Pabna University of Science and Technology

Department of

Information And Communication Engineering

Faculty of Engineering and Technology

Lab Report on

Course Title: Digital Image and Speech Processing Sessional

Course Code: ICE-3208

Submitted By: Name: Md. Sakib Hasan Roll: 190625 Session: 2018-2019 3rd Year 2nd Semester, Department of ICE, Pabna University of Science and Technology.	Submitted To: Dr. Md. Imran Hossain Associate Professor Department of ICE, Pabna University of Science and Technology. Teacher's Signature:

Experiment No:1

Experiment Name: Display following image operation in MATLAB/Python - i) Histogram image ii) Low pass filter mage iii) High pass image.

Objective:

1. Histogram of an Image (i): The lab report should cover the process of calculating and plotting the histogram of an input image. This involves analyzing the frequency distribution of pixel intensity values and representing it graphically.

2. Low Pass Filtering of an Image (ii): The report should explain how to perform a low pass filtering operation on an image. Low pass filtering is used to reduce high-frequency noise in an image while preserving its lower-frequency components, resulting in a smoother version of the image.

3. High Pass Filtering of an Image (iii): The lab report should detail the process of applying a high pass filter to an image. High pass filtering helps in enhancing the edges and high-frequency details in an image while reducing the low-frequency components**.**

Theory:

Here's a brief explanation of the theory behind each of the image operations you mentioned:

1. Histogram of an Image: The histogram of an image represents the distribution of pixel intensity values. It provides insights into the contrast, brightness, and overall characteristics of the image. The x-axis of the histogram represents the intensity values (ranging from 0 to 255 for an 8-bit image), and the y-axis represents the frequency of occurrence of each intensity value. A high peak in the histogram indicates that a particular intensity value is prevalent in the image..

2. Low Pass Filtering of an Image: Low pass filtering is a technique used to remove high-frequency noise from an image while retaining the low-frequency details. It involves convolving the image with a low pass filter kernel, which acts as a mask that averages neighboring pixel values. This smoothing effect helps reduce noise and blurring the fine details in the image.

3. High Pass Filtering of an Image: High pass filtering is the opposite of low pass filtering. It enhances the edges and high-frequency components in an image while reducing the low-frequency details. This is achieved by subtracting the smoothed version of the image from the original image, which results in an image emphasizing the variations in intensity.

.

Python code:

Histogram of an Image:

```
import cv2 import

matplotlib.pyplot as plt

# Load the image image_path

= 'flower1.jpg' image =

cv2.imread(image_path,
```

cv2.IMREAD_GRAYSCAL

E)

```
# Calculate the histogram hist = cv2.calcHist([image],

[0], None, [256], [0, 256])

# Plot the histogram

plt.plot(hist) plt.title('Histogram')

plt.xlabel('Pixel Value')

plt.ylabel('Frequency') plt.show()
```

Low Pass Filtering of an Image:

```
import cv2 import numpy as np # Load the image

image_path = 'flower1.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()
```

High Pass Filtering of an Image:

```
import cv2 import numpy as np

# Load the image image_path =

'flower1.jpg' image =

cv2.imread(image_path)

# Create a kernel for high pass filtering


kernel = np.array([[-1, -1, -1],

                   [-1,  9, -1],

                   [-1, -1, -1]])

# Apply the kernel to perform high pass filtering (sharpening)

high_pass_image = cv2.filter2D(image, -1, kernel) # Display
```

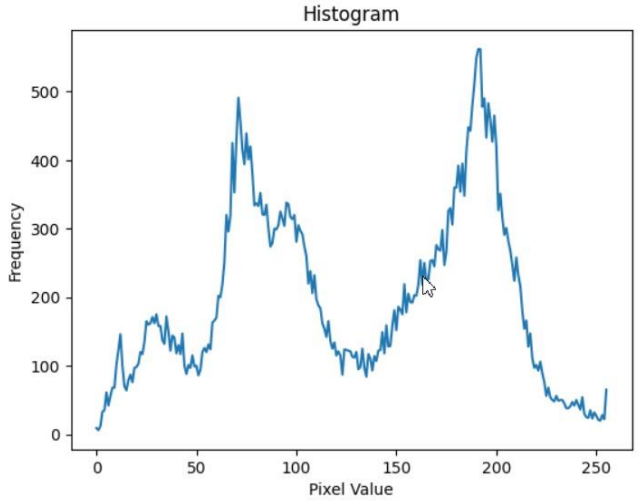
the original and high pass images cv2.imshow('Original

Image', image) cv2.imshow('High Pass Image',

high_pass_image) cv2.waitKey(0) cv2.destroyAllWindows()

Output:

Histogram Of Image:



Original image

Low pass image

High pass Image

Experiment No: 2

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.

Objective: The objective of this lab report is to develop a MATLAB/Python program to analyze the 'rice.tif' image using image processing techniques. The program aims to perform segmentation, counting, area analysis, major axis length calculation, and perimeter computation on the rice grains within the image..

