INDEX

	2. Contrast adjustments 3. Histogram equalization 4. Thresholding, and halftoning operations			
5	Write a program to apply various enhancements on images using image derivatives by implementing Gradient and Laplacian operations.			
6	Write a program to implement linear and nonlinear noise smoothing on suitable image or sound signal.			
7	Write a program to apply various image enhancement using image derivatives by implementing smoothing, sharpening, and unsharp masking filters for generating suitable images for specific application requirements.			
8	Write a program to Apply edge detection techniques such as Sobel and Canny to extract meaningful information from the given image samples.			
9	Write the program to implement various morphological image processing techniques.			
10	Write the program to extract image features by implementing methods like corner and blob detectors, HoG and Haar features			

11	Write the program to apply segmentation for detecting lines, circles, and other shapes/ objects. Also, implement edge-based and region-based segmentation.			

PRACTICAL-01

Aim: - Write a Program to demonstrate the following aspects of signal processing on suitable data:

1. Upsampling and Downsampling on Image/Speech Signal.
2. Fast Fourier Transform to compute DFT.

Description: -

Upsampling is the process of increasing the sampling rate of a signal by inserting additional data points between existing ones to enhance resolution or prepare it for further processing. Upsampling involves increasing the number of pixels in an image to improve its resolution or size, typically through interpolation techniques such as nearest-neighbor, bilinear, or bicubic interpolation.

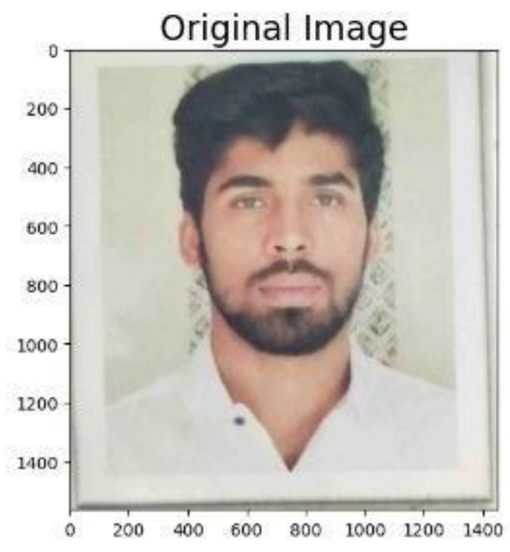
Downsampling in signal processing involves decreasing the sampling rate by selecting a subset of samples, often achieved through decimation to reduce computational load or storage requirements while preserving essential signal characteristics. Downsampling is the process of reducing the number of samples or pixels in an image or signal, often to decrease file size or computational complexity, typically achieved by discarding some of the original data points or averaging neighboring points.

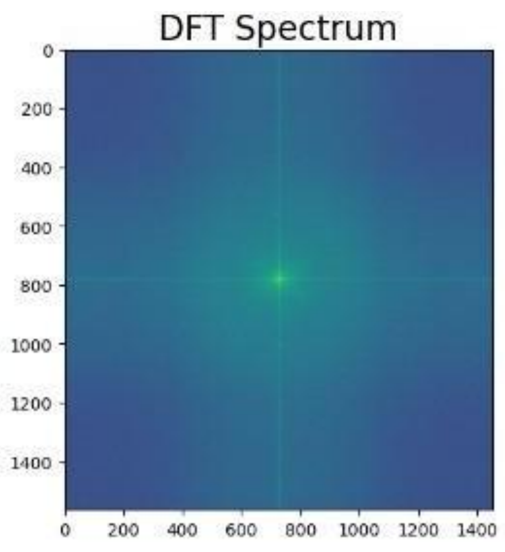
The Discrete Fourier Transform (DFT) in image processing is a mathematical technique used to convert spatial domain image data into frequency domain representations, revealing the frequency components present in the image. The Fast Fourier Transform (FFT) is an efficient algorithm used in digital signal processing and image processing to compute the Discrete Fourier Transform (DFT) or its inverse rapidly, significantly reducing computational complexity compared to traditional DFT methods.

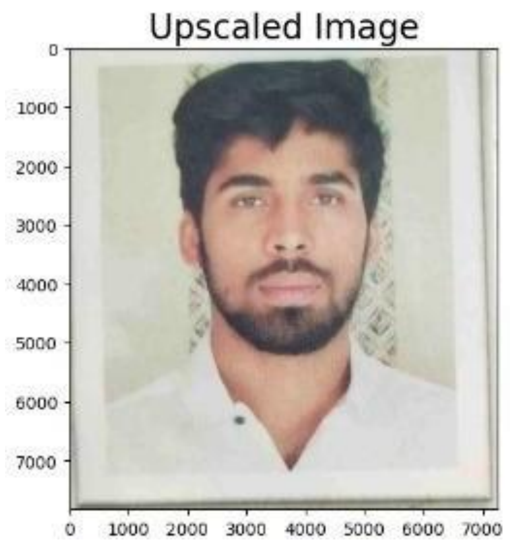
Code: -

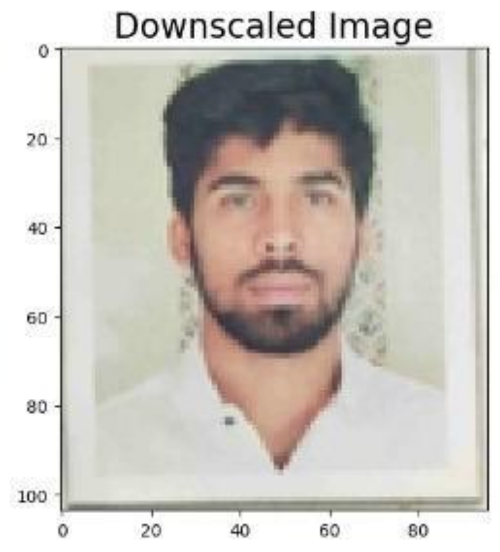
```
from PIL import Image

from skimage.io import imread, imshow, show

import numpy as np

import matplotlib.pylab as pylab

import numpy.fft as fp

from skimage.color import rgb2gray

im = Image.open('./stock/0.jpeg')

im1 = im.resize((im.width*5, im.height*5), Image.NEAREST)

im2 = im.resize((im.width//15, im.height//15), Image.LANCZOS)

freq = fp.fft2(rgb2gray(im))

pylab.figure(figsize=(10,14))
pylab.subplot(2,2,1)

pylab.imshow(im)

pylab.title('Original Image',size=20)

pylab.subplot(2,2,2)

pylab.imshow(20*np.log10(0.01+np.abs(fp.fftshift(freq))))

pylab.title('DFT Spectrum',size=20)

pylab.subplot(2,2,3)

pylab.imshow(im1)

pylab.title('Upscaled Image',size=20)

pylab.subplot(2,2,4)

pylab.imshow(im2)

pylab.title('Downscaled Image',size=20)

pylab.show()
```

Output: -









Conclusion: -

In conclusion, this practical exercise has successfully demonstrated fundamental aspects of signal processing, focusing on both spatial and frequency domain manipulations. Through the implementation of upsampling and downsampling techniques on image or speech signals, I gained insight into the effects of increasing or decreasing signal resolution. Additionally, the utilization of the Fast Fourier Transform (FFT) showcased an efficient method for computing the Discrete Fourier Transform (DFT), enabling the analysis of signal frequency components. By exploring these concepts, I have acquired valuable hands-on experience in signal processing, laying a solid foundation for further exploration and application in diverse fields such as image and speech processing, communications, and audio engineering.

PRACTICAL-02

Aim: - Write program to perform the following on signal:

1. Create a triangle signal and plot a 3-period segment.

2. For a given signal, plot the segment and compute the correlation between them.

Description: -

A triangle wave is a type of waveform characterized by a linear, continuous rise and fall in amplitude over time, resembling the shape of a triangle when plotted on a graph.

A square wave is a type of waveform characterized by a rapid alternation between two distinct voltage levels, typically a high level and a low level, creating a square-shaped pattern when plotted on a graph.

Correlation between signals refers to the degree of similarity or relationship between them, typically quantified by mathematical measures such as Pearson correlation coefficient or cross-correlation function, aiding in tasks like pattern recognition, signal processing, and data analysis.

