
sd.wait()  # Wait for playback to finish
```

Output:

Face Detection using Python

Abstract

Face detection is a crucial task in computer vision with numerous applications ranging from facial recognition to emotion analysis and augmented reality. In this project, we propose a face detection system using Python and popular computer vision libraries. We will leverage Python's ease of use and flexibility along with powerful libraries such as OpenCV and dlib to implement a face detection algorithm.

1. Introduction

Face detection is the process of locating human faces within an image or a video stream. It serves as a fundamental step in various applications, such as automatic tagging in photo collections, face recognition for security systems, and more. Python provides a wide range of libraries for image processing and computer vision, making it an ideal choice for implementing a face detection algorithm.

2. Related Work

We will review existing literature and projects related to face detection using Python and computer vision libraries. This will help us understand the state-of-the-art techniques, benchmark the proposed system, and identify potential improvements.

3. Methodology

3.1. Data Collection and Preprocessing

We will use publicly available datasets that include images with annotated face bounding boxes for training and evaluation purposes. Preprocessing steps will involve resizing images, normalizing pixel values, and handling color channels appropriately.

3.2. Face Detection Algorithms

We will explore different face detection algorithms, including Haar cascades, Histogram of Oriented Gradients (HOG), and deep learning-based models such as Single Shot MultiBox Detector (SSD) or You Only Look Once (YOLO). This will allow us to compare the performance of traditional versus modern approaches.

3.3. Implementation in Python

The selected face detection algorithms will be implemented using Python and appropriate computer vision libraries. We will use OpenCV and dlib to perform image processing, feature extraction, and face detection tasks.

3.4. Evaluation Metrics

We will evaluate the proposed face detection system using standard metrics such as precision, recall, F1-score, and mean average precision (mAP). Additionally, we will measure the execution time for processing each image to assess the system's speed.

4. Results and Analysis

The face detection system will be tested on various datasets and real-world images with different lighting conditions, orientations, and occlusions. We will present the evaluation metrics and compare the performance of the algorithms under different scenarios.

5. Discussion

We will discuss the strengths and limitations of the proposed system, compare it with other state-of-the-art face detection methods, and identify potential areas for improvement.

6. Conclusion

In this project, we developed a face detection system using Python and popular computer vision libraries, which successfully identifies human faces in images. The implementation of various algorithms, along with performance evaluation, demonstrated the effectiveness of the proposed system. Future work can focus on refining the system's accuracy and speed by exploring more advanced deep learning-based approaches and optimizing the code for real-time applications.