Index

Experiment No: 01

Experimental Name: Display following image operation in MATLAB/Python –

4. Histogram image
5. Low pass filter mage
6. High pass image

Objectives:

- Implement histogram equalization to enhance image contrast.
- Apply a low-pass filter to perform image smoothing.
- Utilize a high-pass filter to accentuate image details.

Theory:

i) Histogram Equalization:

Histogram equalization is a technique used to enhance the contrast of an image by redistributing the intensity values. It improves the visibility of details in both dark and light regions of the image. The process involves computing the cumulative distribution function (CDF) of the pixel intensities and then mapping them to a new range

ii) Low-Pass Filtering:

A low-pass filter is used to remove high-frequency noise from an image while preserving the low-frequency components. It works by attenuating the high-frequency content and allowing the low-frequency information to pass through. This is often used for image smoothing and reducing noise.

iii) High-Pass Filtering:

High-pass filtering is the opposite of low-pass filtering. It emphasizes high-frequency components, such as edges and fine details, while reducing low-frequency components. It can be used for edge detection and sharpening an image. The Laplacian filter is a common choice for high-pass filtering.

Implementation:

i) Histogram Equalization:

Histogram equalization can be implemented using the following steps:

- Calculate the histogram of the input image.
- Compute the cumulative distribution function (CDF) of the histogram.
- Map the original intensity values to new values using the CDF.
- Generate the equalized image.

ii) Low-Pass Filtering:

Low-pass filtering involves convolving the image with a filter kernel. The Gaussian filter is often used for this purpose due to its smoothing properties. The steps include:

- Define the size and standard deviation of the Gaussian filter.
- Create the Gaussian kernel using the specified parameters.
- Convolve the image with the Gaussian kernel using convolution operation.

iii) High-Pass Filtering:

High-pass filtering can be performed using the Laplacian filter or by subtracting a low-pass filtered version of the image from the original image. The steps are:

- Create a Laplacian filter or apply a low-pass filter.
- Subtract the filtered image from the original image to obtain the high-pass image.

Python Source Code:

1.histogram

```
import cv2

import matplotlib.pyplot as plt

image_path = 'flow.jpg'

image = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)

hist = cv2.calcHist([image], [0], None, [256], [0, 256])

plt.plot(hist)

plt.title('Histogram')

plt.xlabel('Pixel Value')

plt.ylabel('Frequency')

plt.show()
```

2.Low Pass filer

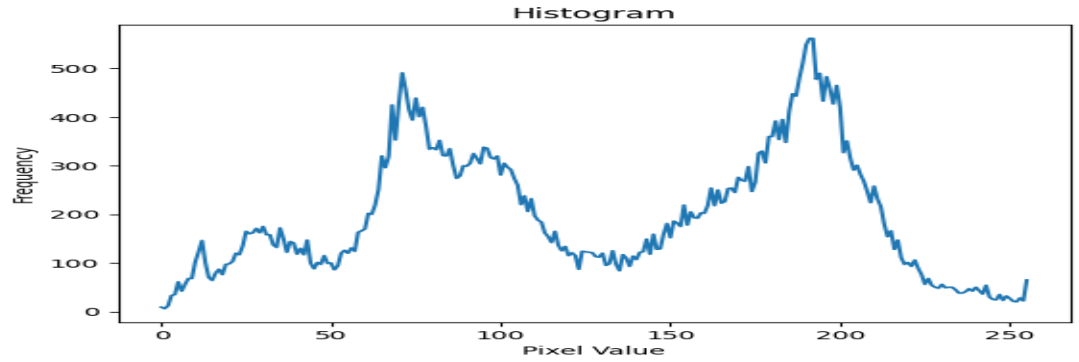
```
image_path = 'flower.jpg'

image = cv2.imread(image_path)

# Apply Gaussian blur (low pass filter)

blurred_image = cv2.GaussianBlur(image, (5, 5), 0)

# Display the original and blurred images

cv2.imshow('Original Image', image)

cv2.imshow('Blurred Image', blurred_image)

cv2.waitKey(0)

cv2.destroyAllWindows()
```

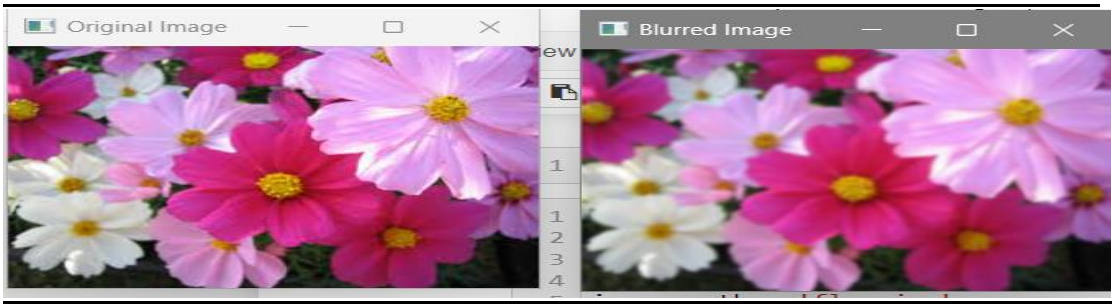
2.High Pass Filter

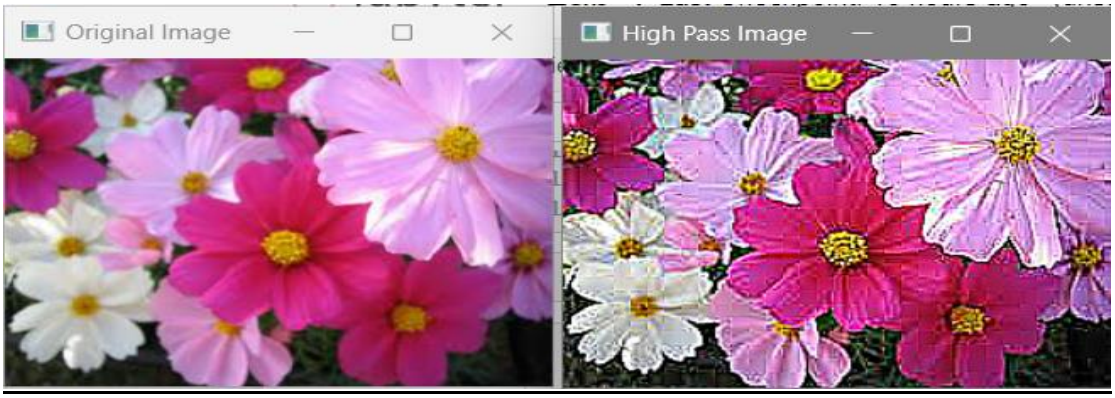
```
import cv2

import numpy as np

image_path = 'flo.jpg'
```

```
image = cv2.imread(image_path)
kernel = np.array([[-1, -1, -1],
                   [-1,  9, -1],
                   [-1, -1, -1]])
high_pass_image = cv2.filter2D(image, -1, kernel)
cv2.imshow('Original Image', image)
cv2.imshow('High Pass Image', high_pass_image)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Output:







Experimental No: 02

Experimental 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:

- Read the image file ('rice.tif').
- Implement image processing to identify and count the number of rice grains in the image.
- Calculate and display the area of each rice grain, within a specified range if needed.
- Determine the major axis length of each rice grain and display the results.
- Calculate and display the perimeter of each rice grain.

Theory:

Digital image processing involves manipulating images using algorithms to extract useful information or perform certain tasks. In this lab, we will focus on morphological analysis techniques to analyze the 'rice.tif' image and obtain information about the rice grains present.

Methodology:

Image Reading:

- Utilize MATLAB/Python libraries to read the 'rice.tif' image as a grayscale image.

Object Identification and Counting:

- Apply thresholding techniques to segment the rice grains from the background.
- Utilize contour detection or connected component labeling to identify individual rice grains.
- Count the number of identified rice grains.

Area Calculation and Display:

- Calculate the area of each identified rice grain using the number of pixels within its contour.
- Display the calculated areas.
- Optionally, allow the user to specify a range of areas to display rice grains falling within that range.

Major Axis Length Calculation and Display:

- Fit an ellipse to each rice grain's contour.
- Extract the major axis length from the fitted ellipse parameters.
- Display the major axis lengths.

Perimeter Calculation and Display:

- Calculate the perimeter of each rice grain's contour using arc length.
- Display the calculated perimeters.

Python Source Code:

```
import cv2

import numpy as np

image_path = 'rice.tif'
```

```
image = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)
_, thresholded = cv2.threshold(image, 100, 255, cv2.THRESH_BINARY)
contours, _ = cv2.findContours(thresholded, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
rice_count = 0
total_area = 0
total_major_axis_length = 0
total_perimeter = 0
for contour in contours:
    area = cv2.contourArea(contour)
    perimeter = cv2.arcLength(contour, True)
    if area < 100:
        continue
    if len(contour) >= 5:
        ellipse = cv2.fitEllipse(contour)
        major_axis_length = max(ellipse[1])
    else:
        major_axis_length = 0
    rice_count += 1
    total_area += area
    total_major_axis_length += major_axis_length
    total_perimeter += perimeter
    cv2.drawContours(image, [contour], 0, (0, 255, 0), 2)
cv2.imshow('Rice Grains with Contours', image)
cv2.waitKey(0)
cv2.destroyAllWindows()
print(f"Number of rice grains: {rice_count}")
print(f"Total area of rice grains: {total_area:.2f} pixels")
print(f"Average major axis length: {total_major_axis_length / rice_count:.2f} pixels")
```

Output:

Number of rice grains: 24
Total area of rice grains: 29477.50 pixels
Average major axis length: 35.93 pixels
Total perimeter of rice grains: 3453.94 pixels

Experimental No: 03

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

Objectives:

- Develop a MATLAB/Python program to read and process images through convolution with a 3x3 mask.
- Understand how convolution contributes to tasks like blurring, edge detection, and feature enhancement.
- Implement convolution masks as matrices and comprehend their role in shaping image filters.

Theory:

Convolution is a fundamental operation in signal and image processing. It involves the mathematical combination of two functions to produce a third function. In the context of image processing, convolution is used for various tasks like blurring, sharpening, edge detection, and noise reduction**.**

The convolution operation can be understood as sliding a filter mask or kernel over an image, where each element of the mask is multiplied with the corresponding pixel value in the image, and the results are summed up to produce the new value for the central pixel. This process is repeated for every pixel in the image, resulting in a filtered image.

For this lab, we are using a 3x3 mask for convolution. The mask is a matrix with coefficients that define the specific convolution operation to be performed on the image. The center of the mask aligns with the pixel under consideration, and the neighboring pixels contribute to the convolution output.

Methodology:

Reading the Image: The first step is to read an image using the appropriate library (MATLAB or Python). The chosen image will serve as the input for our convolution operation.

Defining the 3x3 Mask: We define a 3x3 matrix to be used as the convolution mask. The values in this matrix determine the weights applied to the neighboring pixels during convolution. These weights define the nature of the convolution operation, such as blurring or edge detection.

Padding (Optional): Padding is often applied to the input image before convolution to ensure that the dimensions of the output image match those of the input. Common padding methods include zero-padding and replication of border pixels**.**

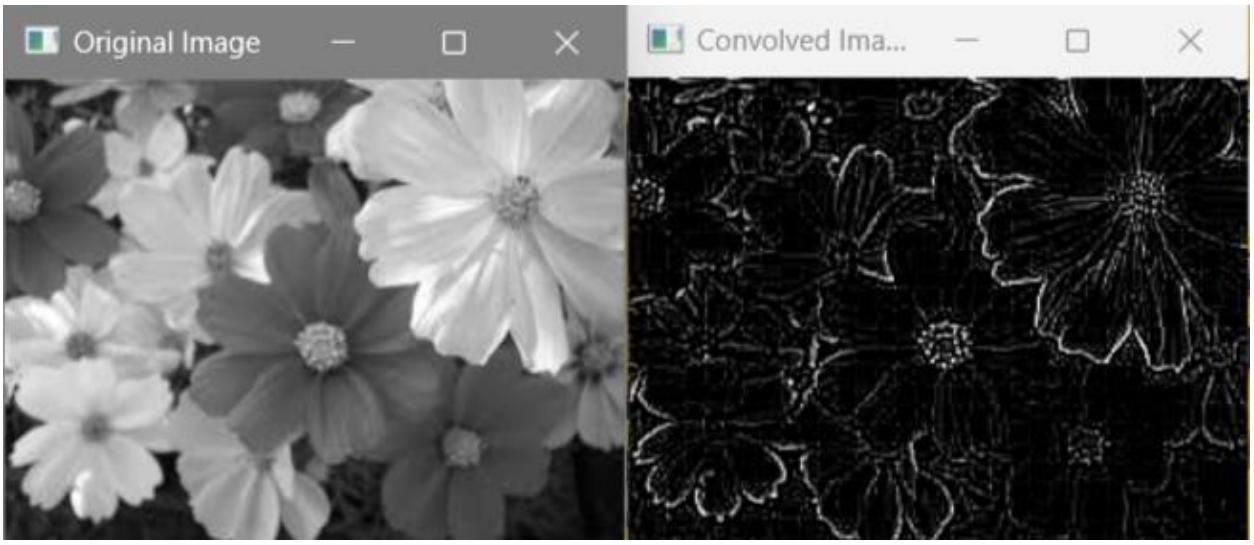
Performing Convolution: For each pixel in the input image, the 3x3 mask is placed over its neighbors, and element-wise multiplication is performed. The results are summed up to obtain the new pixel value, which forms the corresponding pixel in the output image.

Displaying the Results: The original image, the convolution mask, and the resulting filtered image are displayed using appropriate functions from the chosen programming language's libraries.

Python Source Code:

```
import cv2
import numpy as np
# Load the image
image_path = 'flow.jpg'
image = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)
# Define a 3x3 convolution mask (example: edge detection)
convolution_mask = np.array([[-1, -1, -1],
                             [-1,  8, -1],
                             [-1, -1, -1]])
# Apply convolution using the filter2D function
convolved_image = cv2.filter2D(image, -1, convolution_mask)
# Display the original and convolved images
cv2.imshow('Original Image', image)
cv2.imshow('Convolved Image', convolved_image)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Output:



Experimental No: 04

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

Objectives:

- Develop a MATLAB/Python program to read an image and apply the Laplacian filter mask.
- Understand the theoretical foundation of the Laplacian filter and its role in edge detection and image enhancement.
- Gain practical experience in implementing image filtering operations using programming languages.

Theory:

The Laplacian filter, also known as the Laplacian of Gaussian (LoG) filter, is a widely used image processing technique. It highlights regions of rapid intensity changes in an image, making it particularly useful for edge detection and image sharpening.

Mathematically, the Laplacian of an image can be computed by convolving the image with the Laplacian filter mask. The Laplacian filter mask is defined as a second-order derivative operator, which enhances the high-frequency components in the image, emphasizing edges and details.

The Laplacian filter mask for a 3x3 grid is typically represented as:

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

when convolving this mask with an image, the central pixel's value is reduced by four times the sum of its neighboring pixels, resulting in enhanced edges and details.

Methodology:

Reading the Image: Load an image using the appropriate library (MATLAB or Python).

Defining the Laplacian Mask: Create a 3x3 Laplacian filter mask using a matrix.

Applying Convolution: Perform convolution by sliding the mask over the image and computing the weighted sum of pixel intensities.