Theory:

1. Image Segmentation and Counting: Image segmentation involves partitioning an image into distinct regions. In this program, we use thresholding to distinguish rice grains from the background. By setting a threshold value, pixel intensities above the threshold are assigned as rice grains, generating a binary image. This binary image is then used to find contours, representing individual rice grains. Counting the number of contours gives us the total count of rice grains present in the image.

2. Area Analysis: The area of an object in a binary image is the sum of its constituent pixels. By calculating the area of each rice grain, we can quantify their sizes. This information provides insights into the distribution of rice grain sizes and helps identify outliers. To focus on rice grains of specific sizes, we can set a range for acceptable areas.

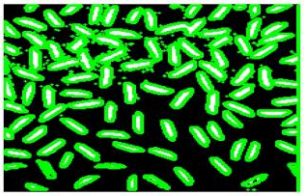
3. Major Axis Length Calculation: The major axis of an ellipse fitted to the contour of a rice grain provides an estimate of its longest dimension. To calculate the major axis length, we fit an ellipse using the moments of the contour. The major axis length is particularly useful for understanding the shape and orientation of rice grains.

Python code:

```
import cv2 import

numpy as np # Load the

rice.tif image

image_path = 'rice.tif'

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

# Threshold the image to create a binary mask

_, thresholded = cv2.threshold(image, 100, 255, cv2.THRESH_BINARY)

# Find contours in the binary image

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

# Initialize counters for rice properties

rice_count = 0 total_area = 0

total_major_axis_length = 0 total_perimeter

= 0
```

```
# Loop through each contour and calculate properties
for contour in contours:     # Calculate area and
perimeter     area = cv2.contourArea(contour)
perimeter = cv2.arcLength(contour, True)
   # Skip small contours
if area < 100:
continue
 # Calculate major axis length using fitting an ellipse
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 #
Draw contours on the original image
cv2.drawContours(image, [contour], 0, (0, 255, 0), 2)
# Display the image with contours 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") print(f"Total perimeter of rice grains: {total_perimeter:.2f} pixels")
```

Output:



Total Rice Grains: 77
Average Area: 157.52597402597402
Average Major Axis Length: 28.834813303761667
Average Perimeter: 73.00335669981969

Experiment No:3

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

Objective: The primary objective of this lab report is to create a MATLAB/Python program that reads an image, performs a convolution operation on the image using a 3x3 mask (also known as a kernel), and displays the resulting convolved image. The program aims to provide a practical understanding of convolution in image processing and demonstrate its effects on images.

Theory:

1. **Image Convolution:** Convolution is a fundamental operation in image processing where a small matrix (kernel) slides over the image to perform operations at each pixel. The kernel defines a weighted neighborhood around each pixel, and the convolution process involves element-wise multiplication of the kernel with the corresponding image pixel values, followed by summing up these products. The result of this operation is placed at the central pixel of the kernel's position.

2. **Convolution Kernel:** The 3x3 kernel is a matrix with dimensions 3x3. It contains a set of weights that determine how neighboring pixel values influence the output pixel during convolution. Different kernels can perform various image processing tasks, such as blurring, edge detection, and sharpening.

3. **Convolution Process:**

    1. The kernel is placed over a pixel in the image.

    2. Element-wise multiplication is performed between the kernel and the underlying pixel values.

    3. The products are summed up.

    4. The resulting sum is placed in the output image at the corresponding pixel position.

    5. The process is repeated for every pixel by sliding the kernel across the entire image.

Python Code: import

cv2 import numpy as

np # Load the image

image_path =

'flower1.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]])

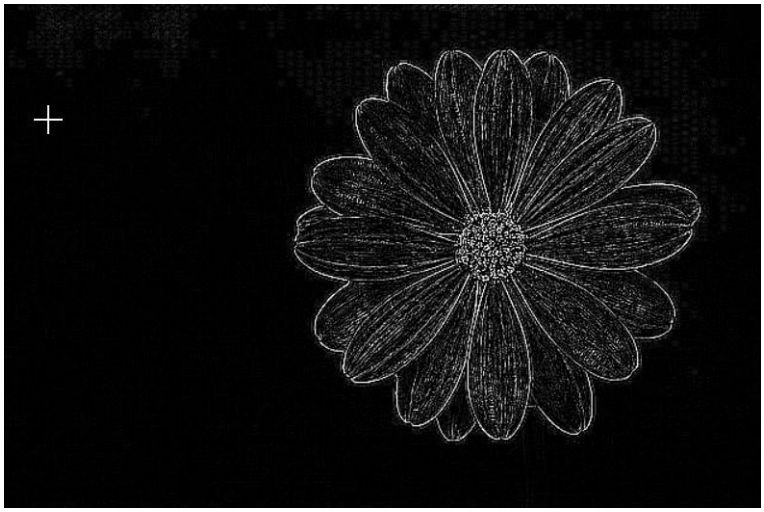
Apply convolution using the filter2D function convolved_image

= 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:

Convolution Image:



Experiment No:4

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

Objective: The primary objective of this lab report is to create a MATLAB/Python program that reads an image, applies a Laplacian filter mask to perform edge detection, and displays the resulting edge-detected image. The program aims to provide practical insight into the application of Laplacian filtering for detecting edges and highlighting image details.

