UNIVERSITY OF MUMBAI

DEPARTMENT OF COMPUTER SCIENCE

मुंबई विद्यापीठ
University of Mumbai
Re-accredited with A++ Grade
(CGPA 3.65) by NAAC (3rd Cycle 2021)

M.Sc. Computer Science – Semester I

Applied Signal and Image Processing Practical

JOURNAL

2023-2024

Seat No. _____

मुंबई विद्यापीठ
University of Mumbai
Re-accredited with A++ Grade
(CGPA 3.65) by NAAC (3rd Cycle 2021)

UNIVERSITY OF MUMBAI

DEPARTMENT OF COMPUTER SCIENCE

CERTIFICATE

This is to certify that the work entered in this journal was done in the University Department of Computer Science laboratory by Mr./Ms._____ Seat No. _____ for the course of M.Sc. (Computer Science) - Semester I (NEP 2020) during the academic year 2023- 2024 in a satisfactory manner.

_____ _____

Subject In-charge **Head of Department**

_____

External Examiner

INDEX

PRACTICAL-01

Aim: Write program to demonstrate the following aspects ofsignal processing on suitable data

1. Upsampling and downsampling on Image/speechsignal.
2. Fast Fourier Transform to compute DFT

Theory:

Up-sampling

The number of pixels in the down-sampled image can be increased by using up-sampling interpolation techniques. The up-sampling technique increases the resolution as well as the size of the image.

Down-sampling

In the down-sampling technique, the number of pixels in the given image is reduced depending on the sampling frequency. Due to this, the resolution and size of the image decrease.

Fast Fourier Transform

To compute the DFT of an N-point sequence using equation (1) would take O(N2) multiplies and adds. The FFT algorithm computes the DFT using O(N log N) multiplies and adds. There are many variants of the FFT algorithm.

Mathematical Equation:

1. **Up-sampling**

$$I_{\text{upscaled}}(x', y') = \sum_{i=0}^{1} \sum_{j=0}^{1} I(x+i, y+j) \cdot w(i,j)$$

2. **Down-sampling**

$$I_{\text{downscaled}}(x', y') = \frac{1}{n} \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} I(x \cdot n + i, y \cdot n + j)$$

3. **Fast Fourier Transform**

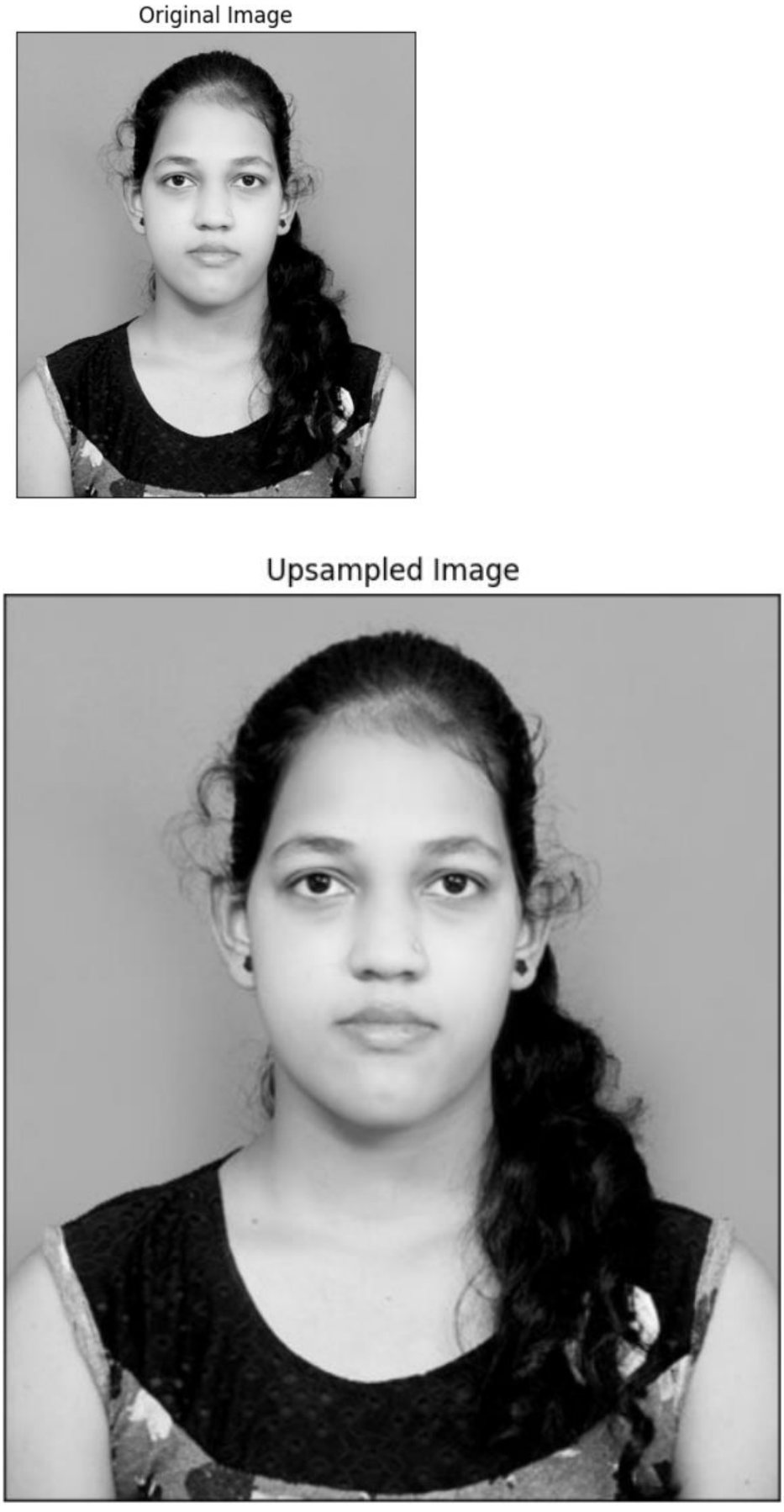
$$X(k) = \sum_{n=0}^{N-1} x(n) \cdot e^{-\frac{2\pi i}{N} nk}$$

Code:

1. Upsampling and downsampling on Image/speech signal.

```
import cv2
import numpy as np
image_path=("Samiksha.jpg")
gray_image=cv2.imread(image_path,cv2.IMREAD_GRAYSCALE)
plt.imshow(gray_image,cmap='gray')
plt.title("Original Image")
plt.axis('off')
plt.show()

upscale_factor= 5
upsampled_image=cv2.resize(gray_image,None,fx=upscale_factor,fy=upscale_factor,inte
rpolation=cv2.INTER_LINEAR)
plt.subplots(figsize=(15,7))
plt.imshow(upsampled_image,cmap='gray')
plt.title("Upsampled Image")
plt.axis('off')
plt.show()


downscale_factor= 0.5
downsampled_image=cv2.resize(gray_image,None,fx=downscale_factor,fy=downscale_f
actor,interpolation=cv2.INTER_LINEAR)
plt.subplots(figsize=(3,3))
plt.imshow(downsampled_image,cmap='gray')
plt.title("Downsampled Image")
plt.axis('off')
plt.show()
```

Output:



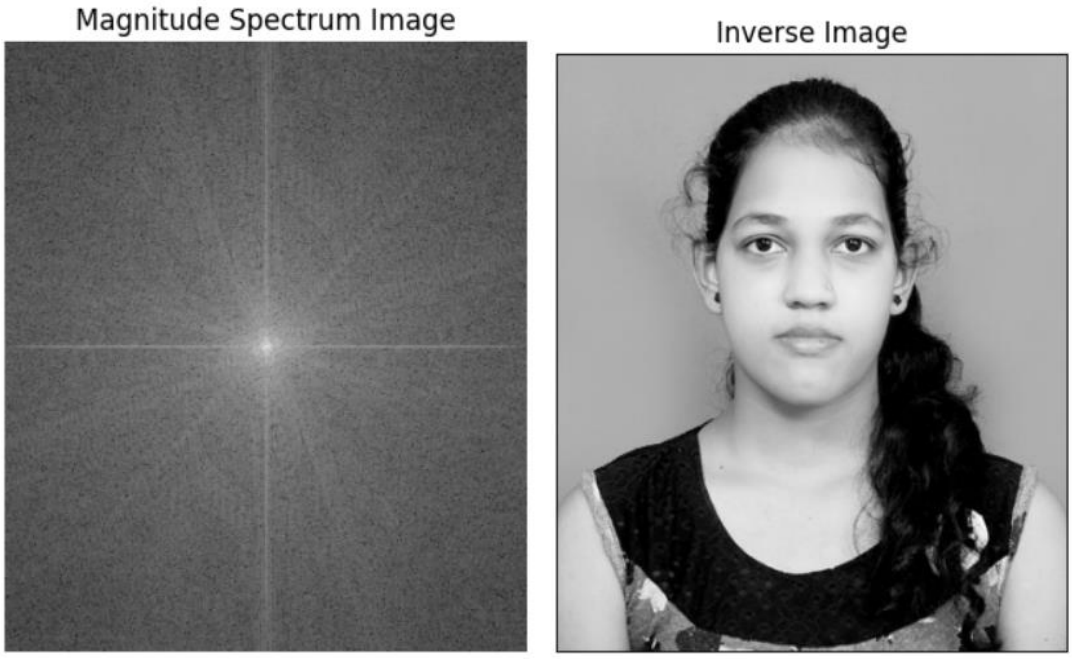
2. Fast Fourier Transform to compute DFT