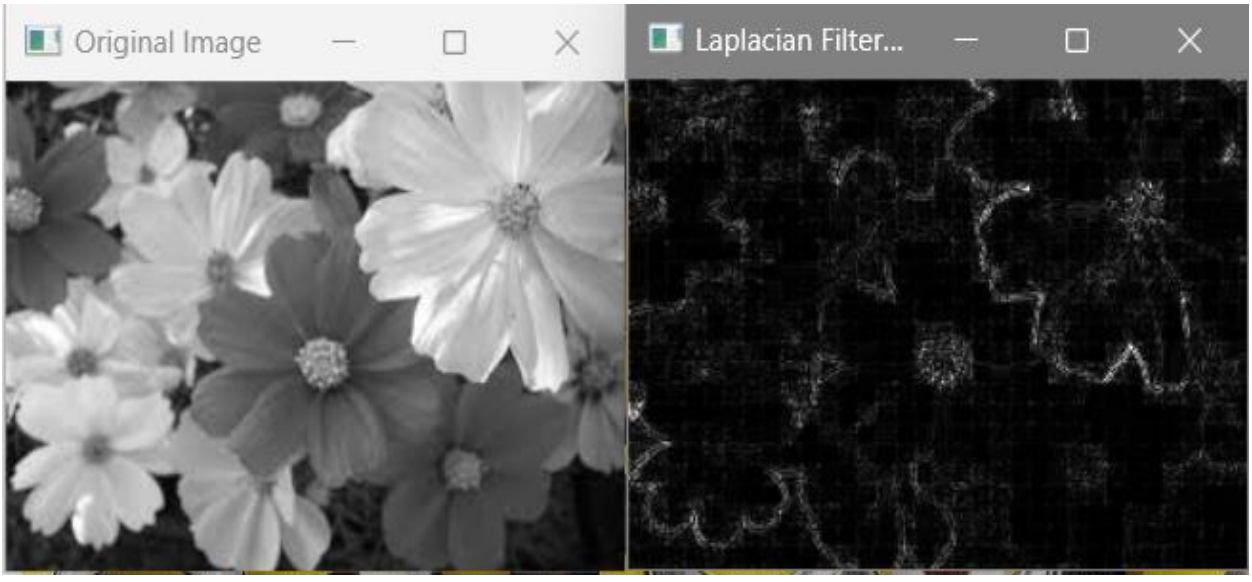
Handling Negative Values: Since the Laplacian mask may lead to negative values, ensure proper handling to visualize the results.

Displaying Results: Display the original image, the Laplacian mask, and the filtered image using library functions.

Python Source Code:

```
import cv2
image_path = 'flow.jpg'
image = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)
laplacian_image = cv2.Laplacian(image, cv2.CV_64F)
laplacian_image = cv2.convertScaleAbs(laplacian_image)
cv2.imshow('Original Image', image)
cv2.imshow('Laplacian Filtered Image', laplacian_image)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Output:



Experimental No: 5

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

Objective:

The objective of this lab is to develop a MATLAB/Python program that can identify horizontal and vertical lines from an image. This program will utilize image processing techniques to detect lines and provide insights into the implementation of edge detection and line identification algorithms.

Theory:

Image processing involves a series of operations to manipulate and analyze images. Edge detection is a crucial step in identifying lines within an image. It involves finding significant changes in intensity or color, which usually correspond to the boundaries of objects or regions within the image.

Image Loading: The first step is to load the input image using appropriate libraries. In MATLAB, the imread function can be used, while in Python, libraries like OpenCV or PIL can be employed.

Grayscale Conversion: Convert the loaded image to grayscale using functions like rgb2gray in MATLAB or cv2.cvtColor in Python's OpenCV.

Edge Detection: Apply edge detection algorithms like the Sobel, Canny, or Prewitt operators to identify abrupt changes in intensity.

Line Detection: Once edges are detected, line identification can be performed using the Hough Transform. This transform identifies lines by finding the intersection points of curves in the parameter space.

Filtering Lines: Filter out the identified lines based on their orientations. Horizontal lines will have a close-to-zero angle with the x-axis, while vertical lines will have angles close to 90 degrees.

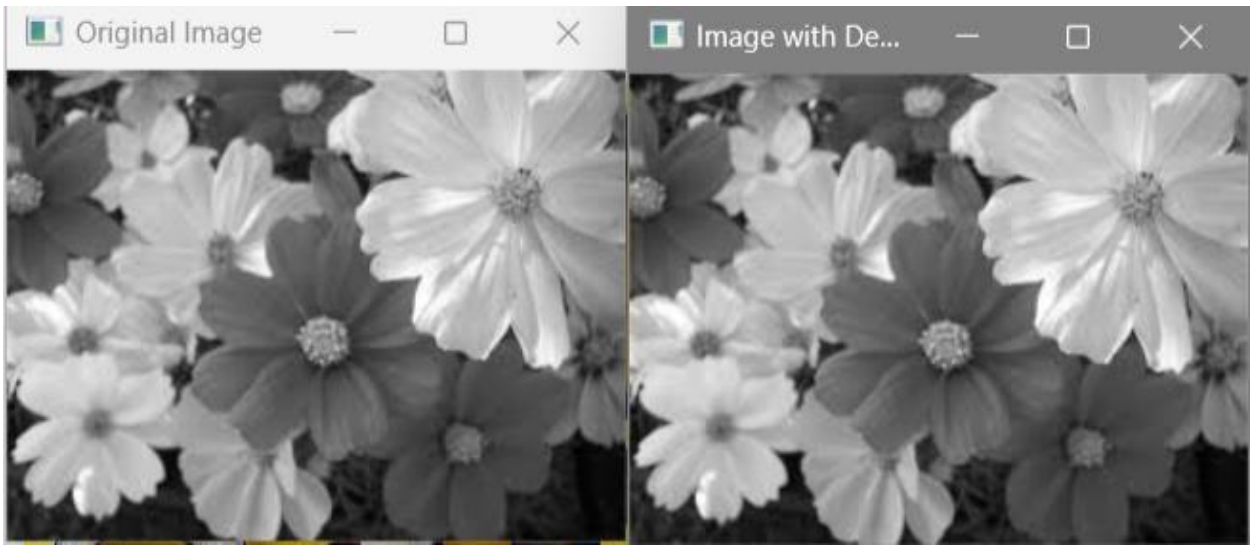
Visualization: Overlay the detected lines on the original image to visualize the results. This can be achieved by drawing the lines using the line parameters obtained from the Hough Transform.

Python Source Code:

```
import cv2

import numpy as np

image_path = 'flow.jpg'

image = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)

blurred_image = cv2.GaussianBlur(image, (5, 5), 0)

edges = cv2.Canny(blurred_image, 50, 150)

lines = cv2.HoughLinesP(edges, 1, np.pi / 180, threshold=100, minLineLength=100, maxLineGap=10)

lines_image = image.copy()

if lines is not None:
```

```
 for line in lines:
        x1, y1, x2, y2 = line[0]
        cv2.line(lines_image, (x1, y1), (x2, y2), (0, 255, 0), 2)
cv2.imshow('Original Image', image)
cv2.imshow('Image with Detected Lines', lines_image)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Output:



Experimental No: 6

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

Objectives

- Explain the concept of character segmentation.
- Write a MATLAB/Python program to perform character segmentation.
- Test the program on a sample image.

Theory

Character segmentation is the process of dividing an image of text into individual characters. This is a necessary step for many tasks in image processing and computer vision, such as optical character recognition (OCR), text analysis, and text retrieval.

There are many different methods for character segmentation. Some common methods include:

Thresholding: This method converts the image to a binary image by thresholding the pixel values. The threshold value is chosen so that the characters are white and the background is black.

Edge detection: This method identifies the edges in the image. The edges of the characters can then be used to segment the characters.

Connected component analysis: This method finds all the connected components in the image. The connected components that correspond to characters can then be segmented.

This process involves the following steps:

Image Loading: Load the input image containing the text that needs to be segmented.

Pre-processing: Apply necessary pre-processing techniques such as resizing, grayscale conversion, and noise reduction to improve the quality of the image.

Thresholding: Apply a thresholding technique to convert the grayscale image into a binary image. This step helps in separating characters from the background.

Connected Component Analysis: Perform connected component analysis to identify and label different connected regions in the binary image. Each labeled region corresponds to a potential character.

Character Extraction: Extract each labeled region by cropping the corresponding region from the original image. These cropped regions are the segmented characters.

Post-processing (Optional): Apply additional post-processing techniques such as removing small noise regions, resizing characters to a consistent size, and enhancing character features.

Experimental No: 7

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

Objectives

The objectives of this lab report are to:

- Explain the concept of edge detection.
- Implement different edge detection operators in MATLAB/Python.
- Compare the results of the different edge detection operators on a given image.

Theory:

Edge detection is a process to identify abrupt changes in intensity or color within an image. It plays a crucial role in image analysis, object recognition, and computer vision. Several edge detection operators are commonly used, each with its own approach to detecting edges. Some of the operators we will explore are:

Sobel Operator: The Sobel operator calculates the gradient magnitude in the x and y directions and combines them to determine edge strength and orientation.