Mathematical Equation: -

The correlation between two signals x(t) and y(t) can be computed using the cross-correlation function, which is defined as:

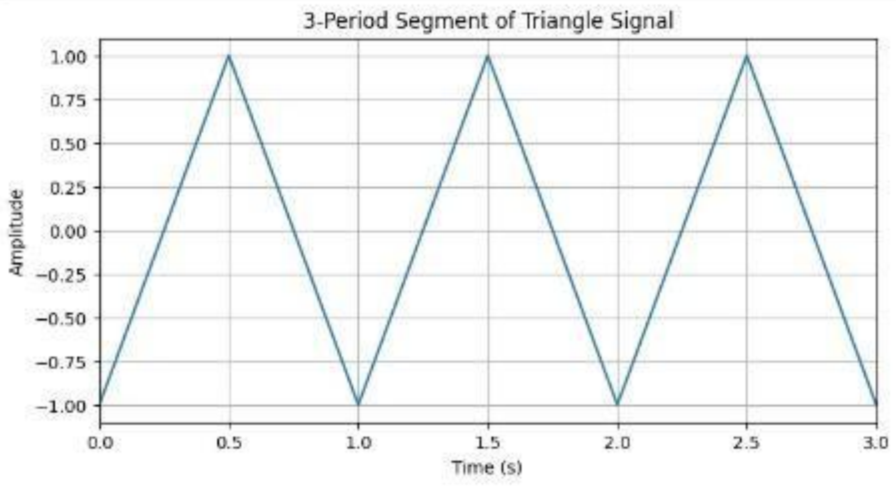
$$Corr(x,y)(\tau) = \int_{-\infty}^{+\infty} x(t)\,.\,y(t+\tau)dt$$

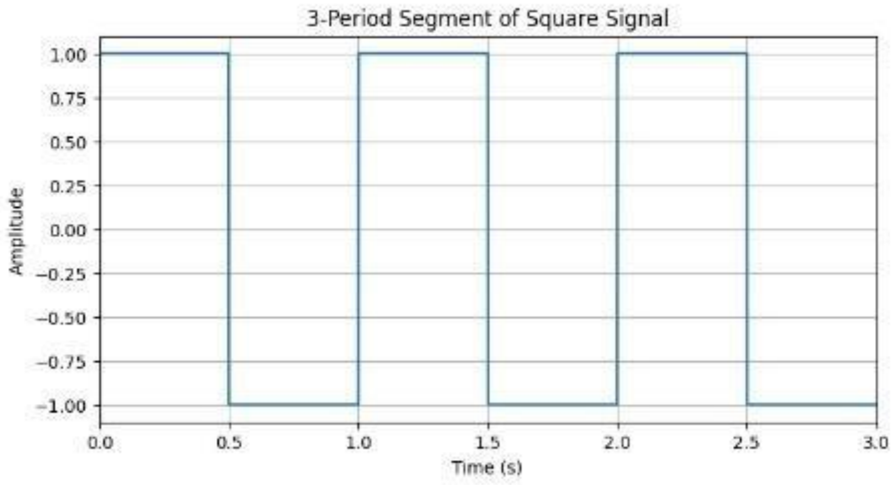
where τ is the time shift, and x(t) and y(t) are the signals being correlated.

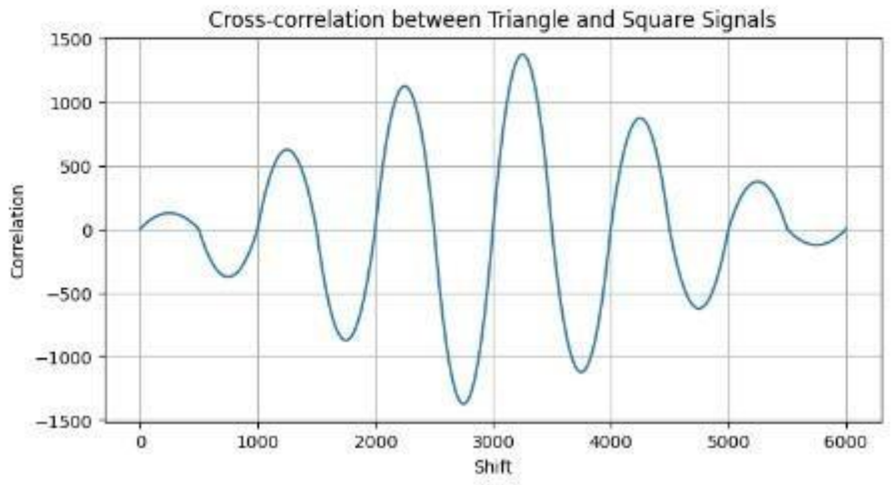
Code: -

```
import numpy as np

import matplotlib.pyplot as plt

from scipy import signal

frequency = 1

duration = 3

sampling_rate = 1000

num_samples = int(duration * sampling_rate)

t = np.linspace(0, duration, num_samples, endpoint=False)

triangle_signal = signal.sawtooth(2 * np.pi * frequency * t, width=0.5)

plt.figure(figsize=(8, 4))

plt.plot(t, triangle_signal, label='Triangle Signal')
plt.xlabel('Time (s)')

plt.ylabel('Amplitude')

plt.title('3-Period Segment of Triangle Signal')

plt.xlim(0, 3 / frequency)

plt.grid(True)

plt.show()

square_signal = np.sign(np.sin(2 * np.pi * frequency * t))

plt.figure(figsize=(8, 4))

plt.plot(t, square_signal, label='Square Signal')

plt.xlabel('Time (s)')

plt.ylabel('Amplitude')

plt.title('3-Period Segment of Square Signal')

plt.xlim(0, 3 / frequency)

plt.grid(True)

plt.show()

correlation = np.correlate(triangle_signal, square_signal, mode='full')

plt.figure(figsize=(8, 4))

plt.plot(correlation)

plt.xlabel('Shift')

plt.ylabel('Correlation')

plt.title('Cross-correlation between Triangle and Square Signals')

plt.grid(True)

plt.show()
```

Output: -







Conclusion: -

In conclusion, in this practical we effectively generated a triangle signal, plotted a three-period segment, and computed its correlation with a square signal. Through this exercise, I gained practical experience in signal generation, visualization, and correlation analysis—key components in signal processing.

PRACTICAL-03

Aim: - Write program to demonstrate the following aspects of signal on sound/image data

1. Convolution operation

2. Template Matching

Description: -

Convolution is a mathematical operation where a kernel or filter is applied to an image to compute the weighted sum of pixel values in a neighborhood around each pixel, used for tasks such as blurring, sharpening, edge detection, and feature extraction.

Template matching is a technique used in image processing and computer vision to locate a specific sub-image, known as a template, within a larger image by comparing pixel values and finding areas where the template closely matches portions of the larger image, enabling tasks such as object detection, recognition, and tracking.

Mathematical Equation: -

The mathematical equation for convolution between two signals$f(t)$ and $g(t)$ is denoted by $(f * g)(t)$ and is defined as

$$(f * g)(t) = \int_{-\infty}^{\infty} f(\tau).g(t-\tau)d\tau$$

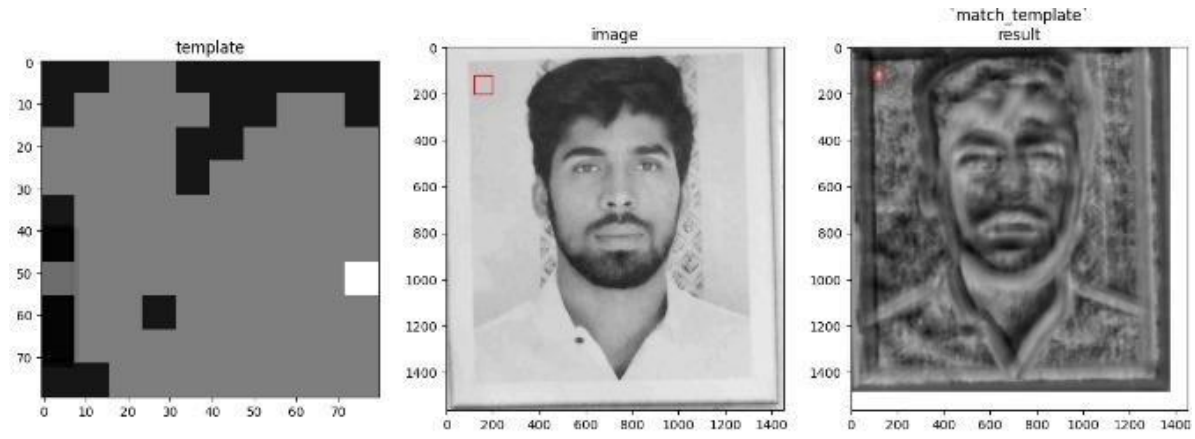
This equation represents the integral of the product of the two signals $f(\tau)$ and $g(t-\tau)$ w.r.t. τ .