Code:

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

#Load a grayscale image
image_path=("Samiksha.jpg")
gray_image=cv2.imread(image_path,cv2.IMREAD_GRAYSCALE)

#Display the original image
plt.imshow(gray_image,cmap='gray')
plt.title("Original Image")
plt.axis('off')
plt.show()

#Compute the 2D FFT of the image
f_transform=np.fft.fft2(gray_image)
f_transform_shifted=np.fft.fftshift(f_transform)

#Compute the magnitude spectrum
magnitude_spectrum=np.abs(f_transform_shifted)

#Display the magnitude spectrum
plt.imshow(np.log1p(magnitude_spectrum),cmap='gray')
plt.title("Magnitude Spectrum Image")
plt.axis('off')
plt.show()

#Compute the inverse FFT to get the original image back
inverse_transform_shifted=np.fft.ifftshift(f_transform_shifted)
```

```
inverse_image=np.fft.ifft2(inverse_transform_shifted).real

#Display the inverse image
plt.imshow(inverse_image,cmap='gray')
plt.title("Inverse Image")
plt.axis('off')
plt.show()
```

Output:



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.

Theory:

1. Create a triangle signal and plot a 3-period segment:
   A triangle signal is a waveform characterized by linearly increasing and decreasing amplitude, creating a triangular shape. To visualize this, you can plot a 3-period segment of the triangle signal, showing three complete cycles. The plot highlights the oscillatory nature of the signal, where amplitude rises and falls in a linear fashion over the specified period. This graphical representation is valuable for understanding the periodic behavior of the triangle wave.

2. For a given signal, plot the segment and compute the correlation between them:
   plotting a segment of a signal helps you visualize it, and computing the correlation helps you understand how similar or different two signals are. This can be useful in various fields like signal processing, finance, and scientific research to analyze and interpret data.

Mathematical Equation :

$$x(t) = A \cdot \left| \frac{4t}{T} - \left\lfloor \frac{4t}{T} + \frac{1}{2} \right\rfloor - 1 \right|$$

Here,

- $x(t)$ represents the amplitude of the triangle wave at time t,
- A is the amplitude of the wave,
- T is the period of one cycle, and
- $\lfloor \rfloor$ denotes the floor function.

Code:

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

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

# Function to create a triangle wave
def triangle_wave(periods, samples_per_period):
  t = np.linspace(0, periods*2*np.pi, periods*samples_per_period)
  return np.abs((2 / np.pi) * (t - np.pi * np.floor((t + np.pi) / (2 * np.pi))))

# Load an example image
```

```
image =cv2.imread('Samiksha.jpg')

# Generate a triangle wave with 3 periods
periods = 3
samples_per_period = 1000
triangle_signal = triangle_wave(periods, samples_per_period)

# Normalize the signal to the range [0, 255]
normalized_signal = (triangle_signal / np.max(triangle_signal) * 255).astype(np.uint8)

# Create a black canvas
canvas = np.zeros_like(image)

# Draw the triangle wave on the canvas
for i, amplitude in enumerate(normalized_signal):
    cv2.line(canvas, (i, 0), (i, amplitude), (255, 255, 255), 1)

# Blend the canvas with the original image
result = cv2.addWeighted(image, 0.7, canvas, 0.3, 0)

# Display the result
cv2.imshow('Image with Triangle Wave', result)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Output:

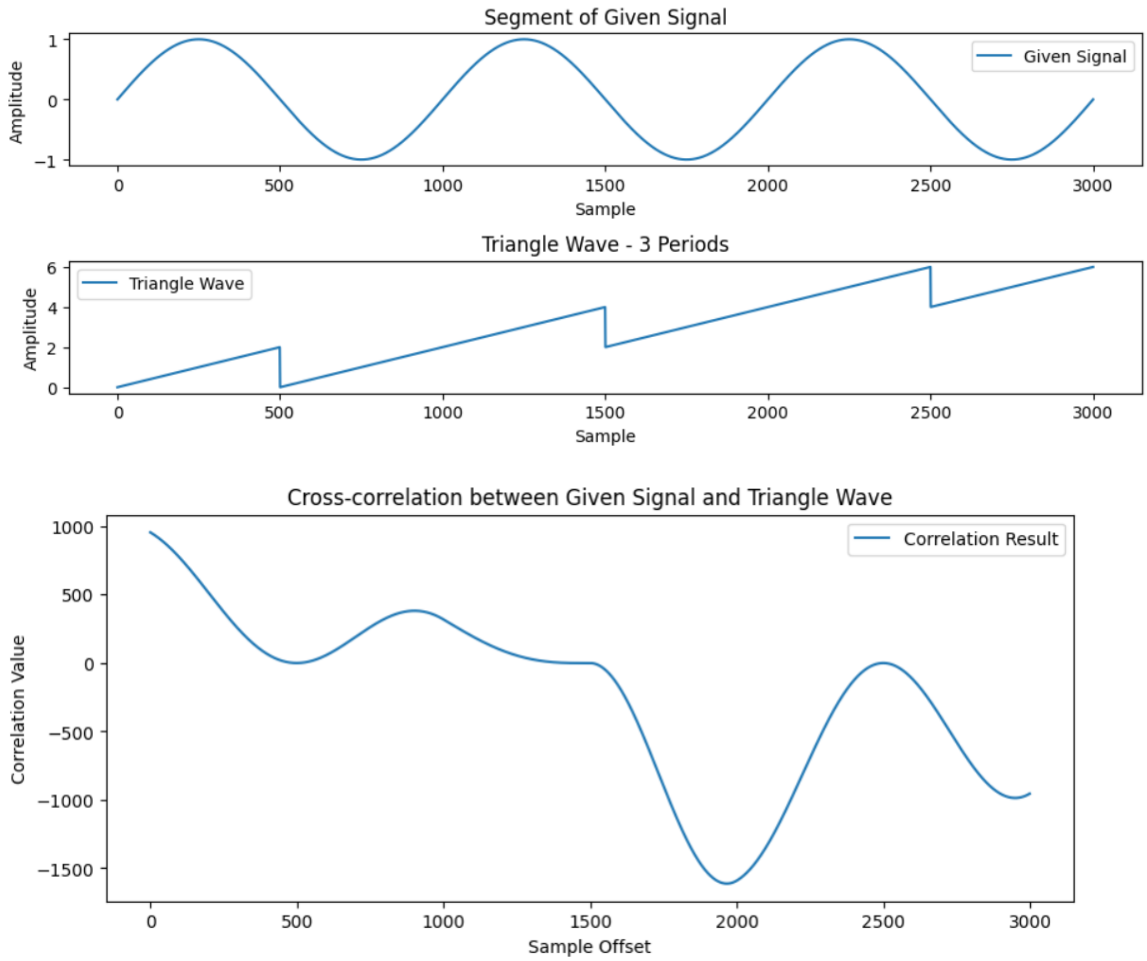
2. For a given signal, plot the segment and compute the correlation between them

Code:

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
from scipy.signal import correlate

# Function to create a triangle wave
def triangle_wave(periods, samples_per_period):
    t = np.linspace(0, periods * 2 * np.pi, periods * samples_per_period)
    return np.abs((2 / np.pi) * (t - np.pi * np.floor((t + np.pi) / (2 * np.pi))))

# Function to create a sine wave (replace this with your own signal)
def sine_wave(periods, samples_per_period):
    t = np.linspace(0, periods * 2 * np.pi, periods * samples_per_period)
    return np.sin(t)

# Load an example image
image = cv2.imread('Samiksha.jpg') # Replace 'Samiksha.jpg' with your image file

# Generate a triangle wave with 3 periods
periods = 3
samples_per_period = 1000
triangle_signal = triangle_wave(periods, samples_per_period)

# Generate a sample sine wave as the given signal
given_signal = sine_wave(periods, samples_per_period)

# Plot the segment of the given signal
plt.figure(figsize=(10, 4))
plt.subplot(2, 1, 1)
plt.plot(given_signal, label='Given Signal')
plt.title('Segment of Given Signal')
plt.xlabel('Sample')
plt.ylabel('Amplitude')
plt.legend()

# Plot the triangle wave
plt.subplot(2, 1, 2)
plt.plot(triangle_signal, label='Triangle Wave')
plt.title('Triangle Wave - 3 Periods')
plt.xlabel('Sample')
plt.ylabel('Amplitude')
plt.legend()
```

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

# Compute the correlation between the given signal and the triangle wave
correlation_result = correlate(given_signal, triangle_signal, mode='same')