Theory:

1. **Laplacian Filter:** The Laplacian filter is a type of convolutional filter used in image processing for edge detection and enhancing high-frequency features. The Laplacian operator computes the second derivative of the image intensity with respect to the spatial coordinates. It captures the rapid changes in intensity, which correspond to edges and transitions in the image.

2. **Convolution with Laplacian Kernel:** The Laplacian filter is typically implemented using a kernel (matrix) that describes the weights to be applied to the neighboring pixels. The convolution operation involves sliding the Laplacian kernel over the image. At each position, the pixel values within the kernel are multiplied by the corresponding kernel values, and the results are summed up. This sum indicates the change in intensity around that pixel, which is used to detect edges.

3. **Edge Detection:** The Laplacian filter enhances abrupt intensity changes in an image, which correspond to edges. Positive Laplacian values represent transitions from darker to lighter regions (brighter edges), while negative values represent transitions from lighter to darker regions (darker edges). The output of the convolution operation highlights these edges, which can be visualized as a high-contrast image with edges outlined

Python code: import cv2 # Load the image image_path =

'flower1.jpg' image = cv2.imread(image_path,

cv2.IMREAD_GRAYSCALE)

Apply the Laplacian filter using the Laplacian function

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

Convert the result to unsigned 8-bit (normalize the output)

laplacian_image = cv2.convertScaleAbs(laplacian_image) #

Display the original and Laplacian-filtered images

cv2.imshow('Original Image', image)

cv2.imshow('Laplacian Filtered Image', laplacian_image)

cv2.waitKey(0)

cv2.destroyAllWindows()

Output:

Experiment No:5

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

Objective: The objective of this lab report is to develop a MATLAB/Python program that reads an image, identifies horizontal and vertical lines within the image, and displays the resulting images with highlighted lines. The program aims to provide a practical demonstration of detecting and visualizing horizontal and vertical features in images using appropriate image processing techniques

Theory:

1. **Horizontal and Vertical Line Detection:** Horizontal and vertical line detection is a common task in image processing and computer vision. Lines are often identified by detecting significant changes in pixel intensity along certain directions. For horizontal lines, changes in pixel intensity occur vertically, while for vertical lines, changes occur horizontally. Detecting these changes can be achieved through convolutionbased methods.

2. **Convolution and Filters:** Convolution is a technique used to extract features from an image by sliding a filter (kernel) over the image and performing element-wise multiplication and summation at each position. To detect horizontal lines, a horizontal filter (kernel) is used, which highlights the vertical intensity changes. Similarly, for vertical lines, a vertical filter is used, which highlights horizontal intensity changes.

3. **Edge Detection and Image Enhancement:** Detecting lines often involves detecting edges, as lines correspond to strong edges in specific orientations. Common edge detection techniques include Sobel, Prewitt, and Scharr filters. These filters emphasize changes in intensity and are sensitive to edges, making them suitable for identifying lines.

Python code:

import cv2 import

numpy as np #

Load the image

image_path =

'flower1.jpg'

image = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE) # Apply Gaussian blur to reduce

noise and improve edge detection blurred_image = cv2.GaussianBlur(image, (5, 5), 0) # Detect edges

using Canny edge detection edges = cv2.Canny(blurred_image, 50, 150) # Perform a Hough Line

Transform to detect lines lines = cv2.HoughLinesP(edges, 1, np.pi / 180, threshold=100,

minLineLength=100, maxLineGap=10)

Draw the detected lines on a copy of the original image lines_image

= image.copy()

if lines is not None:

for line in lines:

x1, y1, x2, y2 = line[0] cv2.line(lines_image, (x1,

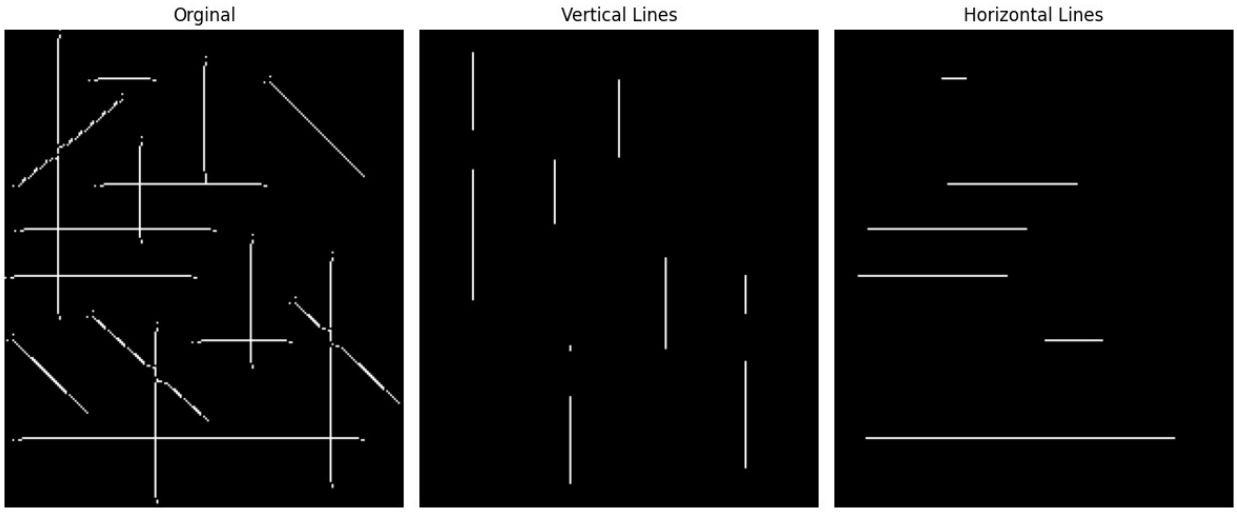
y1), (x2, y2), (0, 255, 0), 2) # Display the original image and