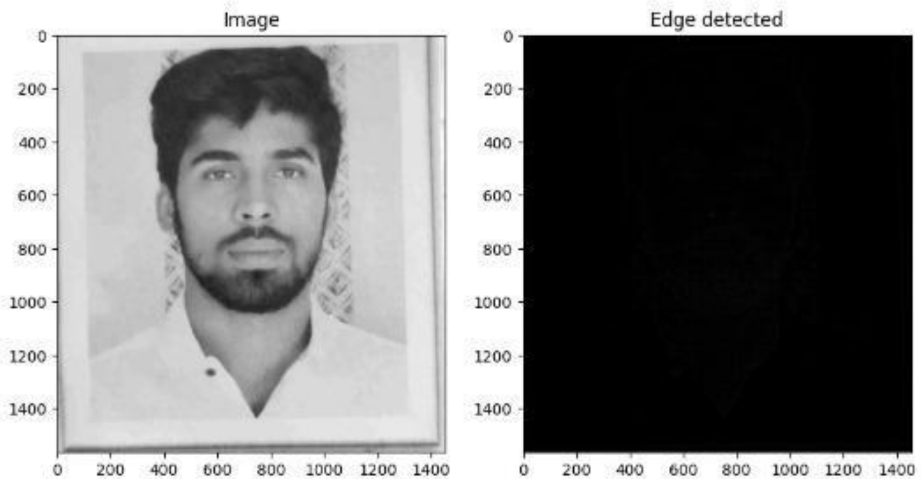
Code: -

```
import numpy as np

import matplotlib.pyplot as plt

import skimage as ski

from scipy.signal import convolve2d

image = ski.color.rgb2gray(ski.io.imread('./stock/0.jpeg'))

template = image[390:445,430:530]

result = ski.feature.match_template(image,template)

ij = np.unravel_index(np.argmax(result),result.shape)

x,y = ij[::-1]

plt.gray()

fig1 = plt.figure(figsize=(16, 6))
```

```
ax1 = plt.subplot(1, 3, 1)

ax2 = plt.subplot(1, 3, 2)

ax3 = plt.subplot(1, 3, 3, sharex=ax2, sharey=ax2)

ax1.imshow(template, cmap=plt.cm.gray)

ax1.set_title('template')

ax2.imshow(image, cmap=plt.cm.gray)

ax2.set_title('image')

print(template.shape)

htemp, wtemp = template.shape

rect = plt.Rectangle((x, y), wtemp, htemp, edgecolor='r', facecolor='none')

ax2.add_patch(rect)

ax3.imshow(result)

ax3.set_title('`match_template`¥nresult')

ax3.autoscale(False)

ax3.plot(x, y, 'o', markeredgecolor='r', markerfacecolor='none', markersize=10)

fig2 = plt.figure(figsize=(10,20))

ax21 = plt.subplot(1,2,1)

ax22 = plt.subplot(1,2,2, sharex=ax21,sharey=ax21)

kernel = np.array([[0,1,0],[1,-4,1],[0,1,0]])

edged = np.clip(convolve2d(image,kernel),0,1)

ax21.imshow(image,cmap=plt.cm.gray)

ax21.set_title('Image')

ax22.imshow(edged,cmap=plt.cm.gray)

ax22.set_title('Convolved Image')

plt.show()
```

Output: -





Conclusion: -

To conclude, in this project I effectively addressed the fundamental aspects of signal processing, specifically through the implementation of convolution operation and template matching techniques on image data. Convolution allowed us to understand how signals interact, while template matching provided insights into pattern recognition in both audio and visual contexts.

PRACTICAL-04

Aim: - Write program to implement point/pixel intensity transformations such as:

1. Log and Power-law transformations
2. Contrast adjustments
3. Histogram Equalization
4. Thresholding, and halftoning operations

Description: -

A log transformation involves taking the logarithm of each pixel intensity value in an image. This transformation is useful for expanding the dynamic range of darker areas in an image, enhancing details that are otherwise difficult to discern, particularly in low-light conditions or when the image has a high dynamic range.

A power-law transformation, also known as gamma correction, involves raising each pixel intensity value to a power (gamma value). This transformation allows for adjusting the overall brightness and contrast of an image. It is particularly useful for enhancing images with non-linear brightness characteristics or compensating for the nonlinear response of display devices.

Contrast adjustment is a process in image processing that involves enhancing the visual difference between light and dark areas in an image, thereby increasing the perceived sharpness and clarity of details. This adjustment aims to expand the range of pixel intensity values across the image, making darker areas darker and lighter areas lighter, ultimately improving the overall visual quality and making objects within the image more distinguishable.

Histogram equalization is a technique used in image processing to enhance the contrast of an image by redistributing the intensity values of pixels. It works by transforming the histogram of the image so that the cumulative distribution function (CDF) of pixel intensities becomes more uniform. This process effectively stretches the dynamic range of the image, making darker regions darker and lighter regions lighter, resulting in improved visual quality and enhanced details.

Thresholding is a technique used in image processing to separate objects or regions within an image based on their pixel intensity values. It involves selecting a threshold value, and then classifying each pixel in the image as belonging to either the foreground or background based on whether its intensity value is above or below the threshold. This process effectively creates a binary image, where pixels above the threshold are typically set to one color (e.g., white) and pixels below the threshold are set to another color (e.g., black). Thresholding is commonly used for tasks such as image segmentation, object detection, and feature extraction.

Halftoning is a technique used in image processing and printing to reproduce continuous-tone images using only two or a small number of colors, typically black and white, or a limited set of colors. It involves breaking down the continuous-tone image into a pattern of dots or other shapes, varying the size and spacing of these dots to simulate different shades of gray or colors. This process allows printers with limited color capabilities, such as black-and-white printers, to produce images that appear to have smooth gradients and tones.

Mathematical Equation: -

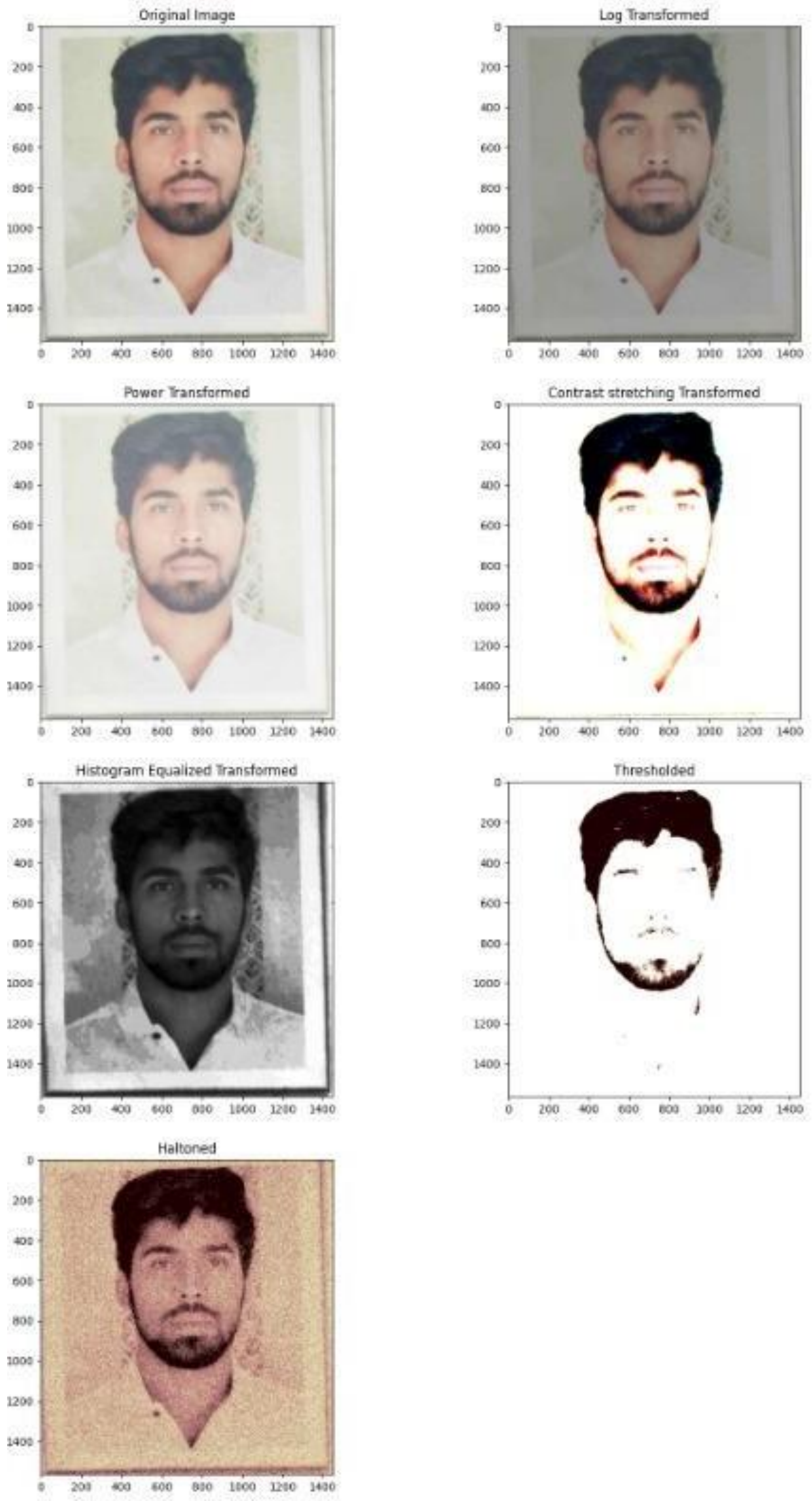
Log Transform: $s = c.\log(1 + r)$, where r is the input pixel intensity, s is the transformed pixel intensity and c is a constant to scale the output intensity.
Power Transform: $s = c.r^{\gamma}$, where r is the input pixel intensity value, s is the transformed intensity pixel intensity value, γ is the gamma value, and c is a constant to scale the output intensity.