# Plot the correlation result
plt.figure(figsize=(10, 4))
plt.plot(correlation_result, label='Correlation Result')
plt.title('Cross-correlation between Given Signal and Triangle Wave')
plt.xlabel('Sample Offset')
plt.ylabel('Correlation Value')
plt.legend()
plt.show()
```

Output:



PRACTICAL-03

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

1. Convolution operation
2. Template Matching

Theory:

CONVOLUTION OPERATION:-

Convolution is a mathematical operation that allows the merging of two sets of information. In the case of CNN, convolution is applied to the input data to filter the information and produce a feature map. This filter is also called a kernel, or feature detector, and its dimensions can be, for example, 3x3.

We've already described how convolution layers work above. They are at the center of CNNs, enabling them to autonomously recognize features in the images.

TEMPLATE MATCHING:-

Template matching is the process of moving the template over the entire image and calculating the similarity between the template and the covered window on the image. Template matching is implemented through two-dimensional convolution .In convolution, the value of an output pixel is computed by multiplying elements of two matrices and summing the results. One of these matrices represents the image itself, while the other matrix is the template, which is known as a convolution kernel.

Mathematical Equation:

1. **Convolution Operation**
   The equation describes the process of sliding the filter over the input and computing the sum of products at each position.

$$y[n] = \sum_{k=0}^{M-1} x[k] \cdot h[n-k]$$

   Here, y[n] represents the output, x[k] is the input sequence, h[n] is the filter or kernel, and M is the length of the filter.

2. **Template Matching**

$$R(u,v) = \frac{\sum_{x,y}(I(x,y)-\bar{I})\cdot(T(x-u,y-v)-\bar{T})}{\sqrt{\sum_{x,y}(I(x,y)-\bar{I})^2 \cdot \sum_{x,y}(T(x-u,y-v)-\bar{T})^2}}$$

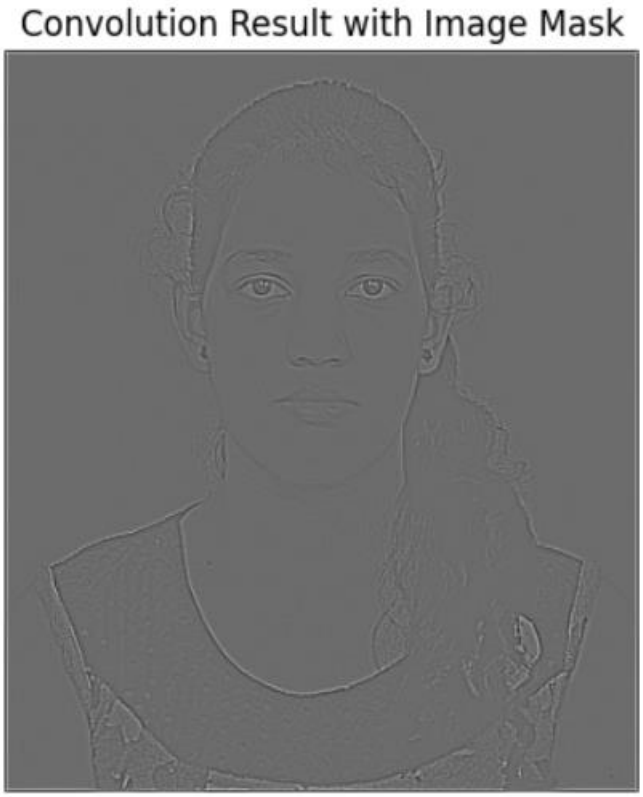
Here, R(u,v) represents the similarity score at position (u,v), $\bar{I}$ and $\bar{T}$ are the mean intensities of the input image and template, respectively, and the summation is performed over all image pixels. This equation quantifies the similarity between the template and the corresponding region in the input image

Code:

1. Convolution Operation

```
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image
from scipy.signal import convolve2d

#Load a sample gray image
image_path="Samiksha.jpg"
image = Image.open(image_path).convert('L')

#Convert to grayscale
image_array=np.array(image)/255.0 #Normalize pixel values to the range [0,1]

#Display the original image
plt.imshow(image_array,cmap='gray')
plt.title("Original Image")
plt.axis('off')
plt.show()

#Define a convolution kernel (image mask)
#Here, we use a simple edge detection kernel
kernel=np.array([[-1,-1,-1],
                 [-1,8,-1],
                 [-1,-1,-1]])
#Perform convolution using the defined kernel
con_result=convolve2d(image_array,kernel,mode='same',boundary='symm')

#Display the result of the convolution
plt.imshow(con_result,cmap='gray')
plt.title("Convolution Result with Image Mask")
plt.axis('off')
plt.show()
```

Output:



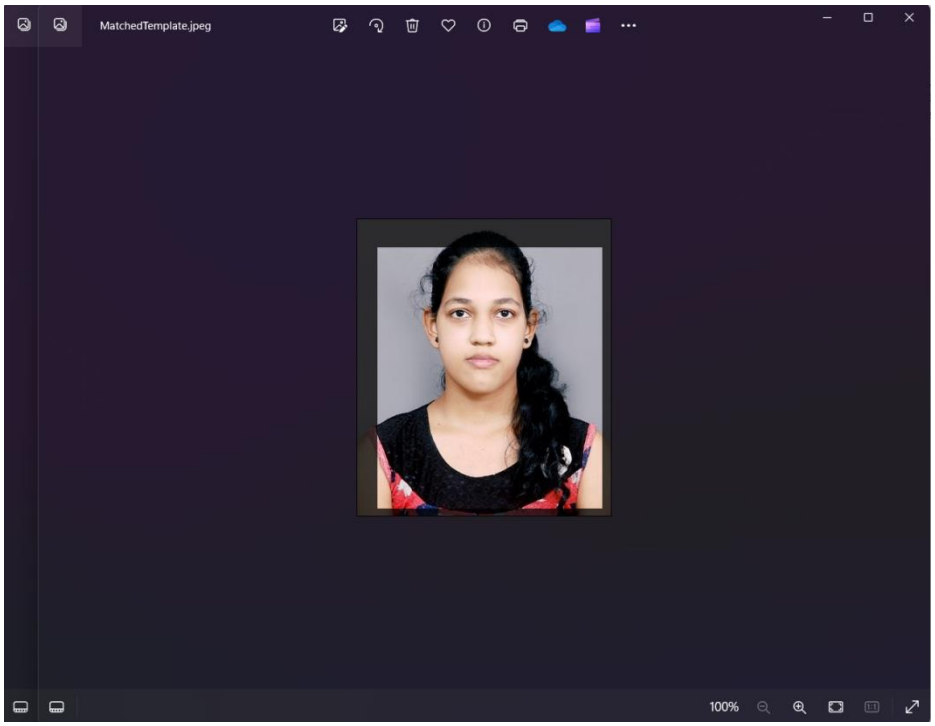
2.Template Matching

```
import cv2
import numpy as np

image=cv2.imread('Samiksha.jpg')
template=cv2.imread("template.png")

(templateHeight,templateWidth)= template.shape[:2]

matchResult=cv2.matchTemplate(image,template,cv2.TM_CCOEFF)
(_,_,minLoc,maxLoc)=cv2.minMaxLoc(matchResult)
topLeft=maxLoc
botRight=(topLeft[0]+templateWidth,topLeft[1]+templateHeight)

roi=image[topLeft[1]:botRight[1],topLeft[0]:botRight[0]]
mask=np.zeros(image.shape,dtype='uint8')
image=cv2.addWeighted(image,0.25,mask,0.75,0)

image[topLeft[1]:botRight[1],topLeft[0]:botRight[0]] =roi

cv2.imwrite("MatchedTemplate.jpeg",image)
```

Output:

```
cv2.imwrite("MatchedTemplate.jpeg",image)
```

```
[5]: True
```



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

Theory:

Log and Power-law transformations :

The general form of the log transformation is s = c * log (1 + r) The log transformation maps [5] a narrow range of low input grey level values into a wider range of output values. The inverse log transformation performs the opposite transformation. Log functions are particularly useful when the input grey level values may have an extremely large range of values. In the following example the Fourier transform of an image is put through a log transform to reveal more detail.

The nth power and nth root curves shown in fig. A can be given by the expression, $s = cr^{\gamma}$. This transformation function is also called as gamma correction [$s = cr^{\gamma}$]. For various values of γ different levels of enhancements can be obtained. This technique is quite commonly called as Gamma Correction. If you notice, different display monitors display images at different intensities and clarity.

The difference between the log transformation function and the power-law functions is that using the power-law function a family of possible transformation curves can be obtained just by varying the λ. These are the three basic image enhancement functions for grey scale images that can be applied easily for any type of image for better contrast and highlighting.

Contrast Adjustment:

Contrast adjustment remaps image intensity values to the full display range of the data type. An image with good contrast has sharp differences between black and white. To illustrate, the image on the left has poor contrast, with intensity values limited to the middle portion of the range.

Histogram equalization

Histogram Equalization is a computer image processing technique used to improve contrast in images . It accomplishes this by effectively spreading out the most frequent intensity values, i.e. stretching out the intensity range of the image.

A color histogram of an image represents the number of pixels in each type of color component. Histogram equalization cannot be applied separately to the red, green and blue components of the image as it leads to dramatic changes in the image's color balance. However, if the image is first converted to another color space, like HSL/HSV color space, then the algorithm can be applied to the luminance or value channel without resulting in changes to the hue and saturation of the image.

The histogram of an image gives important information about the grayscale and contrast of the image. If the entire histogram of an image is cantered towards the left end of the x-axis, then it implies a dark image. If the histogram is more inclined towards the right end, it signifies a white or bright image. A narrow-width histogram plot at the center of the intensity axis shows a low-contrast image, as it has a few levels of grayscale. On the other hand, an evenly distributed histogram over the entire x-axis gives a high-contrast effect to the image.

Thresholding, and halftoning operations:

Thresholding is a type of image segmentation, where we change the pixels of an image to make the image easier to analyze. In thresholding, we convert an image from colour or grayscale into a binary image, i.e., one that is simply black and white. The downside of the simple thresholding technique is that we have to make an educated guess about the threshold t by inspecting the histogram. There are also *automatic thresholding* methods that can determine the threshold automatically for us. One such method is *Otsu's method*. It is particularly useful for situations where the grayscale histogram of an image has two peaks that correspond to background and objects of interest.

Halftoning or analog halftoning is a process that simulates shades of gray by varying the size of tiny black dots arranged in a regular pattern. This technique is used in printers, as well as the publishing industry. If you inspect a photograph in a newspaper, you will notice that the picture is composed of black dots even though it appears to be composed of grays. This is possible because of the spatial integration performed by our eyes.

Mathematical Equation:

1. Log and Power-law transformations

The log transformation is defined as:

$$s = c \cdot \log(1 + r)$$

Where:

- s is the output pixel value after transformation.
- r is the input pixel value.
- c is a constant that scales the result to the desired range. Typically, c is chosen to normalize the output to the range [0, 255] for display purposes.

The log transformation is often used to enhance the contrast of images with a wide dynamic range, such as astronomical images or medical images.

2. Contrast adjustments

The mathematical equation for contrast adjustment typically involves scaling the pixel intensity values of an image to expand or compress the range of intensities. Here's a basic form of the equation:

Given an input pixel intensity value I_{in}, the output pixel intensity value I_{out} after contrast adjustment can be calculated using the following formula:

$$I_{\text{out}} = \text{contrast} \times (I_{\text{in}} - \text{midpoint}) + \text{midpoint}$$

3. Histogram equalization

Let $h(i)$ be the histogram of the input image, where i represents the intensity level.

The cumulative distribution function (CDF) is computed as:

$$CDF(i) = \sum_{j=0}^{i} h(j)$$

Then, normalize the CDF:

$$CDF_{norm}(i) = \frac{CDF(i) - CDF_{min}}{MN - 1}$$

Where:

- CDF_{min} is the minimum value of the CDF.
- MN is the total number of pixels in the image.

The transformation function, $T(i)$, is given by:

$$T(i) = round(CDF_{norm}(i) \times L)$$

4. Thresholding, and halftoning operations

Let $I(x, y)$ be the intensity of the pixel at location (x, y) in the input grayscale image, and T be the threshold value.

The mathematical equation for thresholding operation is:

$$\text{Binary Output Image}(x, y) = \begin{cases} 1 & \text{if } I(x, y) > T \\ 0 & \text{otherwise} \end{cases}$$

Where:

- $\text{Binary Output Image}(x, y)$ is the pixel value of the binary output image at location (x, y).
- $I(x, y)$ is the intensity of the pixel at location (x, y) in the input grayscale image.
- T is the threshold value.
- The output pixel is set to 1 (foreground) if the intensity of the input pixel is greater than the threshold value, otherwise, it is set to 0 (background).

Let $I(x, y)$ be the intensity of the pixel at location (x, y) in the input grayscale image, and M be the halftone matrix.

The mathematical equation for halftoning operation is:

$$\text{Output Image}(x, y) = \begin{cases} 1 & \text{if } I(x, y) > M(x, y) \\ 0 & \text{otherwise} \end{cases}$$

Where:

- $\text{Output Image}(x, y)$ is the pixel value of the halftone output image at location (x, y).
- $I(x, y)$ is the intensity of the pixel at location (x, y) in the input grayscale image.
- $M(x, y)$ is the corresponding element of the halftone matrix.
- The output pixel is set to 1 if the intensity of the input pixel is greater than the corresponding value in the halftone matrix, otherwise, it is set to 0.

Halftoning algorithms vary in the way they construct the halftone matrix and how they distribute the dots or pixels to achieve the desired visual effect. Common halftoning techniques include dithering, error diffusion, and ordered dithering.

Code:

1. Log and Power-law transformations :

```
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image

#Load sample gray image
image_path = 'Samiksha.jpg'
image = Image.open(image_path).convert('L')
image_array = np.array(image)

#Display the original image
plt.figure(figsize=(12, 4))
plt.subplot(1, 3, 1)
plt.imshow(image_array, cmap='gray')
plt.title('Original Image')
plt.axis('off')

#Log transformation
log_transformed_image = np.log1p(image_array)

#Display the log transformed image
plt.subplot(1, 3, 2)
plt.imshow(log_transformed_image, cmap='gray')
plt.title('Log Transformation')
plt.axis('off')

#Power-law (Gamma) Transformation
gamma = 0.5
power_law_transformed_image = np.power(image_array, gamma)

#Display the power-law transformed image
plt.subplot(1, 3, 3)
plt.imshow(power_law_transformed_image, cmap='gray')
plt.title('Power-law Transformation')
plt.axis('off')

plt.show()
```

Output:

2. Contrast Adjustment:

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

#read the image
image_path='Samiksha.jpg'
image = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)


# Check if the image is successfully loaded
if image is None:
    print(f"Error: Unable to load image from {image_path}")
else:
    #Display the original image
    plt.figure(figsize=(12, 4))
    plt.subplot(1, 3, 1)
    plt.imshow(image, cmap='gray')
    plt.title('Original Image')
    plt.axis('off')

#Contrast adjustment using linear scaling
alpha = 1.5
beta = 20

contrast_adjusted_image = cv2.convertScaleAbs(image, alpha=alpha, beta=beta)

#Display the contrast adjusted image
plt.subplot(1,3,2)
plt.imshow(contrast_adjusted_image, cmap='gray')
plt.title('Contrast Adjusted Image')
plt.axis('off')

#Convert the image to 8-bit unsigned integer (required for CLAHE)
image_uint8 = cv2.convertScaleAbs(image)
```

```
#Adoptive histogram equalization for contrast enhancement
clahe = cv2.createCLAHE(clipLimit=2.0, tileGridSize=(8,8))
clahe_result = clahe.apply(image_uint8)

#Display the CLAHE result
plt.subplot(1,3,3)
plt.imshow(clahe_result, cmap='gray')
plt.title('Adaptive Histogram Equalization Result')
plt.axis('off')

plt.show()
```

Output:

3. Histogram equalization

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

# Load your grayscale image (replace 'image_path' with your actual image path)
image_path = 'Samiksha.jpg'
image = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)

#Display the original image
plt.figure(figsize=(12, 4))
plt.subplot(1,3,1)
plt.imshow(image, cmap='gray')
plt.title('Original Image')
plt.axis('off')

# Perform basic histogram equalization
equalized_image = cv2.equalizeHist(image)

#Display the euquilized image
plt.subplot(1,3,2)
plt.imshow(equalized_image, cmap='gray')
plt.title('Equalized Image')
plt.axis('off')
```

```
# Create CLAHE object for adaptive histogram equalization
clahe = cv2.createCLAHE(clipLimit=2.0, tileGridSize=(8, 8))
clahe_image = clahe.apply(image)

# Display the CLAHE images
plt.subplot(1,3,3)
plt.imshow(clahe_image, cmap='gray')
plt.title('CLAHE Image')
plt.axis('off')

plt.show()
```

Output:

4. Thresholding, and halftoning operations:

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

# Load the image
image = cv2.imread('Samiksha.jpg', cv2.IMREAD_GRAYSCALE)

# Thresholding
_, thresholded = cv2.threshold(image, 127, 255, cv2.THRESH_BINARY)

# Halftoning
kernel = cv2.getStructuringElement(2, (3, 7))
halftone = cv2.morphologyEx(image, cv2.MORPH_CLOSE, kernel)

# Display the original, thresholded, and halftoned images
plt.figure(figsize=(12, 4))
plt.subplot(1, 3, 1)
plt.imshow(image, cmap='gray')
plt.title('Original Image')
plt.axis('off')
```

```
plt.subplot(1, 3, 2)
plt.imshow(thresholded, cmap='gray')
plt.title('Thresholded Image')
plt.axis('off')

plt.subplot(1, 3, 3)
plt.imshow(halftone, cmap='gray')
plt.title('Halftoned Image')
plt.axis('off')
plt.show()
```

Output:

PRACTICAL-05

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

Theory:

Recall that the gradient of a two-dimensional function, f, is given by: Then, the Laplacian (that is, the divergence of the gradient) off can be defined by the sum of unmixed second partial derivatives: It can, equivalently, be considered as the trace (tr) of the function's Hessian, H(f).

Gradient descent is the method that iteratively searches for a minimizer by looking in the gradient direction. Conjugate gradient is similar, but the search directions are also required to be orthogonal to each other in the sense that pTiApj=0∀i,j.