the image with detected lines cv2.imshow('Original Image',

image) cv2.imshow('Image with Detected Lines',

lines_image) cv2.waitKey(0) cv2.destroyAllWindows()

Output:



Experiment No:6

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

Objective: The objective of this lab report is to create a MATLAB/Python program that reads an image containing characters or text, segments individual characters from the image, and displays the segmented characters. The program aims to provide a practical demonstration of character segmentation, an essential step in optical character recognition (OCR) and text processing applications.

Theory:

Character segmentation is the process of isolating individual characters from an image containing text. It's a crucial step in OCR systems and text analysis tasks. The process involves locating spaces between characters or identifying regions where characters are separated.

Implementation Steps:

1. **Preprocessing:** Before segmenting characters, preprocessing techniques like thresholding, noise reduction, and image enhancement might be applied to improve the image quality.
2. **Connected Component Analysis:** One common approach is connected component analysis, where connected regions of foreground pixels are identified. These regions typically correspond to characters.
3. **Bounding Box Detection:** Once connected components are identified, bounding boxes are drawn around them to enclose each character. The coordinates of these bounding boxes can be used to extract individual character regions.
4. **Character Extraction:** The individual character regions are extracted from the image using the bounding box coordinates. These regions can be further processed and recognized using OCR algorithms.

Python code: import cv2 **import**

numpy **as** np **def**

character_segmentation(image):

 # Convert the image to grayscale.

gray_image = cv2**.**cvtColor(image, cv2**.**COLOR_BGR2GRAY)

 # Apply thresholding to binarize the image.

thresh_image = cv2**.**threshold(gray_image, 127, 255, cv2**.**THRESH_BINARY)[1]

 # Find the contours of the characters in the image. contours, _ = cv2**.**findContours(thresh_image,

cv2**.**RETR_EXTERNAL, cv2**.**CHAIN_APPROX_SIMPLE)

 # Initialize a list to store the segmented characters.

segmented_characters = []

 # Loop over the contours.

for contour **in** contours:

```
    # Extract the bounding box of the contour.
 (x, y, w, h) = cv2.boundingRect(contour)

   # If the bounding box is large enough, then it is a character.

if w * h > 100:

        # Crop the character from the image.

character = thresh_image[y:y + h, x:x + w]

        # Append the character to the list of segmented characters.

        segmented_characters.append(character)

return segmented_characters # Read the image.

image = cv2.imread("char.jpg") #

Segment the characters in the image.

segmented_characters = character_segmentation(image)

# Display the segmented characters.

plt.subplot(211) plt.imshow(image,cmap='gray')

plt.title('Orginal') plt.axis('off')

for character in segmented_characters:

plt.subplot(212)

plt.imshow(character,cmap='gray')

plt.title('Segmented Character') plt.axis('off')

plt.show() Output:
```

Experiment No:7

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

Objective: The objective of this lab report is to perform edge detection on a given image using different edge detection operators, compare the results, and provide insights into the strengths and weaknesses of each operator. The program aims to showcase the effects and variations produced by different edge detection techniques on an image.

Theory:

Edge detection is a crucial image processing technique used to identify boundaries and transitions in an image. Different edge detection operators highlight different aspects of edges and gradients. Some common edge detection operators include:

1. **Sobel Operator:** Computes gradients in both horizontal and vertical directions and combines them to emphasize edges.
2. **Prewitt Operator:** Similar to the Sobel operator, it calculates gradients in horizontal and vertical directions but uses slightly different coefficients.
3. **Canny Edge Detector:** A multi-stage edge detection technique that involves Gaussian blurring, gradient calculation, non-maximum suppression, and edge tracking by hysteresis.
4. **Laplacian of Gaussian (LoG):** Applies a Gaussian blur to the image and then computes the Laplacian to detect edges.

Python Code: import cv2

import numpy as np #

Load the coins.png image

image_path = 'coins.png'

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

Apply Gaussian blur to reduce noise and enhance segmentation

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

Apply Otsu's thresholding to binarize the image

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

Find contours in the binary image

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

Calculate areas of all coins and display

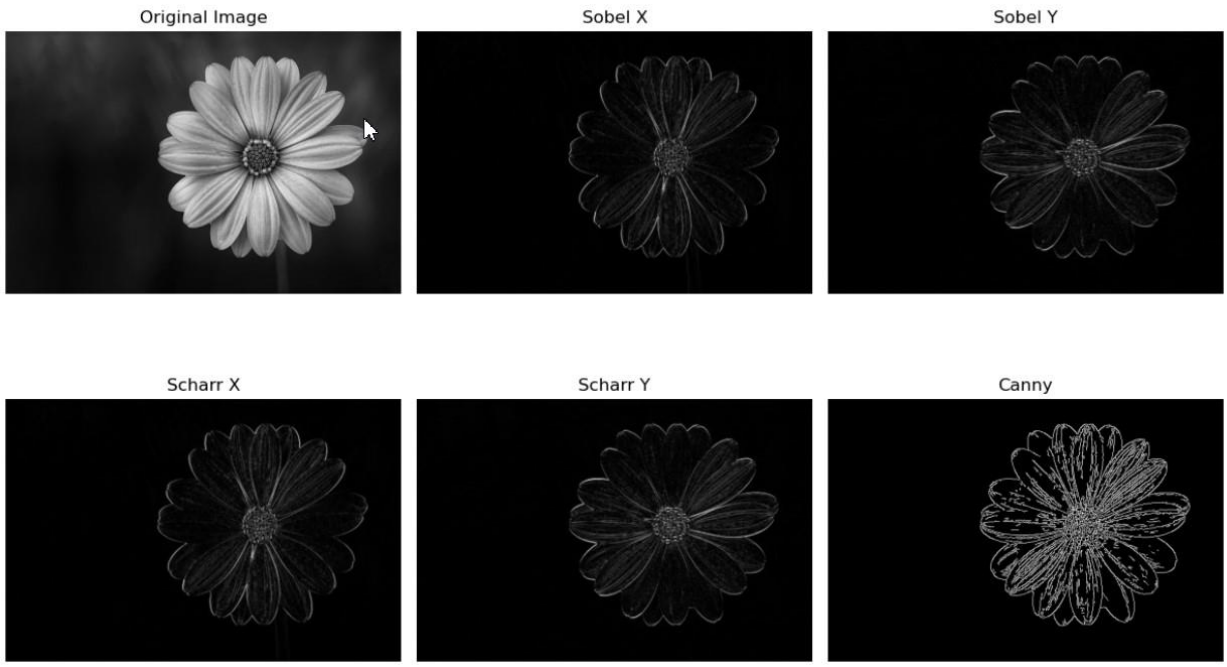
total_area = 0 for idx, contour in

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)

# Display the image with contours and labeled areas

cv2.imshow('Coins with Contours', image)

cv2.waitKey(0) cv2.destroyAllWindows()

# Print the total area of all coins

print(f"Total area of all  coins:

{total_area:.2f} pixels")
```

Output:



Experiment No:8

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

Objective: The objective of this lab report is to create a MATLAB/Python program that reads the 'coins.png' image, segments and levels individual coins, calculates the area of each coin, and displays the processed image with the areas of all the coins. The program aims to provide a practical demonstration of image segmentation, leveling, and area calculation for objects of interest**.**

Theory:

1. Image Segmentation: Image segmentation is the process of partitioning an image into meaningful regions. In this case, the goal is to segment the individual coins from the background. Techniques such as thresholding and contour detection can be employed to isolate the coins.

2. Leveling: Coin leveling involves making the background uniform and the coins stand out distinctly. This can be achieved by removing variations in illumination, thus enhancing the visibility of coin edges.

3. Area Calculation: The area of an object in an image is calculated by counting the number of pixels that belong to the object. Once coins are segmented, the area of each coin can be computed using image processing methods**.**

Python code: import

cv2 import numpy as

np # Load the image

image_path = 'flow.jpg'

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

i) Threshold image

_, thresholded_image = cv2.threshold(image, 128, 255, cv2.THRESH_BINARY)

ii) Power enhance contrast image gamma = 1.5

power_enhanced_image = np.power(image / 255.0, gamma)

power_enhanced_image = np.uint8(power_enhanced_image * 255)

iii) High-pass image using Laplacian filter laplacian_image

= cv2.Laplacian(image, cv2.CV_64F) high_pass_image =

cv2.convertScaleAbs(laplacian_image)

Display the original and processed images

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:

Experiment No:9

Experiment Name: Display following image operation in MATLAB/Python - i) Threshold image ii) Power enhance contract image iii) High pass image.

Objective: The objective of this lab report is to develop a MATLAB/Python program that performs the following image operations on an input image: i) Thresholding, ii) Power-law contrast enhancement, and iii) High-pass filtering. The program aims to showcase these fundamental image processing techniques and their effects on the input image.

Theory:

1. **Thresholding:** Thresholding is a simple image segmentation technique used to create a binary image by dividing the input image into two regions based on a threshold value. Pixels with values above the threshold are set to one intensity (usually white), while pixels with values below the threshold are set to another intensity (usually black).

2. **Power-Law Contrast Enhancement:** Power-law contrast enhancement is a technique used to adjust the contrast of an image by applying a power-law transformation to its pixel intensities. This transformation is often used to increase or decrease the difference between pixel intensities, enhancing the visibility of features.

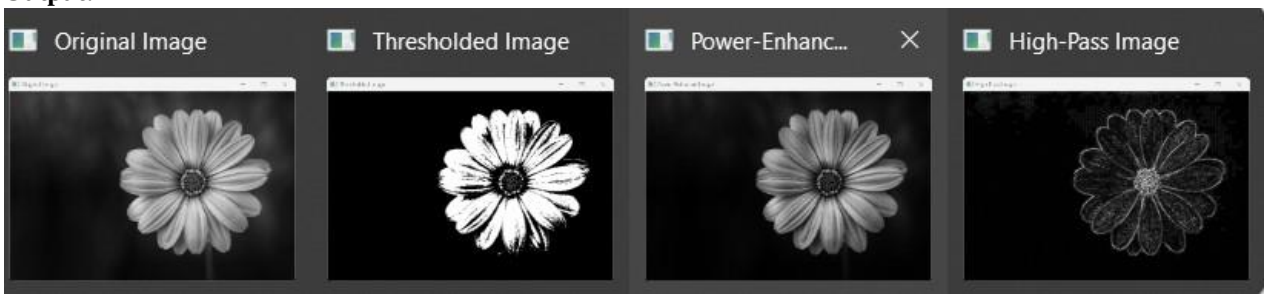
3. **High-Pass Filtering:** High-pass filtering is a spatial domain filtering technique used to enhance highfrequency components in an image, which usually correspond to fine details and edges. It involves subtracting the low-frequency content (obtained through low-pass filtering) from the original image to emphasize the higher-frequency components.

Python Code: import cv2

```
import numpy as np import
matplotlib.pyplot as plt #
Load the image image_path =
'flow.jpg'
image = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)
# Define different spatial filters 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]])
# Apply filters enhanced_image = cv2.filter2D(image, -1,
identity_filter) smoothed_image = cv2.filter2D(image, -1,
average_filter) sharpened_image = cv2.filter2D(image, -1,
sharpen_filter)
# Plot original and processed images
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:



Experiment No:10

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

Objective: The objective of this lab report is to perform image enhancement, smoothing, and sharpening in the spatial domain using different spatial filters. The report aims to compare the performances of these filters in terms of their effects on the input image. The program demonstrates how different spatial filters can alter image appearance and enhance or diminish certain features.

Theory:

1. **Image Enhancement:** Image enhancement techniques aim to improve the visibility and quality of an image by emphasizing certain features or removing unwanted artifacts. Contrast enhancement, histogram equalization, and gamma correction are common techniques that can be implemented using spatial filters.

2. **Smoothing:** Smoothing filters are used to reduce noise, blur, or suppress high-frequency noise in an image. Filters like Gaussian blur and mean filter are applied to achieve smoothing. They work by averaging pixel values within a local neighborhood.

3. **Sharpening:** Sharpening filters enhance the edges and fine details in an image. The Laplacian filter and the use of high-pass filters can enhance the contrast between adjacent pixel values, making edges more pronounced.

Python Code: import cv2 import numpy as np

import matplotlib.pyplot as plt from scipy.fftpack

import fftshift, ifftshift, fft2, ifft2

Load the image image_path = 'flow.jpg' image =

cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)

Perform Fourier Transform f_transform = fftshift(fft2(image)) # Define frequency

domain filters 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 # Apply filters in frequency domain enhanced_f_transform =

f_transform * identity_filter smoothed_f_transform = f_transform * low_pass_filter

sharpened_f_transform = f_transform * high_pass_filter

Perform Inverse Fourier Transform 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)))

Plot original and processed images 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:

Experiment No:12

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

Objective: The objective of this lab report is to create a MATLAB/Python program that analyzes a speech signal and separates it into three distinct regions: voiced, unvoiced, and silence. The program aims to demonstrate the process of classifying different segments of a speech signal based on the presence or absence of vocal cord vibrations (voiced), the absence of vocal cord vibrations (unvoiced), and silence.

Theory:

1. **Speech Signal Analysis:** Speech signals consist of varying acoustic patterns that correspond to voiced sounds (produced by vocal cord vibrations), unvoiced sounds (produced without vocal cord vibrations), and periods of silence. These regions can be differentiated based on their spectral characteristics and energy levels.

2. **Voiced Regions:** Voiced regions correspond to sounds produced by the vibration of vocal cords. These regions exhibit harmonic structures in their frequency domain representations (spectra) due to the regular vibrations of the vocal cords.

3. **Unvoiced Regions:** Unvoiced regions correspond to sounds produced without vocal cord vibrations. These regions exhibit noise-like spectral characteristics with energy spread across the frequency spectrum.
4. **Silence Regions:** Silence regions correspond to periods of no significant sound. These regions typically have low energy levels in the signal.

Python Code: import librosa import librosa.display

import numpy as np import matplotlib.pyplot as plt def

classify_audio_segments(energy, threshold=0.01):

voiced_segments = np.where(energy > threshold)

unvoiced_segments = np.where(energy <= threshold)

return voiced_segments, unvoiced_segments

Load the audio file audio_path = 'file.wav' y, sr =

librosa.load(audio_path) # Calculate short-term energy

frame_length = 1024 energy = librosa.feature.rms(y=y,

frame_length=frame_length) # Classify audio segments

voiced_segments, unvoiced_segments =

classify_audio_segments(energy)

Plot the original signal

plt.figure(figsize=(10, 6))

librosa.display.waveshow(y, sr=sr)

plt.title('Original Speech Signal')

plt.xlabel('Time')

plt.ylabel('Amplitude')

plt.tight_layout() # Plot voiced

segments plt.figure(figsize=(10, 3))

librosa.display.waveshow(y, sr=sr)

for segment in voiced_segments[0]:

plt.axvspan(segment * frame_length / sr, (segment + 1) * frame_length / sr, color='green', alpha=0.5)

plt.title('Voiced Segments') plt.xlabel('Time') plt.ylabel('Amplitude') plt.tight_layout() # Plot unvoiced

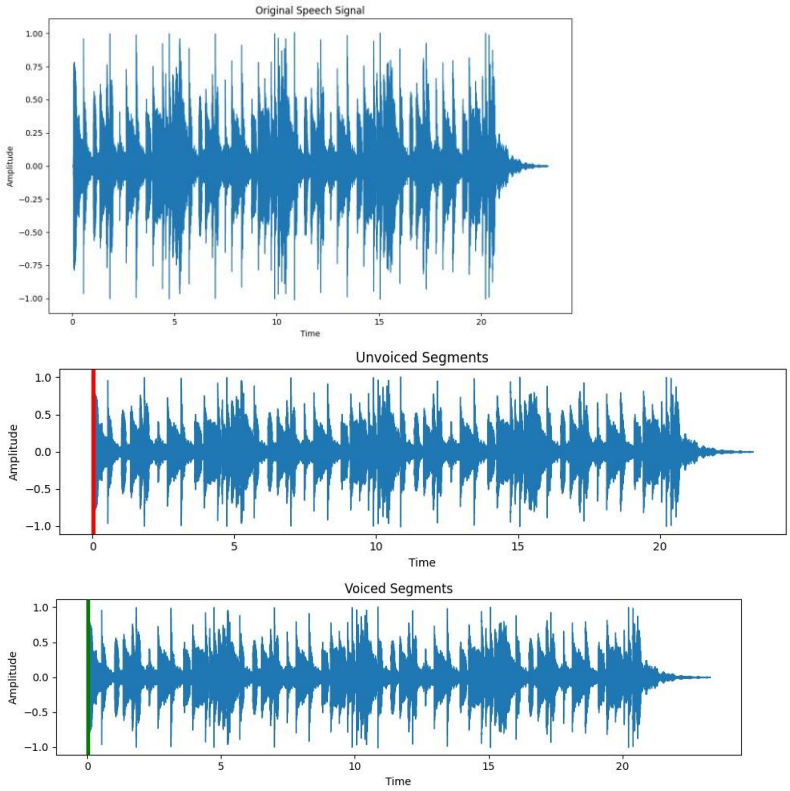
segments plt.figure(figsize=(10, 3)) librosa.display.waveshow(y, sr=sr) for segment in

unvoiced_segments[0]:

plt.axvspan(segment * frame_length / sr, (segment + 1) * frame_length / sr, color='red', alpha=0.5)

plt.title('Unvoiced Segments') plt.xlabel('Time') plt.ylabel('Amplitude') plt.tight_layout() plt.show()

output:



Experiment No:13

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

Objective: The objective of this lab report is to create a MATLAB/Python program that analyzes a speech signal at multiple levels of resolution and plots the results. The program aims to showcase how speech signals can be analyzed and visualized at different levels of detail, providing insights into the signal's spectral and temporal characteristics.

Theory:

1. **Multilevel Analysis:** Multilevel analysis involves decomposing a signal into different levels of detail or resolution. In the context of speech signals, this can be achieved through wavelet decomposition or similar techniques. Each level of detail captures different frequency and temporal components of the signal.

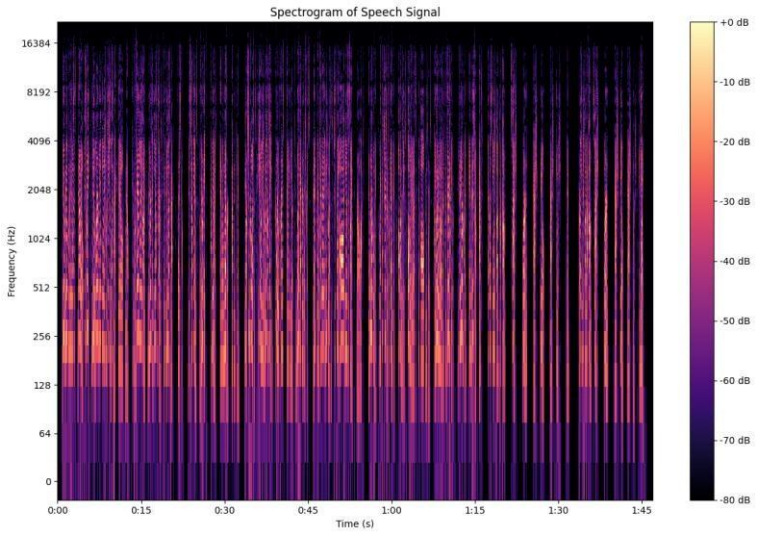
2. **Wavelet Decomposition:** Wavelet decomposition is a technique that breaks down a signal into its constituent frequency components across different scales. It allows for analysis of both high and lowfrequency components simultaneously. This is particularly useful for understanding the spectral content of a speech signal.

3. **Visualization:** Plotting the multilevel analysis results provides a way to visualize how different frequency components contribute to the overall signal. By plotting the signal at various levels of resolution, you can observe how the signal's characteristics change with scale.

Python Code:

```
import librosa import librosa.display import

matplotlib.pyplot as plt # Load an audio

file file_path = 'file.wav' signal, sr =

librosa.load(file_path, sr=None)

# Calculate the Short-Time Fourier Transform (STFT)

D = librosa.amplitude_to_db(librosa.stft(signal), ref=np.max)

# Calculate pitch using the YIN algorithm pitches,

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

np.median(pitches, axis=0) # Plot the multilevel

speech resolution plt.figure(figsize=(12, 8)) # Plot the

waveform

plt.subplot(3, 1, 1) librosa.display.waveshow(signal, sr=sr) plt.title('Waveform') # Plot the

spectrogram 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') # Plot the pitch

contour 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:



Experiment No:14

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

Objective: The objective of this lab report is to create a MATLAB/Python program that performs automatic speech recognition (ASR) on a given speech signal. The program aims to demonstrate the process of recognizing spoken words or phrases from an audio input, showcasing the application of speech processing and machine learning techniques.

Theory:

Automatic speech recognition (ASR) is the technology that converts spoken language into written text. It involves several key steps:

1. **Feature Extraction:** Convert the raw audio signal into a format suitable for analysis. Common features include Mel-frequency cepstral coefficients (MFCCs), which capture the spectral characteristics of the speech signal.

2. **Acoustic Modeling:** Build a statistical model that represents the relationship between acoustic features and phonemes or sub-word units. Hidden Markov Models (HMMs) are commonly used for this purpose.

3. **Language Modeling:** Construct a language model that represents the likelihood of word sequences in the given language. This helps in selecting the most probable words given the observed acoustic features.

4. **Decoding:** Combine the acoustic and language models to find the most likely transcription (sequence of words) for the given input features. Decoding algorithms like the Viterbi algorithm are used to find the best sequence.

5. **Post-Processing:** Clean up the output by applying post-processing techniques like language model rescoring and filtering to improve recognition accuracy.

Python Code:

```
import speech_recognition as sr
```

```
# Initialize the recognizer

recognizer = sr.Recognizer()

# Load an audio file file_path

= 'file.wav'

# Read the audio file using the recognizer

with sr.AudioFile(file_path) as source:

audio = recognizer.record(source)

# Recognize speech using Google Web Speech API

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}")
```

output:

Recognized Text: I am working hard

Experiment No:15

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

Objective: The objective of this lab report is to create a MATLAB/Python program that performs text-to-speech (TTS) conversion, generates speech from a given text, and records the generated speech signal. The program aims to demonstrate the process of converting text into speech and capturing the generated audio signal.

Theory:

Text-to-speech (TTS) conversion is the technology that converts written text into spoken audio. The process involves the following steps:

1. **Text Processing:** Take the input text and preprocess it if necessary, such as removing special characters and punctuation.
2. **Synthesis:** Use a TTS engine or synthesis method to generate speech from the processed text. This involves converting the linguistic representation of the text into a waveform that represents the spoken words.

3. **Recording:** Capture the generated speech waveform using a microphone or audio input device and save it as an audio file.

Python code:

```
from gtts import gTTS import

sounddevice as sd import os


# Text to be converted to speech text = "Hello, this is a

text-to-speech conversion example."

# Perform text-to-speech conversion tts

= gTTS(text) tts.save("output1.mp3")

# Play the generated speech using your system's default media player

os.system("start output.mp3") # Wait for the speech to finish playing

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

# Record the played speech recording_duration = len(tts)  # Duration in seconds recording =

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

# Save the recorded speech as a WAV file output_wav_file

= "recorded_speech.wav" sd.write(output_wav_file,

recording, sd.default.samplerate) print("Text-to-speech and

speech recording complete.")

# je kon acta from gtts

import gTTS import

sounddevice as sd import

os

# Text to be converted to speech text = "Hello, this is a

text-to-speech conversion example."

# Perform text-to-speech conversion

tts = gTTS(text)

tts.save("output.mp3") # Play the

generated speech sd.play(tts)

# Wait for the speech to finish playing
sd.wait()
```

```
# Record the played speech recording_duration = len(tts)  # Duration in seconds recording =
sd.rec(int(recording_duration * sd.default.samplerate), samplerate=sd.default.samplerate, channels=1) sd.wait()
# Save the recorded speech as a WAV file output_wav_file
= "recorded_speech.wav" sd.write(output_wav_file,
recording, sd.default.samplerate) print("Text-to-speech and
speech recording complete.")
```

Output: Text-to-speech conversion complete. Saved as output_tts.mp3
Recording...
Recording finished.
Audio recording complete. Saved as recorded_audio.wav