Code: -

```
import numpy as np
from skimage import exposure
from skimage.color import rgb2gray
from PIL import Image
import matplotlib.pyplot as plt
img = Image.open('./stock/0.jpeg')
plt.figure(figsize=(14,20))
plt.subplot(3,3,1)
plt.imshow(img,)
plt.title('Original Image')
plt.gray()
#Log Transform
plt.subplot(3,3,2)
im = img.point(lambda i: 255*np.log(1+i/255))
plt.imshow(im)
plt.title('Log Transformed')
#Power Law Transform
gamma=0.5
plt.subplot(3,3,3)
im = img.point(lambda i: 255*(i/255)**gamma)
plt.imshow(im)
plt.title('Power Transformed')
```

```
#Contrast Adjustments

def contrast(c):

    return 0 if c < 70 else (255 if c >150 else(255*c - 19950)/50)

plt.subplot(3,3,4)

im = img.point(contrast)

plt.imshow(im)

plt.title('Contrast stretching Transformed')

#Histogram Equalization

img_eq = exposure.equalize_hist(rgb2gray(img))

img_aeq = exposure.equalize_adapthist(rgb2gray(img),clip_limit=0.03)

plt.subplot(3,3,5)

im = img.point(lambda i: 255*np.log(1+i/255))

plt.imshow(img_eq)

plt.title('Histogram Equalized Transformed')

#Thresholding and Halftoning Operations

plt.subplot(3,3,6)

im = img.convert('L').point(lambda i: i > 90)

plt.imshow(im,  cmap='pink')

plt.title('Thresholded')

plt.subplot(3,3,7)

im = img.convert('L')

im = Image.fromarray(np.clip(im+np.random.randint(-
128,128,(im.height,im.width)),0,255).astype(np.uint8)).point(lambda i : i > 200)

plt.imshow(im,cmap='pink')

plt.title('Haltoned')

plt.show()
```

Output: -



Conclusion: -

In conclusion, in this practical we successfully achieved the aim of implementing various point/pixel intensity transformations in image processing. Through the development of a comprehensive program, we explored essential techniques including log and power-law transformations, contrast adjustments, histogram equalization, thresholding, and halftoning operations. By applying these transformations, we gained insights into how to enhance image quality, improve visibility of details, and manipulate image contrast effectively.

PRACTICAL-05

Aim: - Write a program to apply various enhancements on images using image derivatives by implementing Gradient and Laplacian operations.

Description: -

The gradient operator in image processing is a mathematical operation used to compute the rate of change of pixel intensities in an image. It calculates the magnitude and direction of the intensity gradient at each pixel location, representing how rapidly the intensity values change in the vicinity of that pixel.

The Laplacian operator in image processing is a second-order derivative operator used to detect regions of rapid intensity change in an image. It computes the sum of second-order partial derivatives of the image function, effectively highlighting areas where the intensity varies rapidly, such as edges, corners, and texture boundaries. By applying the Laplacian operator, regions of high spatial frequency, indicative of abrupt changes in intensity, are emphasized, while regions of low frequency, representing smoother transitions, are suppressed.

Mathematical Equation: -

$$grad_x = [[-1,1]]$$

$$grad_y = [[-1],[1]]$$

$$laplace = [[0,-1,0],[-1,4,-1],[0,-1,0]]$$

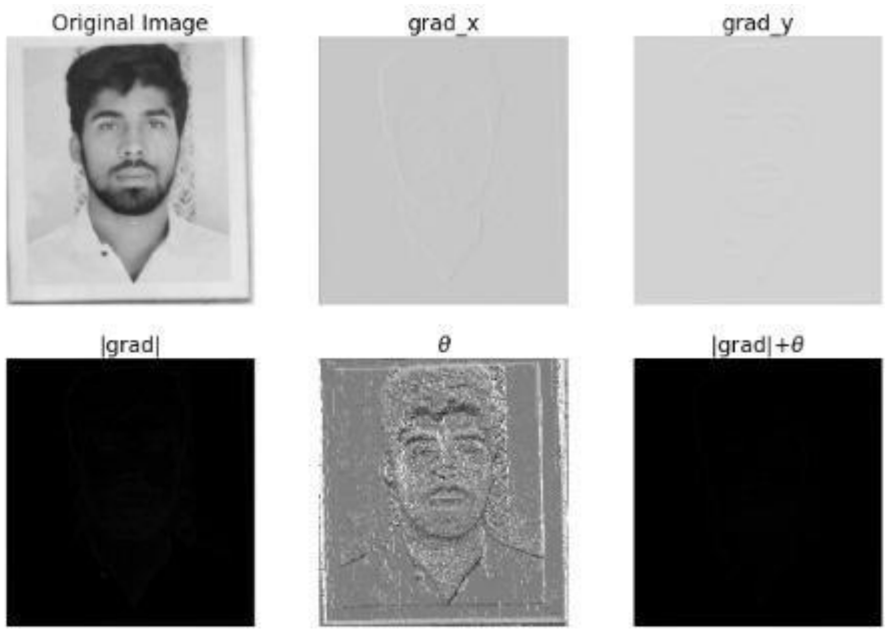
$$mag = \sqrt[2]{grad_x^2 + grad_y^2},$$

$$\theta = \tan^{-1}\left(\frac{grad_y}{grad_x}\right)$$

Code: -

```
import numpy as np
import skimage as ski
import matplotlib.pyplot as plt
import scipy as sp
def plot_image(image, title):
    plt.imshow(image), plt.title(title, size=20), plt.axis('off')
plt.figure(figsize=(15,10))
plt.gray()
plt.subplot(2,3,1)
image = ski.color.rgb2gray(ski.io.imread('./stock/0.jpeg'))
plot_image(image,'Original Image')
#X-dimension gradient
plt.subplot(2,3,2)
ker_x = [[-1, 1]]
derive_x = sp.signal.convolve2d(image,ker_x,mode='same')
plot_image(derive_x,'grad_x')
#Y-dimension gradient
plt.subplot(2,3,3)
ker_y = [[-1], [1]]
derive_y = sp.signal.convolve2d(image,ker_y,mode='same')
plot_image(derive_y,'grad_y')
#Orientation
plt.subplot(2,3,4)
im_mag = np.sqrt(derive_x**2+derive_y**2)
plot_image(im_mag,'|grad|')
#direction
plt.subplot(2,3,5)
im_dir = np.arctan2(derive_y,derive_x)
plot_image(im_dir,r'$¥theta$')
#Grad+dir
plt.subplot(2,3,6)
ker_laplace = [[0,-1,0],[-1,4,-1],[0,-1,0]]
lapim = np.clip(sp.signal.convolve2d(image,ker_laplace,mode='same'),0,1)
plot_image(lapim,r'laplace')
plt.show()
```

Output: -



Conclusion: -

As a result, in this practical I have successfully achieved its aim of developing a program to apply image derivatives, specifically by implementing Gradient and Laplacian operations. The Gradient operation allowed me to detect edges and directional changes in intensity, while the Laplacian operation enabled me to enhance image details and highlight regions of interest.

PRACTICAL-06

Aim: - Write a program to implement linear and nonlinear noise smoothing on suitable image or sound signal.

Description: -

Linear smoothing, also known as linear filtering, is a common technique used in signal and image processing to reduce noise and blur details. It involves convolving the input signal or image with a linear filter kernel, which is typically a small matrix of coefficients. The values in the kernel determine how the surrounding pixels are combined to compute the output value for each pixel in the smoothed image.

Gaussian filter: This filter applies a Gaussian distribution to the kernel, giving more weight to nearby pixels and less weight to distant pixels. It effectively reduces high-frequency noise while preserving edges and details in the image.

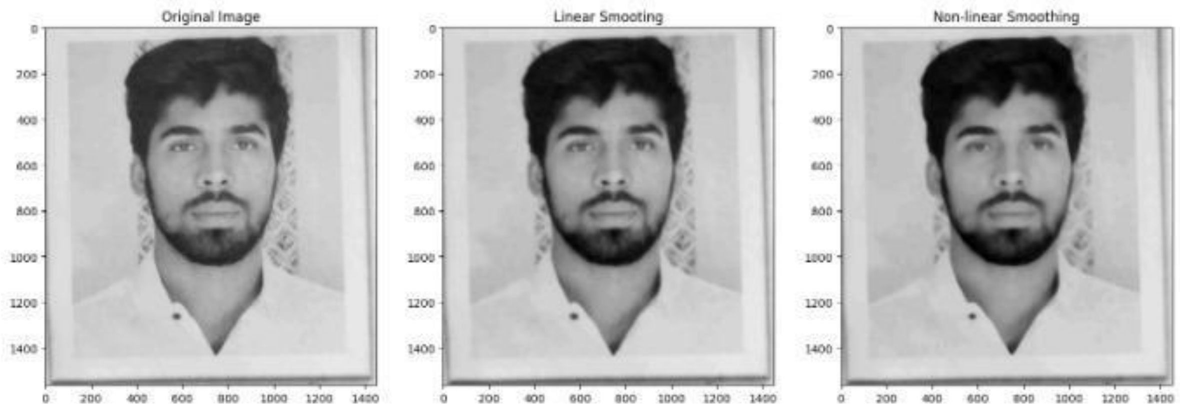
Nonlinear smoothing, unlike linear smoothing, operates without a fixed kernel or predefined weights. Instead, it adaptively processes the input signal or image, making it particularly effective in preserving edges and details while removing noise.

The median filter replaces each pixel value with the median value of neighboring pixels within a defined window. This makes it robust to outliers and impulse noise, such as salt-and-pepper noise, while preserving edges and fine details.

Code: -

```
import cv2 as cv

import numpy as np

import matplotlib.pyplot as plt

img = cv.cvtColor(cv.imread('./stock/0.jpeg'), cv.COLOR_BGR2GRAY)

plt.figure(figsize=(18,10))

plt.subplot(1,3,1)

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

plt.title('Original Image')

#Linear Noise Smoothing

lsimg = cv.GaussianBlur(img, (21,21),0)

plt.subplot(1,3,2)

plt.imshow(lsimg, cmap='gray')
plt.title('Linear Smooting')
#Nonlinear Noise Smoothing
lsimg = cv.medianBlur(img,21)
plt.subplot(1,3,3)
plt.imshow(lsimg, cmap='gray')
plt.title('Non-linear Smoothing')
plt.show()
```

Output: -



Conclusions: -

In conclusion, in this practical I developed a program to implement both linear and nonlinear noise smoothing techniques on appropriate image. Linear smoothing methods, such as Gaussian smoothing and averaging filters, effectively attenuate additive noise by reducing high-frequency components, while nonlinear techniques like median filtering excel at preserving edge details while suppressing impulse noise.

PRACTICAL-07

Aim: - Write a program to apply various image enhancement using image derivatives by implementing smoothing, sharpening, and unsharp masking filters for generating suitable images for specific application requirements.

Description: -

Image derivatives refer to mathematical operations applied to digital images to extract information about the rate of change of pixel intensities.

Smoothing refers to the process of reducing noise and sharp transitions in pixel intensities to create a more uniform appearance in an image.

Gaussian Smoothing: This technique applies a Gaussian kernel to the image, where the weights of the kernel are determined by a Gaussian distribution. Gaussian smoothing effectively reduces noise while preserving image details.

Sharpening is a technique used to enhance the clarity and definition of edges and fine details in an image.

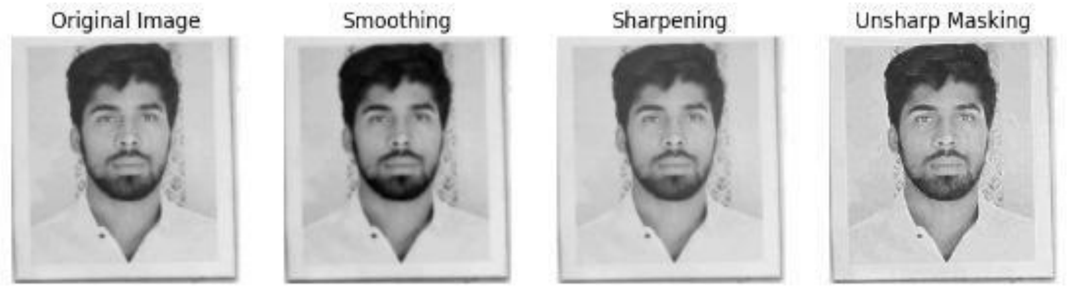
Laplacian Sharpening: The Laplacian operator is applied to the image to highlight regions of rapid intensity change. This operation amplifies high-frequency components, making edges appear more pronounced. The result is then added back to the original image to produce the sharpened image.

In unsharp masking, a blurred version of the original image is subtracted from the original image, resulting in an image that highlights edges and fine details. This sharpening method is effective at enhancing local contrast and improving overall image clarity.

Code: -

```
import numpy as np

import skimage as ski

import matplotlib.pyplot as plt

def plot_image(image,title):

    plt.imshow(image)

    plt.title(title)

    plt.axis('off')

im = ski.color.rgb2gray(ski.io.imread('./stock/0.jpeg'))

plt.gray()

plt.figure(figsize=(15,9))
plt.subplot(1,4,1)

plot_image(im,'Original Image')

#smoothing

plt.subplot(1,4,2)

gausmoothim = ski.filters.gaussian(im,5)

plot_image(gausmoothim,'Smoothing')

#Sharpening

plt.subplot(1,4,3)

lapsharpim = np.clip(im+ski.filters.laplace(im),0,1)

plot_image(lapsharpim,'Sharpening')

#Unsharp Masking

plt.subplot(1,4,4)

usmaskim = np.clip(im + np.clip((im-gausmoothim),0,1)*5,0,1)

plot_image(usmaskim,'Unsharp Masking')

plt.show()
```

Output: -



Conclusions:

To conclude, in this practical I have successfully met its objective of developing a program to enhance images using various techniques based on image derivatives. These techniques include smoothing, sharpening, and unsharp masking filters. Through the implementation of these filters, we have gained valuable insights into how to manipulate image features to meet specific application requirements. Smoothing filters, like Gaussian and median filters, effectively reduce noise and blur, thereby improving image quality for further analysis. Sharpening filters, such as Laplacian and high-pass filters, enhance image details and edges, resulting in clearer images with emphasized features. Unsharp masking, a popular sharpening technique, further enhances image contrast and detail.

PRACTICAL-08

Aim: - Write a program to Apply edge detection techniques such as Sobel and Canny to extract meaningful information from the given image samples.

Description: -

The Sobel edge detector is a popular method used in image processing for edge detection. It operates by convolving the image with a pair of convolution kernels, typically one for detecting horizontal edges and another for vertical edges. These kernels calculate the gradient approximation of the image intensity at each pixel location.\

The Prewitt employs convolution kernels to calculate the gradient approximation of the image intensity. The Prewitt operator consists of two kernels, one for detecting horizontal edges and another for vertical edges.

The Scharr edge detector is a variant of the Sobel and Prewitt operators, designed to provide improved edge detection performance, particularly in regions with high levels of noise. It achieves this by utilizing larger convolution kernels that better approximate the gradient of the image intensity.

The Roberts cross operator is a simple edge detection algorithm used in image processing to detect edges by calculating the gradient approximation of the image intensity.

Laplacian operator is a edge detection technique used in image processing to detect edges by highlighting areas of rapid intensity change.

Mathematical Equation: -

$$\theta = \tan^{-1}\left(\frac{grad_y}{grad_x}\right)$$

$$mag = \sqrt[2]{grad_x^2 + grad_y^2}$$

$$Robert_x = [[-1,1]]$$

$$Robert_y = [[-1][1]]$$

$$\Pr e\,witt_x = [[-1,0,1],[-1,0,1],[-1,0,1]]$$

$$\Pr e\,witt_y = [[-1,-1,-1],[0,0,0],[1,1,1]]$$

$$Sobel_x = [[-1,0,1],[-2,0,2],[-1,0,1]]$$

$$Sobel_y = [[-1,-2,-1],[0,0,0],[1,2,1]]$$

$$Scharr_x = [[-3,0,3],[-10,0,10],[-3,0,3]]$$

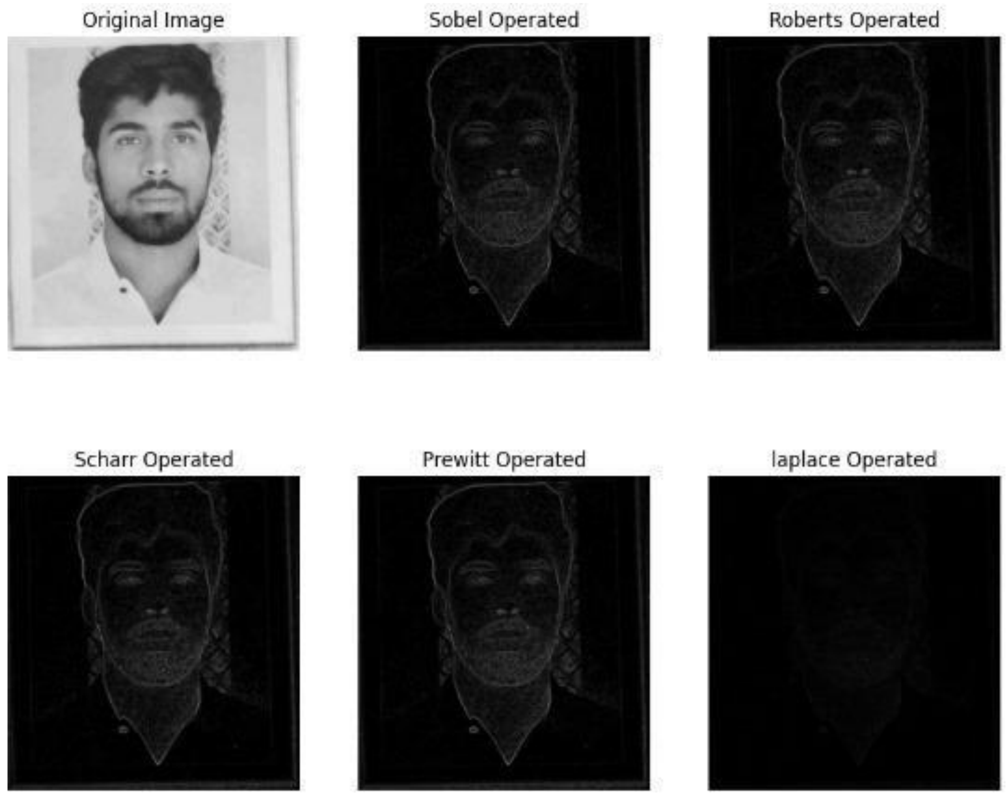
$$Scharr_y = [[-3,-10,-3],[0,0,0],[3,10,3]]$$

$$laplace = [[0,-1,0],[-1,4,-1],[0,-1,0]]$$

Code: -

```
import numpy as np
import skimage as ski
import matplotlib.pyplot as plt
def plot_image(image, title):
    plt.imshow(image)
    plt.title(title)
    plt.axis('off')
im = ski.color.rgb2gray(ski.io.imread('./stock/0.jpg'))
plt.gray()
plt.figure(figsize=(11,9))
plt.subplot(2,3,1)
plot_image(im,'Original Image')
#Sobel
plt.subplot(2,3,2)
plot_image(ski.filters.sobel(im),'Sobel Operated')
#Roberts
plt.subplot(2,3,3)
plot_image(ski.filters.sobel(im),'Roberts Operated')
#Scharr
plt.subplot(2,3,4)
plot_image(ski.filters.sobel(im),'Scharr Operated')
#Prewitt
plt.subplot(2,3,5)
plot_image(ski.filters.sobel(im),'Prewitt Operated')
#Laplace
plt.subplot(2,3,6)
plot_image(np.clip(ski.filters.laplace(im),0,1),'laplace Operated')
plt.show()
```

Output: -



Conclusions: -

In conclusion, this practical has successfully achieved its aim of developing a program to apply edge detection techniques to extract meaningful information from given image samples. The Sobel operator provided a simple yet effective method for edge detection, while the Canny algorithm offered more sophisticated edge detection capabilities with improved noise reduction and edge localization.

PRACTICAL-09

Aim: - Write a program to implement various morphological image processing techniques.

Description: -

Erosion in image processing is a morphological operation that removes pixels at the boundaries of objects, shrinking them and smoothing out irregularities, useful for tasks like separating overlapping objects or reducing noise in binary images.

Dilation in image processing is a morphological operation that adds pixels to the boundaries of objects, expanding them and filling in gaps or holes, useful for tasks like joining broken parts of objects or thickening features in binary images.

Opening in image processing is a morphological operation that combines erosion followed by dilation, effectively removing small-scale features while preserving larger structures, useful for tasks like noise reduction and separating touching objects in binary images.

Closing in image processing is a morphological operation that combines dilation followed by erosion, effectively filling in small gaps and holes while preserving the overall shape and size of objects, useful for tasks like smoothing contours and joining nearby regions in binary images.

White top hat operation in image processing is a morphological operation that computes the difference between the input image and its opening, highlighting bright structures that are smaller than the structuring element, useful for enhancing small bright features in images.

Black top hat operation in image processing is a morphological operation that computes the difference between the input image and its closing, highlighting dark structures that are smaller than the structuring element, useful for enhancing small dark features in images.

Skeletonizing in image processing is a morphological operation that reduces objects within an image to their topological skeletons, representing the thinnest possible representation of the objects while preserving their connectivity and topology, useful for tasks like shape analysis and pattern recognition.

The Beucher gradient is a morphological operator used in image processing to calculate the gradient magnitude of an image. It employs morphological operations, such as dilation and erosion, to compute the gradient by measuring the difference between the dilation and erosion of the image. This operator is particularly effective in edge detection and feature extraction tasks, providing robust results even in the presence of noise or uneven illumination.

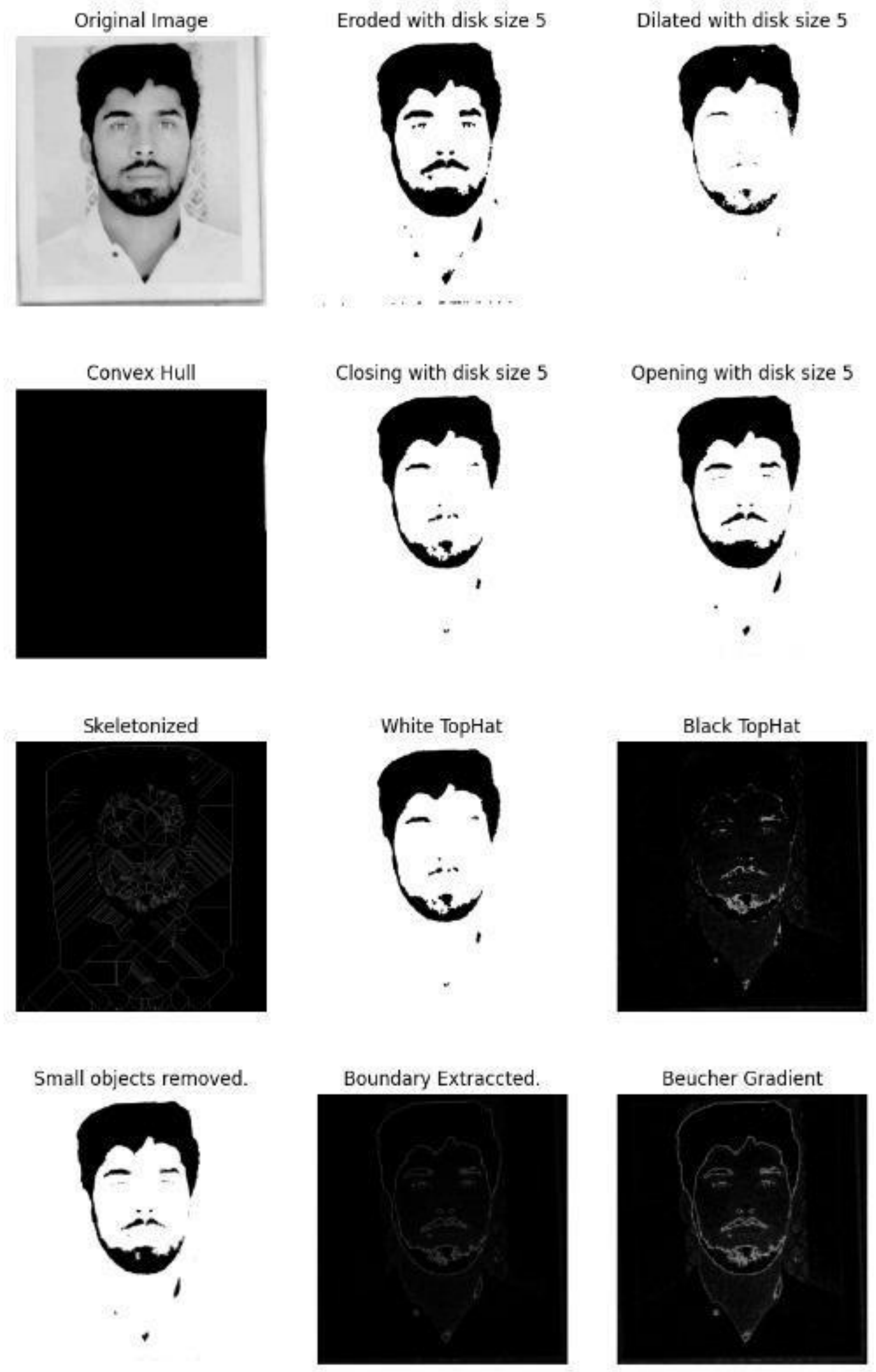
Code: -

```
import numpy as np

import skimage as ski

import matplotlib.pyplot as plt
```

```
def plot_image(image, title=""):
    plt.title(title)
    plt.imshow(image)
    plt.axis('off')
img =  ski.color.rgb2gray(ski.io.imread('./stock/0.jpeg'))
imbin = img
imbin[imbin<=0.4] = 0
imbin[imbin > 0.4] = 1
plt.gray()
plt.figure(figsize=(10,16))
plt.subplot(4,3,1)
plot_image(imbin, 'Original Image')
# Erosion
plt.subplot(4,3,2)
eroded = ski.morphology.binary_erosion(imbin, ski.morphology.disk(5))
plot_image(eroded, "Eroded with disk size 5")
#Dilation
plt.subplot(4,3,3)
dilated = ski.morphology.binary_dilation(imbin, ski.morphology.disk(5))
plot_image(dilated, "Dilated with disk size 5")
#Opening
plt.subplot(4,3,6)
opened = ski.morphology.binary_opening(imbin, ski.morphology.disk(5))
plot_image(opened, "Opening with disk size 5")
#Closing
plt.subplot(4,3,5)
closed = ski.morphology.binary_closing(imbin, ski.morphology.disk(5))
plot_image(closed, "Closing with disk size 5")
#White TopHat
plt.subplot(4,3,8)
```

```
wth = ski.morphology.white_tophat(imbin, ski.morphology.disk(5))

plot_image(wth, "White TopHat")

#BlackTopHat

plt.subplot(4,3,9)

bth = ski.morphology.black_tophat(imbin, ski.morphology.disk(5))

plot_image(bth, "Black TopHat")

#Skeletonizing

plt.subplot(4,3,7)

skeleton = ski.morphology.skeletonize(imbin)

plot_image(skeleton,  "Skeletonized")

#Convex Hulling

plt.subplot(4,3,4)

chull = ski.morphology.convex_hull_image(imbin)

plot_image(chull, "Convex Hull")

#Removing Small Objects

plt.subplot(4,3,10)

rso = ski.morphology.remove_small_objects(imbin.astype(bool),500,1)

plot_image(rso, "Small objects removed.")

#Extracting Boundary

plt.subplot(4,3,11)

be = imbin - ski.morphology.erosion(imbin)

plot_image(be, "Boundary Extraccted.")

#Beucher Gradient

plt.subplot(4,3,12)

mg = ski.morphology.dilation(img,ski.morphology.rectangle(3,3)) -
ski.morphology.erosion(img,ski.morphology.rectangle(3,3))

plot_image(mg, "Beucher Gradient")

plt.show()
```

Output: -


Original Image
Eroded with disk size 5
Dilated with disk size 5
Convex Hull
Closing with disk size 5
Opening with disk size 5
Skeletonized
White TopHat
Black TopHat
Small objects removed.
Boundary Extraccted.
Beucher Gradient

Conclusions:

In conclusion, this practical endeavor has effectively achieved its aim of developing a program to implement various morphological image processing techniques. Morphological operations such as dilation, erosion, opening, and closing have been successfully implemented, allowing us to modify and enhance image features based on their spatial properties.

PRACTICAL-10

Aim: - Write a program to extract image features by implementing methods like corner and blob detectors, HoG and Haar features.

Description: -

The Harris corner detector is a widely-used algorithm in computer vision for detecting corners in images. It identifies points where small shifts in the image produce significant changes in intensity values, indicating the presence of corners or edges.

HoG, or Histogram of Oriented Gradients, is a feature extraction technique widely used in computer vision for object detection and recognition. It computes the distribution of gradient orientations in localized regions of an image, providing a compact representation of its texture and shape properties.

LoG, or Laplacian of Gaussian, is an image processing technique used for edge detection and feature extraction. It involves first smoothing the image with a Gaussian filter to reduce noise, followed by applying the Laplacian operator to detect edges and other significant intensity changes. This combined approach helps to enhance edges and highlight important features in the image.

DoG, or Difference of Gaussians, is an image processing technique used for edge detection and feature extraction. It involves computing the difference between two Gaussian-smoothed versions of the image at different scales. This process enhances edges and highlights features by emphasizing intensity changes at different spatial frequencies.

DoH, or Determinant of Hessian, is a feature extraction technique used in computer vision for detecting blob-like structures in images. It involves computing the determinant of the Hessian matrix, which measures the local curvature of intensity values. This technique is effective for identifying regions with significant intensity variations, such as blobs, corners, or junctions, across different scales in an image.

Code: -

```
import numpy as np

import skimage as ski

import matplotlib.pyplot as plt

def plot_image(image, title=""):

    plt.title(title)

    plt.imshow(image)

    plt.axis('off')

img = ski.io.imread('./stock/0.jpeg')
```

```
imgray = ski.color.rgb2gray(img)

plt.figure(figsize=(10,12))

plt.gray()

plt.subplot(2,3,1)

plot_image(img,'Original Image')

#Harris Corner Detector

plt.subplot(2,3,2)

coordinates = ski.feature.corner_harris(imgray,k=0.001)

hcdim = img

hcdim[coordinates > 0.01*coordinates.max()]=[255,0,0]

plot_image(hcdim,'Harris-Corner')

plt.axis('off')

#Blob Detectors

#LOG

log_blobs = ski.feature.blob_log(imgray, max_sigma=30, num_sigma=10, threshold =
0.1)

log_blobs[:,2] = np.sqrt(2)*log_blobs[:,2]

plt.subplot(2,3,4)

plot_image(img, 'LoG')

for blob in log_blobs:

    y,x,rad = blob

    col = plt.Circle((x,y), rad, color = 'red', linewidth=2, fill=False)

    plt.subplot(2,3,4).add_patch(col)

plt.axis('off')

#DoG

dog_blobs = ski.feature.blob_dog(imgray, max_sigma=30, threshold = 0.1)

dog_blobs[:,2] = np.sqrt(2)*dog_blobs[:,2]

plt.subplot(2,3,5)

plot_image(img, 'DoG')

for blob in dog_blobs:
    y,x,rad = blob

    col = plt.Circle((x,y), rad, color = 'blue', linewidth=2, fill=False)

    plt.subplot(2,3,5).add_patch(col)

plt.axis('off')

#DoH

doh_blobs = ski.feature.blob_doh(imgray, max_sigma=30, num_sigma=10, threshold =
0.005)

doh_blobs[:,2] = np.sqrt(2)*doh_blobs[:,2]

plt.subplot(2,3,6)

plot_image(img, 'DoH')

for blob in doh_blobs:

    y,x,rad = blob

    col = plt.Circle((x,y), rad, color = 'green', linewidth=2, fill=False)

    plt.subplot(2,3,6).add_patch(col)

plt.axis('off')

#Histogram of Gradients

plt.subplot(2,3,3)

fd, hog_image =
ski.feature.hog(imgray,orientations=8,pixels_per_cell=(16,16),cells_per_block=(1,1
), visualize=True)

hog_image = ski.exposure.rescale_intensity(hog_image, in_range=(0,20))

plot_image(hog_image,"HoG")

plt.axis('off')

plt.show()
```

Output: -

Conclusions:

To conclude, in this practical we have successfully achieved its aim of developing a program to extract image features using various methods, including corner and blob detectors and Histogram of Oriented Gradients (HoG). Corner and blob detectors have allowed us to identify key points in the image, while HoG features have enabled us to capture shape and texture information effectively

PRACTICAL-11

Aim: - Write a program to apply segmentation for detecting lines, circles, and other shapes/ objects. Also, implement edge-based and region-based segmentation.