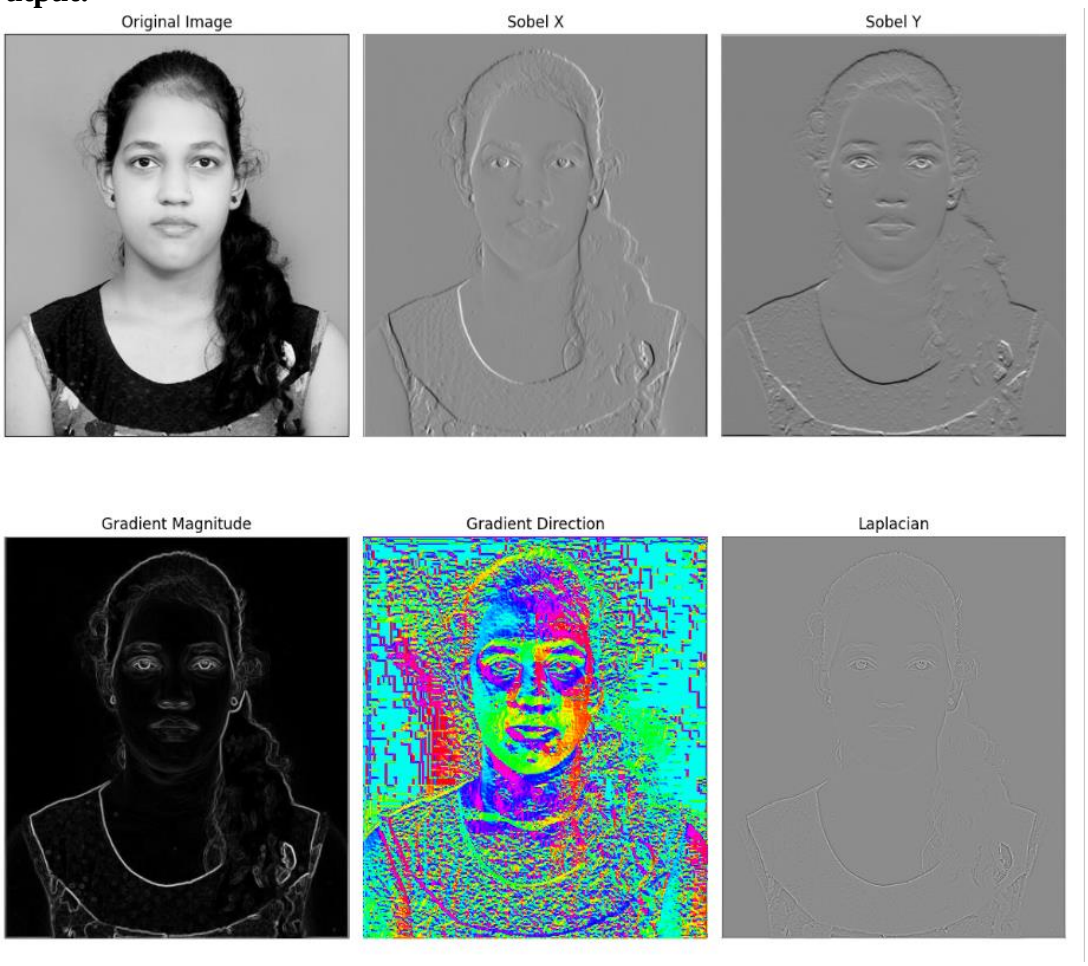
Code:

```
import cv2

import numpy as np

import matplotlib.pyplot as plt


#Load the image

image_path = 'Samiksha.jpg'

image = cv2.imread(image_path)


#Convert the image to grayscale

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


#Sobel gradient operations

sobel_x = cv2.Sobel(gray_image, cv2.CV_64F, 1, 0,ksize=3)

sobel_y = cv2.Sobel(gray_image, cv2.CV_64F, 0, 1,ksize=3)


#Compute gradient magnitude and direction

gradient_magnitude = np.sqrt(sobel_x**2 + sobel_y**2)

gradient_direction = np.arctan2(sobel_y, sobel_x)
```

```
#Laplacian operation
laplacian = cv2.Laplacian(gray_image, cv2.CV_64F)

#Display the result
plt.figure(figsize=(12,12))
plt.subplot(2,3,1)
plt.imshow(gray_image, cmap='gray')
plt.title('Original Image')
plt.axis('off')

plt.subplot(2,3,2)
plt.imshow(sobel_x, cmap='gray')
plt.title('Sobel X')
plt.axis('off')

plt.subplot(2,3,3)
plt.imshow(sobel_y, cmap='gray')
plt.title('Sobel Y')
plt.axis('off')

plt.subplot(2,3,4)
plt.imshow(gradient_magnitude, cmap='gray')
plt.title('Gradient Magnitude')
plt.axis('off')

plt.subplot(2,3,5)
plt.imshow(gradient_direction, cmap='hsv')
plt.title('Gradient Direction')
plt.axis('off')
```

```
plt.subplot(2,3,6)
plt.imshow(laplacian, cmap='gray')
plt.title('Laplacian')
plt.axis('off')

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

Output:



PRACTICAL-06

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

Theory:

Consider a dataset with p features(or independent variables) and one response(or dependent variable). Also, the dataset contains n rows/observations. We define: X (feature matrix) = a matrix of size n X p where x_{ij} denotes the values of jth feature for ith observation.

Linear means something related to a line. All the linear equations are used to construct a line. A non-linear equation is such which does not form a straight line. It looks like a curve in a graph and has a variable slope value.

It forms a straight line or represents the equation for the straight line. It does not form a straight line but forms a curve.

In smoothing, the data points of a signal are modified so individual points higher than the adjacent points (presumably because of noise) are reduced, and points that are lower than the adjacent points are increased leading to a smoother signal. Smoothing is a signal processing technique typically used to remove noise from signals.

Code:

```
import cv2

import numpy as np

import matplotlib.pyplot as plt


#Load the image

image_path = 'Samiksha.jpg'

image = cv2.imread(image_path)


#Convert the image to grayscale

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


#Display the original image

plt.figure(figsize=(15, 12))

plt.subplot(1, 3, 1)

plt.imshow(cv2.cvtColor(image,cv2.COLOR_BGR2RGB))

plt.title('Original Image')
```

```
plt.axis('off')

#Linear Noise Smoothing
linear_smoothed_image = cv2.GaussianBlur(gray_image, (5, 5), 0)
plt.subplot(1, 3, 2)
plt.imshow(linear_smoothed_image, cmap='gray')
plt.title('Linear Noise Smoothing (Gaussian Blur)')
plt.axis('off')

#Nonlinear Noise Smoothing(Median Filter)
nonlinear_smoothed_image = cv2.medianBlur(gray_image, 5)
plt.subplot(1, 3, 3)
plt.imshow(nonlinear_smoothed_image, cmap='gray')
plt.title( 'Nonlinear Noise Smoothing (Median Filter)')
plt.axis('off')

plt.show()
```

Output:

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.

Theory:

- Output is simply the average of the pixels contained in the neighbourhood of the filter mask. Also called averaging filters or low pass filters
- Replacing the value of every pixel in an image by the average of the gray levels in the neighbourhood will reduce the "sharp " transitions in gray levels.
- Sharp transitions
  - random noise in the image
  - edges of objects in the image
  - Smoothing can reduce noises (desirable) and blur edges (undesirable)

$\frac{1}{9} \times$

1	1	1
1	1	1
1	1	1

box filter

$\frac{1}{16} \times$

1	2	1
2	4	2
1	2	1

weighted average

Sharpen

Given the fact that we can blur an image, it seems that one should be able to sharpen an image as well. To a certain extent this is possible, although the most commonly used method is really somewhat of a trick. The Sharpen operator actually works by increasing the contrast between areas of transition in an image. This, in turn, is perceived by the eye as an increased sharpness. Sharpening can either be done by using a convolve with a filter such as

−1	−1	−1
−1	9	−1
−1	−1	−1

or via other techniques, such as unsharp masking.

Unsharp masking, an old technique known to photographers, is used to change the relative high pass content in an image by subtracting a blurred (lowpass filtered) version of the image [5]. This can be done optically by first developing an unsharp picture on a negative film and then using this film as a mask in a second development step.

Mathematically, unsharp masking can be expressed as

F = af- Bfip

where a and ß are positive constants, a ≥ B. When processing digital image data, it is desirable to keep the local mean of the image unchanged. If the coefficients in the lowpass filter fip are normalized, i.e., their sum equals 1, the following formulation of unsharp masking ensures unchanged local mean in the image:

By expanding the expression in the parentheses (af - Bfip) = afip + alf

- fip) - Bfip, we can write Eq. (9) as

f = fip + a-s (f - fip),

which provides a more intuitive way of writing unsharp masking.

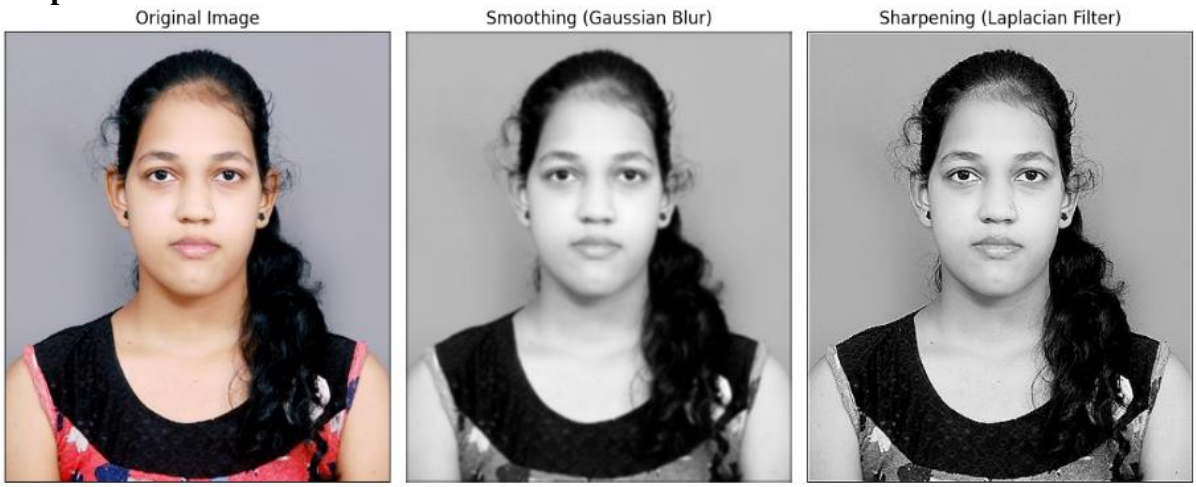
Further rewriting yields

Code:

```
import cv2

import numpy as np

import matplotlib.pyplot as plt


#Load the image

image_path = 'Samiksha.jpg'

image = cv2.imread(image_path)


#Convert the image to grayscale

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


#Display the original image

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

plt.subplot(2, 3, 1)

plt.imshow(cv2.cvtColor(image,cv2.COLOR_BGR2RGB))

plt.title('Original Image')
```

```
plt.axis('off')

#Smoothing (Gaussian Blur)
smoothed_image = cv2.GaussianBlur(gray_image, (5, 5), 0)
plt.subplot(2, 3, 2)
plt.imshow(smoothed_image, cmap='gray')
plt.title('Smoothing (Gaussian Blur)')
plt.axis('off')
plt.axis('off')

#Sharpening using Laplacian filter
laplacian = cv2.Laplacian(gray_image, cv2.CV_64F)
sharpened_image = np.uint8(np.clip(gray_image - 0.5*laplacian, 0, 255))
plt.subplot(2, 3, 3)
plt.imshow(sharpened_image, cmap='gray')
plt.title('Sharpening (Laplacian Filter)')
plt.axis('off')

#Unsharp masking
blurred_image = cv2.GaussianBlur(gray_image, (5, 5), 10)
unsharp_mask = cv2.addWeighted(gray_image, 2, blurred_image, -1, 0)
plt.subplot(2, 3, 4)
plt.imshow(unsharp_mask, cmap='gray')
plt.title('Unsharp Masking')
plt.axis('off')

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

Output:


Original Image
Smoothing (Gaussian Blur)
Sharpening (Laplacian Filter)


Unsharp Masking

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.

Theory:

Sobel Edge Detection: The Sobel edge detection is a popular method for detecting edges in images. It operates by convolving the image with a pair of 3x3 kernels, one for horizontal changes and the other for vertical changes. These kernels emphasize the rate of intensity change in the x and y directions. The magnitude of the gradient is computed using these partial derivatives, highlighting regions with significant intensity transitions. Sobel filtering is effective in detecting edges but may be sensitive to noise. It is commonly used as a preprocessing step in computer vision applic

Canny Edge Detection: The Canny edge detection algorithm is a multi-stage process designed to detect a wide range of edges in an image while minimizing false positives. The key stages include smoothing the image with a Gaussian filter to reduce noise, computing the gradient magnitude and direction, applying non- maximum suppression to thin the detected edges, and finally, using hysteresis thresholding to link adjacent edge pixels. Canny edge detection is known for its ability to provide high-precision edge maps with reduced susceptibility to noise. It is widely used in image processing and computer vision tasks, especially when robust edges detection is required.

Mathematical Equation: Certainly, let's provide the mathematical equations for Sobel and Canny edge detection: Sobel Edge Detection: The Sobel operators for detecting changes in intensity in the x-direction (Gx) and y-direction (Gy) are given by the convolution with the following kernels:

$$Gx = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

$$Gy = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

The gradient magnitude (G) is calculated as:

$$G = \sqrt{(Gx^2 + Gy^2)}$$

And the gradient direction (θ) is given by:

θ =arctan(Gy/Gx)

Canny Edge Detection:

Smoothing with Gaussian Filter:

Ismoothed (x,y)=I* Gkernel

Gradient Gx =Ismoothed * [−1 −2 −1
0 0 0
1 2 1]

Gy=Ismoothed * [−1 0 1
−2 0 2
−1 0 1]

G= √(Gx 2 +Gy 2)

θ =arctan(Gy/Gx)

Non-Maximum Suppression: For each pixel, compare the gradient magnitude with its neighbors along the gradient direction and keep the maximum value.

Hysteresis Thresholding: Set two thresholds, high and low. Pixels with gradient magnitude above the high threshold are considered strong edges. Pixels between the high and low thresholds are considered weak edges if they are connected to strong edges; otherwise, they are

Code:

```
# Import necessary libraries
import cv2
import numpy as np
import matplotlib.pyplot as plt

# Load the image
image_path = "Samiksha.jpg"
image = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)

# Apply Sobel edge detection
sobel_x = cv2.Sobel(image, cv2.CV_64F, 1, 0, ksize=5)
sobel_y = cv2.Sobel(image, cv2.CV_64F, 0, 1, ksize=5)
sobel_combined = np.sqrt(sobel_x**2 + sobel_y**2)

# Apply Canny edge detection
```

```
canny_edges = cv2.Canny(image, 100, 200)

# Plot the original image and edge-detected 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(sobel_combined, cmap= 'gray')
plt.title('Sobel Edge Detection')
plt.axis('off')

plt.subplot(2, 2, 3)
plt.imshow(canny_edges, cmap='gray')
plt.title('Canny Edge Detection')
plt.axis('off')
plt.show()
```

Output:

PRACTICAL-09

AIM: Write the program to implement various morphological image processing techniques.

Theory:

Erosion is a morphological operation in image processing that involves shrinking or wearing away the boundaries of objects in a binary image. It is typically achieved by convolving the image with a structuring element, and a pixel in the resulting image is set to 1 if all the pixels under the structuring element in the original image are also 1. Erosion is useful for removing small, isolated regions and for separating objects that are close to each other.

Dilation is another morphological operation that expands or thickens the boundaries of objects in a binary image. Like erosion, it involves convolving the image with a structuring element, and a pixel in the resulting image is set to 1 if at least one pixel under the structuring element in the original image is one. Dilation is effective in joining broken parts of objects and enlarging regions, making it useful in tasks like noise reduction and feature enhancement.

Opening is a morphological operation that combines erosion followed by dilation. It is particularly effective in removing small objects preprocess images before further analysis, as it helps eliminate noise and fine details while preserving the overall structure of the larger objects.

Closing is the reverse of opening and involves dilation followed by erosion. This operation is useful in closing small gaps and joining nearby object boundaries. Closing is commonly applied to images where objects are partially separated or have small interruptions. It helps in completing the contours of objects and making them more solid. Like opening, closing is an essential tool in morphological image processing for tasks such as object recognition and segmentation