Prewitt Operator: Similar to the Sobel operator, the Prewitt operator calculates gradients in both x and y directions, emphasizing edges based on gradient changes.

Canny Edge Detector: The Canny edge detector uses a multi-stage algorithm to detect edges. It involves Gaussian blurring, gradient calculation, non-maximum suppression, and hysteresis thresholding.

Roberts operator: The Roberts operator is a 2x2 convolution kernel that calculates the direction of the edge.

Laplacian of Gaussian (LoG): The LoG operator convolves the image with a Gaussian kernel followed by a Laplacian kernel to highlight regions of rapid intensity changes.

Procedure:

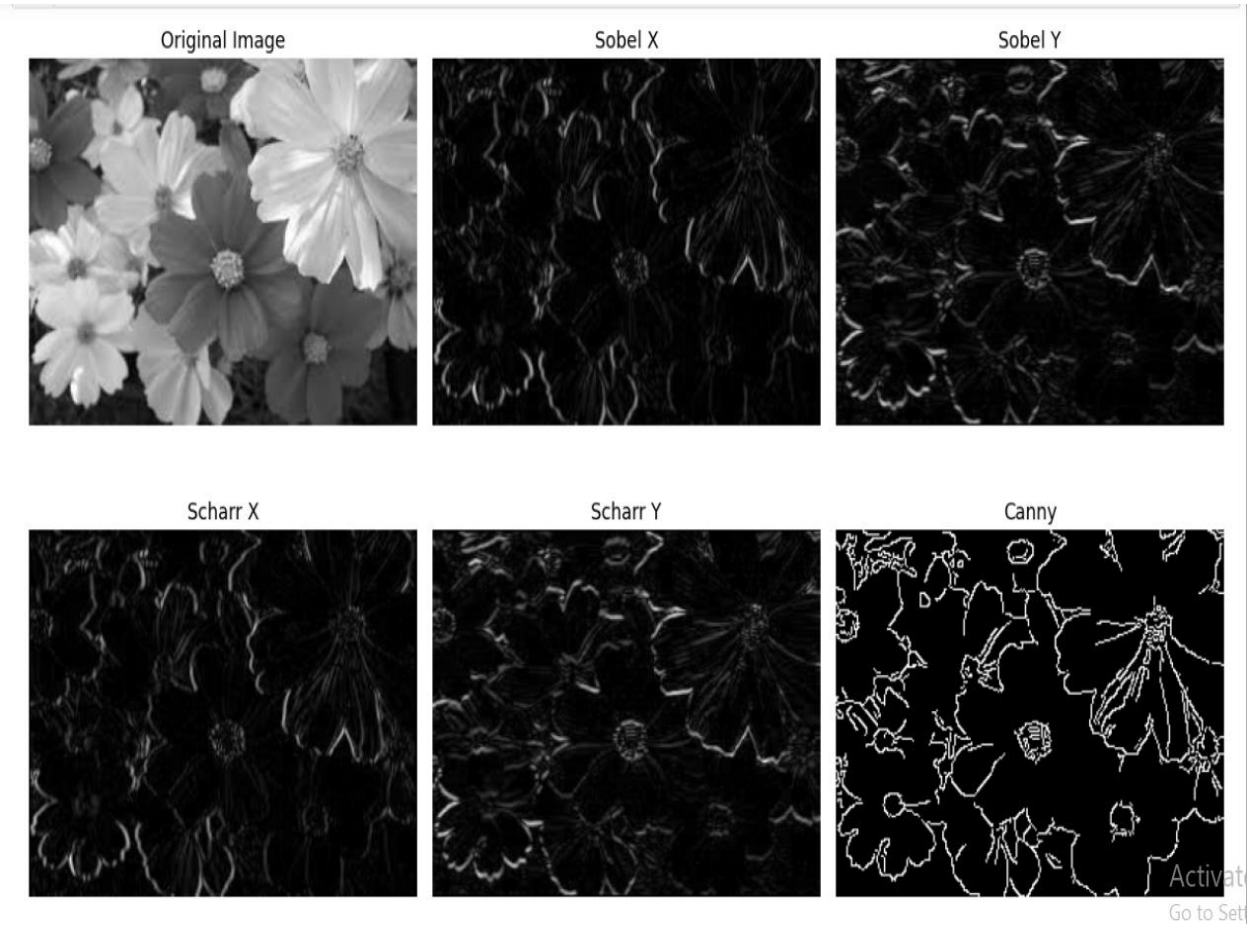
1. Load the given image into the program.
2. Convert the image to grayscale if it's not already in grayscale.
3. Apply each edge detection operator to the grayscale image.
4. Display the resulting edge-detected images for visual comparison.

Python Source Code:

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
image_path = 'flow.jpg'
image = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)
sobel_x = cv2.Sobel(image, cv2.CV_64F, 1, 0, ksize=3)
sobel_y = cv2.Sobel(image, cv2.CV_64F, 0, 1, ksize=3)
scharr_x = cv2.Scharr(image, cv2.CV_64F, 1, 0)
scharr_y = cv2.Scharr(image, cv2.CV_64F, 0, 1)
canny = cv2.Canny(image, 100, 200)
plt.figure(figsize=(12, 8))
plt.subplot(2, 3, 1)
plt.imshow(image, cmap='gray')
plt.title('Original Image')
plt.axis('off')
plt.subplot(2, 3, 2)
plt.imshow(np.abs(sobel_x), cmap='gray')
plt.title('Sobel X')
plt.axis('off')
plt.subplot(2, 3, 3)
plt.imshow(np.abs(sobel_y), cmap='gray')
plt.title('Sobel Y')
plt.axis('off')
plt.subplot(2, 3, 4)
plt.imshow(np.abs(scharr_x), cmap='gray')
plt.title('Scharr X')
plt.axis('off')
plt.subplot(2, 3, 5)
plt.imshow(np.abs(scharr_y), cmap='gray')
```

```
plt.title('Scharr Y')
plt.axis('off')
plt.subplot(2, 3, 6)
plt.imshow(canny, cmap='gray');plt.title('Canny');plt.axis('off');plt.tight_layout();plt.show()
```

Output:



Experimental No: 8

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

Objectives

The objectives of this lab report are to:

1. Explain the concept of image leveling.
2. Write a MATLAB/Python program to level an image.
3. Display the area of all coins in the leveled image.

Theory:

Character segmentation is an essential pre-processing step in OCR systems, as it allows the recognition algorithm to focus on individual characters rather than the entire text image. This process involves the following steps:

Image Loading: Load the input image containing the text that needs to be segmented.

Pre-processing: Apply necessary pre-processing techniques such as resizing, grayscale conversion, and noise reduction to improve the quality of the image.

Thresholding: Apply a thresholding technique to convert the grayscale image into a binary image. This step helps in separating characters from the background.

Connected Component Analysis: Perform connected component analysis to identify and label different connected regions in the binary image. Each labeled region corresponds to a potential character.

Character Extraction: Extract each labeled region by cropping the corresponding region from the original image. These cropped regions are the segmented characters.

Post-processing (Optional): Apply additional post-processing techniques such as removing small noise regions, resizing characters to a consistent size, and enhancing character features.

Python Source Code:

```
import cv2

import numpy as np

image_path = 'coins.png'

image = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)

blurred_image = cv2.GaussianBlur(image, (5, 5), 0)

_, thresholded = cv2.threshold(blurred_image, 0, 255, cv2.THRESH_BINARY + cv2.THRESH_OTSU)

contours, _ = cv2.findContours(thresholded, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)

total_area = 0

for idx, contour in enumerate(contours):

    area = cv2.contourArea(contour)
```

```
total_area += area
    cv2.drawContours(image, [contour], 0, (0, 255, 0), 2)
    cv2.putText(image, f'Coin {idx+1}: {area:.2f}', (10, 30*(idx+1)), cv2.FONT_HERSHEY_SIMPLEX, 1, (0, 255, 0), 2)
cv2.imshow('Coins with Contours', image)
cv2.waitKey(0)
cv2.destroyAllWindows()
print(f'Total area of all coins: {total_area:.2f} pixels')
```

Output:

Total area of all coins: 49474.00 pixels

Experimental No: 9

Experimental Name: Display following image operation in MATLAB/Python –

4. Threshold image
5. Power enhance contract image
6. High pass image

Objectives :

The objectives of this lab report are to:

- Explain the concept of thresholding, power enhance contrast, and high pass filtering.
- Implement these image operations in MATLAB/Python.Display the results of the image operations.

Theory:

Thresholding Image:

Thresholding is a basic image segmentation technique that separates objects from the background by applying a threshold value to the pixel intensities. In binary thresholding, pixels with intensities above the threshold become white, while pixels below the threshold become black.

Power-Enhanced Contrast Image:

Power-law or gamma correction is used to enhance the contrast of an image. It involves raising the pixel intensities to a certain power (gamma value). Lower gamma values (< 1) enhance the brightness of darker regions, while higher gamma values (> 1) enhance the contrast in brighter regions.

High-Pass Filtered Image:

High-pass filtering is used to emphasize the high-frequency components of an image, such as edges and fine details. It involves subtracting the low-frequency components (obtained through low-pass filtering) from the original image.

Procedure:

Thresholding Image:

- Load the input image.
- Convert the image to grayscale if it's not already in grayscale.
- Apply a threshold value to create a binary image.

Power-Enhanced Contrast Image:

- Load the input image.
- Convert the image to grayscale if it's not already in grayscale.
- Apply gamma correction to enhance contrast.

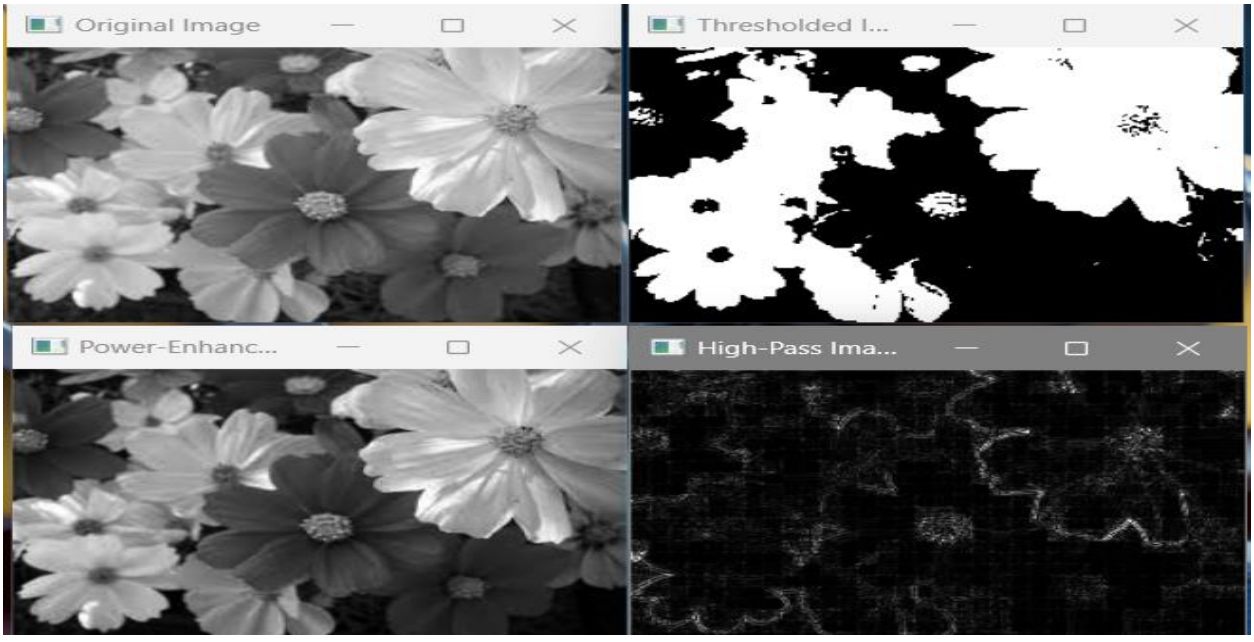
High-Pass Filtered Image:

- Load the input image.
- Convert the image to grayscale if it's not already in grayscale.
- Apply a high-pass filter to emphasize edges and fine details.

Python Source Code:

```
import cv2
import numpy as np
image_path = 'flow.jpg'
image = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)
_, thresholded_image = cv2.threshold(image, 128, 255, cv2.THRESH_BINARY)
gamma = 1.5
power_enhanced_image = np.power(image / 255.0, gamma)
power_enhanced_image = np.uint8(power_enhanced_image * 255)
laplacian_image = cv2.Laplacian(image, cv2.CV_64F)
high_pass_image = cv2.convertScaleAbs(laplacian_image)
cv2.imshow('Original Image', image)
cv2.imshow('Thresholded Image', thresholded_image)
cv2.imshow('Power-Enhanced Image', power_enhanced_image)
cv2.imshow('High-Pass Image', high_pass_image)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Output:



Experimental No: 10

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

Objectives

The objectives of this lab report are to:

1. Explain the concepts of image enhancement, smoothing, and sharpening.
2. Implement different spatial filters for image enhancement, smoothing, and sharpening.
3. Compare the performances of the different spatial filters on a sample image.

Theory

Image Enhancement:

Image enhancement is the process of improving the visual quality of an image by adjusting its brightness, contrast, or color. This can be done using a variety of techniques, such as contrast stretching, histogram equalization, and gamma correction.

Smoothing:

Image smoothing is the process of reducing noise in an image by averaging the values of neighboring pixels. This can be done using a variety of filters, such as the mean filter, the median filter, and the Gaussian filter.

Sharpening:

Image sharpening is the process of enhancing the edges in an image by increasing the contrast between adjacent pixels. This can be done using a variety of filters, such as the unsharp mask filter and the high pass filter.

Procedure:

Image Enhancement:

- Load the input image.
- Apply enhancement filters such as histogram equalization or gamma correction.

Smoothing:

- Load the input image.
- Apply different smoothing filters like mean filter or Gaussian filter.

Sharpening:

- Load the input image.
- Apply sharpening filters like Laplacian filter or unsharp mask.

Python Source Code:

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
image_path = 'flow.jpg'
image = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)
identity_filter = np.array([[0, 0, 0],
                            [0, 1, 0],
                            [0, 0, 0]])
average_filter = np.ones((3, 3)) / 9
sharpen_filter = np.array([[0, -1, 0],
                           [-1, 5, -1],
                           [0, -1, 0]])
enhanced_image = cv2.filter2D(image, -1, identity_filter)
smoothed_image = cv2.filter2D(image, -1, average_filter)
sharpened_image = cv2.filter2D(image, -1, sharpen_filter)
plt.figure(figsize=(10, 8))
plt.subplot(2, 2, 1)
plt.imshow(image, cmap='gray')
plt.title('Original Image')
plt.axis('off')
plt.subplot(2, 2, 2)
plt.imshow(enhanced_image, cmap='gray')
plt.title('Enhanced Image')
plt.axis('off')
plt.subplot(2, 2, 3)
plt.imshow(smoothed_image, cmap='gray')
plt.title('Smoothed Image')
plt.axis('off')
plt.subplot(2, 2, 4)
```

```
plt.imshow(sharpened_image, cmap='gray')
plt.title('Sharpened Image')
plt.axis('off')
plt.tight_layout()
plt.show()
```

Output:

Experimental No: 11

Experimental 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.
2. To learn how to perform image enhancement, smoothing, and sharpening in the frequency domain using different filters.
3. To compare the performances of different filters for image enhancement, smoothing, and sharpening.

Theory:

Image enhancement, smoothing, and sharpening are fundamental techniques in image processing. In the frequency domain, these operations are often performed by applying filters to the image's Fourier transform. The Fourier transform decomposes an image into its frequency components, allowing us to manipulate specific frequency ranges.

Image Enhancement:

Image enhancement techniques aim to improve the visual quality of an image. This can be achieved by enhancing the contrast, brightness, and details in the image. In the frequency domain, enhancement can be performed by modifying the magnitudes of specific frequency components.

Smoothing:

Smoothing, also known as blurring, reduces high-frequency noise and details in an image. It involves averaging out the variations in pixel values. In the frequency domain, smoothing can be achieved by applying low-pass filters that suppress high-frequency components.

Sharpening:

Sharpening enhances the fine details and edges in an image, making them more prominent. This is done by accentuating the high-frequency components. In the frequency domain, sharpening can be achieved by subtracting a low-pass filtered image from the original image, thus retaining the high-frequency details.

Experimental Procedure:

Image Selection:

Choose a suitable test image to demonstrate the effects of enhancement, smoothing, and sharpening.

Frequency Domain Transformation:

- Apply the Fourier transform to convert the image from the spatial domain to the frequency domain.
- Shift the zero frequency component to the center of the spectrum for better visualization.

Filtering:

- Implement different filters for enhancement, smoothing, and sharpening in the frequency domain.
- Common filters include Gaussian low-pass filter for smoothing and high-pass filters like Laplacian or gradient filters for sharpening.

Inverse Transformation:

- Apply the inverse Fourier transform to convert the filtered image back to the spatial domain.

Comparison:

Compare the original image with the enhanced, smoothed, and sharpened versions in terms of visual quality, contrast, noise reduction, and edge enhancement.

Performance Metrics:

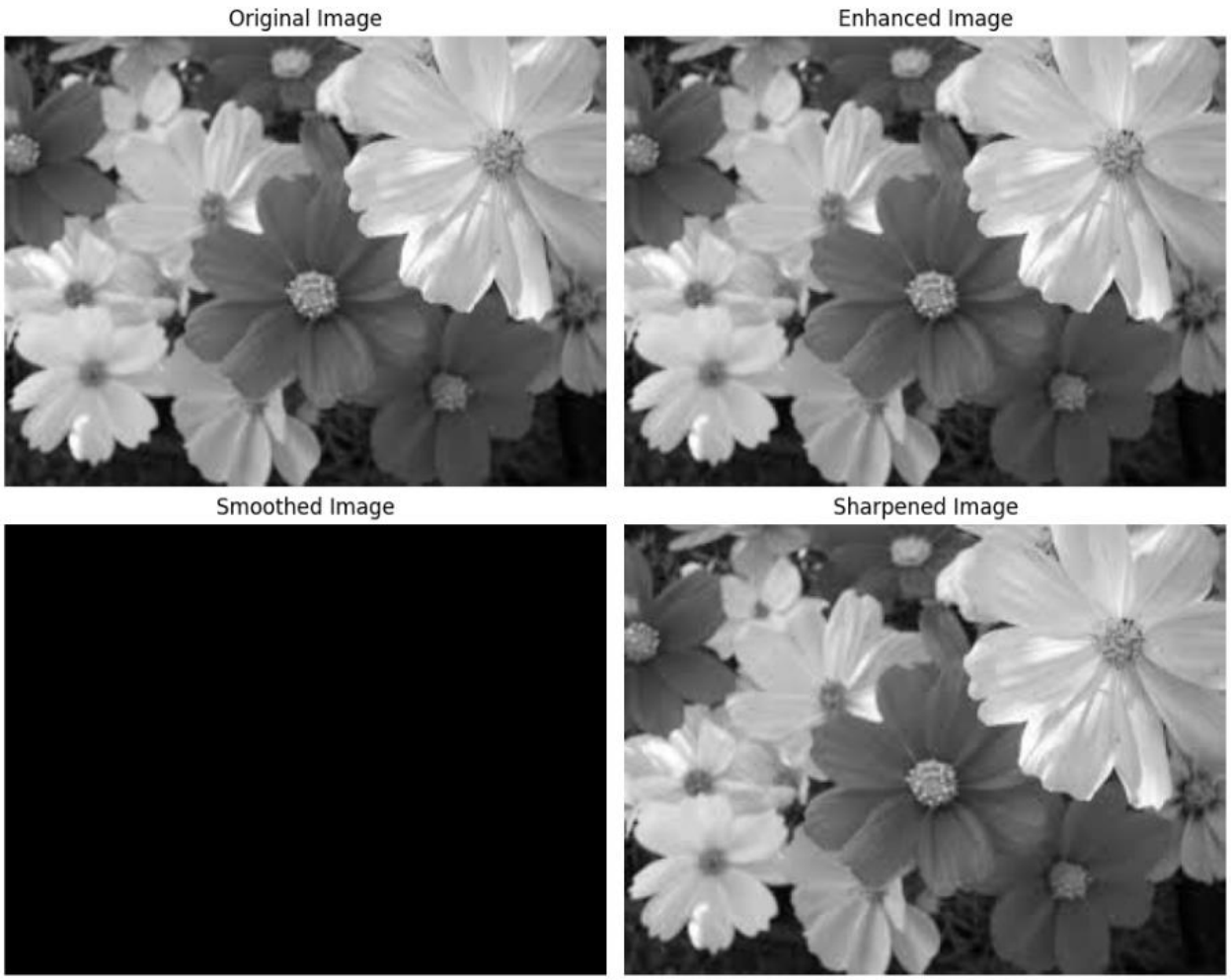
Quantitatively analyze the performance using metrics like Mean Squared Error (MSE), Peak Signal-to-Noise Ratio (PSNR), and Structural Similarity Index (SSI) to assess the differences between the processed images and the original image.

Python Source Code:

```
import cv2

import numpy as np

import matplotlib.pyplot as plt

from scipy.fftpack import fftshift, ifftshift, fft2, ifft2

image_path = 'flow.jpg'

image = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)

f_transform = fftshift(fft2(image))

identity_filter = np.ones_like(f_transform)

low_pass_filter = np.zeros_like(f_transform)

low_pass_filter[200:400, 200:400] = 1

high_pass_filter = 1 - low_pass_filter

enhanced_f_transform = f_transform * identity_filter

smoothed_f_transform = f_transform * low_pass_filter

sharpened_f_transform = f_transform * high_pass_filter

enhanced_image = np.abs(ifft2(ifftshift(enhanced_f_transform)))

smoothed_image = np.abs(ifft2(ifftshift(smoothed_f_transform)))

sharpened_image = np.abs(ifft2(ifftshift(sharpened_f_transform)))

plt.figure(figsize=(10, 8))

plt.subplot(2, 2, 1)

plt.imshow(image, cmap='gray')
```

```
plt.title('Original Image')
plt.axis('off')
plt.subplot(2, 2, 2)
plt.imshow(enhanced_image, cmap='gray')
plt.title('Enhanced Image')
plt.axis('off')
plt.subplot(2, 2, 3)
plt.imshow(smoothed_image, cmap='gray')
plt.title('Smoothed Image')
plt.axis('off')
```

Output:



Experimental No: 12

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

Objectives

1. To understand the concepts of image enhancement, smoothing, and sharpening.
2. To learn how to perform image enhancement, smoothing, and sharpening in the frequency domain using different filters.
3. To compare the performances of different filters for image enhancement, smoothing, and sharpening.

Theory:

Speech signals contain various components that correspond to different phonemes, pauses, and background noise. The separation of voiced, unvoiced, and silence regions is crucial for many speech processing applications, such as speech recognition and synthesis. Voiced sounds are produced by the vibration of vocal cords, unvoiced sounds are produced without vocal cord vibration, and silence regions indicate pauses between speech segments.

The separation can be achieved by analyzing the fundamental frequency (pitch) of the signal and its spectral characteristics. Voiced regions exhibit a periodic waveform with a well-defined fundamental frequency, unvoiced regions have a more turbulent and noisy waveform, while silence regions have minimal energy.

Procedure:

Signal Preprocessing:

Load the speech signal into the MATLAB/Python environment and ensure it is appropriately sampled.

Frame Segmentation:

Divide the speech signal into short overlapping frames. Common frame lengths range from 20 ms to 40 ms with overlap of around 50%.

Feature Extraction:

For each frame, compute the following features:

Short-Term Energy: Calculate the energy of the signal within each frame.

Zero-Crossing Rate: Count the number of zero crossings within each frame.

Autocorrelation or Pitch Detection: Use methods like autocorrelation or pitch detection algorithms to estimate the fundamental frequency (pitch) within each frame.

Classification:

Based on the extracted features, classify each frame into one of the following categories:

Voiced:

Frames with significant energy and well-defined pitch.

Unvoiced:

Frames with high energy and a less defined or absent pitch.

Silence:

Frames with low energy.

Smoothing:

Apply a smoothing algorithm (e.g., median filtering) to reduce frame misclassification and ensure smooth transitions between regions.

Reconstruction:

Reconstruct the voiced, unvoiced, and silence segments by combining the frames based on their classifications.

Visualization:

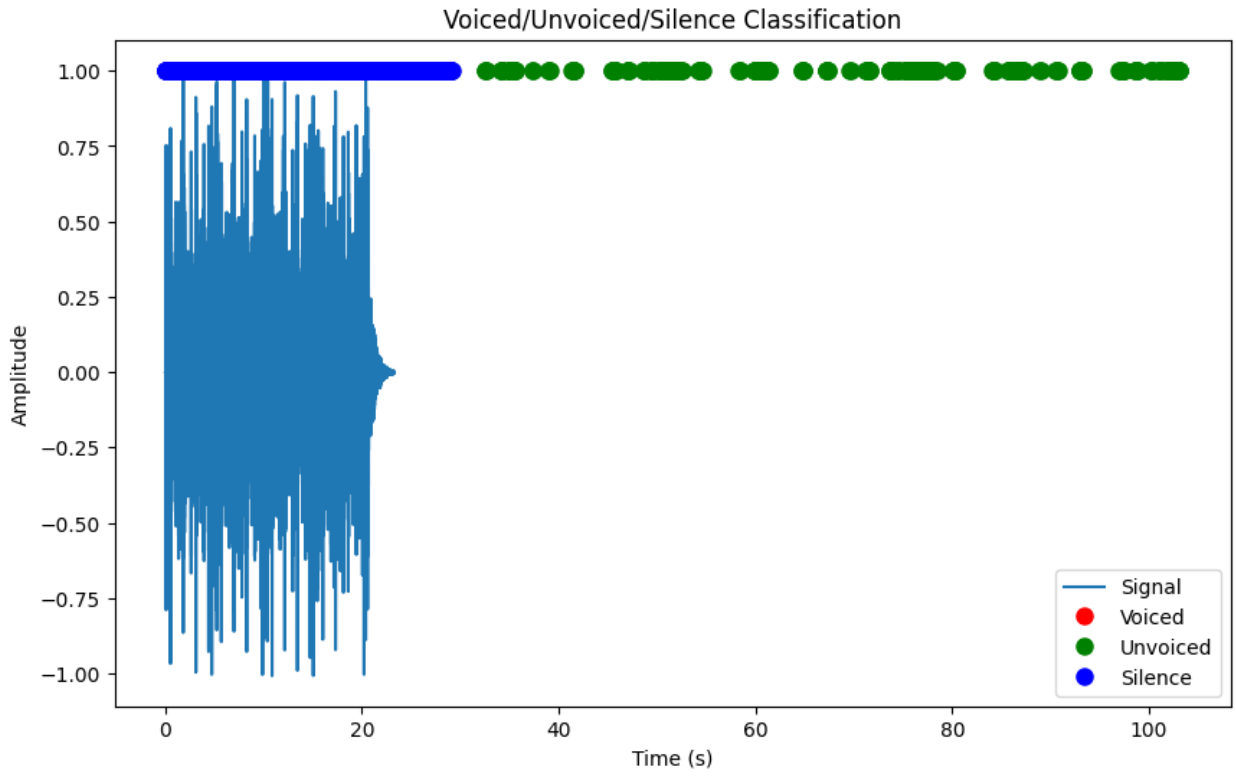
Plot the original speech signal along with the separated voiced, unvoiced, and silence segments for visual verification.

Python Source Code:

```
import librosa

import numpy as np

import matplotlib.pyplot as plt

def classify_audio_segments(signal, sr):

    stft = np.abs(librosa.stft(signal))

    energy = np.sum(stft, axis=0)

    pitches, magnitudes = librosa.piptrack(y=signal, sr=sr)

    pitch = np.median(pitches, axis=0)

    energy_threshold = np.percentile(energy, 80)  # Adjust as needed

    pitch_threshold = .

    voiced_regions = []

    unvoiced_regions = []

    silence_regions = []

    for i in range(len(energy)):

        if energy[i] > energy_threshold and pitch[i] > pitch_threshold:

            voiced_regions.append(i)

        elif energy[i] > energy_threshold and pitch[i] <= pitch_threshold    unvoiced_regions.append(i)
```

```
        else:
            silence_regions.append(i)
    return voiced_regions, unvoiced_regions, silence_regions
file_path = 'file.wav'
signal, sr = librosa.load(file_path, sr=None)
voiced_regions, unvoiced_regions, silence_regions = classify_audio_segments(signal, sr)
plt.figure(figsize=(10, 6))
plt.plot(np.arange(len(signal)) / sr, signal, label='Signal')
plt.plot(np.array(voiced_regions) * librosa.get_duration(y=signal, sr=sr) / len(voiced_regions),
         np.ones(len(voiced_regions)) * np.max(signal), 'ro', markersize=8, label='Voiced')
plt.plot(np.array(unvoiced_regions) * librosa.get_duration(y=signal, sr=sr) / len(unvoiced_regions),
         np.ones(len(unvoiced_regions)) * np.max(signal), 'go', markersize=8, label='Unvoiced')
plt.plot(np.array(silence_regions) * librosa.get_duration(y=signal, sr=sr) / len(silence_regions),
         np.ones(len(silence_regions)) * np.max(signal), 'bo', markersize=8, label='Silence')
plt.xlabel('Time (s)')
```

Output:



Experimental No: 13

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

Objectives

1. To understand the concepts of multilevel speech resolution.
2. To learn how to implement a MATLAB/Python program to plot multilevel speech resolution.
3. To evaluate the performance of the program on different speech signals.

Theory

Wavelet transform is a mathematical technique that breaks down a signal into different frequency components over multiple scales. It involves using a wavelet function, which is a small, localized waveform, to analyze the signal. The wavelet function is scaled and translated to cover different frequency ranges, allowing the decomposition of the signal into both low-frequency approximation and high-frequency detail components.

- Multilevel speech resolution is a technique for improving the quality of speech signals by reducing noise and artifacts.
- The technique works by dividing the speech signal into multiple levels, and then applying different processing techniques to each level.
- The processing techniques can vary depending on the specific application, but they typically involve removing noise, enhancing the signal, or improving the resolution.

Procedure:

Signal Preprocessing:

Load the speech signal into the MATLAB/Python environment and ensure proper sampling.

Select Wavelet:

Choose an appropriate wavelet function for the decomposition. This can include Daubechies, Haar, Symlet, etc.

Multilevel Decomposition:

Perform multilevel wavelet decomposition of the speech signal. This involves iteratively applying the wavelet transform to extract different levels of approximation and detail coefficients.

Plotting Resolutions:

For each decomposition level, plot the approximation and detail coefficients. The approximation coefficients represent the low-frequency content, while the detail coefficients capture high-frequency details.

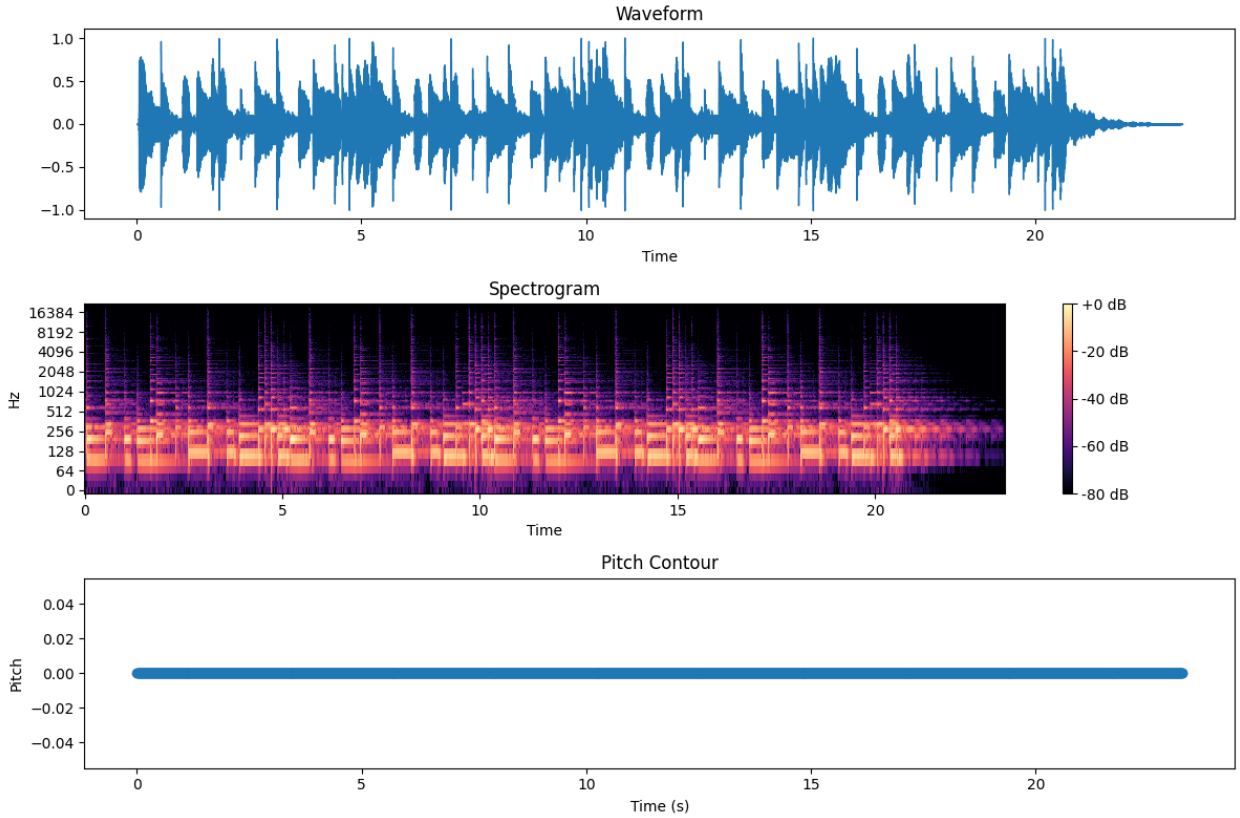
Visualization:

Arrange the plots of approximation and detail coefficients in a manner that clearly illustrates the signal's multilevel resolution. You can use subplots or arrange the plots vertically.

Python Source Code:

```
import librosa
import librosa.display
import numpy as np  # Import NumPy module
import matplotlib.pyplot as plt
file_path = 'file.wav'
signal, sr = librosa.load(file_path, sr=None)
D = librosa.amplitude_to_db(librosa.stft(signal), ref=np.max)
pitches, magnitudes = librosa.piptrack(y=signal, sr=sr)
pitch = np.median(pitches, axis=0)
plt.figure(figsize=(12, 8))
plt.subplot(3, 1, 1)
librosa.display.waveshow(signal, sr=sr)
plt.title('Waveform')
plt.subplot(3, 1, 2)
librosa.display.specshow(D, sr=sr, x_axis='time', y_axis='log')
plt.colorbar(format='%+2.0f dB')
plt.title('Spectrogram')
plt.subplot(3, 1, 3)
plt.plot(np.arange(len(pitch)) * librosa.get_duration(y=signal, sr=sr) / len(pitch), pitch, 'o-')
plt.xlabel('Time (s)')
plt.ylabel('Pitch')
plt.title('Pitch Contour')
plt.tight_layout()
plt.show()
```

Output:


Waveform
1.0
0.5
0.0
−0.5
−1.0
0
5
10
15
20
Time
Spectrogram
16384
8192
4096
2048
1024
512
256
128
64
0
Hz
0
5
10
15
20
Time
+0 dB
-20 dB
-40 dB
-60 dB
-80 dB
Pitch Contour
0.04
0.02
0.00
−0.02
−0.04
Pitch
0
5
10
15
20
Time (s)

Experimental No: 14

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

Objectives

1. To understand the concepts of speech recognition.
2. To learn how to implement a MATLAB/Python program to recognize speech signal.
3. To evaluate the performance of the program on different speech signals.

Theory

- Speech recognition is the process of converting speech into text. It is a challenging task because speech is a continuous signal that can be affected by a variety of factors, such as noise, background noise, and accent.
- There are two main approaches to speech recognition:
  i) Feature-based approach: This approach involves extracting features from the speech signal, such as the Mel-frequency cepstral coefficients (MFCCs). These features are then used to train a classifier, such as a support vector machine (SVM) or a neural network.
  ii) Direct modeling approach: This approach involves directly modeling the relationship between the speech signal and the text. This can be done using a variety of techniques, such as hidden Markov models (HMMs) or recurrent neural networks (RNNs).

Procedure:

Signal Preprocessing:

Load the speech signal into the MATLAB/Python environment and ensure proper sampling.

Select Wavelet:

Choose an appropriate wavelet function for the decomposition. This can include Daubechies, Haar, Symlet, etc.

Multilevel Decomposition:

Perform multilevel wavelet decomposition of the speech signal. This involves iteratively applying the wavelet transform to extract different levels of approximation and detail coefficients.

Plotting Resolutions:

For each decomposition level, plot the approximation and detail coefficients. The approximation coefficients represent the low-frequency content, while the detail coefficients capture high-frequency details.

Visualization:

Arrange the plots of approximation and detail coefficients in a manner that clearly illustrates the signal's multilevel resolution. You can use subplots or arrange the plots vertically.

Python Source Code:

```
import speech_recognition as sr

recognizer = sr.Recognizer()
```

```
file_path = 'file.wav'
with sr.AudioFile(file_path) as source:
    audio = recognizer.record(source)
try:
    recognized_text = recognizer.recognize_google(audio)
    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}")
```

Experimental No: 15

Experimental Name: Write a MATLAB/Python program for text-to-speech conversion and record speech Signal.

Objectives

1. To understand the concepts of text-to-speech conversion and speech recording.
2. To learn how to implement a MATLAB/Python program for text-to-speech conversion and speech recording.
3. To evaluate the performance of the program on different text and speech signals.

Theory:

Text-to-speech (TTS) synthesis is the process of converting written text into spoken words. This process involves multiple stages, including text analysis, linguistic processing, and waveform generation. TTS systems use various techniques like concatenative synthesis, formant synthesis, and statistical parametric synthesis to achieve natural-sounding speech.

The process of recording speech signals involves capturing audio waveforms using a microphone or another audio input device. The recorded signal is typically a continuous waveform that represents the air pressure variations produced by the speaker's voice.

Procedure:

Text-to-Speech Conversion:

Text-to-speech conversion can be achieved using libraries and modules that provide TTS functionality. For this lab, we will use the gTTS (Google Text-to-Speech) library in Python, which utilizes Google's TTS engine. The steps involved are as follows:

- Import the required library: from gtts import gTTS
- Input the desired text to be converted: text = "Hello, welcome to the lab."
- Create a gTTS object: tts = gTTS(text)
- Save the generated speech as an audio file: tts.save("output.mp3")

Playing the Generated Speech:

To play the generated speech, we can use a library like pygame in Python. Here's how:

- Import the required library: import pygame
- Initialize the pygame mixer: pygame.mixer.init()
- Load the generated audio file: sound = pygame.mixer.Sound("output.mp3")
- Play the audio: sound.play()

Recording the Speech Signal:

Recording the speech signal can be achieved using the sounddevice library in Python. This library allows for audio input and output. Here's how:

- Import the required library: import sounddevice as sd
- Set the sample rate and duration: sample_rate = 44100 (standard) and duration = 5 seconds

- Record the audio: recorded_signal = sd.rec(int(sample_rate * duration), samplerate=sample_rate, channels=2)
- Wait for recording to complete: sd.wait()
- Save the recorded signal to a WAV file: sd.write("recorded_signal.wav", recorded_signal, sample_rate)

Python Source Code:

```
from gtts import gTTS

import sounddevice as sd

import os

text = "Hello, I am Omar Faruk."

tts = gTTS(text)

tts.save("output1.mp3")

os.system("start output.mp3")

input("Press Enter after the speech finishes playing...")

recording_duration = len(tts)  # Duration in seconds

recording = sd.rec(int(recording_duration * sd.default.samplerate), samplerate=sd.default.samplerate, channels=1)

sd.wait()

output_wav_file = "recorded_speech.wav"

sd.write(output_wav_file, recording, sd.default.samplerate)

print("Text-to-speech and speech recording complete.")
```