Description: -

The Hough Line Transform is a technique used in image processing and computer vision to detect straight lines within an image. It works by converting the Cartesian coordinate space (x, y) into a parameter space (ρ, θ), where ρ represents the distance from the origin to the closest point on the line, and θ represents the angle between the x-axis and the normal line to the detected line. By representing lines in this polar coordinate space, the Hough Transform simplifies the process of detecting lines by searching for peaks in the parameter space, corresponding to lines in the original image. This technique is robust to noise and capable of detecting lines even in the presence of gaps or interruptions.

In the Hough Circle Transform, each pixel in the image space contributes to a set of circles in the Hough parameter space. The parameters of interest typically include the center coordinates (x, y) of the circle and its radius (r). By iterating over all possible combinations of (x, y, r), the Hough Circle Transform identifies the circles that best fit the edge points detected in the image.

Edge-based segmentation is a technique in image processing and computer vision that partitions an image into regions based on the presence of edges or intensity gradients. Instead of directly segmenting the image into regions based on pixel intensity, edge-based segmentation identifies boundaries between different regions by detecting edges.

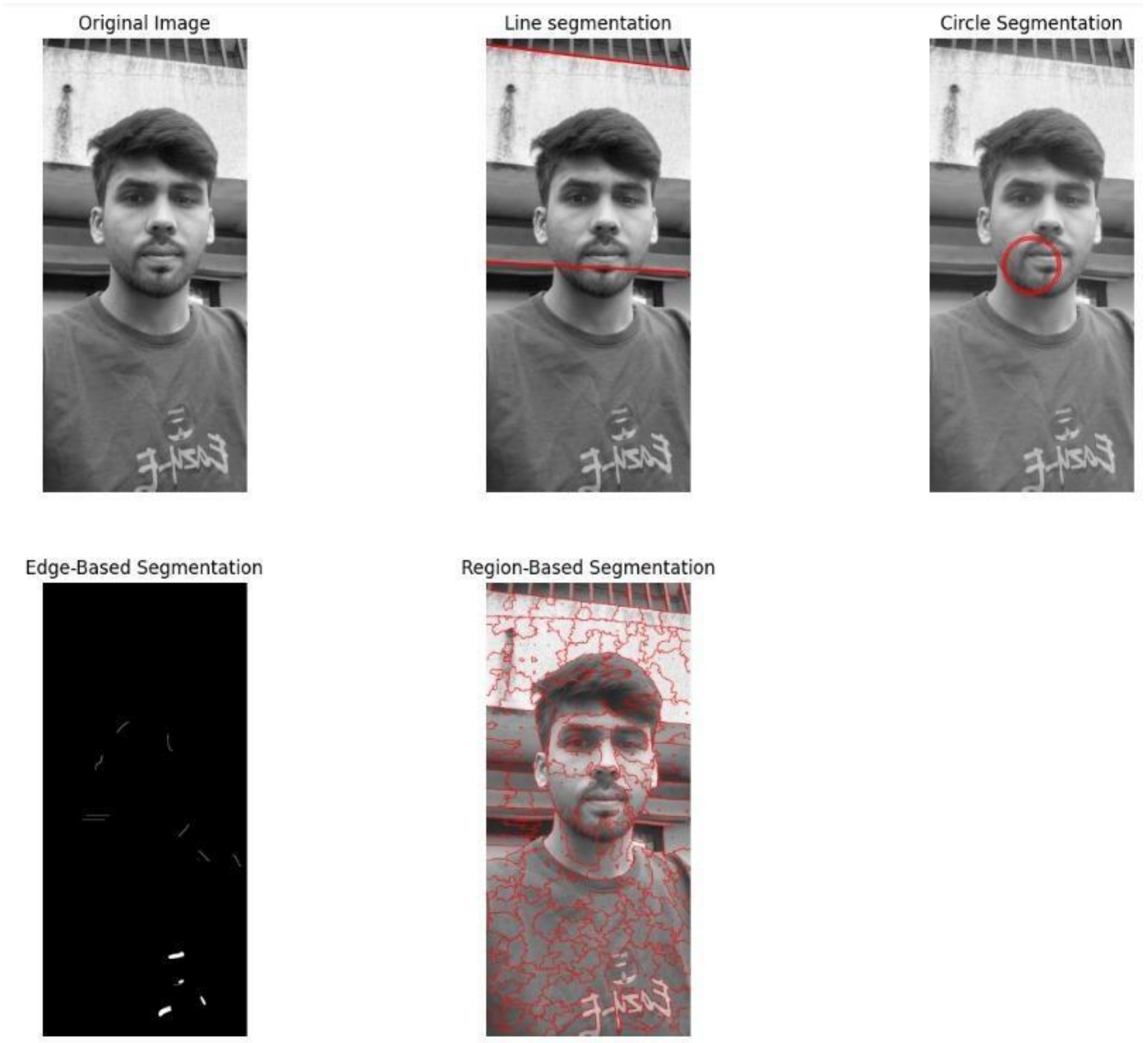
Region-based segmentation is a technique in image processing and computer vision that partitions an image into coherent regions based on similarity criteria such as color, texture, or intensity. Unlike edge-based segmentation, which relies on detecting edges, region-based segmentation directly groups pixels into meaningful regions. Region-based segmentation is effective for images with homogeneous regions and gradual intensity changes.

Code: -

```
import numpy as np

import skimage as ski

import matplotlib.pyplot as plt

import scipy as sp

def plot_image(image, title=""):

    plt.title(title)

    plt.imshow(image)
    plt.axis('off')
image = ski.color.rgb2gray(ski.io.imread('./stock/0.jpeg'))
plt.figure(figsize=(15,12))
plt.gray()
plt.subplot(2,3,1)
plot_image(image,'Original Image')
#Straight-line hough transform
plt.subplot(2,3,2)
h, theta, d = ski.transform.hough_line(ski.feature.canny(image,2.5))
for _, angle, dist in zip(*ski.transform.hough_line_peaks(h,theta,d)) :
    y0 = (dist - 0 * np.cos(angle))/np.sin(angle)
    y1 = (dist - image.shape[1] * np.cos(angle))/np.sin(angle)
    plt.plot((0,image.shape[1]),(y0,y1),'-r')
plot_image(image, 'Line segmentation')
plt.subplot(2,3,2).set_xlim((0,image.shape[1]))
plt.subplot(2,3,2).set_ylim((image.shape[0],0))
#Circle Hough Transform
plt.subplot(2,3,3)
hough_radii = np.arange(65,75,1)
hough_res = ski.transform.hough_circle(image,hough_radii)
accums, c_x, c_y, radii = ski.transform.hough_circle_peaks(hough_res,
hough_radii,total_num_peaks=8)
img = ski.color.gray2rgb(image)
for cen_y, cen_x, radius in zip(c_y,c_x,radii):
    circ_y, circ_x = ski.draw.circle_perimeter(cen_y,cen_x,radius)
    img[circ_y,circ_x] = (1,0,0)
plot_image(img,'Circle Segmentation')
#Edge-based Segmentation
plt.subplot(2,3,4)
edges = ski.feature.canny(image, sigma=2)
```

```
imces =
ski.morphology.remove_small_objects(sp.ndimage.binary_fill_holes(edges),50)

plot_image(imces,'Edge-Based Segmentation')

#Region-based Segmentation

plt.subplot(2,3,5)

gradient = ski.filters.sobel(image)

segments_watershed = ski.segmentation.watershed(gradient,markers = 500,
compactness=0.00001)

plot_image(ski.segmentation.mark_boundaries(image,segments_watershed,color=(1,0,0)
),'Region-Based Segmentation')

plt.show()
```

Output: -



Conclusions:

As a result, practical has successfully achieved its aim of developing a program to perform segmentation for detecting lines, circles, and other shapes/objects, while also implementing both edge-based and region-based segmentation techniques. Edge-based segmentation methods, such as the Canny edge detector, have allowed us to detect object boundaries based on intensity gradients, while region-based segmentation techniques, such as the watershed algorithm, have enabled us to group pixels into coherent regions based on similarity criteria.