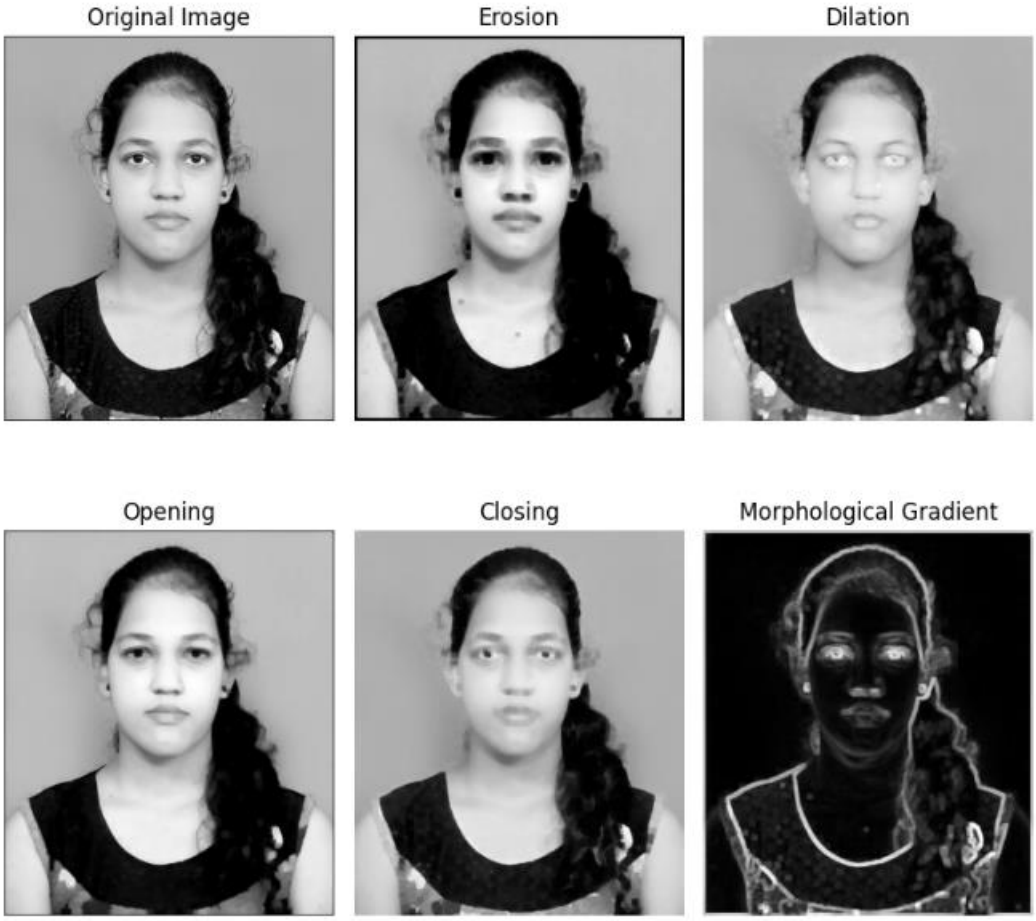
Code:

```
import cv2

import numpy as np

import matplotlib.pyplot as plt
```

```
# Load the image
image_path = 'Samiksha.jpg'
image = cv2. imread(image_path, cv2.IMREAD_GRAYSCALE)

# Display the original image
plt.figure(figsize=(8, 8))
plt.subplot(2, 3, 1)
plt.imshow(image, cmap='gray')
plt.title('Original Image')
plt.axis('off')

#Erosion
kernel = np.ones( (5, 5), np.uint8)
erosion = cv2.erode(image, kernel, iterations=1)
plt.subplot(2, 3, 2)
plt.imshow(erosion, cmap='gray')
plt.title('Erosion')
plt.axis('off')

# Dilation
dilation = cv2.dilate(image, kernel, iterations=1)
plt.subplot(2, 3, 3)
plt.imshow(dilation, cmap='gray')
plt.title('Dilation')
plt.axis('off')

# Opening (Erosion followed by Dilation)
opening = cv2.morphologyEx(image, cv2.MORPH_OPEN, kernel)
plt.subplot(2, 3, 4)
```

```
plt.imshow(opening, cmap='gray')
plt.title('Opening')
plt.axis('off')

# Closing (Dilation followed by Erosion)
closing = cv2.morphologyEx(image, cv2.MORPH_CLOSE, kernel)
plt.subplot(2, 3, 5)
plt.imshow(closing, cmap='gray')
plt.title('Closing')
plt.axis('off')

# Morphological Gradient (Difference between Dilation and Erosion)
gradient = cv2.morphologyEx(image, cv2.MORPH_GRADIENT, kernel)
plt.subplot(2, 3, 6)
plt.imshow(gradient, cmap='gray')
plt.title('Morphological Gradient')
plt.axis('off')
plt.tight_layout()
plt.show()
```

Output:


Original Image
Erosion
Dilation
Opening
Closing
Morphological Gradient

PRACTICAL-10

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

Theory:

1) **Corner detectors:**
   A corner detector in image processing is a method used to identify and locate corners or junctions in an image where there are significant changes in intensity or color. These corners typically represent points where the brightness or color changes in multiple directions. Corner detection algorithms are designed to highlight these key points, which are useful for tasks like image stitching, object recognition, and navigation. corner detectors help to pinpoint locations in an image where there are sharp changes or intersections in features, kind of like the corners of a room. These corners are distinctive points that can be used as landmarks for various computer vision tasks.
2) **Blob detectors:**
   Blob detection in image processing refers to the process of identifying regions or areas in an image that have similar properties or characteristics, such as color, intensity, or texture. These regions are typically referred to as "blobs". A blob detector looks for regions in an image that stand out in some way, such as being brighter or darker than their surroundings, having a particular color, or having a distinct texture. These regions can represent objects, features, or anomalies in the image.
3) **HoG Features:**
   HoG (Histogram of Oriented Gradients) is a technique used in computer vision for object detection. It works by dividing an image into small regions and computing histograms of gradient orientations within each region. These histograms capture the distribution of edge directions in different parts of the image. HoG features are then used to represent the shape and texture of objects in the image, making them useful for tasks like pedestrian detection, face detection, and general object recognition.
4) **Haar Features:**

   Haar features are basic rectangular patterns used in image processing for object detection. These features work by calculating the difference in pixel intensities between adjacent rectangular regions of an image. Haar features are particularly known for their use in the Viola-Jones face detection framework, where they help efficiently detect faces by identifying characteristic patterns such as edges, corners, and texture variations. They are computationally efficient and provide a simple yet effective way to capture important information about the structure of objects in images, making them widely used in various applications beyond face detection.

Mathematical Equation:

1. Corner detectors:

Given an input grayscale image I, the Harris corner detector computes a score R for each pixel location (x, y) based on the local intensity gradients. The score R is calculated using the following equation:

$$R = \det(M) - k \cdot \text{trace}^2(M)$$

Where:

- $\det(M)$ is the determinant of the auto-correlation matrix M,
- $\text{trace}(M)$ is the trace (sum of diagonal elements) of M,
- k is a constant usually between 0.04 to 0.06 (empirically determined),
- M is the auto-correlation matrix, defined as:

$$M = \sum_{x',y'} w(x', y') \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix}$$

Where:

- I_x and I_y are the derivatives of the image I with respect to x and y respectively,
- $w(x', y')$ is a window function that specifies the neighborhood around the pixel (x', y') over which the gradients are computed. It's often a Gaussian window or a simple square window.

2. Blob detectors:

$$LoG(x, y) = \nabla^2(G * I) = \frac{\partial^2}{\partial x^2}(G * I) + \frac{\partial^2}{\partial y^2}(G * I)$$

Where:

- $G * I$ represents the image I convolved with a Gaussian kernel G.
- ∇^2 denotes the Laplacian operator.

3. HoG Features:

The first step involves calculating the gradient magnitude (M) and orientation (θ) of each pixel in the image. This is typically done using gradient filters like Sobel or Scharr operators. Let $I(x, y)$ denote the intensity of the pixel at coordinates (x, y). The gradient magnitude (M) and orientation (θ) are given by:

$$M(x, y) = \sqrt{(G_x(x, y))^2 + (G_y(x, y))^2}$$
$$\theta(x, y) = \arctan\left(\frac{G_y(x, y)}{G_x(x, y)}\right)$$

Where $G_x(x, y)$ and $G_y(x, y)$ are the gradients in the x and y directions respectively.

4. Haar Features:

Let's denote $I(x, y)$ as the pixel intensity at position (x, y) in the image.

A Haar feature is defined by a rectangular region within the image, typically described by two points: the top-left corner (x_1, y_1) and the bottom-right corner (x_2, y_2).

For a single rectangular Haar-like feature, the mathematical expression to compute the feature value is given by:

$$\text{Haar\_feature} = \sum_{(x,y)\in\text{Rectangle\_1}} I(x, y) - \sum_{(x,y)\in\text{Rectangle\_2}} I(x, y)$$

Where:

- Rectangle_1 represents the first rectangular region defined by (x_1, y_1) and (x_2, y_2).
- Rectangle_2 represents the second rectangular region adjacent to the first one.
- $\sum$ denotes the sum of pixel intensities over all pixels within each rectangular region.

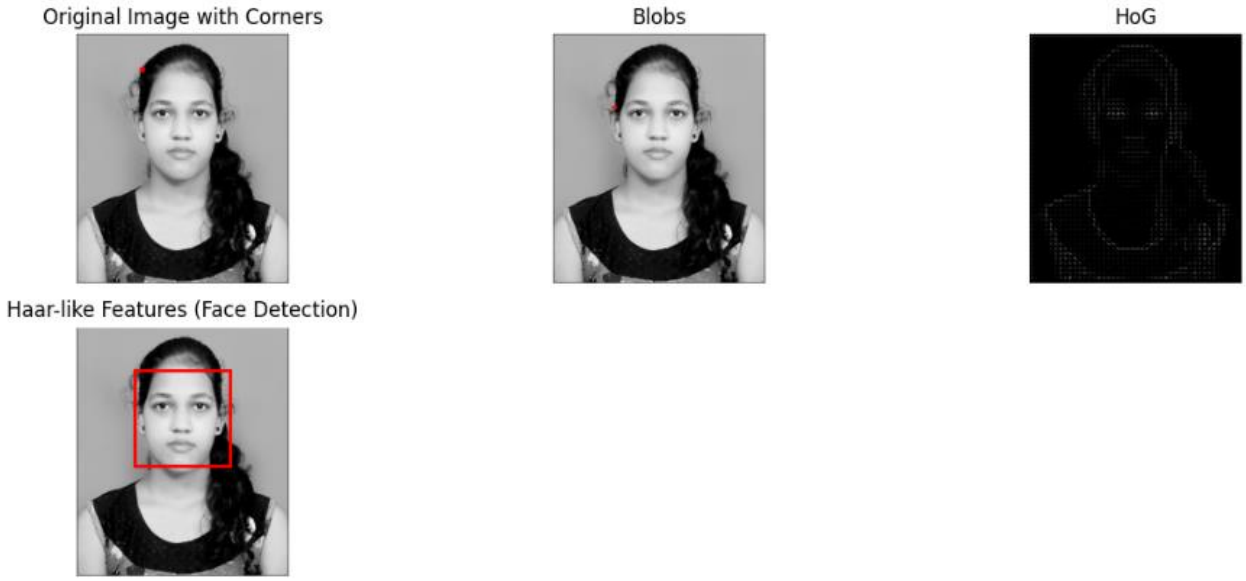
Code:

```
import cv2
import numpy as np
from skimage import feature
from skimage import exposure
import matplotlib.pyplot as plt

# Load an image
image_path = 'Samiksha.jpg'
image = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)

# Corner Detection (using Shi-Tomasi corner detector)
corners = cv2.goodFeaturesToTrack(image, maxCorners=100, qualityLevel=0.01, minDistance=10)
corners = np.intp(corners)

# Blob Detection (using Difference of Gaussians)
blobs = feature.blob_dog(image, max_sigma=30, threshold=0.1)
```

```
# HoG (Histogram of Oriented Gradients)
hog_features, hog_image = feature.hog(image, visualize=True)

# Haar-like Features (using OpenCV's Haar Cascade Classifier)
# Note: You need a pre-trained Haar Cascade XML file for face detection
face_cascade = cv2.CascadeClassifier(cv2.data.haarcascades +
'haarcascade_frontalface_default.xml')
faces = face_cascade.detectMultiScale(image, scaleFactor=1.1, minNeighbors=5)

# Displaying Results
plt.figure(figsize=(12, 5))

# Original Image with Corners
plt.subplot(2, 3, 1)
plt.imshow(image, cmap='gray')
plt.axis('off')
plt.title('Original Image with Corners')
for corner in corners:
    x,y = corner.ravel()
plt.scatter(x, y, c='red', s=5)

# Blobs
plt.subplot(2, 3, 2)
plt.imshow(image, cmap='gray')
plt.title('Blobs')
plt.axis('off')
for blob in blobs:
    y, x, r = blob
c = plt.Circle((x, y), r, color='red', linewidth=2, fill=False)
plt.gca().add_patch(c)
```

```
# HoG
plt.subplot(2, 3, 3)
plt.imshow(hog_image, cmap='gray')
plt.title('HoG')
plt.axis('off')

# Haar-like Features
plt.subplot(2, 3, 4)
plt.imshow(image, cmap='gray')
plt.axis('off')
plt.title('Haar-like Features (Face Detection)')
for (x, y, w, h) in faces:
    plt.Rectangle((x, y), w, h, color='red', fill=False, linewidth=2)
    plt.gca().add_patch(plt.Rectangle((x, y), w, h, color='red', fill=False, linewidth=2))
    plt.tight_layout()
plt.show()
```

Output:



PRACTICAL NO-11

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

Theory:

- Segmentation in image processing is the process of partitioning an image into multiple segments or regions to simplify its representation and make it easier to analyze. In the context of detecting lines, circles, and other shapes/objects, segmentation aims to identify distinct regions in the image that correspond to these shapes.
- Line Detection: In line detection, segmentation involves separating regions of the image that represent lines from the background or other objects. This can be done using techniques like edge detection to find abrupt changes in intensity, followed by line fitting algorithms to group edge pixels into straight lines.
- Circle Detection: Similarly, for circle detection, segmentation involves isolating regions in the image that form circular shapes. Techniques such as the Hough Transform can be used, where the accumulator space is segmented to identify peaks corresponding to circles of specific radii.
- Other Shapes/Objects: Segmentation can also be used to detect various other shapes or objects in an image. This can involve thresholding, region growing, or contour detection to identify areas with similar properties (e.g., color, texture) and group them into meaningful segments representing different objects or shapes.

1. Edge-based segmentation:
   - What it does: Edge-based segmentation identifies object boundaries by detecting sudden changes (edges) in intensity or color within an image.
   - How it works: It looks for sharp transitions in pixel intensity or color, indicating where one object ends and another begins.
   - Example: Imagine looking at a black circle on a white background. Edge-based segmentation would detect the sharp transition between black and white pixels around the circle's perimeter, outlining its shape.
2. Region-based segmentation:
   - What it does: Region-based segmentation groups pixels together based on similar characteristics (like color or intensity) to identify distinct objects or regions within an image.
   - How it works: It examines the overall similarity of neighboring pixels, grouping together those that are alike and forming distinct regions.
   - Example: In an image of a red apple on a green table, region-based segmentation would group together all pixels with similar red values to identify the apple, and all pixels with similar green values to identify the table.

Mathematical Equation:

1. **Line Detection**:
   - One of the most common methods for line detection is the Hough Transform.
   - The standard form of the Hough Transform for detecting lines can be expressed as:
     $r = x \cdot \cos(\theta) + y \cdot \sin(\theta)$
     where (r, θ) are the polar coordinates of a line passing through a point (x, y).
2. **Circle Detection**:
   - Circle detection can also be done using the Hough Transform, but a specific variant known as the Circular Hough Transform is often used.
   - The equation for detecting circles using the Circular Hough Transform is more complex, involving three parameters: the circle's center (a, b) and its radius r.
     $(x - a)^2 + (y - b)^2 = r^2$
3. **Other Shapes/Objects**:
   - Detection of other shapes or objects depends on the specific characteristics of those shapes and the algorithms used.
   - For example, rectangles or squares can be detected by analyzing the edges and corners of an image using techniques like the Hough Transform for lines and corner detection methods.

- ## Edge-based segmentation

  - **Mathematical Representation**: One common approach is to compute the gradient of the image, which represents the rate of change of intensity. This can be achieved using operators like the Sobel operator or the Canny edge detector.
    Mathematically, this can be represented as:
    $\nabla f = \sqrt{(G_x)^2 + (G_y)^2}$
    Where:
    - ∇f is the gradient magnitude,
    - G_x is the gradient in the x-direction,
    - G_y is the gradient in the y-direction.

- ## Region-based segmentation

  - **Mathematical Representation**: One common approach is to use clustering algorithms such as K-means or Gaussian Mixture Models (GMM) to group similar pixels together. Mathematically, this can be represented as:
    $R_i = \{p | d(p, c_i) < d(p, c_j) \ \forall j \neq i\}$
    Where:
    - R_i represents region i,
    - p is a pixel in the image,
    - c_i is the centroid of region i,
    - $d(p, c_i)$ is a distance measure between pixel p and centroid c_i,
    - The condition $d(p, c_i) < d(p, c_j) \ \forall j \neq i$ ensures that pixel p is closer to the centroid of region i than to the centroids of other regions.

Code:

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

# Load an image
image_path = 'Samiksha.jpg'
original_image = cv2.imread(image_path)
gray_image = cv2.cvtColor(original_image, cv2.COLOR_BGR2GRAY)

# Edge-based Segmentation (Canny edge detector)
edges = cv2.Canny(gray_image, 50, 150)

# Region-based Segmentation (Simple Thresholding)
_, binary_image = cv2.threshold(gray_image, 127, 255, cv2.THRESH_BINARY)

# Displaying Results
plt.figure(figsize=(10, 5))

# Original Image
plt.subplot(2, 3, 1)
plt.imshow(cv2.cvtColor(original_image, cv2.COLOR_BGR2RGB))
plt.title('Original Image')
plt.axis('off')

# Gray Image
plt.subplot(2, 3, 2)
plt.imshow(gray_image, cmap='gray')
plt.title('Gray Image')
plt.axis('off')
```

```
# Edge-based Segmentation (Canny)
plt.subplot(2, 3, 3)
plt.imshow(edges, cmap='gray')
plt.title('Edge-based Segmentation (Canny)')
plt.axis('off')

# Region-based Segmentation (Thresholding)
plt.subplot(2, 3, 4)
plt.imshow(binary_image, cmap='gray')
plt.title('Region-based Segmentation (Thresholding)')
plt.axis('off')

# Additional Processing (e.g., Hough Line Transform for Lines, Hough Circle Transform for Circles)
# Lines
lines = cv2.HoughLines(edges, 1, np.pi / 180, 100)
for line in lines:
    rho, theta = line[0]
a = np.cos(theta)
b = np.sin(theta)
x0 = a * rho
y0 = b * rho
x1 = int(x0 + 1000 * (-b))
y1 = int(y0 + 1000 * (a))
x2 = int(x0 - 1000 * (-b))
y2 = int(y0 - 1000 * (a))
cv2.line(original_image, (x1, y1), (x2, y2), (0, 0, 255), 2)

# Circles
```

```
circles = cv2.HoughCircles(edges, cv2.HOUGH_GRADIENT, dp=1, minDist=20,
param1=50, param2=30, minRadius=10, maxRadius=50)
if circles is not None:
    circles = np.uint16(np.around(circles))
for circle in circles[0, :]:
    center = (circle[0], circle[1])
    radius = circle[2]
cv2.circle(original_image, center, radius, (0, 255, 0), 2)


# Display Results with Detected Lines and Circles
plt.subplot(2, 3, 5)
plt.imshow(cv2.cvtColor(original_image, cv2.COLOR_BGR2RGB))
plt.title('Detected Lines and Circles')
plt.axis('off')
plt.tight_layout()
plt.show()
```

Output: