MADRAS INSTITUTE OF TECHNOLOGY
ANNA UNIVERSITY

DEPARTMENT OF INFORMATION TECHNOLOGY
AD23402-COMPUTER VISION

RECORD

REGISTER NUMBER: 2023510041

NAME: M.NATRAJAN

SEMESTER : 4

DEPARTMENT OF INFORMATION TECHNOLOGY

ANNA UNIVERSITY , MIT CAMPUS

CHROMEPET, CHENNAI – 600 044

BONAFIDE CERTIFICATE

Certified that the Bonafide record of the practical work done by **M.NATRAJAN**, Register Number **(2023510041)** Of **Four** Semester **B.Tech Artificial Intelligence and Data Science** in the **– Computer Vision Laboratory.**

Date:1/1/2025 Course Instructor: Ms M Hemalatha

TABLE OF CONTENT:

Exercise No:1 Date:9/01/2025	BASICS OF IMAGE PROCESSING

AIM :

To explore the some basics concepts of image processing in lena and sample image.

FUNCTIONS :

1. cv2.imread():

- Reads an image from a specified file.
- Example: img = cv2.imread('path_to_image').

2. cv2.cvtColor():

- Converts an image from one color space to another (e.g., BGR to RGB or RGB to grayscale).
- Example: RGB_img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB).

3. cv2.equalizeHist():

- Performs histogram equalization to improve the contrast of the image.
- Example: equalized_image = cv2.equalizeHist(gray_image).

4. sns.histplot():

- Calculates the histogram of an image, which shows the distribution of pixel intensities.
- sns.histplot(gray_image.ravel(), bins=30, kde=True, color='skyblue', alpha=0.6)

5. plt.subplot():

- Creates a subplot within a grid for multiple images or plots.
- Example: plt.subplot(2, 2, 1) for a 2x2 grid, 1st plot.

6. plt.imshow():

- Displays an image in a plot.
- Example: plt.imshow(gray_image, cmap='gray') for showing a grayscale image.

7. plt.plot():

- Plots a graph (typically used for histograms or data).
- Example: plt.plot(hist_orig, color='black').

8. plt.tight_layout():

- Adjusts subplots to prevent overlapping of elements (like labels).
- Example: plt.tight_layout().

9. cv2.resize():

- Resizes an image to the specified dimensions.
- Example: resized_img = cv2.resize(img, (256, 256)).

10. plt.show():

- Displays the figure created by matplotlib with all the plots and images.
- Example: plt.show().

ALGORITHM :

USING CUSTOM FUNCTION :

1.**Load the Image**:

- Read the image from a specified file using cv2.imread().

2.**Convert Image Color**:

- Convert the image from BGR (default format in OpenCV) to RGB format using cv2.cvtColor().

3.**Convert to Grayscale**:

- Convert the RGB image to grayscale using cv2.cvtColor().

4.**Histogram Equalization**:

- Apply histogram equalization on the grayscale image using cv2.equalizeHist() to enhance contrast.

5.**Calculate Histograms**:

- Compute the histogram of the original grayscale image using sns.histplot().
- Compute the histogram of the equalized grayscale image using sns.histplot().

6.**Plot Histograms**:

- Create subplots to display:
  - Histogram of the original grayscale image.
  - Histogram of the equalized grayscale image.

7.**Display Images**:

- Display the grayscale image and the equalized grayscale image using plt.imshow().

8.**Resize Image**:

- Resize the image to a specific size (e.g., 256x256 pixels) using cv2.resize().

9.**Convert Resized Image**:

- Convert the resized image from BGR to RGB using cv2.cvtColor() for proper visualization.

10.**Display Resized Image**:

- Display the resized image using plt.imshow().

11. **Final Display**:

- Adjust layout and show all subplots using plt.tight_layout() and plt.show().

WITHOUT USING CUSTOM FUNCTION :

1.**Load Image in RGB Format**:

- Load the image using OpenCV (cv2.imread()).
- Convert the image from BGR to RGB using cv2.cvtColor().

2. **Extract RGB Channels**:

- Extract the Red (R), Green (G), and Blue (B) channels from the RGB image.

3. **Apply Grayscale Conversion Formula**:

- Convert the image to grayscale using the standard formula:
- **Gray Value = 0.2989 * R + 0.5870 * G + 0.1140 * B**.

4. **Display the Grayscale Image**:

- Use plt.imshow() with the cmap='gray' option to display the resulting grayscale image.

1. **Load the Grayscale Image**:

- Read the image in grayscale format (using cv2.imread() with cv2.IMREAD_GRAYSCALE).

2. **Calculate Histogram of Original Image**:
    - Compute the histogram of the grayscale image using cv2.calcHist(), which gives the frequency distribution of pixel intensities.
3. **Compute CDF (Cumulative Distribution Function)**:
    - Compute the cumulative sum of the histogram to obtain the CDF.
    - Normalize the CDF by scaling it to the maximum value of the original histogram.
4. **Normalize the CDF to the Full Pixel Range (0-255)**:
    - Mask the zero values in the CDF to prevent errors in calculations.
    - Stretch the CDF to span the full pixel intensity range (0-255) by adjusting its minimum and maximum values.
5. **Apply the CDF to the Original Image**:
    - Use the normalized CDF to map the pixel values of the original image to new values, thus performing histogram equalization.
6. **Calculate Histogram of Equalized Image**:
    - Compute the histogram of the equalized image using sns.histplot().
7. **Plot Histograms**:
    - Plot the histogram of the original image (before equalization).
    - Plot the histogram of the equalized image (after equalization).
8. **Display Original and Equalized Images**:
    - Display the original grayscale image.
    - Display the equalized grayscale image.
9. **Show Results**:
    - Use plt.tight_layout() to adjust the layout and avoid overlap.
    - Display all the images and histograms using plt.show().

LENNA IMAGE:

```
#image processing

import cv2 as c
```

```
import matplotlib.pyplot as plt
import numpy as np
img=c.imread("lena.bmp")
plt.figure(figsize=(3,3))
plt.imshow(img)
plt.title('Lenna Image')
plt.axis('off')
plt.show()
```


```
#library
img_color=c.cvtColor(img,c.COLOR_BGR2RGB)
plt.figure(figsize=(5,5))
plt.subplot(1,2,1)
plt.imshow(img_color)
plt.title("RGB Image")
plt.axis('off')
#manual
img_man=img[:,:,[2,1,0]]
plt.subplot(1,2,2)
plt.imshow(img_man)
plt.title("RGB Manual")
plt.axis("off")
```

plt.show()


```
#grayscale-inbuilt
image_gray = cv2.cvtColor(image, cv2.COLOR_RGB2GRAY)
plt.imshow(image_gray , cmap='gray')
plt.title('Original grayscale image')
plt.axis('off')
plt.show()
#constant
R = img_color[..., 0]
G = img_color[..., 1]
B = img_color[..., 2]
gray_manual = (0.299 * R + 0.587 * G + 0.114 * B).astype(np.uint8)
plt.imshow(gray_manual, cmap='gray')
plt.title("Grayscale Image(industrial formula)")
plt.axis('off')
plt.show()
```

height, width, channels = img.shape

print(f"Height: {height}, Width: {width}, Channels: {channels}")

output:

```
Height: 512, Width: 512, Channels: 3
```

#resize

new_size = (400, 300)

img_resized = c.resize(img_man, new_size)

plt.subplot(1,2,1)

plt.imshow(img_resized)

plt.title("Resize Img")

plt.axis('off')

#resize manual

img_resized_manual = c.resize(img_color,(int(img_color.shape[1]/5),int(img_color.shape[0]/4)))

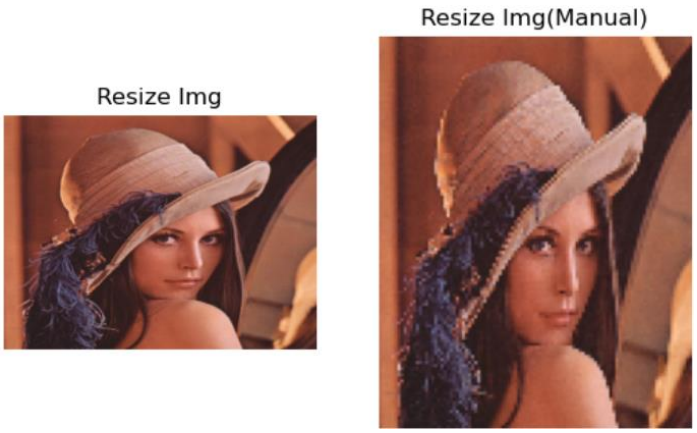
plt.subplot(1,2,2)

plt.imshow(img_resized_manual)

plt.title("Resize Img(Manual)")

plt.axis('off')

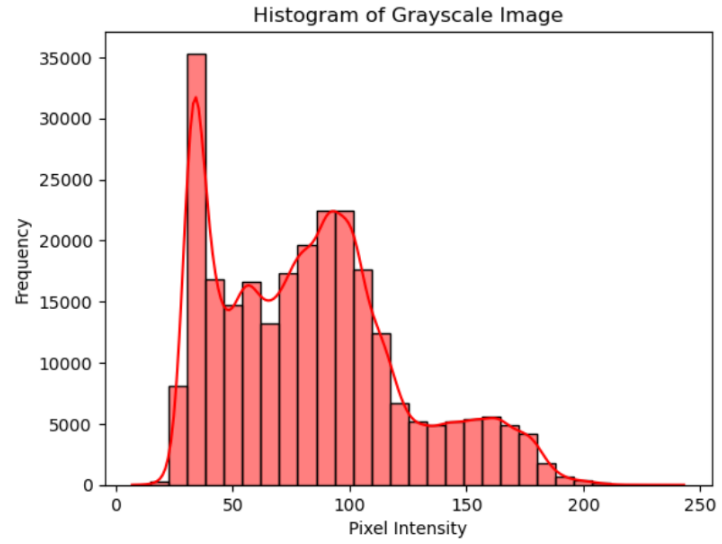
plt.show()

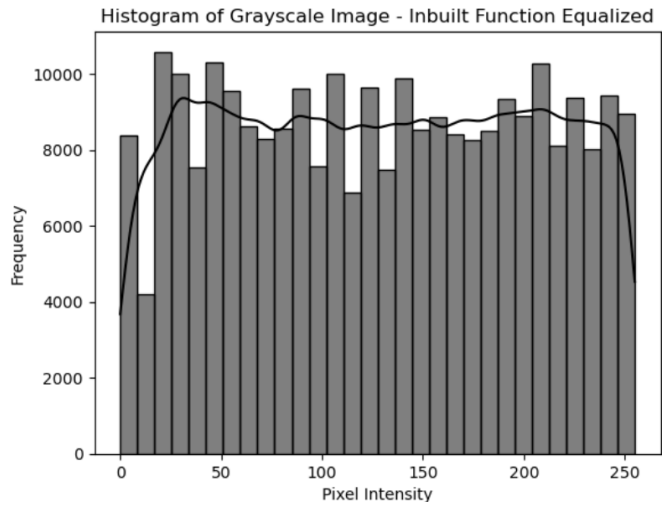



```
#equalize
img_gy=c.cvtColor(img,c.COLOR_RGB2GRAY)
img_eq = c.equalizeHist(img_gy)
plt.figure(figsize=(6,6))
plt.subplot(1, 2, 1)
plt.imshow(img_gy, cmap='gray')
plt.title("Original Grayscale Image")
plt.axis('off')
plt.subplot(1, 2, 2)
plt.imshow(img_eq, cmap='gray')
plt.title("Histogram Equalized Image")
plt.axis('off')
plt.show()
```


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

```
import numpy as np
import seaborn as sns
sns.histplot(img_gy.ravel(), kde=True, bins=30, color='red')
plt.title("Histogram of Grayscale Image")
plt.xlabel('Pixel Intensity')
plt.ylabel('Frequency')
plt.show()
sns.histplot(img_eq.ravel(), kde=True, bins=30, color='black')
plt.title("Histogram of Grayscale Image - Inbuilt Function Equalized")
plt.xlabel('Pixel Intensity')
plt.ylabel('Frequency')
plt.show()
```

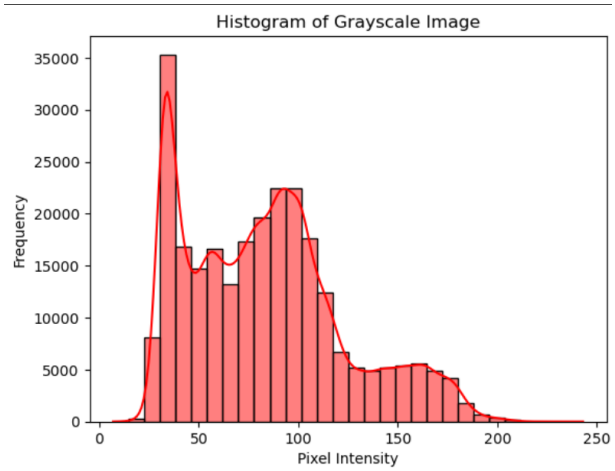


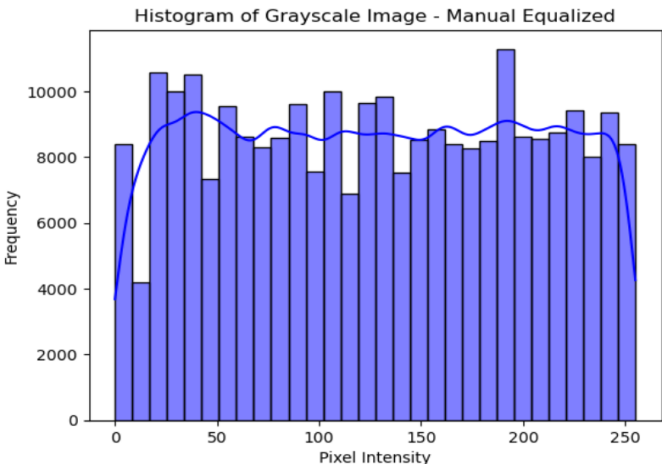


```
#histrogram equalisation manual
histogram = np.bincount(img_gy.flatten(), minlength=256)
cdf = histogram.cumsum()
cdf_normalized = (cdf - cdf.min()) * 255 / (cdf.max() - cdf.min())
cdf_normalized = cdf_normalized.astype('uint8')
equalized_image = cdf_normalized[img_gy]
plt.figure(figsize=(6, 6))
plt.subplot(1, 2, 1)
plt.imshow(img_gy, cmap='gray')
plt.title("Original Image")
plt.axis("off")
plt.subplot(1, 2, 2)
plt.imshow(equalized_image, cmap='gray')
plt.title("Equalized Image-Manual")
plt.axis("off")
plt.show()
```


```
sns.histplot(img_gy.ravel(), kde=True, bins=30, color='red')
plt.title("Histogram of Grayscale Image")
plt.xlabel('Pixel Intensity')
plt.ylabel('Frequency')
plt.show()
sns.histplot(equalized_image.ravel(), kde=True, bins=30, color='blue')
```

```
plt.title("Histogram of Grayscale Image - Manual Equalized")
plt.xlabel('Pixel Intensity')
plt.ylabel('Frequency')
plt.show()
```

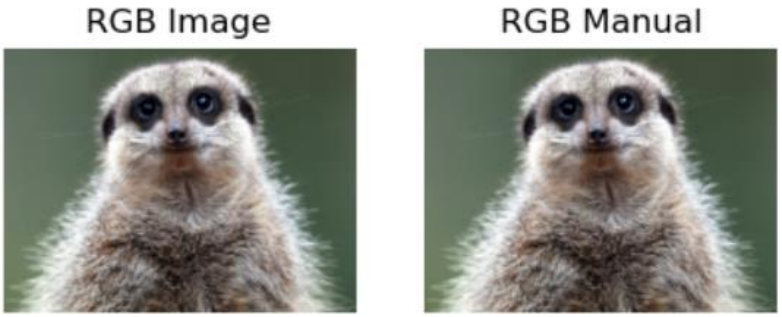




OTHER IMAGE:

```
#image processing
import cv2 as c
import matplotlib.pyplot as plt
import numpy as np
img=c.imread("len_hyna.jpg")
plt.figure(figsize=(3,3))
plt.imshow(img)
plt.title('Lenna Image')
plt.axis('off')
plt.show()
```


```
#library
img_color=c.cvtColor(img,c.COLOR_BGR2RGB)
plt.figure(figsize=(5,5))
plt.subplot(1,2,1)
plt.imshow(img_color)
plt.title("RGB Image")
plt.axis('off')
#manual
img_man=img[:,:,[2,1,0]]
plt.subplot(1,2,2)
plt.imshow(img_man)
plt.title("RGB Manual")
plt.axis("off")
plt.show()
```




```
image_gray = cv2.cvtColor(image, cv2.COLOR_RGB2GRAY)
plt.imshow(image_gray , cmap='gray')
plt.title('original graysce image')
```

```
plt.axis('off')

plt.show()

#constant

R = img_color[..., 0]

G = img_color[..., 1]

B = img_color[..., 2]

gray_manual = (0.299 * R + 0.587 * G + 0.114 * B).astype(np.uint8)

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

plt.title("Grayscale Image(industrial formula)")

plt.axis('off')

plt.show()
```

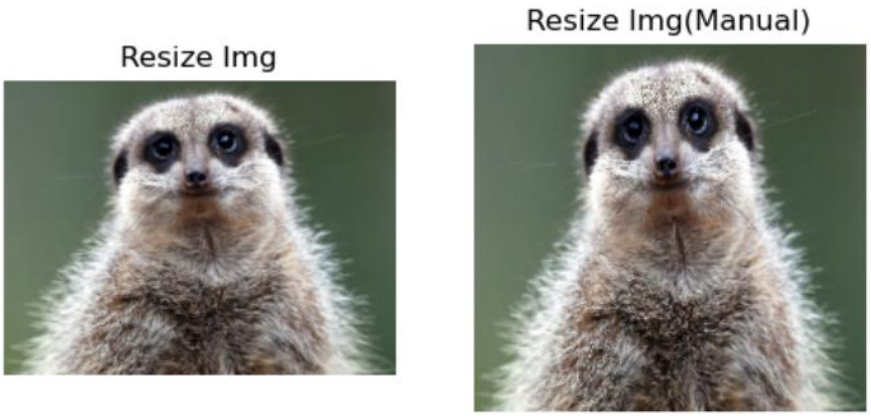

```
height, width, channels = img.shape

print(f"Height: {height}, Width: {width}, Channels: {channels}")
```

output:

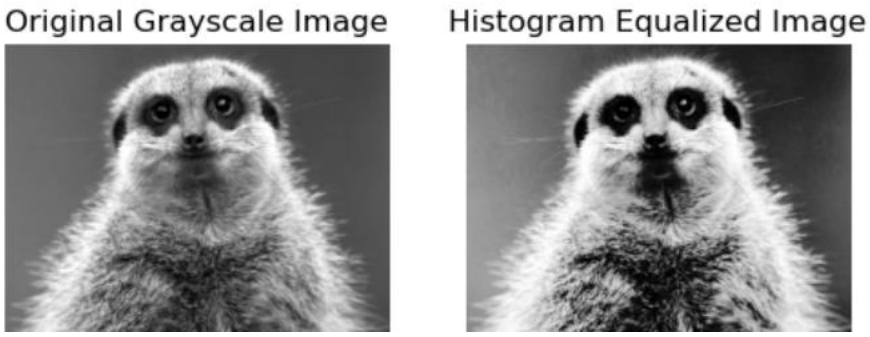
```
Height: 960, Width: 1280, Channels: 3
```

```
#resize

new_size = (400, 300)

img_resized = c.resize(img_man, new_size)

plt.subplot(1,2,1)

plt.imshow(img_resized)

plt.title("Resize Img")

plt.axis('off')

#resize manual

img_resized_manual = c.resize(img_color,(int(img_color.shape[1]/5),int(img_color.shape[0]/4)))
```

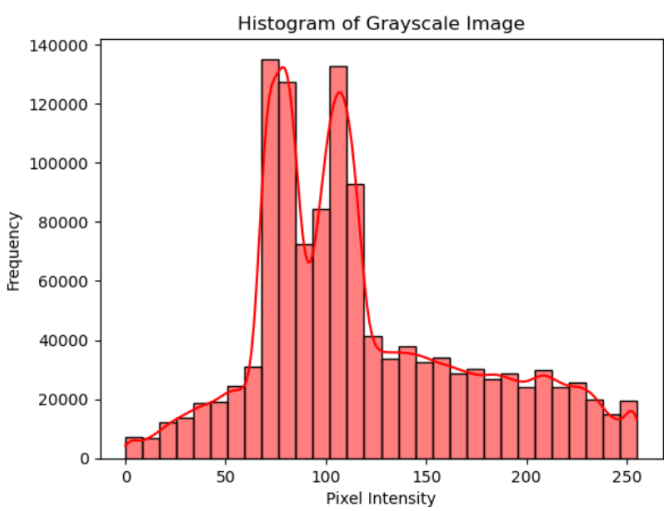
```
plt.subplot(1,2,2)
plt.imshow(img_resized_manual)
plt.title("Resize Img(Manual)")
plt.axis('off')
plt.show()
```

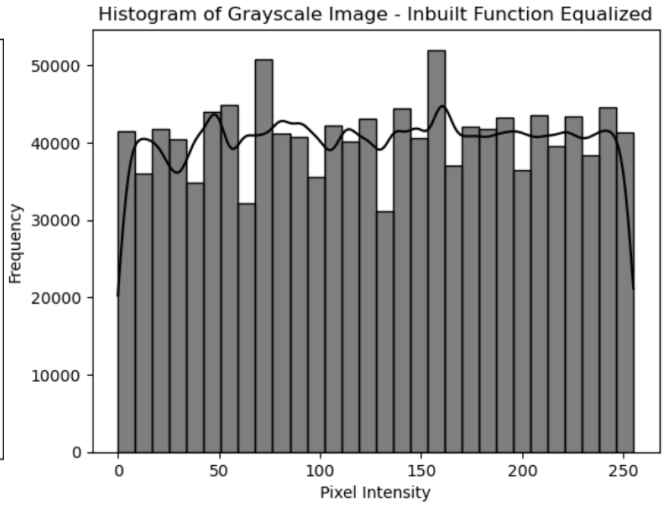



```
#equalize
img_gy=c.cvtColor(img,c.COLOR_RGB2GRAY)
img_eq = c.equalizeHist(img_gy)
plt.figure(figsize=(6,6))
plt.subplot(1, 2, 1)
plt.imshow(img_gy, cmap='gray')
plt.title("Original Grayscale Image")
plt.axis('off')
plt.subplot(1, 2, 2)
plt.imshow(img_eq, cmap='gray')
plt.title("Histogram Equalized Image")
plt.axis('off')
plt.show()
```



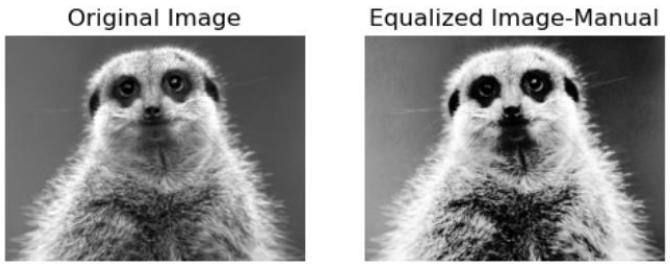

```
import cv2

import matplotlib.pyplot as plt

import numpy as np

import seaborn as sns

sns.histplot(img_gy.ravel(), kde=True, bins=30, color='red')

plt.title("Histogram of Grayscale Image")

plt.xlabel('Pixel Intensity')

plt.ylabel('Frequency')

plt.show()

sns.histplot(img_eq.ravel(), kde=True, bins=30, color='black')

plt.title("Histogram of Grayscale Image - Inbuilt Function Equalized")

plt.xlabel('Pixel Intensity')

plt.ylabel('Frequency')

plt.show()
```





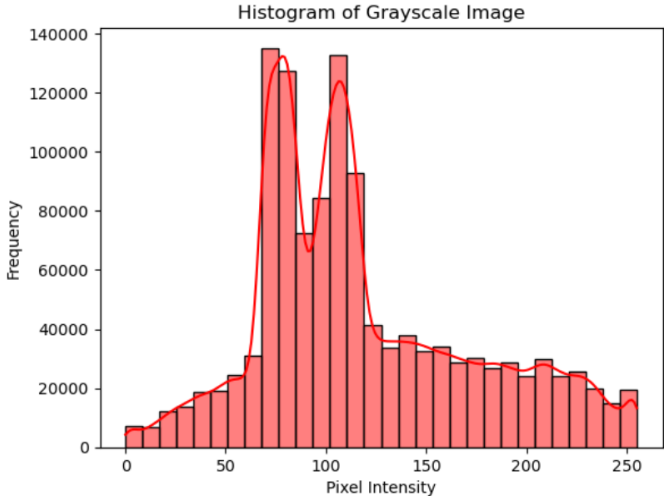

```
#histrogram equalisation manual
```

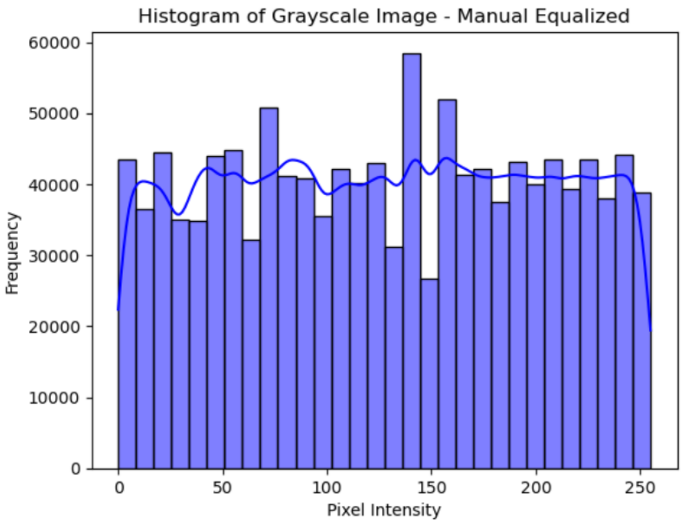
```
histogram = np.bincount(img_gy.flatten(), minlength=256)
cdf = histogram.cumsum()
cdf_normalized = (cdf - cdf.min()) * 255 / (cdf.max() - cdf.min())
cdf_normalized = cdf_normalized.astype('uint8')
equalized_image = cdf_normalized[img_gy]
plt.figure(figsize=(6, 6))
plt.subplot(1, 2, 1)
plt.imshow(img_gy, cmap='gray')
plt.title("Original Image")
plt.axis("off")
plt.subplot(1, 2, 2)
plt.imshow(equalized_image, cmap='gray')
plt.title("Equalized Image-Manual")
plt.axis("off")
plt.show()
```




```
sns.histplot(img_gy.ravel(), kde=True, bins=30, color='red')
plt.title("Histogram of Grayscale Image")
plt.xlabel('Pixel Intensity')
plt.ylabel('Frequency')
plt.show()
sns.histplot(equalized_image.ravel(), kde=True, bins=30, color='blue')
plt.title("Histogram of Grayscale Image - Manual Equalized")
plt.xlabel('Pixel Intensity')
plt.ylabel('Frequency')
```

plt.show()





RESULT :

Thus, the some basic concepts of image processing was successfully explored and analysed.

Exercise No:2 Date:23/01/2025	COLOR MODELS

AIM:

To show color models of differ types for images (RGB,CMY,HSV,YIQ).

FUNCTIONS:

1. **cv2.imread():**

   Loads an image from a specified file path.

2. **cv2.cvtColor():**

   Converts an image from one color space to another (e.g., BGR to RGB, RGB to HSV).

3. **cv2.split():**

   Splits a multi-channel image into its individual channels (e.g., H, S, V channels for HSV).

4. **cv2.merge():**

   Combines individual image channels into a multi-channel image.

5. **plt.imshow():**

   Displays an image in a specific color map using Matplotlib.

6. **plt.title():**

   Sets the title of a plot for better visualization.

7. **plt.axis("off"):**

   Hides the axes in a plot for a cleaner display of the image.

8. **plt.tight_layout():**

   Automatically adjusts subplot parameters for better spacing and layout.

9. **Image.copy():**

   Creates the copy of the image.

ALGORITHM :

1. **HSV Visualization Code :**

   · Creates an empty HSV image.

· Updates the HSV values with predefined combinations.

· Converts each HSV combination to RGB.

· Displays the results as small color patches, annotated with the HSV values for easy reference**.**

2. Load and Prepare the Image

- Input: A color image (sports.png) in BGR format (default for OpenCV).
- Processing:
  - Use cv2.imread() to read the image.
  - Convert the image from BGR to RGB using cv2.cvtColor() for accurate color representation.
- Output: The image in RGB format for further processing.

3. Convert RGB to HSV Color Space

- Input: RGB image.
- Processing:
  - Use cv2.cvtColor(image_rgb, cv2.COLOR_RGB2HSV) to convert the RGB image to HSV (Hue, Saturation, Value) color space.
  - Split the HSV image into three separate channels: H, S, and V using cv2.split(image_hsv).
- Output: Separate H, S, and V channels.

4. Modify the Hue Channel

- Input: Hue channel H.
- Processing:
  - Increase the Hue value by 10 using H1 = (H + 10) % 180.
  - The modulo operator ensures the Hue values wrap around (valid range is [0, 179] in OpenCV).
  - Merge the modified Hue channel back with original Saturation (S) and Value (V) channels using cv2.merge([H1, S, V]).
  - Convert the modified HSV back to RGB using cv2.cvtColor(modified_hsv1, cv2.COLOR_HSV2RGB).
- Output: Image with adjusted Hue.

5. Modify the Saturation Channel

- Input: Saturation channel S.
- Processing:
    - Double the Saturation values using np.clip(S * 2, 0, 255) to ensure values remain in the valid range [0, 255].
    - Merge the original Hue (H) and Value (V) with the modified Saturation channel using cv2.merge([H, S1, V]).
    - Convert the modified HSV back to RGB using cv2.cvtColor(modified_hsv2, cv2.COLOR_HSV2RGB).
- Output: Image with enhanced Saturation.

6. Modify the Value Channel

- Input: Value channel V.
- Processing:
    - Decrease the Value by 30 using np.clip(V - 30, 0, 255) to avoid negative values.
    - Merge the original Hue (H) and Saturation (S) with the modified Value channel using cv2.merge([H, S, V3]).
    - Convert the modified HSV back to RGB using cv2.cvtColor(modified_hsv3, cv2.COLOR_HSV2RGB).
- Output: Image with decreased brightness.

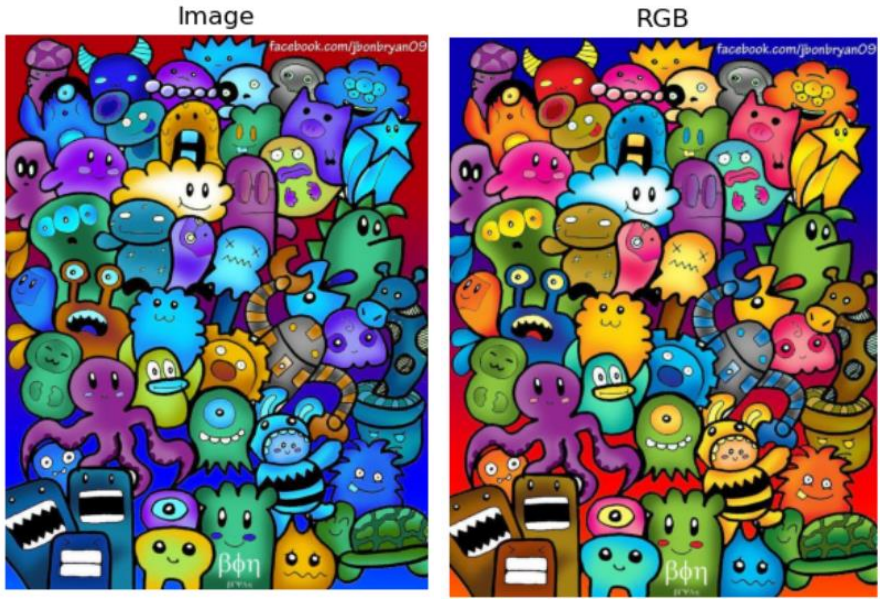
7. Visualize the Results

- Input: Original and modified RGB images.
- Processing:
    - Use Matplotlib (plt.imshow) to display the images side-by-side.
    - Add titles to each subplot to describe the modification applied.
    - Arrange the plots using plt.subplot and finalize the layout using plt.tight_layout.
- Output: A figure displaying the original image and modified images (Hue, Saturation, and Value adjustments).

Code Snippet:

```
import cv2 as c
import matplotlib.pyplot as plt
import numpy as np
img=c.imread("color.jpg")
plt.title('Image')
plt.axis('off')
plt.imshow(img)
plt.show()
img_rgb=c.cvtColor(img,c.COLOR_BGR2RGB)
plt.imshow(img_rgb)
plt.title('RGB')
plt.axis("off")
plt.show()
```

Output:



Code Snippet:

```
h,w,ch=img_rgb.shape
print(f'Heigth:{h},Width:{w},Channels:{ch}')
```

Output:

Heigth:1024,Width:780,Channels:3

Code Snippet:

#Red,Green,Blue plane

```
plt.figure(figsize=(10,10))
img_red=np.zeros_like(img_rgb,dtype=float)
img_red[:,:,0]=img_rgb[:,:,0]
img_red/=255
plt.subplot(2,2,1)
plt.title("Red Plane")
plt.axis('off')
plt.imshow(img_red)

img_green=np.zeros_like(img_rgb,dtype=float)
img_green[:,:,1]=img_rgb[:,:,1]
img_green/=255
plt.subplot(2,2,2)
plt.title("Green Plane")
plt.axis('off')
plt.imshow(img_green)

img_blue=np.zeros_like(img_rgb,dtype=float)
img_blue[:,:,2]=img_rgb[:,:,2]
img_blue/=255
plt.subplot(2,2,3)
plt.axis('off')
plt.title("Blue Plane")
plt.imshow(img_blue)
plt.tight_layout()
```

Output:

Code Snippet:

#CMY

```
img_cmy=np.zeros_like(img_rgb,dtype=float)
img_cmy[:,:,0]=1-(img_rgb[:,:,0]/255.0)
img_cmy[:,:,1]=1-(img_rgb[:,:,1]/255.0)
img_cmy[:,:,2]=1-(img_rgb[:,:,2]/255.0)
img_cmy=(img_cmy*255).astype(np.uint8)
plt.imshow(img_cmy)
plt.axis('off')
plt.title('CMY-Image')
plt.show()
```

Output:

Code Snippet:

#YIQ

```
img_yiq=np.zeros_like(img_rgb,dtype=float)
img_yiq[:,:,0]=(img_rgb[:,:,0]*0.299+img_rgb[:,:,1]*0.587+img_rgb[:,:,2]*0.114)
img_yiq[:,:,1]=(img_rgb[:,:,0]*0.596-img_rgb[:,:,1]*0.274-img_rgb[:,:,2]*0.322)
img_yiq[:,:,2]=(img_rgb[:,:,0]*0.211-img_rgb[:,:,1]*0.523+img_rgb[:,:,2]*0.312)
img_yiq[:,:,0]=np.clip(img_yiq[:,:,0],0,255)
img_yiq[:,:,1]=np.clip(img_yiq[:,:,1]+128,0,255)
img_yiq[:,:,2]=np.clip(img_yiq[:,:,2]+128,0,255)
img_yiq=img_yiq.astype(np.uint8)
plt.imshow(img_yiq)
plt.axis('off')
plt.title('YIQ-Image')
plt.show()
```

Output:

YIQ-Image

Code Snippet:

#HSV

```
img_hsv=c.cvtColor(img_rgb,c.COLOR_RGB2HSV)

plt.imshow(img_hsv)

plt.axis('off')

plt.title('HSV-Image')

plt.show()
```

Output:

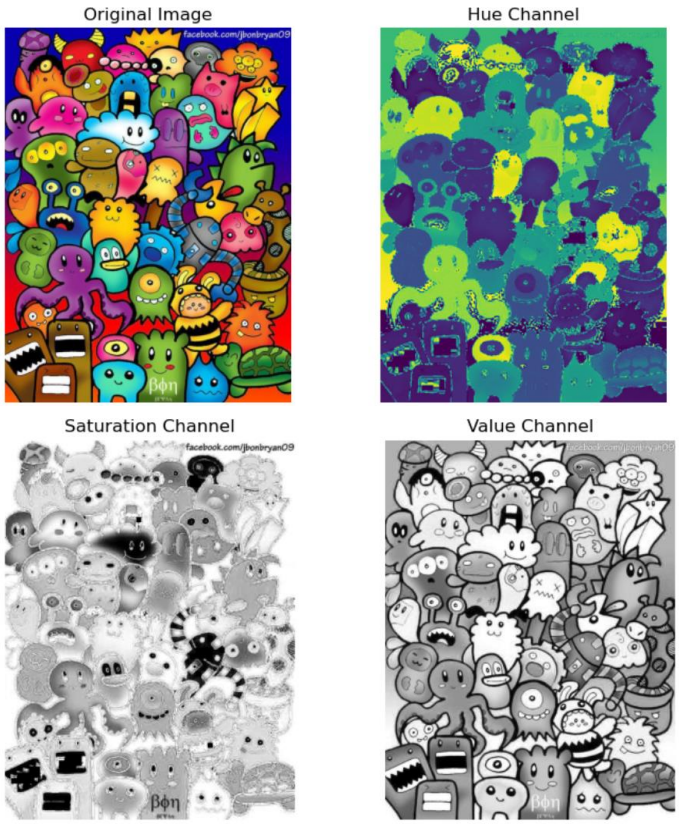
HSV-Image

Code Snippet:

#HSV Split

```
h, s, v = c.split(img_hsv)
plt.figure(figsize=(8, 8))
plt.subplot(2, 2, 1)
plt.imshow(img_rgb)
plt.title('Original Image')
plt.axis('off')
plt.subplot(2, 2, 2)
plt.imshow(h)
plt.title('Hue Channel')
plt.axis('off')
plt.subplot(2, 2, 3)
plt.imshow(s, cmap='gray')
plt.title('Saturation Channel')
plt.axis('off')
plt.subplot(2, 2, 4)
plt.imshow(v, cmap='gray')
plt.title('Value Channel')
plt.axis('off')
plt.tight_layout()
plt.show()
```

Output:



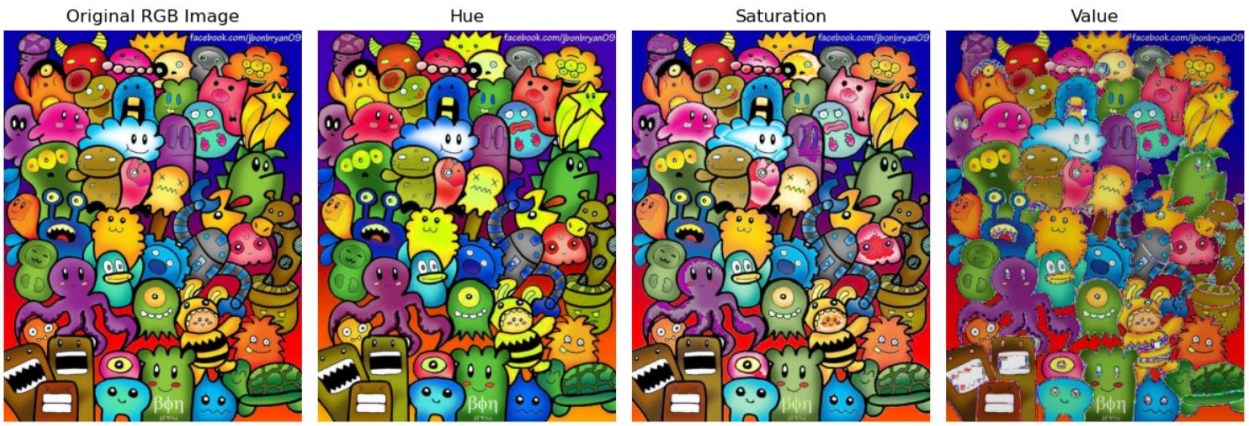
Code Snippet:

#HSV VALUE CHANGE OR DOUBLE

```
H1 = (h + 10) % 180

modified_hsv1 = c.merge([H1, s, v])

modified_rgb1= c.cvtColor(modified_hsv1, c.COLOR_HSV2RGB)


S1= np.clip(s*2, 0, 255)

modified_hsv2 = c.merge([h, S1, v])

modified_rgb2= c.cvtColor(modified_hsv2, c.COLOR_HSV2RGB)


V3 =np.clip( v- 30, 0, 255)

modified_hsv3 = c.merge([h, s, V3])

modified_rgb3 = c.cvtColor(modified_hsv3, c.COLOR_HSV2RGB)


plt.figure(figsize=(15, 6))

plt.subplot(1, 5, 1)
```

```
plt.imshow(img_rgb)
plt.title("Original RGB Image")
plt.axis("off")
plt.subplot(1, 5, 2)
plt.imshow(modified_rgb1)
plt.title("Hue")
plt.axis("off")
plt.subplot(1, 5, 3)
plt.imshow(modified_rgb2)
plt.title("Saturation")
plt.axis("off")
plt.subplot(1, 5, 4)
plt.imshow(modified_rgb3)
plt.title("Value")
plt.axis("off")
plt.tight_layout()
plt.show()
```

Output:

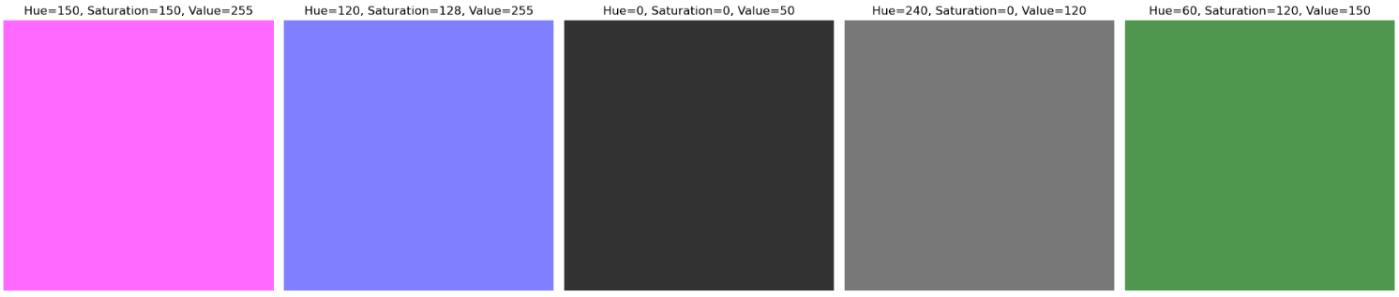


Code Snippet:

```
height, width = 25, 25
image_hsv = np.zeros((height, width, 3), dtype=np.uint8)
hsv_values = [
```

```
    [150, 150, 255],
    [120, 128, 255],
    [0, 0, 50],
    [240, 0, 120],
    [60, 120, 150]
]
plt.figure(figsize=(20, 10))
for idx, (hue, saturation, value) in enumerate(hsv_values):
    image_hsv[:, :, 0] = hue
    image_hsv[:, :, 1] = saturation
    image_hsv[:, :, 2] = value
    image_rgb = c.cvtColor(image_hsv, c.COLOR_HSV2RGB)
    plt.subplot(1, len(hsv_values), idx + 1)
    plt.imshow(image_rgb)
    plt.title(f"Hue={hue}, Saturation={saturation}, Value={value}")
    plt.axis('off')
plt.tight_layout()
plt.show()
```

Output:



OTHER IMAGE:

Code Snippet:

```
import cv2 as c
import matplotlib.pyplot as plt
import numpy as np
img=c.imread("multi-color-4.jpg")
plt.title('Image')
plt.axis('off')
plt.imshow(img)
plt.show()
img_rgb=c.cvtColor(img,c.COLOR_BGR2RGB)
plt.imshow(img_rgb)
plt.title('RGB')
plt.axis("off")
plt.show()
```

Output:

Code Snippet:

```
h,w,ch=img_rgb.shape
print(f'Heigth:{h},Width:{w},Channels:{ch}')
```

Output:

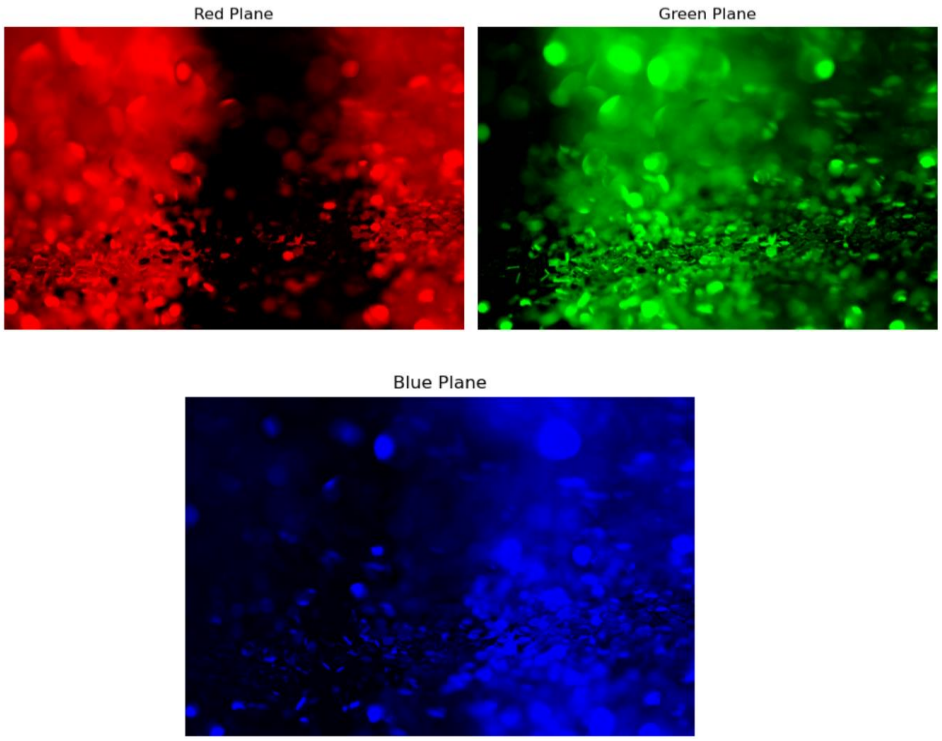
```
Heigth:3744,Width:5616,Channels:3
```

Code Snippet:

```
#Red,Green,Blue plane
plt.figure(figsize=(10,10))
img_red=np.zeros_like(img_rgb,dtype=float)
img_red[:,:,0]=img_rgb[:,:,0]
img_red/=255
plt.subplot(2,2,1)
plt.title("Red Plane")
plt.axis('off')
plt.imshow(img_red)

img_green=np.zeros_like(img_rgb,dtype=float)
img_green[:,:,1]=img_rgb[:,:,1]
img_green/=255
plt.subplot(2,2,2)
plt.title("Green Plane")
plt.axis('off')
plt.imshow(img_green)

img_blue=np.zeros_like(img_rgb,dtype=float)
img_blue[:,:,2]=img_rgb[:,:,2]
img_blue/=255
plt.subplot(2,2,3)
plt.axis('off')
plt.title("Blue Plane")
plt.imshow(img_blue)
```

plt.tight_layout()

Output:

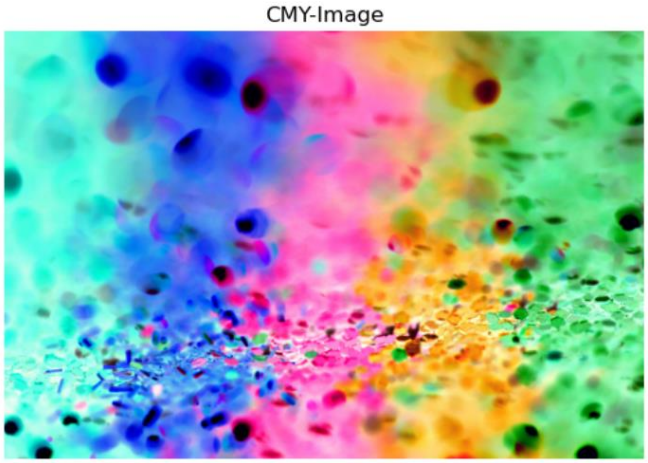


Code Snippet:

#CMY

```
img_cmy=np.zeros_like(img_rgb,dtype=float)
img_cmy[:,:,0]=1-(img_rgb[:,:,0]/255.0)
img_cmy[:,:,1]=1-(img_rgb[:,:,1]/255.0)
img_cmy[:,:,2]=1-(img_rgb[:,:,2]/255.0)
img_cmy=(img_cmy*255).astype(np.uint8)
plt.imshow(img_cmy)
plt.axis('off')
plt.title('CMY-Image')
plt.show()
```

Output:

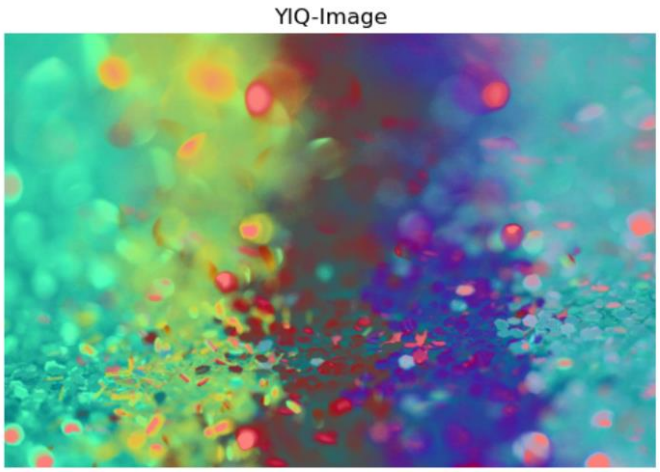


Code Snippet:

#YIQ

```
img_yiq=np.zeros_like(img_rgb,dtype=float)
img_yiq[:,:,0]=(img_rgb[:,:,0]*0.299+img_rgb[:,:,1]*0.587+img_rgb[:,:,2]*0.114)
img_yiq[:,:,1]=(img_rgb[:,:,0]*0.596-img_rgb[:,:,1]*0.274-img_rgb[:,:,2]*0.322)
img_yiq[:,:,2]=(img_rgb[:,:,0]*0.211-img_rgb[:,:,1]*0.523+img_rgb[:,:,2]*0.312)
img_yiq[:,:,0]=np.clip(img_yiq[:,:,0],0,255)
img_yiq[:,:,1]=np.clip(img_yiq[:,:,1]+128,0,255)
img_yiq[:,:,2]=np.clip(img_yiq[:,:,2]+128,0,255)
img_yiq=img_yiq.astype(np.uint8)
plt.imshow(img_yiq)
plt.axis('off')
plt.title('YIQ-Image')
plt.show()
```

Output:

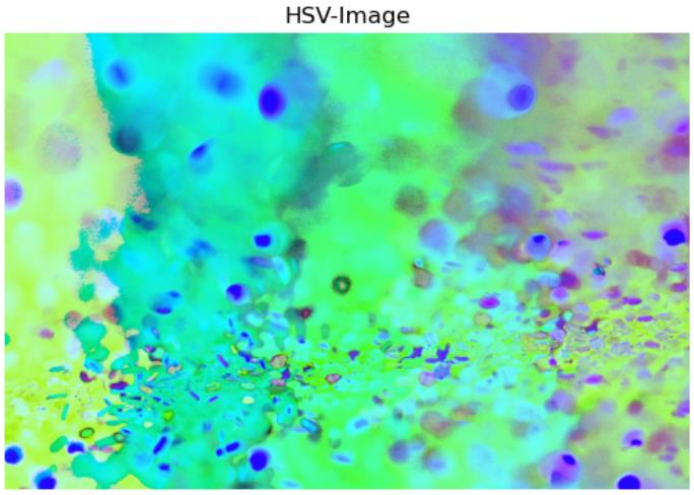


Code Snippet:

#HSV

```
img_hsv=c.cvtColor(img_rgb,c.COLOR_RGB2HSV)

plt.imshow(img_hsv)

plt.axis('off')

plt.title('HSV-Image')

plt.show()
```

Output:



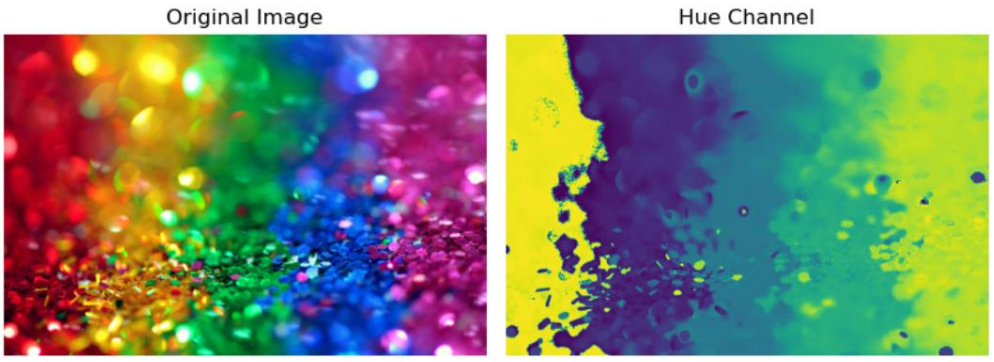
Code Snippet:

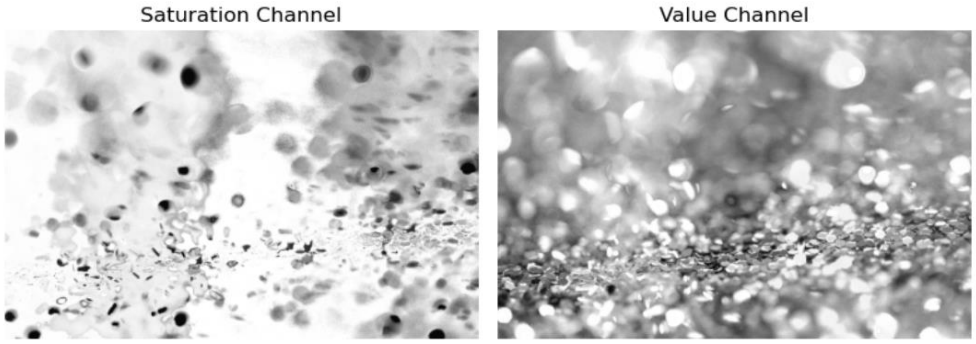
#HSV Split

```
h, s, v = c.split(img_hsv)

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

plt.subplot(2, 2, 1)
```

```
plt.imshow(img_rgb)
plt.title('Original Image')
plt.axis('off')
plt.subplot(2, 2, 2)
plt.imshow(h)
plt.title('Hue Channel')
plt.axis('off')
plt.subplot(2, 2, 3)
plt.imshow(s, cmap='gray')
plt.title('Saturation Channel')
plt.axis('off')
plt.subplot(2, 2, 4)
plt.imshow(v, cmap='gray')
plt.title('Value Channel')
plt.axis('off')
plt.tight_layout()
plt.show()
```

Output:





Code Snippet:

#HSV VALUE CHANGE OR DOUBLE

```
H1 = (h + 10) % 180

modified_hsv1 = c.merge([H1, s, v])

modified_rgb1= c.cvtColor(modified_hsv1, c.COLOR_HSV2RGB)


S1= np.clip(s*2, 0, 255)

modified_hsv2 = c.merge([h, S1, v])

modified_rgb2= c.cvtColor(modified_hsv2, c.COLOR_HSV2RGB)


V3 =np.clip( v- 30, 0, 255)

modified_hsv3 = c.merge([h, s, V3])

modified_rgb3 = c.cvtColor(modified_hsv3, c.COLOR_HSV2RGB)


plt.figure(figsize=(15, 6))

plt.subplot(1, 5, 1)

plt.imshow(img_rgb)

plt.title("Original RGB Image")

plt.axis("off")

plt.subplot(1, 5, 2)

plt.imshow(modified_rgb1)

plt.title("Hue")

plt.axis("off")
```

```
plt.subplot(1, 5, 3)
plt.imshow(modified_rgb2)
plt.title("Saturation")
plt.axis("off")
plt.subplot(1, 5, 4)
plt.imshow(modified_rgb3)
plt.title("Value")
plt.axis("off")
plt.tight_layout()
plt.show()
```

Output:

Result:

Color models of different types are shown.

EX NO:03 DATE:30/01/2025	IMAGE TRANSFORMATION

AIM:

To perform the image transformation like translation, rotation, scaling, shearing, reflection etc..

READ THE IMAGE:

ALGORITHM:

1. **Import Libraries**: Import OpenCV (cv2) and Matplotlib (plt).
2. **Read Image**: Load the image using cv2.imread().
3. **Display Image**: Use plt.imshow() to display the image.
4. **Hide Axes**: Remove axes with plt.axis('off').
5. **Show Image**: Display the image with plt.show().

CONVERSION TO BGR TO RGB:

ALGORITHM:

1. **Import Libraries**: Import OpenCV and Matplotlib.
2. **Load Image**: Use OpenCV to load the image (BGR format).
3. **Convert to RGB**: Convert the image from BGR to RGB using OpenCV.
4. **Display Image**: Show the image with Matplotlib, without axes.

CODE:

```
import cv2 as c
import numpy as np
import matplotlib.pyplot as plt
img=c.imread("iron man.jpg")
img_rgb=c.cvtColor(img,c.COLOR_BGR2RGB)
plt.title("iron man")
plt.axis('off')
plt.imshow(img_rgb)
plt.show()
```

img_rgb.shape

OUTPUT:

```
(1080, 1920, 3)
```

TRANSLATION:

ALGORITHM:

1. Import Libraries: Import cv2 for image processing and numpy for array manipulation.

2. Load Image: Load the image using OpenCV and get its dimensions.

3. Create Coordinate Grids: Generate x and y coordinate grids for the image.

4. Flatten Coordinates: Flatten the x and y grids into 1D arrays.

5. Translate Image: Create a translation matrix to shift the image and apply the translation to the coordinates.

6. Create Empty Image: Create a blank image to hold the translated result.

7. Map Translated Pixels: Map valid pixel positions from the original image to the translated image.

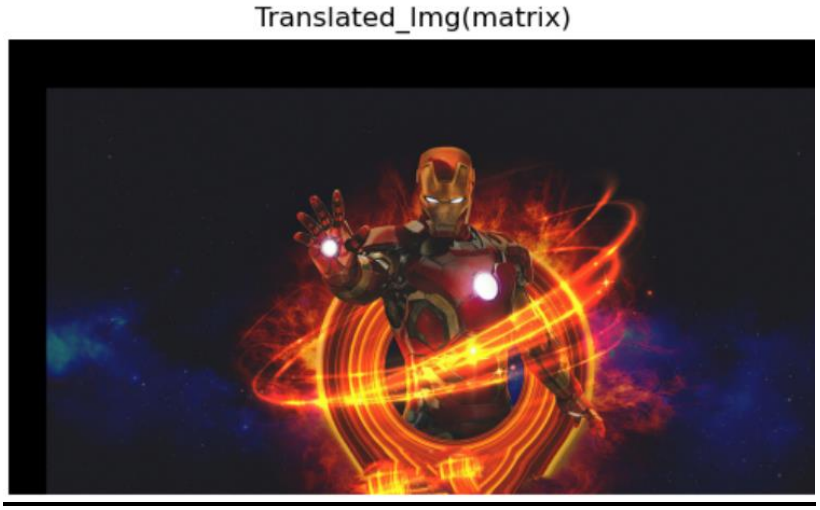
8. Display Translated Image: Show the translated image using Matplotlib

CODE:

```
def translation2(x, y, img):
    mat = np.array([[1, 0, x],[0, 1, y],[0, 0, 1]])
    h, w, ch = img.shape
```

```
    img2 = np.zeros_like(img)
    y_indices, x_indices = np.indices((h, w))
    orig_coords = np.stack([x_indices, y_indices, np.ones_like(x_indices)], axis=-1)
    orig_coords = orig_coords.reshape(-1, 3).T
    new_coords = mat @ orig_coords
    new_x, new_y = new_coords[0].astype(int), new_coords[1].astype(int)
    mask = (0 <= new_x) & (new_x < w) & (0 <= new_y) & (new_y < h)
    orig_x, orig_y = x_indices.flatten()[mask], y_indices.flatten()[mask]
    new_x, new_y = new_x[mask], new_y[mask]
    img2[new_y, new_x] = img[orig_y, orig_x]
    return img2
translate2=translation2(90,115,img_rgb)
plt.imshow(translate2)
plt.title("Translated_Img")
plt.axis("off")
plt.show()
```

OUTPUT:



ROTATION:

ALGORITHIM:

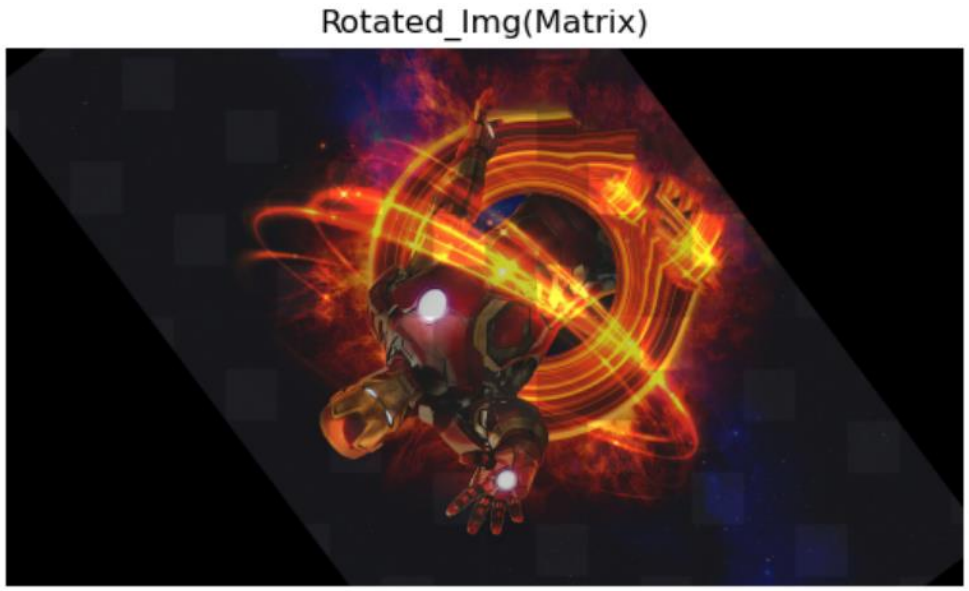
- **Load Image**: Load the image and get its dimensions.
- **Set Rotation Angle**: Define the rotation angle (theta).
- **Generate Coordinate Grid**: Create 2D grids for x and y pixel coordinates.
- **Center Coordinates**: Translate coordinates to rotate around the center of the image.
- **Rotate Coordinates**: Apply a rotation matrix to the coordinates.
- **Map to New Image**: Map the rotated coordinates back onto a new image.
- **Display the Image**: Show the rotated image.

CODE:

```
def rotation2(a, img):
    h, w, ch = img.shape
    center_x, center_y = w // 2, h // 2
    mat = np.array([[m.cos(a), -m.sin(a), 0],
                    [m.sin(a), m.cos(a), 0],
                    [0, 0, 1]])
    img4 = np.zeros_like(img)
    y_indices, x_indices = np.indices((h, w))
    orig_coords = np.stack([x_indices - center_x, y_indices - center_y,
np.ones_like(x_indices)], axis=-1)
    orig_coords = orig_coords.reshape(-1, 3).T
    new_coords = mat @ orig_coords
    new_x, new_y = new_coords[0].astype(int) + center_x, new_coords[1].astype(int) +
center_y
    mask = (0 <= new_x) & (new_x < w) & (0 <= new_y) & (new_y < h)
    orig_x, orig_y = x_indices.flatten()[mask], y_indices.flatten()[mask]
    new_x, new_y = new_x[mask], new_y[mask]
    img4[new_y, new_x] = img[orig_y, orig_x]
    return img4
```

```
rot2=rotation2(180,img_rgb)
plt.imshow(rot2)
plt.title("Rotated_Img")
plt.axis("off")
plt.show()
```

OUTPUT:



SCALING:

ALGORITHM:

- **Load Image**: Load the image and extract its height, width, and channels.
- **Set Scaling Factors**: Define the scaling factors for the x and y axes (scale_x, scale_y).
- **Create Grid of Coordinates**: Generate a grid of x and y coordinates for the image using np.indices().
- **Apply Scaling**: Apply scaling to the x and y coordinates, ensuring the scaling is centered around the image center.
- **Round and Convert Coordinates**: Round the new coordinates and convert them to integers.
- **Map Valid Coordinates**: Check which coordinates are within the image bounds and map valid coordinates to the scaled image.

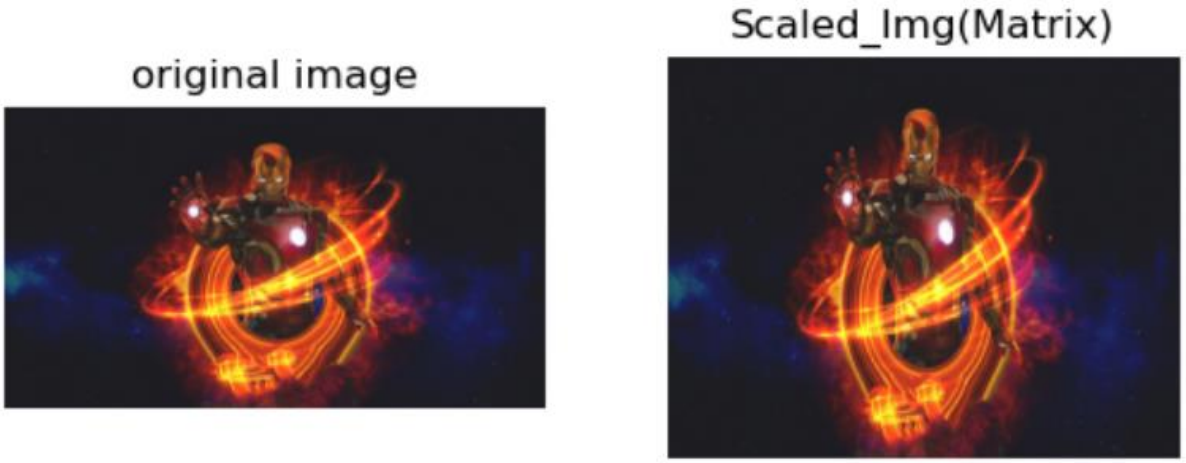
 Display the Image: Display the scaled image.

CODE:

```
def scaling2(s1, s2, img):
    mat = np.array([[s1, 0, 0],
                    [0, s2, 0],
                    [0, 0, 1]])

    h, w, ch = img.shape
    new_h, new_w = int(h * s2), int(w * s1)
    img3 = np.zeros((new_h, new_w, ch), dtype=np.uint8)
    y_indices, x_indices = np.indices((new_h, new_w))
    orig_coords = np.stack([x_indices / s1, y_indices / s2, np.ones_like(x_indices)], axis=-
1)
    orig_coords = orig_coords.reshape(-1, 3).T
    orig_x, orig_y = orig_coords[0].astype(int), orig_coords[1].astype(int)
    mask = (0 <= orig_x) & (orig_x < w) & (0 <= orig_y) & (orig_y < h)
    new_x, new_y = x_indices.flatten()[mask], y_indices.flatten()[mask]
    orig_x, orig_y = orig_x[mask], orig_y[mask]
    img3[new_y, new_x] = img[orig_y, orig_x]
    return img3
scale2=scaling2(0.5,0.7,img_rgb)
plt.subplot(2,2,1)
plt.imshow(img_rgb)
plt.title("original image")
plt.axis("off")
plt.subplot(2,2,2)
plt.imshow(scale2)
```

```
plt.title("Scaled_Img")
plt.axis("off")
plt.show()
```

OUTPUT:



SHEARING:

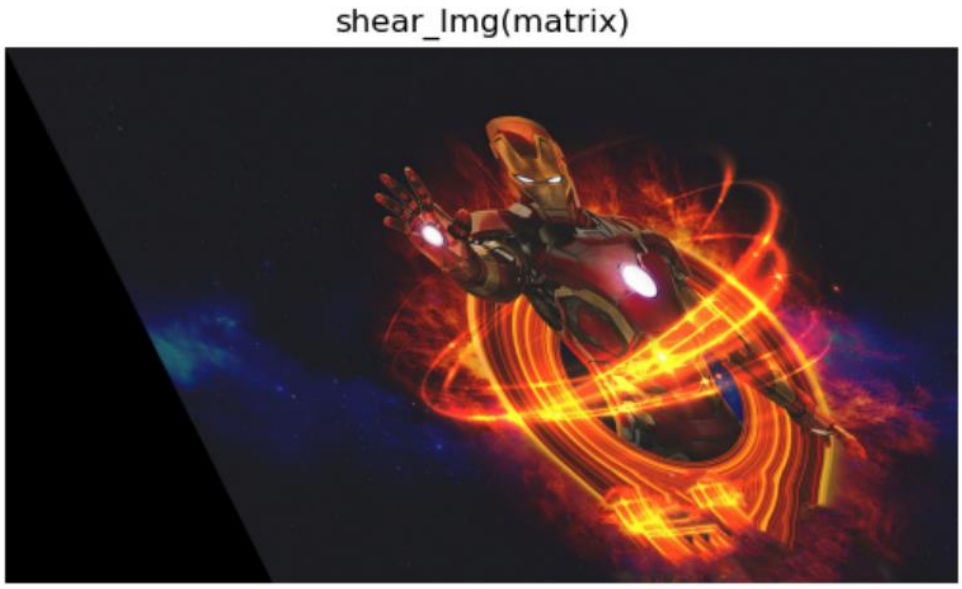
ALGORITHM:

- **Load Image**: Load the image and get its dimensions (height, width).
- **Set Shear Factors**: Define the shear factors for the x and y directions (sh_x and sh_y).
- **Generate Coordinate Grid**: Create a grid of x and y coordinates for the image.
- **Apply Shear Transformation**: Apply the shear to the coordinates:
  - sheared_x = x + sh_x * y: Shears the x-coordinates by shifting them based on the y-coordinates.
  - sheared_y = y + sh_y * x: Shears the y-coordinates by shifting them based on the x-coordinates.
- **Round and Convert Coordinates**: Round the transformed coordinates and convert them to integers.
- **Check Valid Indices**: Ensure the transformed coordinates are within the bounds of the image.
- **Map Pixels**: Map the valid pixels from the original image to the sheared image.
- **Display Image**: Display the sheared image.

CODE:

```
def shear_img2(axis, factor, img):
    h, w, ch = img.shape
    if axis == 'x':
        mat = np.array([[1, factor, 0],
                        [0, 1, 0],
                        [0, 0, 1]])
    elif axis == 'y':
        mat = np.array([[1, 0, 0],
                        [factor, 1, 0],
                        [0, 0, 1]])
    img6 = np.zeros_like(img)
    y_indices, x_indices = np.indices((h, w))
    orig_coords = np.stack([x_indices, y_indices, np.ones_like(x_indices)], axis=-1)
    orig_coords = orig_coords.reshape(-1, 3).T
    new_coords = mat @ orig_coords
    new_x, new_y = new_coords[0].astype(int), new_coords[1].astype(int)
    mask = (0 <= new_x) & (new_x < w) & (0 <= new_y) & (new_y < h)
    orig_x, orig_y = x_indices.flatten()[mask], y_indices.flatten()[mask]
    new_x, new_y = new_x[mask], new_y[mask]
    img6[new_y, new_x] = img[orig_y, orig_x]
    return img6
shear2= shear_img2('x',0.5, img_rgb)
plt.imshow(shear2)
plt.title("shear_Img")
plt.axis("off")
plt.show()
```

OUTPUT:



REFLECTION:

ALGORITHM:

- **Load Image**: Load the image using OpenCV and get its dimensions (height, width).
- **Generate Coordinates**: Create coordinate grids for x and y pixels.
- **Reflection Calculation**: Reflect the x coordinates by subtracting from width - 1. This gives the mirrored x positions.
- **Clip Coordinates**: Ensure the reflected x coordinates stay within the valid bounds of the image (0 to width - 1).
- **Create Output Image**: Create a blank image to store the reflected result.
- **Map Pixels**: Map the original pixels to the reflected positions on the new image.
- **Display Image**: Show the reflected image using Matplotlib.

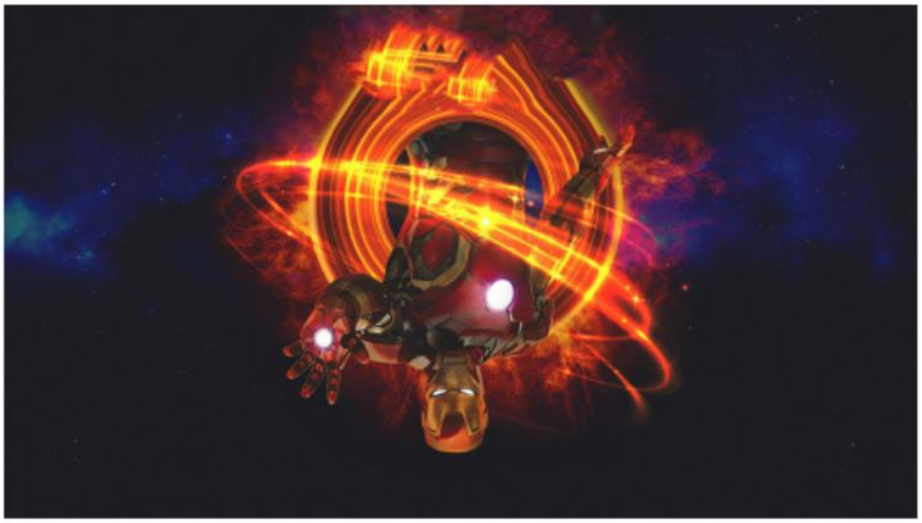
CODE:

```
def reflect2(axis, img):
    h, w, ch = img.shape
    if axis == 'x':
        mat = np.array([[1, 0, 0],
                        [0, -1, h],
```

```
                [0, 0, 1]])
    elif axis == 'y':
        mat = np.array([[-1, 0, w],
                [0, 1, 0],
                [0, 0, 1]])
    img5= np.zeros_like(img)
    y_indices, x_indices = np.indices((h, w))
    orig_coords = np.stack([x_indices , y_indices, np.ones_like(x_indices)], axis=-1)
    orig_coords = orig_coords.reshape(-1, 3).T
    new_coords = mat @ orig_coords
    new_x, new_y = new_coords[0].astype(int) , new_coords[1].astype(int)
    mask = (0 <= new_x) & (new_x < w) & (0 <= new_y) & (new_y < h)
    orig_x, orig_y = x_indices.flatten()[mask], y_indices.flatten()[mask]
    new_x, new_y = new_x[mask], new_y[mask]
    img5[new_y, new_x] = img[orig_y, orig_x]
    return img5
ref_img2= reflect2('x', img_rgb)
plt.imshow(ref_img2)
plt.title("Reflected_Img")
plt.axis("off")
plt.show()
```

OUTPUT:

Reflected_Img(Matrix)



RESULT:

The transformation in image is analysed.

EX NO:04 DATE:06/02/2025	DISCRETE FOURIER TRANSFORM, HISTOGRAM PROCESSING, LINEAR FILTERING

AIM:

To perform discrete fourier transform , histogram processing, linear filtering on certain image.

DFT:

ALGORITHM:

- Import the necessary libraries (cv2 for image reading and matplotlib for plotting).
- Read the image from the given path using cv2.imread.
- Create a figure for displaying the image with specific dimensions using plt.figure.
- Show the image in the created figure using plt.imshow.
- Hide axes for a clean display with plt.axis('off').
- Display the image with plt.show.

CODE:

```
import cv2 as c
import matplotlib.pyplot as plt
import numpy as np
img=c.imread("leo.jpg")
img_rgb=c.cvtColor(img,c.COLOR_BGR2GRAY)
plt.imshow(img_rgb,cmap="gray")
plt.axis("off")
plt.show()
img.shape
```

OUTPUT:

(219, 123)

DFT WITH ORIGINAL IMAGE:

ALGORITHM:

1. Load the grayscale image.
2. Apply DFT on the image to get frequency components.
3. Shift the DFT result to center the zero-frequency component.
4. Calculate the magnitude of the DFT components.
5. Display the original and magnitude spectrum images side by side.
6. Plot the histogram of the magnitude spectrum to show frequency distribution.

DFT WITH HISTOGRAM EQUALISED IMAGE:

ALGORITHM:

1. **Read the image** in grayscale.
2. **Apply histogram equalization** to improve the contrast.
3. **Display** the original image and the equalized image side by side.
4. **Apply DFT** on the equalized image to convert it into the frequency domain.
5. **Shift** the zero-frequency component to the center of the DFT result.

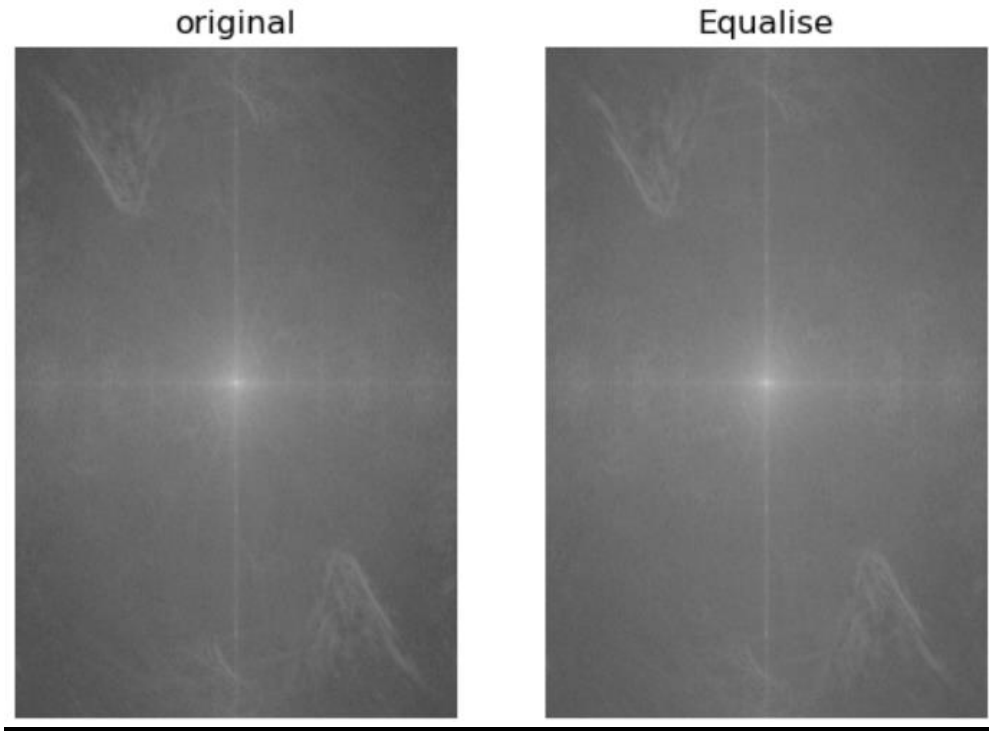
6. **Calculate the magnitude spectrum** of the equalized image and display it.
7. **Plot a histogram** of the magnitude spectrum values to analyze the frequency distribution.

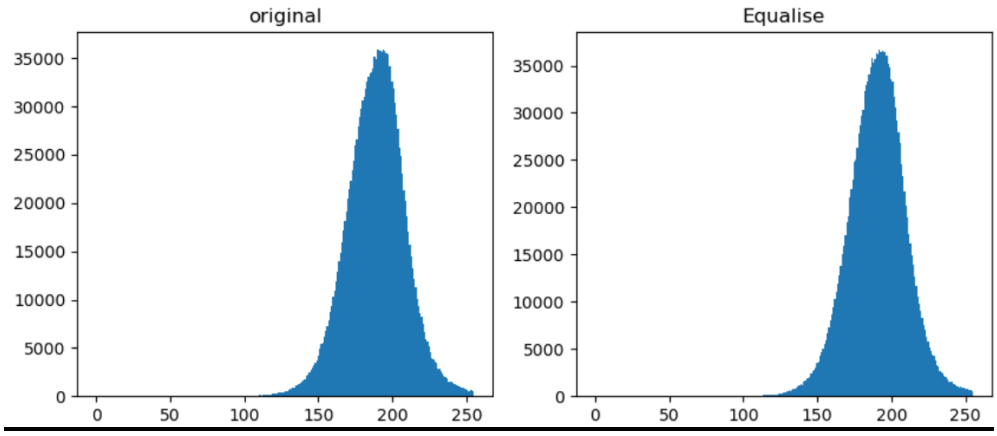
CODE:

```
img_eq=c.equalizeHist(img_rgb)
def DFT(img):
    dft=np.fft.fft2(img)
    dft_shift=np.fft.fftshift(dft)
    mg=20*np.log(np.abs(dft_shift))
    return mg
original=DFT(img_rgb)
equalise=DFT(img_eq)
plt.subplot(1,2,1)
plt.imshow(original,cmap="gray")
plt.axis("off")
plt.title("original")
plt.subplot(1,2,2)
plt.imshow(equalise,cmap="gray")
plt.axis("off")
plt.title("Equalise")
plt.show()
plt.figure(figsize=(10,4))
plt.subplot(1,2,1)
plt.hist(original.ravel(),bins=256,range=(0,255))
plt.title("original")
plt.subplot(1,2,2)
```

```
plt.hist(equalise.ravel(),bins=256,range=(0,255))
plt.title("Equalise")
plt.show()
```

OUTPUT:





BLURRING:

ALGORITHM:

Step 1: Read the Image

- Load the image in grayscale so that each pixel is represented by a single intensity value (between 0 and 255).

Step 2: Create an Empty Output Image

- We will create a new image to store the blurred result. It will have the same dimensions as the original image.

Step 3: Define a 3x3 Box Blur Kernel

- A **3x3 box blur kernel** is just a 3x3 matrix where each element is 1. When we normalize this kernel by dividing by 9 (since there are 9 elements in total), we get an average of all surrounding pixels.

Step 4: Loop Through the Image

- Iterate over each pixel in the image, excluding the borders (since the kernel would go outside the image at the edges).
- For each pixel, get the surrounding **3x3 neighborhood** of pixels.

Step 5: Apply the Kernel

- Multiply each pixel in the 3x3 neighborhood by the corresponding value in the kernel (in this case, 1/9 for each pixel).
- Sum all the values in the neighborhood, and assign this sum as the new value for the current pixel in the blurred image.

Step 6: Display the Original and Blurred Image

- Show the original image alongside the blurred result for comparison.

SHARPENING:

ALGORITHM:

1. **Input**: A grayscale image.
2. **Define the Sharpening Kernel**:
    - The 3x3 sharpening kernel is:

[0, -1, 0]

[-1, 5, -1]

[0, -1, 0]

3. **Create an Empty Image**:
    - Initialize an empty image with the same size as the original image to store the sharpened result.
4. **Iterate Through the Image**:

- For each pixel in the image (excluding the borders):
  - Extract the **3x3 region** around the current pixel.
  - Apply the **sharpening kernel** by multiplying the corresponding values in the 3x3 region with the kernel and summing the results.

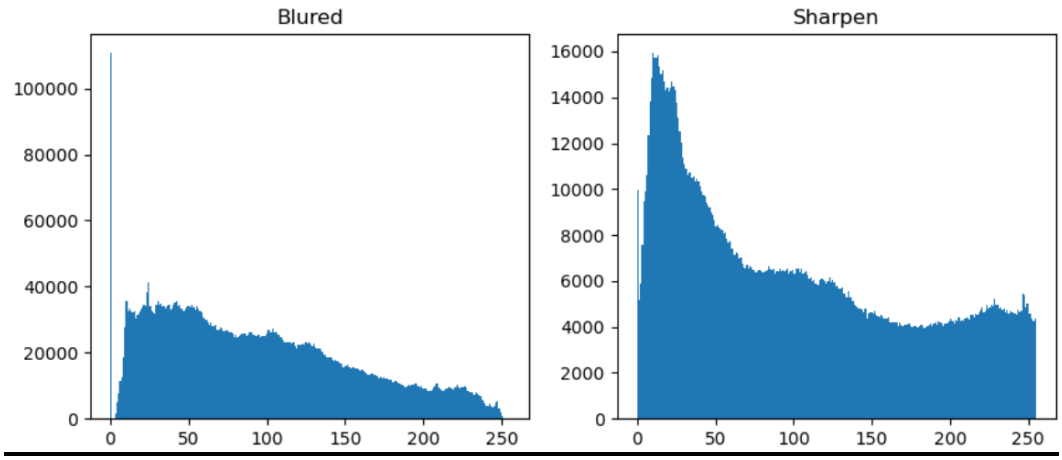
5. **Update the Pixel Value**:
   - The result from applying the kernel (sum of products) will be the new value for the current pixel in the sharpened image.
6. **Edge Handling**:
   - To avoid going out of bounds, skip processing the borders of the image (pixels on the first and last row/column).
7. **Output**: Return or display the sharpened image.

CODE:

```
#BLURED IMAGE
blur=np.ones((15,15),np.float32)/(225)
blured=np.zeros_like(img)
h,w=img_rgb.shape
off=15//2
for i in range(off,h-off):
    for j in range(off,w-off):
        region = img_rgb[i - off:i + off + 1, j - off:j + off + 1]
        blurr= np.sum(region * blur)
        blured[i,j]=np.clip(blurr,0,255)
plt.imshow(blured,cmap="gray")
plt.axis("off")
plt.title("Blured")
plt.show()
```

```
#SHARPHEN IMAGE
sharpen = np.array([[0, -1, 0],
                    [-1,  5, -1],
                    [0, -1, 0]], dtype=np.float32)
sharpened = np.zeros_like(img_rgb)
h, w= img_rgb.shape
off = 3//2
for i in range(off, h - off):
    for j in range(off, w - off):
        region = img_rgb[i - off:i + off + 1, j - off:j + off + 1]
        sharpened[i, j] = np.sum(region * sharpen)
        sharpened[i, j] = np.clip(sharpened[i, j], 0, 255)
plt.imshow(sharpened,cmap="gray")
plt.axis("off")
plt.title("Sharpened Image")
plt.show()
plt.figure(figsize=(10,4))
plt.subplot(1,2,1)
plt.hist(blured.ravel(),bins=256,range=(0,255))
plt.title("Blured")
plt.subplot(1,2,2)
plt.hist(sharpened.ravel(),bins=256,range=(0,255))
plt.title("Sharpen")
plt.show()
```

OUTPUT:



NEGATIVE:

ALGORITHM:

- **Read the Image in Grayscale**:
  - You load the image in grayscale (cv2.IMREAD_GRAYSCALE), so the pixel values range from 0 (black) to 255 (white).
- **Apply the Negative Effect**:
  - For each pixel in the image, the negative effect is applied by subtracting the pixel value from 255. This inverts the grayscale values, creating a negative version of the image. A pixel value of 0 becomes 255 (white), and a pixel value of 255 becomes 0 (black).

negative_image = 255 - image

 Display the Images:

- You use matplotlib to display:
  - The **original image**.
  - The **negative image**.
- These are shown in a 2x2 grid, where the first row contains the original and negative images, and the second row contains the histograms of both images.

 Plot Histograms:

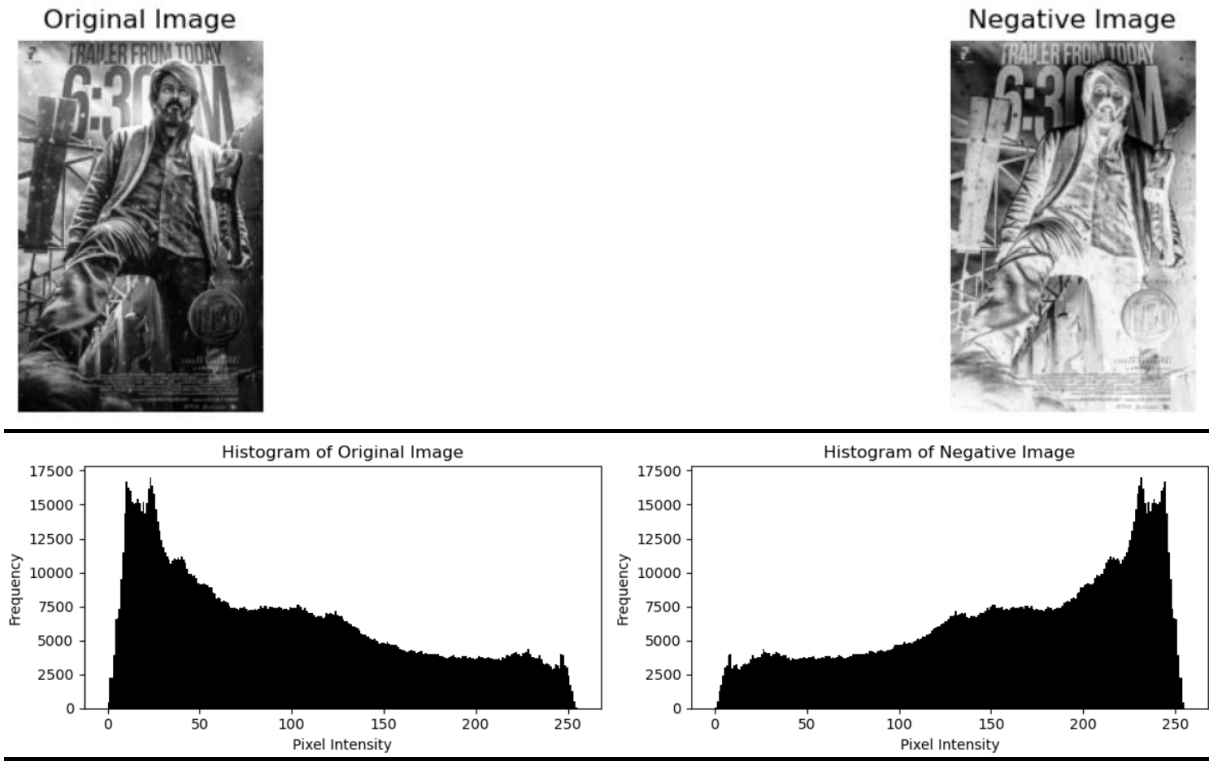
- For both the original and negative images, you generate and display the histograms. The histograms show the distribution of pixel intensities (how often each pixel value occurs).
- **Original Image Histogram**: Shows the distribution of pixel intensities in the original grayscale image.
- **Negative Image Histogram**: Shows how the negative effect has altered the distribution. It will be roughly a mirror image of the original histogram.

CODE:

```
negative_image = 255 - img_rgb

plt.figure(figsize=(12, 6))

plt.subplot(2, 2, 1)

plt.imshow(img_rgb, cmap="gray")

plt.title("Original Image")

plt.axis("off")

plt.subplot(2, 2, 2)

plt.imshow(negative_image, cmap="gray")

plt.title("Negative Image")

plt.axis("off")

plt.subplot(2, 2, 3)

plt.hist(img_rgb.ravel(), bins=256, color='black')

plt.title("Histogram of Original Image")

plt.xlabel("Pixel Intensity")
```

```
plt.ylabel("Frequency")
plt.subplot(2, 2, 4)
plt.hist(negative_image.ravel(), bins=256, color='black')
plt.title("Histogram of Negative Image")
plt.xlabel("Pixel Intensity")
plt.ylabel("Frequency")
plt.tight_layout()
plt.show()
```

OUTPUT:



HISTOGRAM VARIATIONS:

RIGHT SHIFT:

ALGORITHM:

- **Reading the Image**:
  - The image is loaded in grayscale (cv2.IMREAD_GRAYSCALE), where pixel values range from 0 (black) to 255 (white).
- **Brightness Adjustment**:

- The cv2.add() function is used to **add 50** to all pixel values in the image:

i2 = cv2.add(image, +50)

- This operation **increases** the brightness by adding 50 to each pixel. Any pixel value greater than 255 will be capped at 255.

 Display the Images:

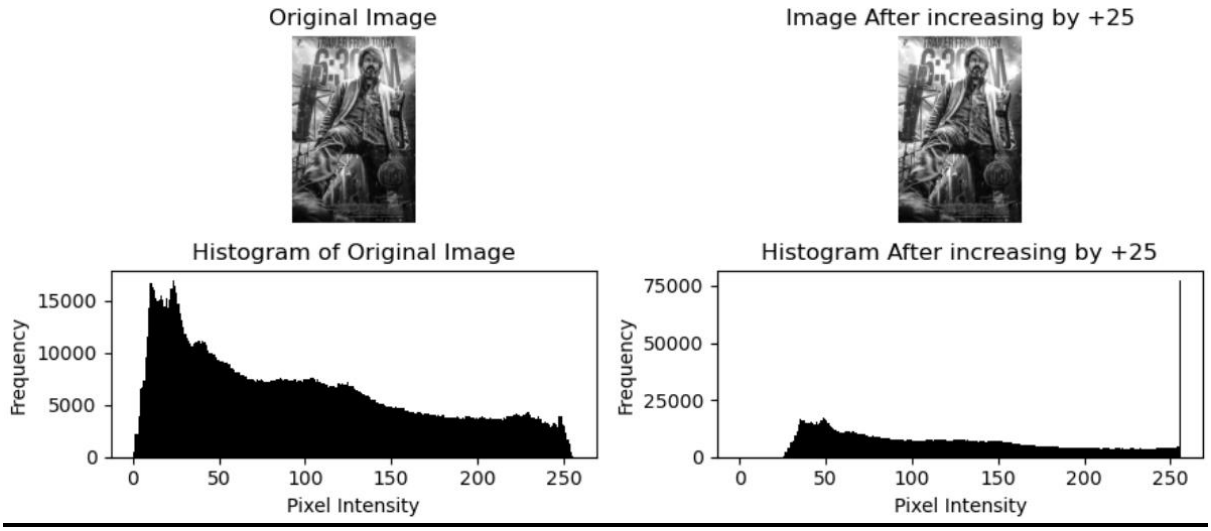
- The original image and the adjusted image (brighter) are displayed side by side in a 2x2 grid.
- The histograms of both images are also plotted. Histograms show the distribution of pixel intensities for both the original and the adjusted images.

CODE:

```
i2 = c.add(img_rgb, +25)

plt.figure(figsize=(9, 4))

plt.subplot(221)

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

plt.axis('off')

plt.title("Original Image")

plt.subplot(222)

plt.imshow(i2, cmap='gray')

plt.axis('off')

plt.title("Image After Decreasing by -25")

plt.subplot(223)

plt.hist(img_rgb.ravel(), bins=256, range=[0, 256], color='black')

plt.title("Histogram of Original Image")

plt.xlabel("Pixel Intensity")

plt.ylabel("Frequency")

plt.subplot(224)

plt.hist(i2.ravel(), bins=256, range=[0, 256], color='black')

plt.title("Histogram After Decreasing by -25")
```

```
plt.xlabel("Pixel Intensity")
plt.ylabel("Frequency")
plt.tight_layout()
plt.show()
```

OUTPUT:



LEFT SHIFT:

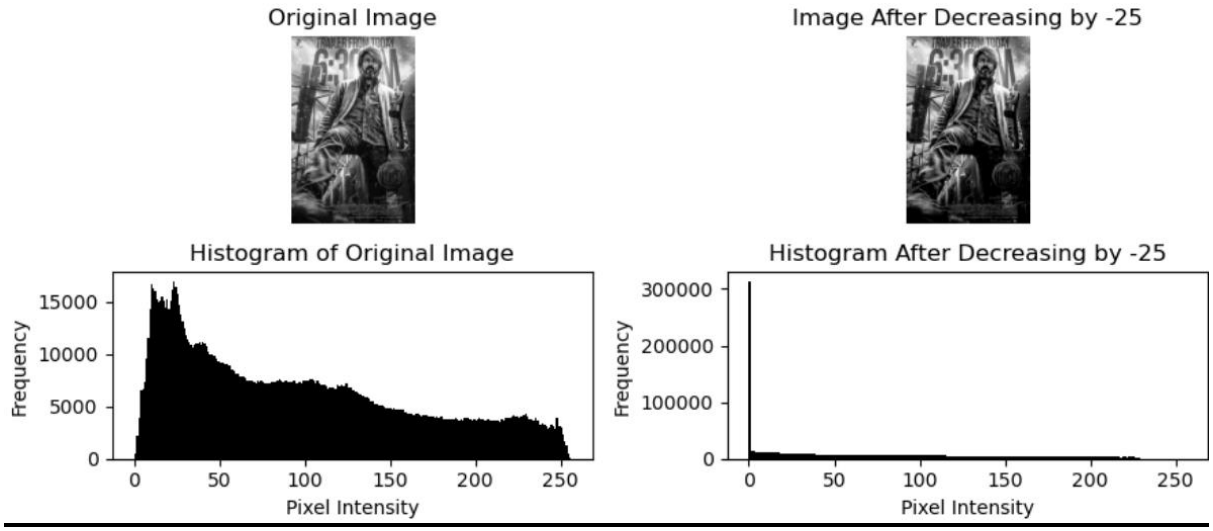
ALGORITHM:

- **Read the image**: Load the image in grayscale mode (black and white).
- **Decrease pixel intensity**:
  - For each pixel in the image, subtract 50 from its intensity value.
  - If the result is less than 0, set the pixel value to 0 (avoid negative values).
- **Display the images**:
  - Show the original image.
  - Show the modified (darker) image.
- **Plot histograms**:
  - Plot the histogram of pixel intensities for the original image.
  - Plot the histogram of pixel intensities for the modified (darker) image.

CODE:

```
i2 = c.add(img_rgb, -25)
plt.figure(figsize=(9, 4))
plt.subplot(221)
plt.imshow(img_rgb, cmap='gray')
plt.axis('off')
plt.title("Original Image")
plt.subplot(222)
plt.imshow(i2, cmap='gray')
plt.axis('off')
plt.title("Image After Decreasing by -25")
plt.subplot(223)
plt.hist(img_rgb.ravel(), bins=256, range=[0, 256], color='black')
plt.title("Histogram of Original Image")
plt.xlabel("Pixel Intensity")
plt.ylabel("Frequency")
plt.subplot(224)
plt.hist(i2.ravel(), bins=256, range=[0, 256], color='black')
plt.title("Histogram After Decreasing by -25")
plt.xlabel("Pixel Intensity")
plt.ylabel("Frequency")
plt.tight_layout()
plt.show()
```

OUTPUT:



CONTRAST STRETCHING:

ALGORITHM:

- **Read the Image in Grayscale**:
  - The image is loaded in grayscale using cv2.IMREAD_GRAYSCALE, where pixel values range from 0 (black) to 255 (white).
- **Find Minimum and Maximum Pixel Intensities**:
  - The minimum (I_min) and maximum (I_max) pixel values in the original image are found using np.min() and np.max() functions.
- **Apply Contrast Stretching**:
  - The contrast stretching formula is applied: stretched_image=(gray_image−Imin)×255Imax−Imin\text{stretched_image} = \frac{(\text{gray_image} - I_{\text{min}}) \times 255}{I_{\text{max}} - I_{\text{min}}}stretched_image=Imax−Imin(gray_image−Imin)×255
  - This operation rescales the pixel values, stretching the intensity range of the image so that the darkest pixel becomes black (0), and the brightest pixel becomes white (255).
  - The result is clipped to ensure pixel values remain within the valid range of 0-255, and it's converted to np.uint8 for proper image representation.
- **Display the Images**:
  - Both the **original image** and the **contrast-stretched image** are displayed side by side using matplotlib.
- **Plot Histograms**:

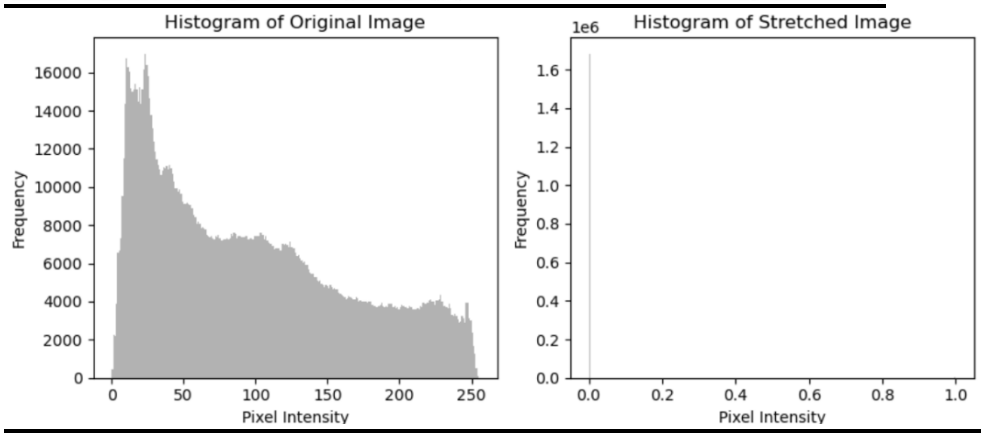
- Histograms are plotted for both the original image and the contrast-stretched image.
- The histograms show how the pixel intensity distribution has shifted due to the stretching. Typically, after contrast stretching, the pixel intensities will be more evenly distributed across the full range (0-255), making the image appear with higher contrast

CODE:

```
I_min = np.min(img_rgb)

I_max = np.max(img_rgb)

stretched_image = np.clip(((img_rgb - I_min) * 255) / (I_max - I_min), 0, 255).astype(np.uint8)

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

plt.subplot(1, 2, 1)

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

plt.title('Original Image')

plt.axis('off')

plt.subplot(1, 2, 2)

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

plt.title('Contrast Stretched Image')

plt.axis('off')

plt.tight_layout()

plt.show()

plt.figure(figsize=(9,4))

plt.subplot(1, 2, 1)

plt.hist(img_rgb.flatten(), bins=256, color='gray', alpha=0.6)

plt.title('Histogram of Original Image')

plt.xlabel('Pixel Intensity')

plt.ylabel('Frequency')

plt.subplot(1, 2, 2)
```

```
plt.hist(stretched_image.flatten(), bins=256, color='gray', alpha=0.6)
plt.title('Histogram of Stretched Image')
plt.xlabel('Pixel Intensity')
plt.ylabel('Frequency')
plt.tight_layout()
plt.show()
```

OUTPUT:



THRESHOLDING:

ALGORITHM:

- **Read the Image**:

- The image is loaded in grayscale using cv2.IMREAD_GRAYSCALE, where pixel values range from 0 (black) to 255 (white).

 Apply Thresholding:

- The cv2.threshold() function is used to apply binary thresholding. In this case:

python

Copy

val, t = cv2.threshold(image, 90, 255, cv2.THRESH_BINARY)

  - **Threshold value**: 90 — Pixels with values greater than 90 are set to 255 (white), and those less than or equal to 90 are set to 0 (black).
  - **Resulting image (t)**: This is the binary (thresholded) image where the pixel values are either 0 (black) or 255 (white).

 Display the Images:

- The original and thresholded images are displayed side by side using matplotlib:
  - **Original Image**: Displays the original grayscale image.
  - **Thresholded Image**: Displays the binary image after thresholding.

 Plot Histograms:

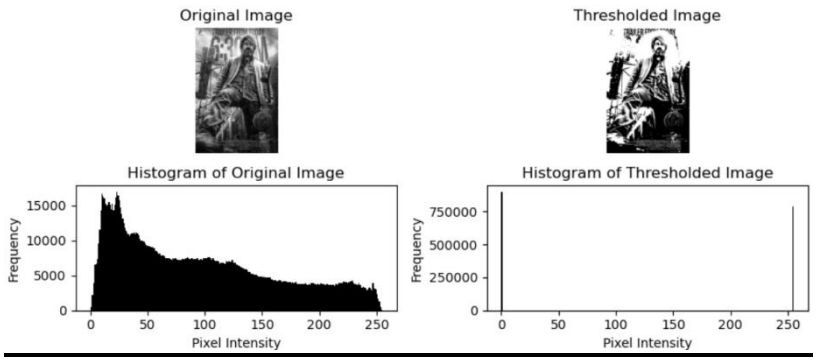
- Histograms for both the **original image** and the **thresholded image** are plotted:
  - The **original histogram** will show the distribution of pixel intensities, which could span the entire range from 0 to 255.
  - The **thresholded histogram** will show a binary distribution, with two distinct peaks at 0 (black) and 255 (white).

CODE:

```
val, t = c.threshold(img_rgb, 90, 255, c.THRESH_BINARY)

plt.figure(figsize=(9, 4))

plt.subplot(2, 2, 1)

plt.imshow(img_rgb, cmap="gray")

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

```
plt.axis("off")
plt.subplot(2, 2, 2)
plt.imshow(t, cmap="gray")
plt.title("Thresholded Image")
plt.axis("off")
plt.subplot(2, 2, 3)
plt.hist(img_rgb.ravel(), bins=256, color='black')
plt.title("Histogram of Original Image")
plt.xlabel("Pixel Intensity")
plt.ylabel("Frequency")
plt.subplot(2, 2, 4)
plt.hist(t.ravel(), bins=256, color='black')
plt.title("Histogram of Thresholded Image")
plt.xlabel("Pixel Intensity")
plt.ylabel("Frequency")
plt.tight_layout()
plt.show()
```

OUTPUT:



RESULT:

The discrete fourier transform , histogram processing, linear filtering on certain image is analysed.

EX NO:05 DATE:13/02/2025	IMAGE WITH NOISE, RESTORING THE IMAGE, EDGE DETECTION-SOBEL

AIM:

To deduct the noise in the image and restoring the original image and finally perform the edge detection-sobel.

ALGORITHM:

- **Read an image** from your computer using the cv2.imread() function.
- **Set up a window** to display the image with the desired size using plt.figure().
- **Show the image** using plt.imshow().
- **Remove axes** around the image to make it cleaner with plt.axis('off').
- **Display** the image with plt.show().

CODE:

```
import cv2 as c
import matplotlib.pyplot as plt
import numpy as np
img=c.imread("moon.jpg")
image=c.cvtColor(img,c.COLOR_BGR2GRAY)
plt.imshow(image,cmap="gray")
plt.axis("off")
image.shape
```

OUTPUT:

```
(266, 474)
```

GAUSSIAN NOISE

ALGORITHM:

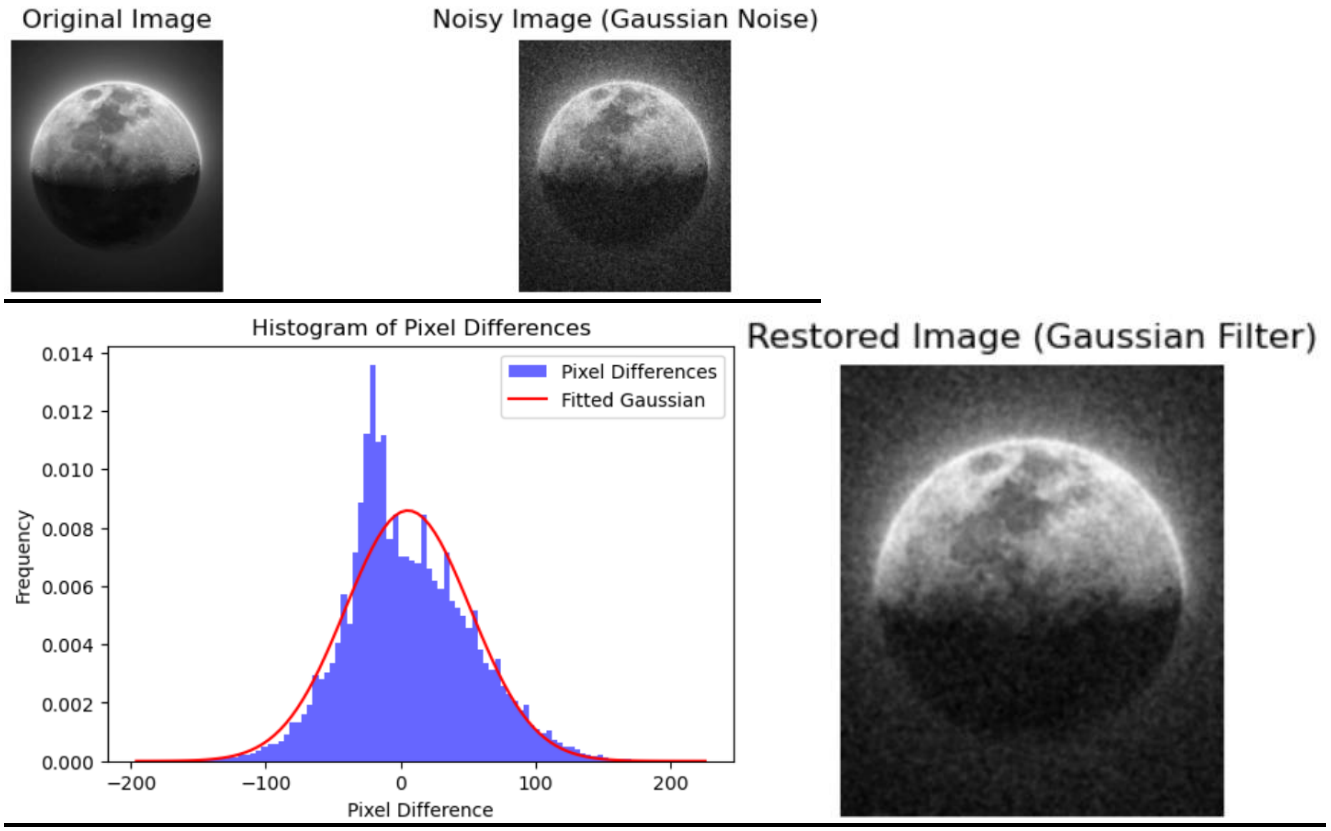
- **Read Image**: Load the image in grayscale.
- **Add Gaussian Noise**: Simulate random noise and add it to the image.
- **Display Images**: Show the original and noisy images side by side.
- **Plot Pixel Differences**: Show how different the noisy image is from the original.
- **Fit a Gaussian Curve**: Fit and display a Gaussian distribution for the pixel differences.
- **Restore Image**: Apply a Gaussian filter to remove some of the noise and display the result.

CODE:

```
gaussian=np.random.normal(0,55,image.shape)
noisy= image.astype(np.float32) + gaussian
noisy_image = np.clip(noisy, 0, 255).astype(np.uint8)
plt.figure(figsize=(6, 4))
plt.subplot(1,3, 1)
plt.imshow(image, cmap='gray')
plt.title("Original Image")
plt.axis("off")
plt.subplot(1, 3, 3)
```

```
plt.imshow(noisy_image, cmap='gray')
plt.title("Noisy Image (Gaussian Noise)")
plt.axis("off")
plt.show()
diff = noisy_image.astype(np.float32) - image.astype(np.float32)
plt.figure(figsize=(6, 4))
plt.hist(diff.ravel(), bins=100, density=True, color='blue', alpha=0.6,
label="Pixel Differences")
mean, std_dev = np.mean(diff), np.std(diff)
x = np.linspace(diff.min(), diff.max(), 100)
gaussian_curve = (1 / (std_dev * np.sqrt(2 * np.pi))) * np.exp(-((x - mean) ** 2)
/ (2 * std_dev ** 2))
plt.plot(x, gaussian_curve, color='red', label="Fitted Gaussian")
plt.xlabel("Pixel Difference")
plt.ylabel("Frequency")
plt.legend()
plt.title("Histogram of Pixel Differences")
plt.show()
plt.figure(figsize=(8, 5))
sigma = 2
from scipy.ndimage import gaussian_filter
restored_image = gaussian_filter(noisy_image, sigma=sigma)
plt.subplot(1, 3, 3)
plt.imshow(restored_image, cmap='gray')
plt.title("Restored Image (Gaussian Filter)")
plt.axis("off")
plt.show()
```

OUTPUT:



UNIFORM NOISE

ALGORITHM:

- **Add Noise**:
  - We start with an image and add **uniform noise** to it. This means we randomly change pixel values within a certain range to simulate noise.
- **Display Images**:
  - We show the **original image** and the **noisy image** side by side to compare how the noise affects the image.
- **Show Pixel Differences**:
  - We calculate the difference between the noisy and original image pixel by pixel, and then create a **histogram** to show how the noise is distributed.
- **Apply Filters**:
  - We use two **filters** to remove the noise:
    - **Min Filter**: Looks at each pixel's neighbors and keeps the smallest value. This helps remove bright spots (like noise).

- **Max Filter**: Looks at each pixel's neighbors and keeps the largest value. This helps remove dark spots.

 Display Filtered Images:

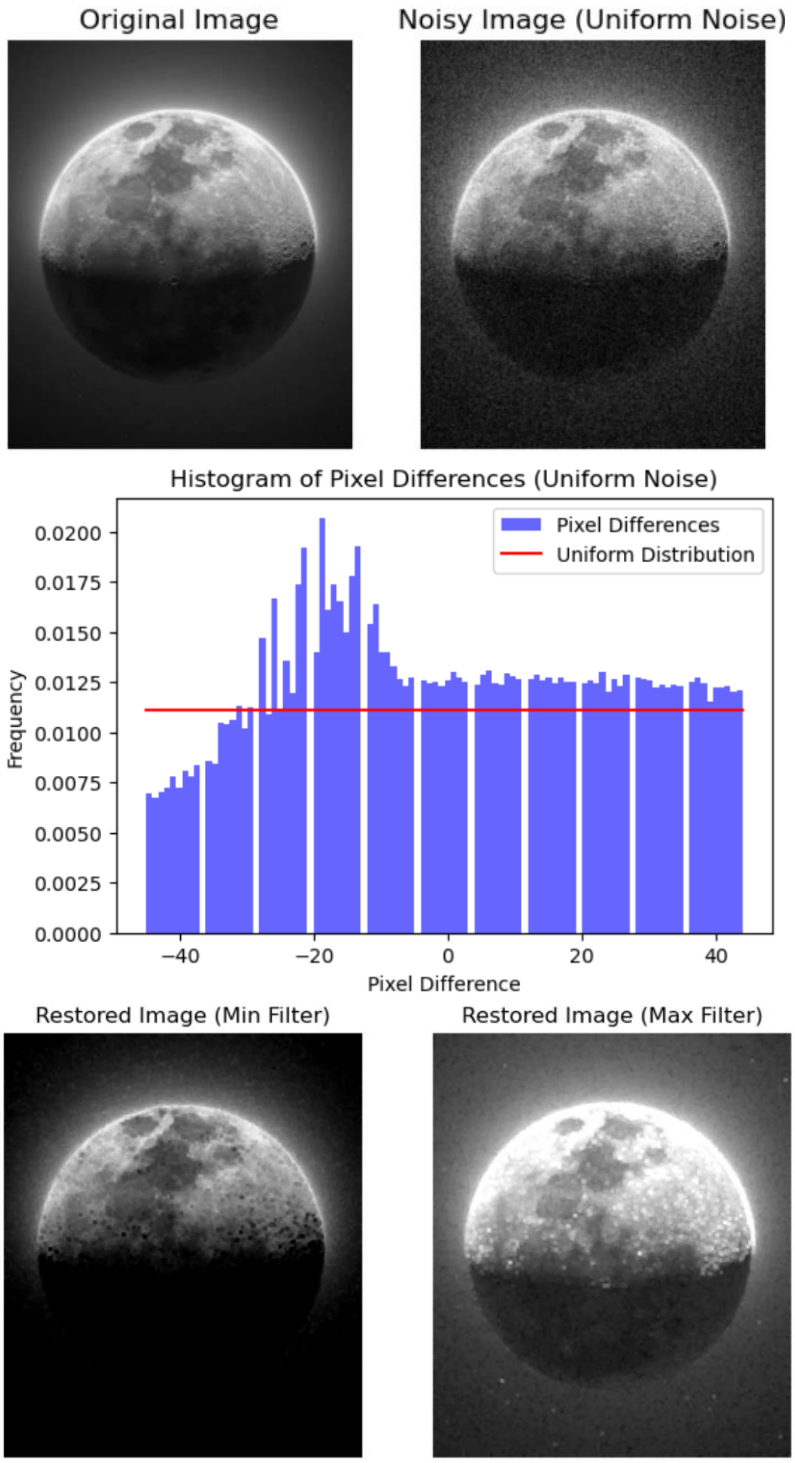
- After applying both filters, we show the **restored images** to see how much the noise was reduced.

CODE:

```
std_dev = 45
uniform_noise = np.random.uniform(-std_dev, std_dev, image.shape)
noisy_image = image.astype(np.float32) + uniform_noise
noisy_image = np.clip(noisy_image, 0, 255).astype(np.uint8)
plt.figure(figsize=(6, 4))
plt.subplot(1, 2, 1)
plt.imshow(image, cmap='gray')
plt.title("Original Image")
plt.axis("off")
plt.subplot(1, 2, 2)
plt.imshow(noisy_image, cmap='gray')
plt.title("Noisy Image (Uniform Noise)")
plt.axis("off")
plt.show()
diff = noisy_image.astype(np.float32) - image.astype(np.float32)
plt.figure(figsize=(6, 4))
plt.hist(diff.ravel(), bins=100, density=True, color='blue', alpha=0.6,
label="Pixel Differences")
x = np.linspace(diff.min(), diff.max(), 100)
uniform_curve = np.ones_like(x) / (2 * std_dev)
plt.plot(x, uniform_curve, color='red', label="Uniform Distribution")
```

```
plt.xlabel("Pixel Difference")
plt.ylabel("Frequency")
plt.legend()
plt.title("Histogram of Pixel Differences (Uniform Noise)")
plt.show()
size = 5
min_filter_image = c.erode(noisy_image, np.ones((size,
size), np.uint8))
max_filter_image = c.dilate(noisy_image, np.ones((size,
size), np.uint8))
plt.figure(figsize=(12, 6))
plt.subplot(1, 3, 1)
plt.imshow(min_filter_image, cmap='gray')
plt.title("Restored Image (Min Filter)")
plt.axis("off")
plt.subplot(1, 3, 2)
plt.imshow(max_filter_image, cmap='gray')
plt.title("Restored Image (Max Filter)")
plt.axis("off")
```

OUTPUT:



SALT AND PEPPER NOISE

ALGORITHM:

- **Adding Noise**:
  - We add salt-and-pepper noise by randomly setting some pixels to white (255) and some to black (0).

 Displaying Images:

- We show the **original** image and the **noisy** image to see the effect of the added noise.

 Histogram:

- The histogram of the noisy image shows how many pixels have each intensity value (brightness). This helps you understand the spread of salt-and-pepper noise in the image.

 Median Filter:

- The median filter helps remove the salt-and-pepper noise by looking at each pixel and replacing it with the median value of its neighboring pixels. This is good for removing small isolated white and black spots (salt and pepper).

 Displaying Restored Image:

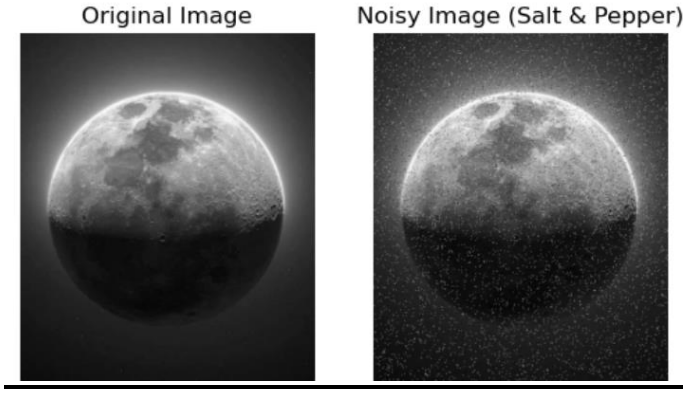
- Finally, the **restored image** after applying the median filter is displayed, showing how the noise is reduced.

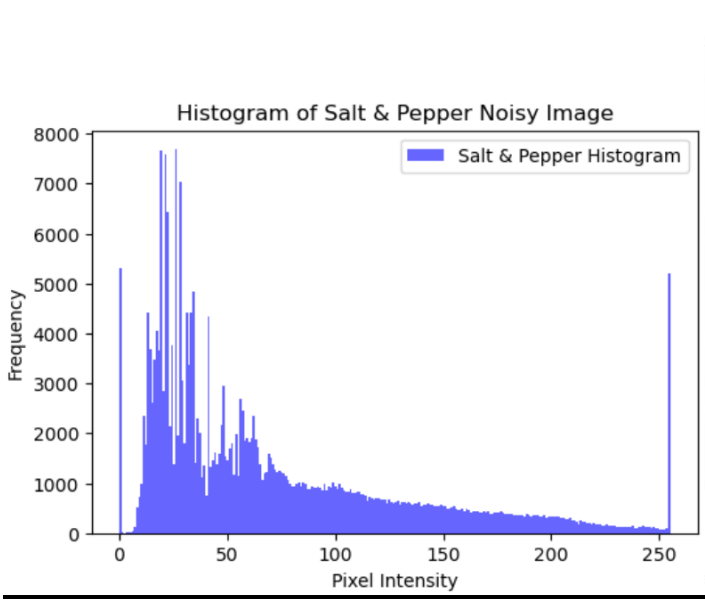
CODE:

```
salt_prob = 0.02

pepper_prob = 0.02

noisy = np.copy(image)

salt_mask = np.random.rand(*image.shape) < salt_prob

noisy[salt_mask] = 255

pepper_mask = np.random.rand(*image.shape) < pepper_prob

noisy[pepper_mask] = 0

plt.figure(figsize=(6, 4))

plt.subplot(1, 2, 1)

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

plt.title("Original Image")

plt.axis("off")

plt.subplot(1, 2, 2)

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

```
plt.title("Noisy Image (Salt & Pepper)")
plt.axis("off")
plt.show()
plt.figure(figsize=(6, 4))
plt.hist(noisy_image.ravel(), bins=256, range=(0, 255), color='blue', alpha=0.6,
label="Salt & Pepper Histogram")
plt.xlabel("Pixel Intensity")
plt.ylabel("Frequency")
plt.legend()
plt.title("Histogram of Salt & Pepper Noisy Image")
plt.show()
kernel_size = 5
median_filtered_image = c.medianBlur(noisy_image, kernel_size)
plt.figure(figsize=(6, 4))
plt.imshow(median_filtered_image, cmap='gray')
plt.title("Restored Image (Median Filter)")
plt.axis("off")
plt.show()
```

OUTPUT:





EDGE DETECTION-SOBERT

WITH INBUILT FUNCTION:

ALGORITHM:

- **Sobel Operator**:
  - The **Sobel filter** is a simple edge detection filter that highlights regions where the intensity of the image changes significantly.
  - **sobel_x** detects edges in the **horizontal** direction (left to right).
  - **sobel_y** detects edges in the **vertical** direction (top to bottom).
- **Gradient Magnitude**:
  - The **magnitude** of the gradient is calculated by combining the results from **sobel_x** and **sobel_y**. This gives a stronger indication of the edges' presence regardless of the direction of change.
  - sobel_magnitude = np.sqrt(sobel_x**2 + sobel_y**2) combines the two gradients to compute the overall edge strength.
- **Normalize Values**:
  - The gradient values are then clipped to the range [0, 255] and converted to uint8 (standard image format) to make sure the output image can be properly visualized.
- **Visualization**:
  - First, it displays the **original grayscale image**.

- Then it displays the **Sobel edge-detected image** to show the result of edge detection.

CODE:

```
sobel_x = c.Sobel(image, c.CV_64F, 1, 0, ksize=3)

sobel_y = c.Sobel(image, c.CV_64F, 0, 1, ksize=3)

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

sobel_magnitude = np.clip(sobel_magnitude, 0, 255).astype(np.uint8)

plt.figure(figsize=(10, 4))

plt.subplot(1, 3, 1)

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

plt.title("Original Image")

plt.axis("off")

plt.subplot(1,3,3)

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

plt.title("Sobel Edge Detection")

plt.axis("off")

plt.show()
```

OUTPUT:

WITHOUT INBUILT FUNCTION

ALGORITHM:

CODE:

```
sobel_x= np.array([[-1, 0, 1],
                   [-2, 0, 2],
                   [-1, 0, 1]], dtype=np.float32)
sobel_y= np.array([[-1, -2, -1],
                   [ 0,  0,  0],
                   [ 1,  2,  1]], dtype=np.float32)
height, width = image.shape
x = np.zeros_like(image, dtype=np.float32)
y = np.zeros_like(image, dtype=np.float32)
for i in range(1, height - 1):
    for j in range(1, width - 1):

        region = image[i-1:i+2, j-1:j+2]
        gx = np.sum(region * sobel_x)
        gy = np.sum(region * sobel_y)



        x[i, j] = gx
        y[i, j] = gy
sobel_magnitude = np.sqrt(x**2 + y**2)
sobel_magnitude = np.clip(sobel_magnitude, 0, 255).astype(np.uint8)
plt.figure(figsize=(10, 4))
plt.subplot(1, 3, 1)
plt.imshow(image, cmap='gray')
plt.title("Original Image")
plt.axis("off")
plt.subplot(1,3,3)
```

```
plt.imshow(sobel_magnitude, cmap='gray')
plt.title("Sobel Edge Detection(Manual)")
plt.axis("off")
plt.show()
```

OUTPUT:

RESULT:

The restoration, noise and edge detection is analysed.

EX NO:06 DATE:13/02/2025	FILTER-ROBERT, PREWITT , CANNY EDGE DETECTION , LOG AND DOG

AIM:

To work with filters in edge detection , canny edge detection , LOG and DOG.

READ THE IMAGE:

ALGORITHM:

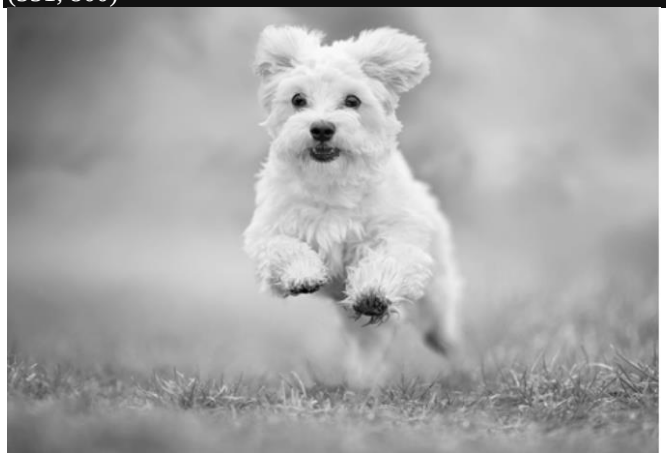
- **Import required libraries** (cv2, matplotlib.pyplot, numpy).
- **Read the image** from the given file path using cv2.imread().
- **Display the image** using plt.imshow().
- **Show the image** using plt.show()

CODE:

```
import cv2 as c
import matplotlib.pyplot as plt
import numpy as np
img=c.imread("dog.png",c.IMREAD_GRAYSCALE)
plt.imshow(img,cmap="gray")
plt.axis("off")
plt.show()
img.shape
```

OUTPUT:

```
(551, 800)
```



ROBERT CROSS DETECTION:

ALGORITHM:

1. **Import necessary libraries** (cv2, numpy, matplotlib.pyplot).
2. **Read the image in grayscale** using cv2.imread().
3. **Define Roberts Cross operator kernels** (roberts_x and roberts_y).
4. **Apply convolution** using cv2.filter2D() to get edge responses (edges_x and edges_y).
5. **Compute edge magnitude** using np.sqrt(edges_x**2 + edges_y**2).
6. **Convert the result to an 8-bit image** using np.uint8().
7. **Display the original and edge-detected images** using plt.subplot() and plt.imshow().
8. **Show the final output** using plt.show()

PREWITT FILTER:

ALGORITHM:

- **Import necessary libraries** (cv2, numpy, matplotlib.pyplot).
- **Read the image in grayscale** using cv2.imread().
- **Define Prewitt operator kernels** (prewitt_x and prewitt_y).
- **Apply convolution** using cv2.filter2D() to detect edges in the X and Y directions.
- **Compute the combined edge magnitude** using np.sqrt(edges_x**2 + edges_y**2).
- **Display the results** using plt.subplot() for X, Y, and combined edges.
- **Show the final output** using plt.show().

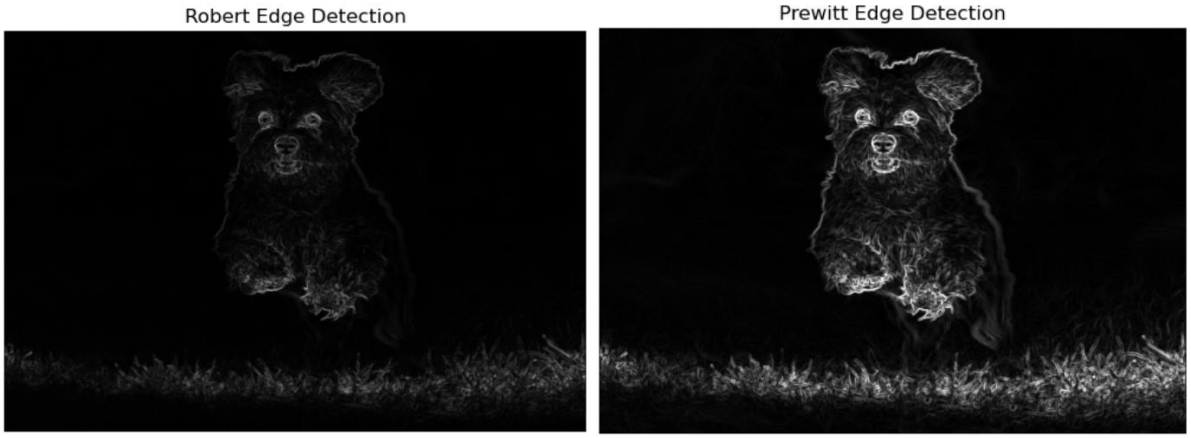
CODE:

```
blur=c.GaussianBlur(img,(5,5),0)
def edge_det(image,a):
    if a=='r':
        mat_x=np.array([[1,0],[0,-1]])
        mat_y=np.array([[0,-1],[1,0]])
    elif a=='s':
```

```
        mat_x=np.array([[-1,0,1],[-2,0,2],[-1,0,1]])
        mat_y=mat_x.T
    elif a=='p':
        mat_x=np.array([[-1,0,1],[-1,0,1],[-1,0,1]])
        mat_y=np.array([[-1,-1,-1],[0,0,0],[1,1,1]])
    h,w=image.shape
    g_x=np.zeros_like(image,dtype=np.float32)
    g_y=np.zeros_like(image,dtype=np.float32)
    for i in range(1,h-1):
        for j in range(1,w-1):
            if a == 'r':
                region=img[i-1:i+1,j-1:j+1]
            else:
                region=img[i-1:i+2,j-1:j+2]
            gx=np.sum(region*mat_x)
            gy=np.sum(region*mat_y)
            g_x[i,j]=gx
            g_y[i,j]=gy
        sob_mag=np.sqrt(g_x**2 + g_y**2)
        sob_mag=np.clip(sob_mag,0,255).astype(np.uint8)
    return sob_mag,np.arctan(g_x,g_y)
robert,angle=edge_det(img,'r')
plt.imshow(robert,cmap='gray')
plt.axis("off")
plt.title("Robert Edge Detection")
per,angle=edge_det(img,'p')
plt.imshow(per,cmap='gray')
```

```
plt.axis("off")
plt.title("Prewitt Edge Detection")
```

OUTPUT:



CANNY EDGE DETECTION:

ALGORITHM:

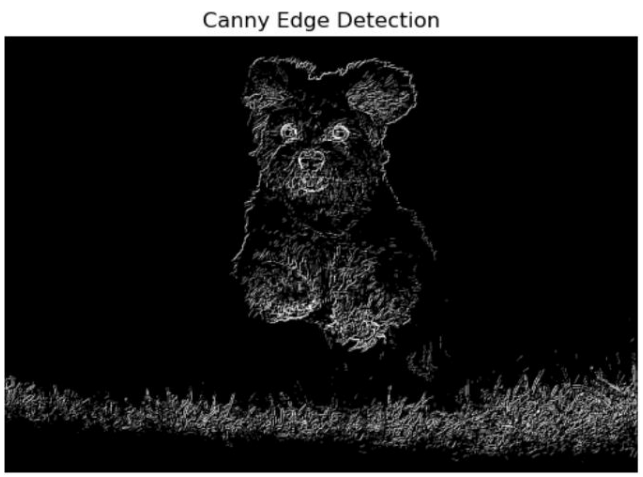
- **Import necessary libraries** (cv2, numpy, matplotlib.pyplot).
- **Read the image in grayscale** using cv2.imread().
- **Apply Gaussian Blur** to reduce noise and smooth the image.
- **Compute gradients using Sobel operators** (cv2.Sobel()) in the X and Y directions.
- **Calculate gradient magnitude and direction** using np.sqrt() and np.arctan2().
- **Apply Non-Maximum Suppression (NMS)** using cv2.Canny().
- **Perform Double Thresholding** to classify strong and weak edges.
- **Display results** at each step using plt.subplot() and plt.imshow().
- **Show final Canny edge-detected image** using plt.show().

CODE:

```
edge,angle=edge_det(blur,'s')
h,w=edge.shape
angle=angle*180/np.pi
angle=np.mod(angle,180)
output = np.zeros_like(edge, dtype=np.float32)
```

```
for i in range(1,edge.shape[0]-1):
    for j in range(1,edge.shape[1]-1):
        an=angle[i,j]
        if (an >= 0 and an < 22.5) or (an >= 157.5 and an < 180):
                neighbor1 =edge[i, j + 1]
                neighbor2 = edge[i, j - 1]
        elif (an >= 22.5 and an < 67.5):
                neighbor1 = edge[i + 1, j]
                neighbor2 = edge[i - 1, j]
        elif (an >= 67.5 and an < 112.5):
                neighbor1 = edge[i + 1, j + 1]
                neighbor2 = edge[i - 1, j - 1]
        else:
                neighbor1 = edge[i + 1, j - 1]
                neighbor2 = edge[i - 1, j + 1]
        if edge[i,j]>=neighbor1 and edge[i,j]>=neighbor2:
            output[i,j]=edge[i,j]
        else:
            output[i,j]=0
```

OUTPUT:



LOG:

ALGORITHM:

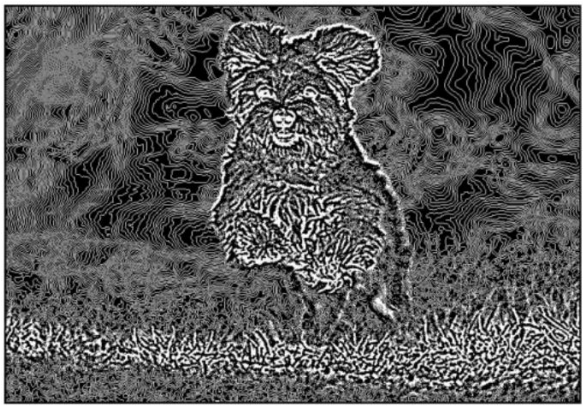
1. **Import necessary libraries** (cv2, numpy, matplotlib.pyplot).
2. **Read the image in grayscale** using cv2.imread().
3. **Apply Gaussian Blur** to smooth the image and reduce noise.
4. **Define the Laplacian kernel** for edge detection.
5. **Apply convolution** using cv2.filter2D() to detect edges.
6. **Convert edge values to absolute and normalize** for better visualization.
7. **Display the original and edge-detected images** using plt.subplot() and plt.imshow().
8. **Show the final results** using plt.show().

CODE:

```
def gaussian_kernel(size, sigma):
    kernel = np.fromfunction(
        lambda x, y: (1 / (2 * np.pi * sigma**2)) * np.exp(-((x - (size - 1) / 2)**2 + (y - (size - 1) / 2)**2) / (2 * sigma**2)),
        (size, size)
    )
    return kernel / np.sum(kernel)
def apply_convolution(image, kernel):
    h, w = image.shape
    kernel_size = kernel.shape[0]
    pad = kernel_size // 2
    padded_image = np.pad(image, pad, mode='constant', constant_values=0)
    result = np.zeros_like(image)
    for i in range(h):
        for j in range(w):
            region = padded_image[i:i+kernel_size, j:j+kernel_size]
```

```
            result[i, j] = np.sum(region * kernel)
    return result
def LoG(image, sigma=1.0, kernel_size=5):
    gaussian_k = gaussian_kernel(kernel_size, sigma)
    smoothed_image = apply_convolution(image, gaussian_k)
    laplacian_kernel = np.array([[0,-1,0],[-1,4,-1],[0,-1,0]], dtype=np.float32)
    log_image = apply_convolution(smoothed_image, laplacian_kernel)
    return log_image
log=LoG(blur)
plt.imshow(log,cmap='gray')
plt.axis('off')
```

OUTPUT:



DOG:

ALGORITHM:

CODE:

```
def DoG(image, sigma1=1.0, sigma2=2.0):
    blurred1 = c.GaussianBlur(image, (0, 0), sigma1)
    blurred2 = c.GaussianBlur(image, (0, 0), sigma2)
    dog_image = blurred1 - blurred2
    return dog_image
dog=DoG(blur,2,1)
```

```
plt.imshow(dog,cmap='gray')
plt.axis('off')
```

OUTPUT:

RESULT:

The filters like Robert, prewitt, canny edge detection, LOG and DOG is analysed.

EXERCISE NO:07	CANNY EDGE DETECTION USING PREWITT AND ROBERT OPERATOR
DATE:27/02/2025	

AIM:

To perform the canny edge detection using Robert and prewitt operator.

READ THE IMAGE:

ALGORITHM:

1. **Start**
2. **Import necessary libraries**
    - Import cv2 (OpenCV) for image handling.
    - Import matplotlib.pyplot for displaying the image.
3. **Read the image**
    - Use cv2.imread("image_path") to load the image from the specified file path.
4. **Display the image using Matplotlib**
    - Use plt.imshow(image) to display the image.
    - Use plt.show() to render the output.
5. **End**

CODE:

```
import cv2 as c
import matplotlib.pyplot as plt
import numpy as np
image=c.imread("dog.png")
img=c.cvtColor(image,c.COLOR_RGB2GRAY)
plt.imshow(img,cmap="gray")
plt.axis("off")
plt.show()
img.shape
```

OUTPUT:

PREWITT OPERATOR:

ALGORITHM:

- **Start**
- **Load the grayscale image** using OpenCV (cv2.imread).
- **Define Prewitt kernels** for detecting edges in the X and Y directions.
- **Apply the Prewitt operator** using cv2.filter2D to compute:
  - Horizontal gradient (grad_x)
  - Vertical gradient (grad_y)
- **Compute the edge magnitude** using the formula:
edges=(grad_x)2+(grad_y)2\text{edges} = \sqrt{(\text{grad_x})^2 + (\text{grad_y})^2}edges=(grad_x)2+(grad_y)2 Convert to an 8-bit image for display.
- **Apply Canny edge detection** using cv2.Canny().
- **Display the original, Prewitt, and Canny edge images** using matplotlib.pyplot.
- **End**

CODE:

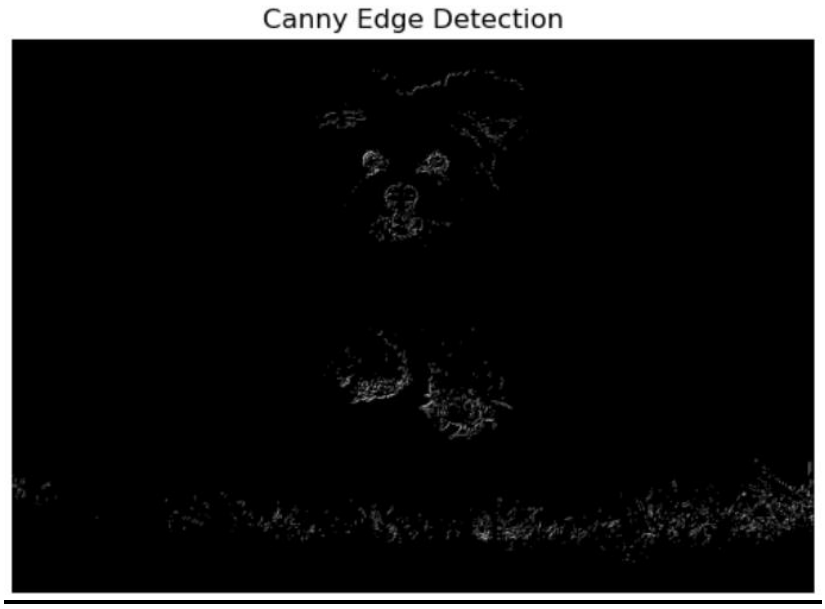
```
blur=c.GaussianBlur(img,(5,5),0)

plt.imshow(blur,cmap="gray")
```

```
plt.axis("off")
def edge_det(image,a):
    if a=='r':
        mat_x=np.array([[1,0],[0,-1]])
        mat_y=np.array([[0,-1],[1,0]])
    elif a=='s':
        mat_x=np.array([[-1,0,1],[-2,0,2],[-1,0,1]])
        mat_y=mat_x.T
    elif a=='p':
        mat_x=np.array([[-1,0,1],[-1,0,1],[-1,0,1]])
        mat_y=np.array([[-1,-1,-1],[0,0,0],[1,1,1]])
    h,w=image.shape
    g_x=np.zeros_like(image,dtype=np.float32)
    g_y=np.zeros_like(image,dtype=np.float32)
    for i in range(1,h-1):
        for j in range(1,w-1):
            if a == 'r':
                region=img[i-1:i+1,j-1:j+1]
            else:
                region=img[i-1:i+2,j-1:j+2]
            gx=np.sum(region*mat_x)
            gy=np.sum(region*mat_y)
            g_x[i,j]=gx
            g_y[i,j]=gy
        sob_mag=np.sqrt(g_x**2 + g_y**2)
        sob_mag=np.clip(sob_mag,0,255).astype(np.uint8)
    return sob_mag,np.arctan(g_x,g_y)
```

```
edge,angle=edge_det(blur,'r')
h,w=edge.shape
angle=angle*180/np.pi
angle=np.mod(angle,180)
output = np.zeros_like(edge, dtype=np.float32)
for i in range(1,edge.shape[0]-1):
    for j in range(1,edge.shape[1]-1):
        an=angle[i,j]
        if (an >= 0 and an < 22.5) or (an >= 157.5 and an < 180):
                neighbor1 =edge[i, j + 1]
                neighbor2 = edge[i, j - 1]

        elif (an >= 22.5 and an < 67.5):
                neighbor1 = edge[i + 1, j]
                neighbor2 = edge[i - 1, j]
        elif (an >= 67.5 and an < 112.5):
                neighbor1 = edge[i + 1, j + 1]
                neighbor2 = edge[i - 1, j - 1]
        else:
                neighbor1 = edge[i + 1, j - 1]
                neighbor2 = edge[i - 1, j + 1]
        if edge[i,j]>=neighbor1 and edge[i,j]>=neighbor2:
            output[i,j]=edge[i,j]
        else:
            output[i,j]=0
def double_thres(image,lT,hT):
    result=np.zeros_like(image,dtype=np.uint8)
```

```
    result[image>hT]=255
    result[(image>=lT) & (image<hT)]=128

    return result
canny=double_thres(output,50,100)
plt.imshow(canny,cmap='gray')
plt.axis("off")
plt.title("Canny Edge Detection")
```

OUTPUT:



ROBERT :

ALGORITHM:

- **Start**
- **Load the grayscale image** using OpenCV (cv2.imread).
- **Define the Roberts Cross operator kernels** for detecting edges in the X and Y directions.
- **Apply the Roberts operator** using cv2.filter2D() to compute:

- Horizontal gradient (gradient_x).
- Vertical gradient (gradient_y).

 Compute the edge magnitude using the formula:
edges=(gradient_x)2+(gradient_y)2\text{edges} = \sqrt{(\text{gradient_x})^2 + (\text{gradient_y})^2}edges=(gradient_x)2+(gradient_y)2 Convert to an 8-bit image.

 Normalize the gradient magnitude using cv2.normalize() for better visualization.

 Apply Canny edge detection using cv2.Canny().

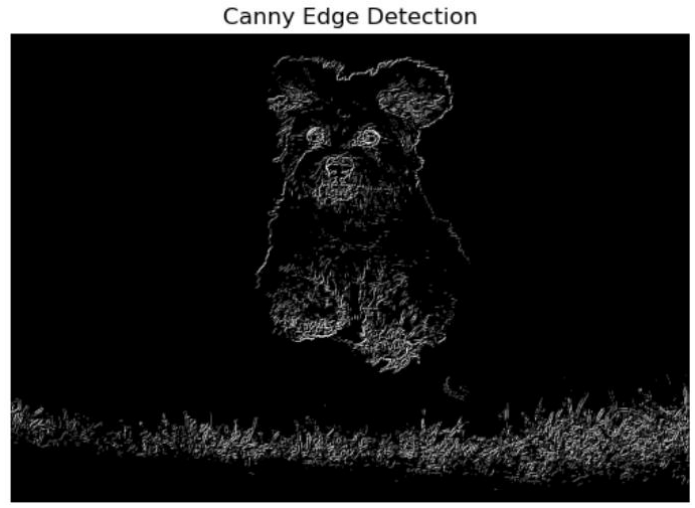
 Display the original, Roberts, and Canny edge images using matplotlib.pyplot.

 End

CODE:

```
edge,angle=edge_det(blur,'p')

canny=double_thres(output,50,100)

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

plt.axis("off")

plt.title("Canny Edge Detection")
```

OUTPUT:



RESULT:

The Robert and prewitt operator in canny edge detection is performed.

EXERCISE NO:8	HARRIS CORNER AND HOUGH TRANSFORM LINE DETECTION
DATE:7/03/25	

AIM:

To perform the Harris corner detection and hough transform line detection.

Read the image:

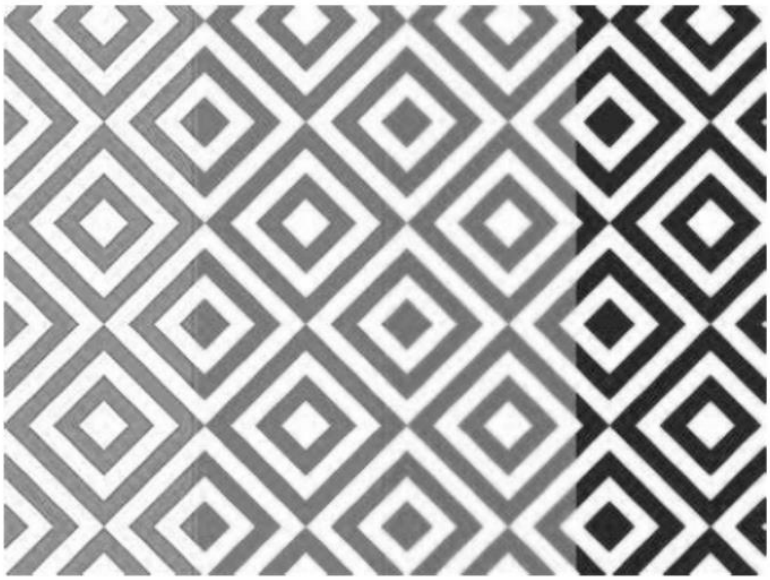
ALGORITHM:

- **Import Required Libraries**
  - Import cv2 for image processing.
  - Import matplotlib.pyplot for displaying the image.
- **Read the Image**
  - Use cv2.imread() to load the image from the given file path.
- **Display the Image**
  - Use plt.imshow() to visualize the image.
  - Use plt.show() to display the image in a window.

CODE:

```
import cv2
import matplotlib.pyplot as plt
import numpy as np
image=cv2.imread('OIP.jpg')
image=cv2.cvtColor(image,cv2.COLOR_BGR2RGB)
gray=cv2.cvtColor(image,cv2.COLOR_BGR2GRAY)
plt.imshow(gray,cmap='gray')
plt.axis('off')
```

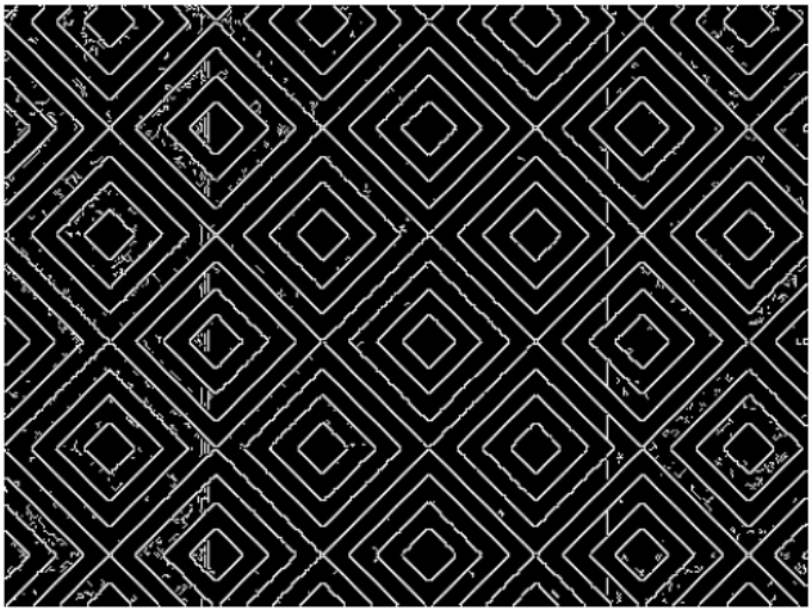
OUTPUT:



CODE:

```
edges=cv2.Canny(gray,50,100,apertureSize=3)
plt.imshow(edges,cmap='gray')
plt.axis('off')
```

OUTPUT:



HOUGH LINE TRANSFORM:

ALGORITHM:

1. **Import Required Libraries**

- Import numpy, cv2, and matplotlib.pyplot.

2. **Read the Image**
   - Use cv2.imread() to load the image in **color mode** (cv2.IMREAD_COLOR).
3. **Convert the Image to Grayscale**
   - Use cv2.cvtColor() with cv2.COLOR_BGR2GRAY to convert the image to grayscale.
4. **Apply Canny Edge Detection**
   - Use cv2.Canny() with threshold values **50** and **200** to detect edges.
5. **Apply Hough Line Transform (Probabilistic Version)**
   - Use cv2.HoughLinesP() with the following parameters:
     - Distance resolution = **1 pixel**
     - Angle resolution = **π/180 radians**
     - Threshold = **80** (minimum number of intersections to consider a line)
     - minLineLength = 15 (minimum line segment length)
     - maxLineGap = 250 (maximum gap between line segments to be considered a single line)
6. **Draw the Detected Lines on the Original Image**
   - Copy the original image to img_with_lines.
   - Iterate through the detected lines and draw them using cv2.line() with **red color (255, 0, 0)** and a thickness of **3 pixels**.
7. **Display the Original and Processed Images**
   - Create a figure using plt.figure(figsize=(12, 6)).
   - Use plt.subplot(1, 2, 1) to display the **original image** (converted from BGR to RGB).
   - Use plt.subplot(1, 2, 2) to display the **image with detected lines**.
   - Use plt.tight_layout() to optimize the layout.
   - Use plt.show() to display the images.

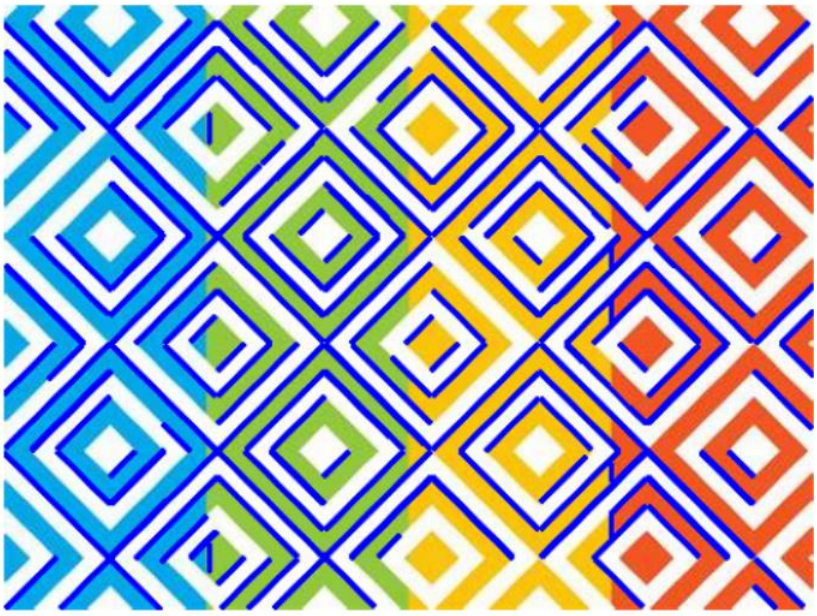
CODE:

```
lines=cv2.HoughLinesP(edges,1,np.pi/180,100,50,10)
lines.shape
```

```
(201, 1, 4)
```

```
if lines is not None:
    for line in lines:
        x1,y1,x2,y2=line[0]
        cv2.line(image,(x1,y1),(x2,y2),(0,0,255),2)
plt.imshow(image,cmap='gray')
plt.axis('off')
```

OUTPUT:



HARRIS CORNER:

ALGORITHM:

1. **Import Required Libraries**
    - cv2 for image processing.
    - numpy for numerical operations.
    - matplotlib.pyplot for displaying images.

2. **Load the Image in Grayscale**
    - Read the image using cv2.imread() with cv2.IMREAD_GRAYSCALE to convert it to grayscale.
3. **Apply Gaussian Blur**
    - Use cv2.GaussianBlur() with a kernel size of **5×5** to smooth the image and reduce noise.
4. **Apply Harris Corner Detection**
    - Use cv2.cornerHarris() with parameters:
        - Block size = **2** (size of the neighborhood).
        - Aperture size = **3** (Sobel kernel size).
        - Harris detector free parameter = **0.04**.
5. **Enhance the Detected Corners**
    - Use cv2.dilate() to expand the detected corner regions for better visualization.
6. **Convert Grayscale Image to Color**
    - Use cv2.cvtColor() to convert the grayscale image to **BGR** for better visualization.
7. **Mark the Corners in Red**
    - Identify the strong corner responses (dst > 0.01 * dst.max()).
    - Set the corresponding pixels in the color image to **Red (255, 0, 0)**.
8. **Display the Results Using Matplotlib**
    - Create a figure with **3 subplots**:
        - **Original grayscale image**.
        - **Harris response heatmap** (hot colormap).
        - **Image with detected corners** (converted to RGB for correct display in Matplotlib).
9. **Show the Plots**
    - Use plt.show() to display the results.

CODE:

```
dst=cv2.cornerHarris(gray,2,3,0.04)
dst=cv2.dilate(dst,None)
image[dst>0.01*dst.max()]=(255,0,0)
plt.imshow(image,cmap='gray')
plt.axis('off')
```

OUTPUT:

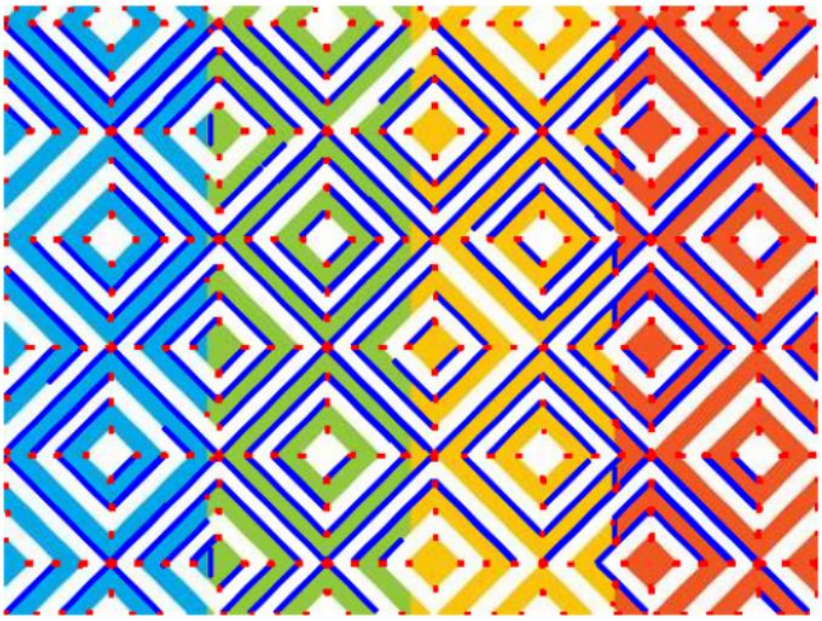
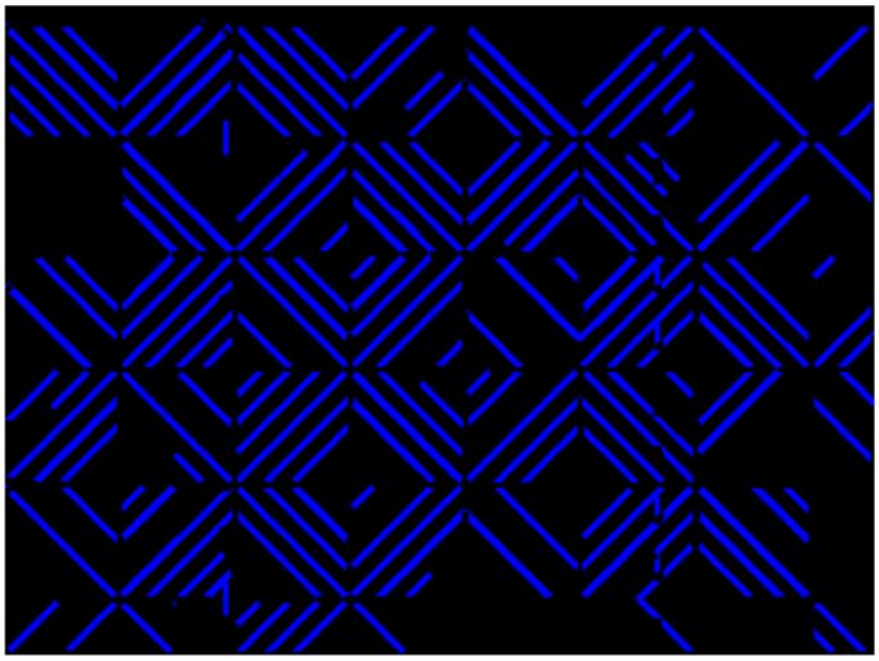


IMAGE SEGMENTATION:

CODE:

```
segment=np.zeros_like(image)
if lines is not None:
    for line in lines:
        x1,y1,x2,y2=line[0]

        cv2.line(segment,(x1,y1),(x2,y2),(0,0,255),2)
segmented = cv2.bitwise_and(image,segment)
plt.imshow(segmented)
plt.axis('off')
```

OUTPUT:



RESULT:

The harris corner and hough transform is analysed.

EXERCISE NO:9	IMAGE CLASSIFICATION AND OBJECT DETECTION
DATE:14/03/25	

AIM:

to perform the image classification and object detection using SIFT

IMAGE CLASSIFICATION:

ALGORITHM:

Step 1: Import Required Libraries

- Import necessary libraries such as cv2, numpy, os, matplotlib.pyplot, and skimage.feature.hog.
- Import machine learning tools from sklearn.

Step 2: Load Dataset and Define Classes

- Set the dataset path.
- Retrieve class names from the dataset folder using os.listdir().

Step 3: Define Feature Extraction Functions

1. **SIFT Feature Extraction**
    - Use cv2.SIFT_create() to create a SIFT object.
    - Detect keypoints and compute descriptors.
    - If descriptors are None, return a zero vector.
    - Otherwise, compute the mean of descriptors to get a fixed-length feature vector.
2. **HOG Feature Extraction**
    - Use skimage.feature.hog() with specified parameters.
    - Return the extracted HOG feature vector.
3. **GLOH Feature Extraction**
    - Similar to SIFT, but return the **variance** of descriptors instead of the mean.

Step 4: Prepare Feature Vectors and Labels

- Initialize empty lists X (features) and y (labels).
- Loop through each class in the dataset:
    - For each image in the class folder:
        - Read and convert the image to grayscale.
        - Resize it to **128×128 pixels**.
        - Extract **SIFT, HOG, and GLOH** features.
        - Stack the features together using np.hstack().
        - Append the feature vector to X and the corresponding label to y.

Step 5: Preprocess Data

- Convert X and y into numpy arrays.
- Apply **Standard Scaling** (StandardScaler()) to normalize feature values.
- Split the dataset into **80% training** and **20% testing** using train_test_split().

Step 6: Train the Random Forest Model

- Initialize a **Random Forest Classifier** (RandomForestClassifier).
- Train the classifier using the training set (clf.fit(X_train, y_train)).

Step 7: Evaluate the Model

- Predict labels for the test set.
- Compute **accuracy** using accuracy_score().
- Print the model's accuracy.

Step 8: Define Image Classification Function

- Define classify_image(image_path) to predict a new image's class:

- Read and process the input image (convert to grayscale and resize).
- Extract **SIFT, HOG, and GLOH** features.
- Standardize the features.
- Predict the class using the trained classifier.
- Print the predicted class.

Step 9: Load and Classify a Test Image

- Read the test image (lamp.jpeg).
- Convert it from BGR to RGB.
- Display the image using matplotlib.
- Call classify_image() to predict its class.

CODE:

```
import cv2
import numpy as np
import os
from skimage.feature import hog
from sklearn.ensemble import RandomForestClassifier as rf
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score
import matplotlib.pyplot as plt
dataset_path = r"C:\Users\madhu\Downloads\madhu dataset - Copy"
classes = os.listdir(dataset_path)  # Get class names from dataset folder
def extract_sift_features(image):
    sift = cv2.SIFT_create()
    keypoints, descriptors = sift.detectAndCompute(image, None)
```

```
    if descriptors is None:
        return np.zeros((128,))
    return np.mean(descriptors, axis=0)  # Take mean for fixed-length feature
def extract_hog_features(image):
    features, _ = hog(image, pixels_per_cell=(8, 8), cells_per_block=(2, 2),
                      orientations=9, visualize=True)
    return features
def extract_gloh_features(image):
    sift = cv2.SIFT_create()
    keypoints, descriptors = sift.detectAndCompute(image, None)
    if descriptors is None:
        return np.zeros((128,))
    return np.var(descriptors, axis=0)
X, y = [], []
for label, flower in enumerate(classes):
    flower_path = os.path.join(dataset_path, flower)
    for img_name in os.listdir(flower_path):
        img_path = os.path.join(flower_path, img_name)
        image = cv2.imread(img_path, cv2.IMREAD_GRAYSCALE)
        image = cv2.resize(image, (128, 128))

        sift_feat = extract_sift_features(image)
        hog_feat = extract_hog_features(image)
        gloh_feat = extract_gloh_features(image)

        features = np.hstack((sift_feat, hog_feat, gloh_feat))
        X.append(features)
```

```
        y.append(label)

X = np.array(X)
y = np.array(y)
scaler = StandardScaler()
X = scaler.fit_transform(X)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
clf = rf(n_estimators=100, random_state=42)
clf.fit(X_train, y_train)
y_pred = clf.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
print(f"Model Training Complete. Accuracy: {accuracy:.4f}")
def classify_image(image_path):
    image = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)
    image = cv2.resize(image, (128, 128))
     sift_feat = extract_sift_features(image)
    hog_feat = extract_hog_features(image)
    gloh_feat = extract_gloh_features(image)

    features = np.hstack((sift_feat, hog_feat, gloh_feat)).reshape(1, -1)
    features = scaler.transform(features)

    pred_label = clf.predict(features)[0]
    predicted_class = classes[pred_label]

    print(f"Predicted  Class: {predicted_class}")
```

```
t = r"C:\Users\madhu\Downloads\lamp.jpeg"
image = cv2.imread(t)
image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
plt.axis('off')
plt.imshow(image)
plt.show()
classify_image(t)
```

OUTPUT:

```
Model Training Complete. Accuracy: 0.7692
```

```
Predicted  Class: lamp
```

OBJECT DETECTION:

ALGORITHM:

Step 1: Import Required Libraries

- cv2 for image processing.
- matplotlib.pyplot for visualization.
- numpy for numerical operations.
- sklearn.cluster for Mean-Shift clustering.

Step 2: Read Images

- Load the **scene image (img2)** where the object is to be found.
- Load the **template image (img1)** which is the object to be detected.

Step 3: Convert Image for Visualization

- Make a copy of the **scene image** (img_rgb) to draw detections later.

Step 4: Initialize and Compute SIFT Features

- Create a **SIFT detector** using cv2.SIFT_create().
- Detect **keypoints and descriptors** in both images.

Step 5: Prepare Data for Clustering

- Extract keypoint locations from img2.
- Convert keypoint locations into a NumPy array for clustering.

Step 6: Apply Mean-Shift Clustering

- **Estimate bandwidth** using estimate_bandwidth().
- Apply **Mean-Shift clustering** to group similar keypoints.
- Store **cluster centers** and count the estimated number of clusters.

Step 7: Group Keypoints by Cluster

- Iterate over each **cluster** and store keypoints.

Step 8: Perform Feature Matching in Each Cluster

For each cluster:

1. **Check if keypoints are sufficient**

- If a cluster has less than 2 keypoints, skip it.

2. **Use FLANN-based Feature Matching**
   - Convert descriptors to float32 type.
   - Use **FLANN KNN Matcher** to find **good matches** (Lowe's ratio test).
3. **Filter Good Matches**
   - Keep matches where distance(m) < 0.5 * distance(n).
   - Ensure len(good) > MIN_MATCH_COUNT to proceed.

Step 9: Find Homography & Transform Points

- **Compute Homography Matrix (M)**
  - Use cv2.findHomography() with **RANSAC** to find transformation.
  - If M is None, skip the cluster.
- **Transform Object Corners**
  - Define the four corners of img1.
  - Use cv2.perspectiveTransform() to **map them onto img2**.

Step 10: Draw Bounding Box

- **Draw a rectangle** around the detected object in img_rgb.
- **Overlay a polygon** on img2 to visualize the transformed object.

Step 11: Convert Images for Display

- Convert **BGR to RGB** for proper visualization in matplotlib.

Step 12: Display Results

- Plot **template image** (img1).
- Plot **scene image** (img_rgb) with detected object.

CODE:

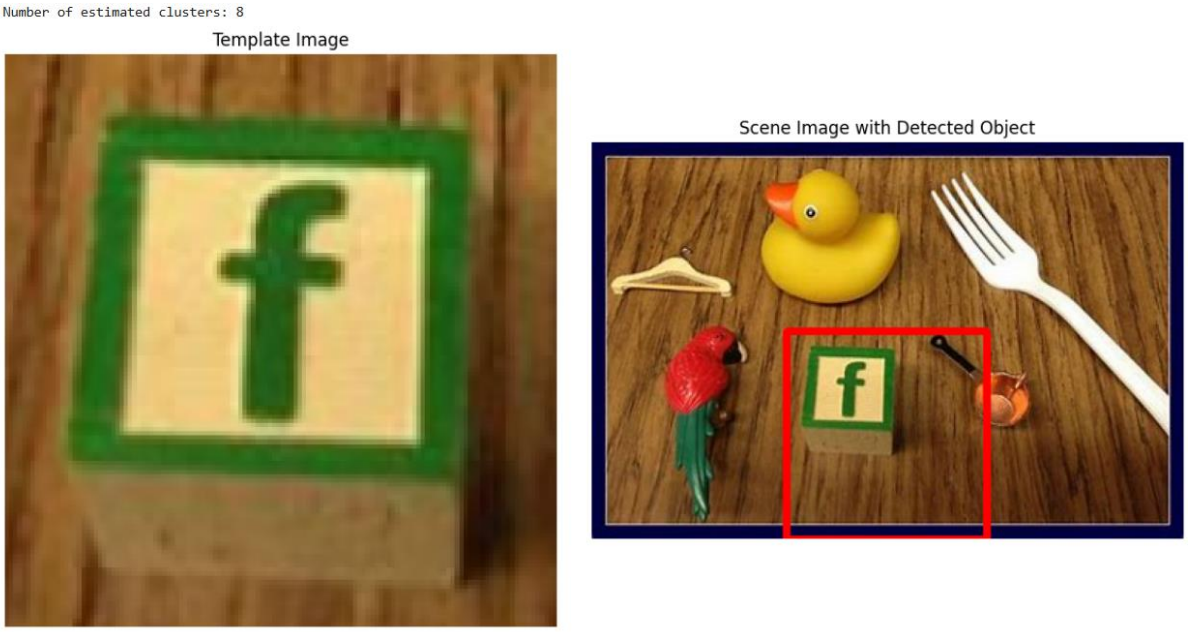
```
import cv2
from matplotlib import pyplot as plt
import numpy as np
from sklearn.cluster import MeanShift, estimate_bandwidth
MIN_MATCH_COUNT = 3
# Read images
img2 = cv2.imread(r"C:\Users\madhu\Downloads\objects.jpeg")  # Scene image
img1 = cv2.imread(r"C:\Users\madhu\OneDrive\Pictures\Screenshots\Screenshot 2025-03-17 211340.png")  # Template image
# Convert img2 for drawing later
img_rgb = img2.copy()
# Initialize SIFT
alg = cv2.SIFT_create()  # Use xfeatures2d.SIFT_create() for older OpenCV versions
# Detect keypoints and descriptors
kp1, des1 = alg.detectAndCompute(img1, None)
kp2, des2 = alg.detectAndCompute(img2, None)
# Prepare data for clustering
x = np.array([kp2[0].pt])
for i in range(len(kp2)):
    x = np.append(x, [kp2[i].pt], axis=0)
x = x[1:len(x)]  # Remove duplicate of first point
# Estimate bandwidth and apply MeanShift clustering
bandwidth = estimate_bandwidth(x, quantile=0.1, n_samples=500)
ms = MeanShift(bandwidth=bandwidth, bin_seeding=True, cluster_all=True)
ms.fit(x)
```

```
labels = ms.labels_
cluster_centers = ms.cluster_centers_
labels_unique = np.unique(labels)
n_clusters_ = len(labels_unique)
print("Number of estimated clusters: %d" % n_clusters_)
# Group keypoints by cluster
s = [None] * n_clusters_
for i in range(n_clusters_):
    l = ms.labels_
    d, = np.where(l == i)
    s[i] = list(kp2[xx] for xx in d)
des2_ = des2
# Loop through clusters
for i in range(n_clusters_):
    kp2 = s[i]
    d, = np.where(labels == i)
    des2 = des2_[d, :]
    if len(kp2) < 2 or len(kp1) < 2:
        continue
    # FLANN matcher
    FLANN_INDEX_KDTREE = 0
    index_params = dict(algorithm=FLANN_INDEX_KDTREE, trees=5)
    search_params = dict(checks=50)
    flann = cv2.FlannBasedMatcher(index_params, search_params)
    des1 = np.float32(des1)
    des2 = np.float32(des2)
    matches = flann.knnMatch(des1, des2, 2)
```

```
    good = []
    for m, n in matches:
        if m.distance < 0.5 * n.distance:
            good.append(m)
    # Proceed if enough good matches
    if len(good) > MIN_MATCH_COUNT:
        src_pts = np.float32([kp1[m.queryIdx].pt for m in good]).reshape(-1, 1, 2)
        dst_pts = np.float32([kp2[m.trainIdx].pt for m in good]).reshape(-1, 1, 2)
        M, mask = cv2.findHomography(src_pts, dst_pts, cv2.RANSAC, 2)
        if M is None:
            print("No Homography")
            continue
        # Draw bounding box
        h, w = img1.shape[:2]
        corners = np.float32([[0, 0], [0, h-1], [w-1, h-1], [w-1, 0]]).reshape(-1, 1, 2)
        transformedCorners = cv2.perspectiveTransform(corners, M)
        x = int(transformedCorners[0][0][0])
        y = int(transformedCorners[0][0][1])

        # Draw rectangle and polygon
        cv2.rectangle(img_rgb, (x, y), (x+w, y+h), (0, 0, 255), 3)
        img2 = cv2.polylines(img2, [np.int32(transformedCorners)], True, (0, 0, 255), 2,
cv2.LINE_AA)
# Convert images for matplotlib
img1_rgb = cv2.cvtColor(img1, cv2.COLOR_BGR2RGB)
img_rgb = cv2.cvtColor(img_rgb, cv2.COLOR_BGR2RGB)
# Plot using matplotlib
```

```
plt.figure(figsize=(12, 6))
plt.subplot(121)
plt.title('Template Image')
plt.axis('off')
plt.imshow(img1_rgb)
plt.subplot(122)
plt.title('Scene Image with Detected Object')
plt.axis('off')
plt.imshow(img_rgb)
plt.tight_layout()
plt.show()
```

OUTPUT:



RESULT:

Image classification and object detection using SIFT is performed .

EXERCISE NO:10	SEGMENTATION TECHNIQUES
DATE:20/03/25	

AIM:

To perform segmentation techniques like threshold , k-means etc on certain image.

K-MEANS SEGMENTATION:

ALGORITHM:

1. **Read and Preprocess Image**
    - Load the image using OpenCV (cv2.imread).
    - Convert it from BGR (default OpenCV format) to RGB for proper visualization.
    - Reshape the image into a 2D array where each row represents a pixel with three color channels.
2. **Convert Data Type**
    - Convert pixel values to float32 for numerical stability in computations.
3. **Initialize Cluster Centers**
    - Set a random seed for reproducibility.
    - Randomly select KKK initial cluster centers from the pixel values.
4. **Iterative Clustering (K-Means Algorithm)**
    - Repeat for a maximum of max_iters iterations or until cluster centers converge:
      a. **Compute Distance**: Calculate the Euclidean distance between each pixel and the cluster centers.
      b. **Assign Labels**: Assign each pixel to the nearest cluster center.
      c. **Update Centers**: Compute new cluster centers as the mean of all pixels in each cluster.
      d. **Check for Convergence**: If the centers do not change significantly (using np.allclose with a small tolerance), stop the iteration.
5. **Reconstruct Segmented Image**
    - Replace each pixel with its corresponding cluster center.

- Reshape the processed pixel array back to the original image dimensions.
- Convert the pixel values back to uint8 for proper image display.

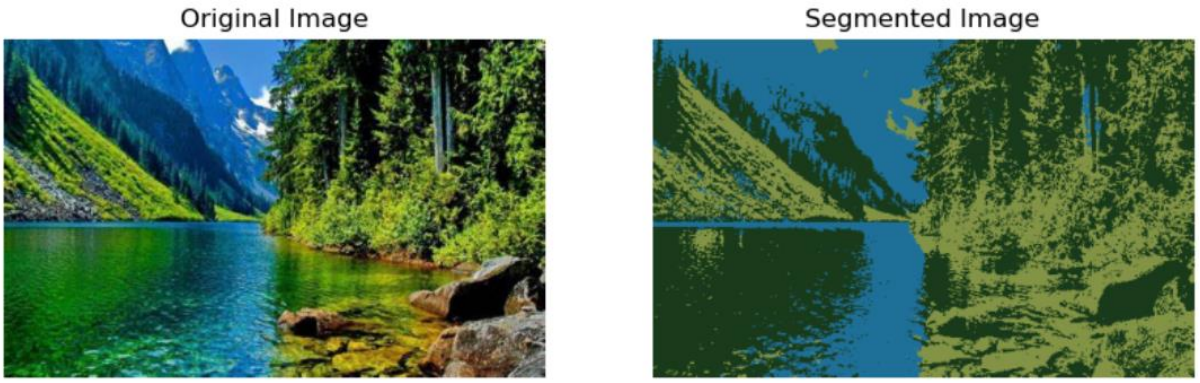
6. **Display Original and Segmented Images**
   - Use matplotlib.pyplot to show both the original and segmented images side by side.

CODE:

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
image = cv2.imread('OIP.jfif')
image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
def euclidean_distance(point1, point2):
    return np.sqrt(np.sum((point1 - point2) ** 2))
def kmeans_custom(image, k, max_iter=100):
    pixel_values = image.reshape((-1, 3))
    pixel_values = np.float32(pixel_values)
    np.random.seed(42)
    centroids = pixel_values[np.random.choice(pixel_values.shape[0], k, replace=False)]
     for _ in range(max_iter):
        labels = np.array([np.argmin([euclidean_distance(pixel, centroid) for centroid in centroids]) for pixel in pixel_values])
        new_centroids = np.array([pixel_values[labels == i].mean(axis=0) if len(pixel_values[labels == i]) > 0 else centroids[i] for i in range(k)])
        if np.all(centroids == new_centroids):
            break
        centroids = new_centroids
    return labels, centroids
k = 3
```

```
labels, centroids = kmeans_custom(image, k)
seg_image = centroids[labels].astype(np.uint8).reshape(image.shape)
plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.title("Original Image")
plt.imshow(image)
plt.axis('off')
plt.subplot(1, 2, 2)
plt.title("Segmented Image")
plt.imshow(seg_image)
plt.axis('off')
```

OUTPUT:



THRESHOLD SEGMENTATION:

ALGORITHM:

- **Load the Image**
  - Open the image using PIL (Image.open).
  - Convert the image to grayscale (convert("L")).
  - Convert the grayscale image into a NumPy array.
- **Define the Threshold Value**

- Set a threshold TTT (e.g., 100), which determines the cutoff for pixel intensities.

 Apply Manual Thresholding

- Iterate through each pixel:
  - If the pixel intensity is greater than TTT, set it to 255 (white).
  - Otherwise, set it to 0 (black).

 Display the Images

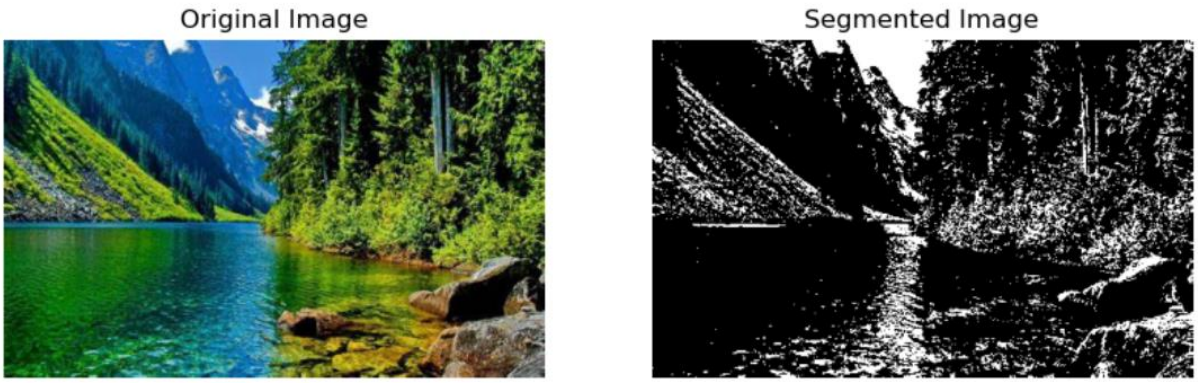
- Use matplotlib.pyplot to:
  - Display the original grayscale image.
  - Display the thresholded (binary) image.

CODE:

```
def custom_threshold(image, threshold_value):
    if len(image.shape) > 2:
        gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    else:
        gray = image
    binary = np.zeros_like(gray)
    binary[gray > threshold_value] = 255
    return binary
threshold_value = 128
segmented_image = custom_threshold(image, threshold_value)
plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.title("Original Image")
plt.imshow(image)
plt.axis('off')
plt.subplot(1, 2, 2)
plt.title("Segmented Image")
```

```
plt.imshow(segmented_image, cmap='gray')
plt.axis('off')
plt.show()
```

OUTPUT:



GRAB CUT SEGMENTATION:

ALGORITHM:

- **Load and Preprocess Image**
  - Read the image using OpenCV (cv2.imread).
  - Convert it from BGR (default OpenCV format) to RGB for proper visualization.
- **Initialize Mask and Models**
  - Create an empty mask of the same spatial dimensions as the image but with a single channel.
  - Initialize background (bgd_model) and foreground (fgd_model) models required for the GrabCut algorithm.
- **Define a Bounding Box**
  - Specify a rectangular region (x, y, width, height) that encloses the foreground object.
- **Apply GrabCut Algorithm**
  - Use cv2.grabCut() with the initialized mask, rectangle, and models.
  - Run the algorithm for a set number of iterations (e.g., 5) to refine the segmentation.
- **Process the Mask**

- Convert the mask values:
    - Pixels marked as **background (0,2)** are set to 0.
    - Pixels marked as **foreground (1,3)** are set to 1.

Extract the Foreground Object

- Multiply the original image with the processed mask to retain only the segmented object.

Display Results

- Use matplotlib.pyplot to:
    - Show the original image.
    - Show the segmented image after applying GrabCut.

CODE:

```
mask = np.zeros(image.shape[:2], np.uint8)
rect = (50, 50, 450, 290)
bgd_model = np.zeros((1, 65), np.float64)
fgd_model = np.zeros((1, 65), np.float64)
cv2.grabCut(image, mask, rect, bgd_model, fgd_model, 5, cv2.GC_INIT_WITH_RECT)
mask2 = np.where((mask == 2) | (mask == 0), 0, 1).astype('uint8')
result = image * mask2[:, :, np.newaxis]
plt.imshow(result)
plt.title("Grabcut Result")
plt.axis("off")
plt.show()
```

OUTPUT:

MEAN SHIFT SEGMENTATION:

ALGORITHM:

- **Load and Preprocess Image**
  - Read the image using OpenCV (cv2.imread).
  - Convert the image from BGR (default OpenCV format) to RGB for correct visualization.
- **Apply Mean Shift Filtering**
  - Use cv2.pyrMeanShiftFiltering() with the following parameters:
    - sp = 20: Defines the spatial window radius for filtering.
    - sr = 40: Defines the color window radius for filtering.
  - This filtering groups similar color regions together, smoothing the image while preserving edges.
- **Display Results**
  - Use matplotlib.pyplot to:
    - Show the original image.
    - Show the segmented image after Mean Shift filtering.

CODE:

```
segmented_image = cv2.pyrMeanShiftFiltering(image, sp=20, sr=40)

plt.figure(figsize=(10, 5))

plt.subplot(1, 2, 1)
```

```
plt.imshow(image)
plt.title("Original Image")
plt.axis("off")
plt.subplot(1, 2, 2)
plt.imshow(segmented_image)
plt.title("Segmented Image (Mean Shift)")
plt.axis("off")
plt.show()
```

OUTPUT:

REGION GROWING:

ALGORITHM:

- **Load and Preprocess Image**
  - Open the image using PIL.Image.open().
  - Convert it to grayscale (convert("L")).
  - Convert the grayscale image into a NumPy array.
- **Initialize Segmentation Variables**
  - Choose a **seed point** (x,y)(x, y)(x,y) manually.
  - Define a **threshold** for pixel similarity.
  - Create an empty **segmented mask** initialized to zero.

- Use a **stack** to store pixels to be visited (for DFS-like traversal).
- Use a **visited set** to track processed pixels.

 Define Neighborhood Connectivity

- Use **8-connected neighbors** (including diagonals) to allow smoother region expansion.

 Apply Region Growing Algorithm

- While the **stack is not empty**:
  - Pop a pixel (x,y)(x, y)(x,y) from the stack.
  - Check if it is already visited.
  - Compare the intensity difference between the current pixel and the seed pixel.
  - If the difference is **less than the threshold**, mark the pixel as part of the segmented region.
  - Add its **unvisited neighbors** to the stack for further exploration.

 Display Results

- Use matplotlib.pyplot to:
  - Show the **original grayscale image**.
  - Show the **segmented image** after region growing.

CODE:

```
def region(image, seed, threshold):
    rows, cols = image.shape
    segmented_image = np.zeros_like(image)
    segmented_image[seed] = 255
    neighbors = [(-1, 0), (1, 0), (0, -1), (0, 1)]
    to_process = [seed]
    while to_process:
        current_pixel = to_process.pop()
        x, y = current_pixel
```

```
        for dx, dy in neighbors:
            nx, ny = x + dx, y + dy
            if 0 <= nx < rows and 0 <= ny < cols:
                if segmented_image[nx, ny] == 0:
                    if abs(int(image[x, y]) - int(image[nx, ny])) <= threshold:
                        segmented_image[nx, ny] = 255
                        to_process.append((nx, ny))
    return segmented_image
image_gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
seed = (5, 5)
threshold = 10
segmented_image = region(image_gray, seed, threshold)
plt.subplot(1, 2, 1)
plt.imshow(image_gray, cmap='gray')
plt.title('Original Image')
plt.axis("off")
plt.subplot(1, 2, 2)
plt.imshow(segmented_image, cmap='gray')
plt.title('Region Growing Segmentation')
plt.axis("off")
plt.show()
```

OUTPUT:

Original Image

Region Growing Segmentation

RESULT:

The segmentation techniques is successfully performed and analysed.

EXERCISE NO:11	BACKGROUND SUBTRACTION
DATE:27/03/25	

AIM:

To perform background subtraction in video for motion detection.

ALGORITHM:

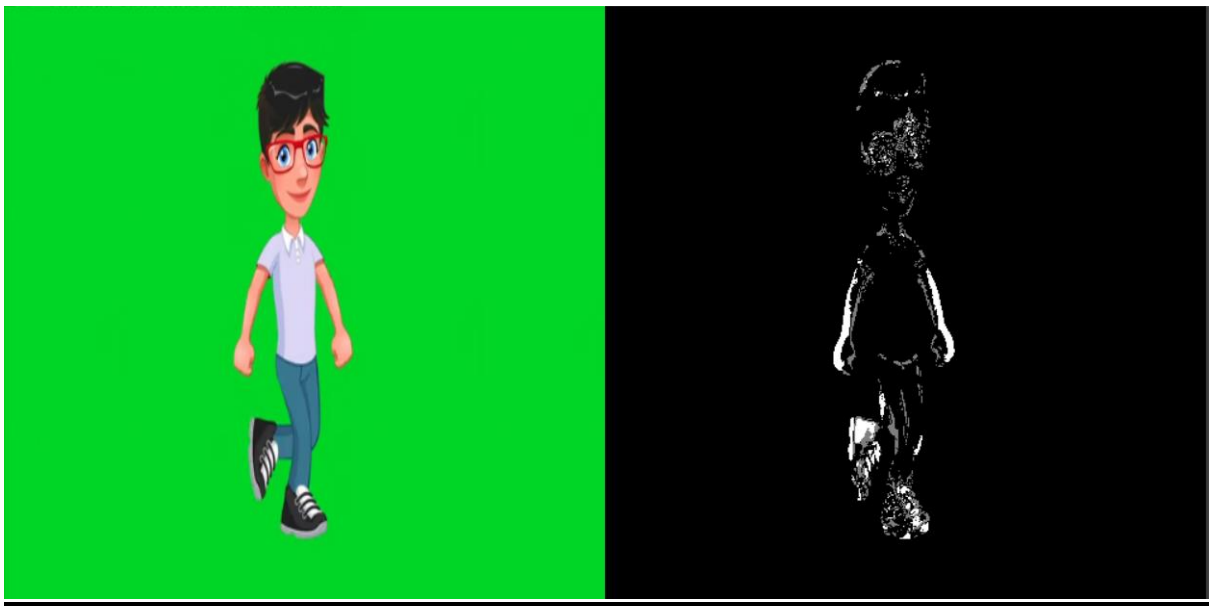
- **Start**
- **Load the video** from the given path.
- **Check if the video is opened** successfully. If not, exit.
- **Initialize the background subtractor** using MOG2.
- **Process each frame** of the video:
  - Read the frame. If reading fails, exit.
  - Keep a copy of the original frame.
  - Convert the frame to **grayscale**.
  - Resize the frame to **500x500** pixels.
  - Apply **Gaussian blur** to smoothen the image.
  - Apply **background subtraction** to get the foreground mask.
  - Resize the original frame to match the processed frame size.
  - **Concatenate** the original and processed frames side by side.
  - Display the combined result.
- **Wait for user input (q)** to exit.
- **Release video resources** and close the display window.
- **End**

CODE:

```
import cv2
def background_subtraction(video_path):
    cap = cv2.VideoCapture(video_path)
    if not cap.isOpened():
```

```
        print("Error: Unable to open video.")
        return
    bg_subtractor = cv2.createBackgroundSubtractorMOG2(detectShadows=True)
while True:
        ret, frame = cap.read()
 if not ret:
            break
 original_frame = frame.copy()  # Keep a copy of the original frame
        frame = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
        frame = cv2.resize(frame, (500, 500))
        frame = cv2.GaussianBlur(frame, (5, 5), 0)
        fg_mask = bg_subtractor.apply(frame)
       original_frame = cv2.resize(original_frame, (500, 500)
        combined = cv2.hconcat([original_frame, cv2.cvtColor(fg_mask,
cv2.COLOR_GRAY2BGR)])
cv2.imshow('Original Video and Foreground Mask', combined)
  if cv2.waitKey(30) & 0xFF == ord('q'):
            break
cap.release()
    cv2.destroyAllWindows()
background_subtraction(r"C:\Users\madhu\Downloads\video.mp4")
```

OUTPUT:



BACKGROUND SUBTRACTION BY AVERAGING:

ALGORITHM:

- **Start**
- **Load the video** from the given path.
- **Check if the video is opened** successfully. If not, exit.
- **Initialize the background model** as None.
- **Process each frame** of the video:
  - Read the frame. If reading fails, exit.
  - Keep a **copy of the original frame**.
  - Convert the frame to **grayscale**.
  - Resize the frame to **500×500** pixels.
  - If the background model is **empty**, initialize it using the current frame and continue.
  - Update the **background model** using **accumulateWeighted()** with a given learning rate.
  - Compute the **absolute difference** between the current frame and the background model.
  - Apply **thresholding** to get the **foreground mask** (highlight moving objects).
  - Resize the original frame to match the processed frame.

- Convert the foreground mask to **3 channels** to match the original frame.
- **Concatenate** and display the original and foreground mask side by side.

 Wait for user input (q) to exit.

 Release video resources and close the display window.

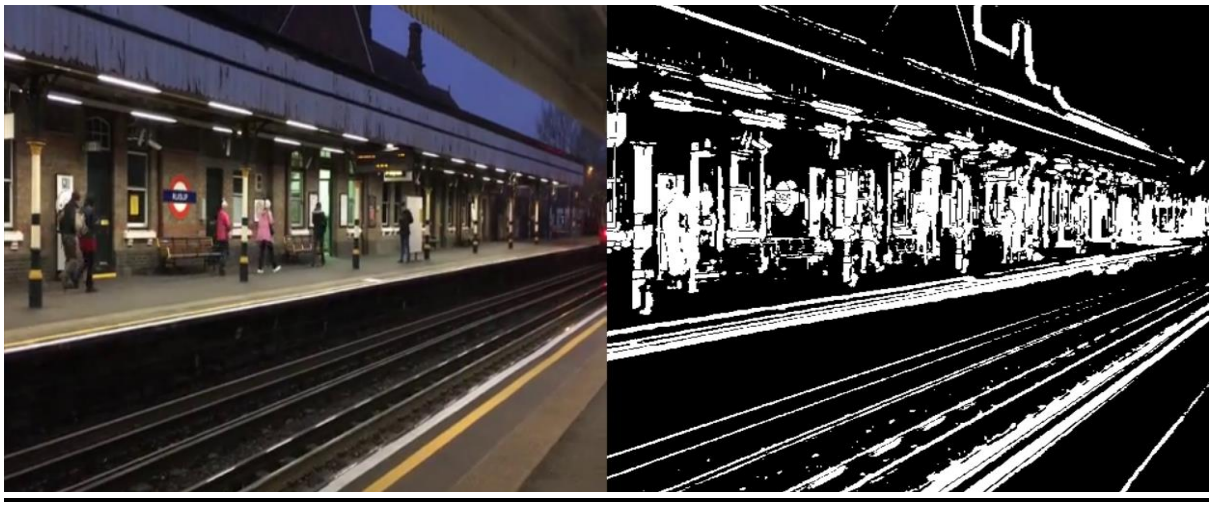
 End

CODE:

```
import cv2
import numpy as np
def background_subtraction_by_averaging(video_path, learning_rate=0.01):
    cap = cv2.VideoCapture(video_path)
     if not cap.isOpened():
        print("Error: Unable to open video.")
        return
   background = None
    while True:
        ret, frame = cap.read()
         if not ret:
            break

        original_frame = frame.copy()  # Keep a copy of the original frame
        gray_frame = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
        gray_frame = cv2.resize(gray_frame, (500, 500))

        if background is None:
            background = gray_frame.astype("float")
            continue
        cv2.accumulateWeighted(gray_frame, background, learning_rate)
```

```
        Diff_frame = cv2.absdiff(gray_frame, cv2.convertScaleAbs(background))


        _, fg_mask = cv2.threshold(diff_frame, 25, 255, cv2.THRESH_BINARY)
original_frame = cv2.resize(original_frame, (500, 500))
        fg_mask_colored = cv2.cvtColor(fg_mask, cv2.COLOR_GRAY2BGR
        combined = cv2.hconcat([original_frame, fg_mask_colored])
          cv2.imshow('Original Video and Foreground Mask', combined)


        if cv2.waitKey(30) & 0xFF == ord('q'):
            break
          cap.release()
    cv2.destroyAllWindows()
background_subtraction_by_averaging(r"C:\Users\madhu\Downloads\video 2.mp4")
```

OUTPUT:



MOTION DETECCTION WITH SCENE CHANGE:

ALGORITHM:

- **Start**
- **Load the video** from the given file path.
- **Check if the video is opened** successfully. If not, exit.
- **Initialize the background model** as None.

 Process each frame of the video:

- Read the frame. If reading fails, exit.
- Keep a **copy of the original frame** for display.
- Convert the frame to **grayscale**.
- Resize the frame to **500×500** pixels.
- If the background model is **empty**, initialize it with the current frame and continue.
- Update the **background model** using **accumulateWeighted()** with a specified learning rate.
- Compute the **absolute difference** between the current frame and the background model.
- Apply **thresholding** to get a **foreground mask** (highlighting scene changes).
- Count the **number of changed pixels** (non-zero values in the mask).
- If the count **exceeds the threshold**, print "Scene change detected!".
- Resize the original frame for display.
- Convert the **foreground mask to 3 channels** for side-by-side visualization.
- **Concatenate** and display both the original frame and the foreground mask.

 Wait for user input (q) to exit.

 Release video resources and close the display window.

 End

CODE:

```
import cv2
import numpy as np
def detect_scene_change(video_path, threshold=50000, learning_rate=0.01):
    cap = cv2.VideoCapture(video_path)
        if not cap.isOpened():
        print("Error: Unable to open video.")
        return
```

```
background = None
    while True:
        ret, frame = cap.read()
         if not ret:
            break
  original_frame = frame.copy()  # Keep a copy of the original frame
     gray_frame = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
        gray_frame = cv2.resize(gray_frame, (500, 500))

        if background is None:
            background = gray_frame.astype("float")
            continue
        cv2.accumulateWeighted(gray_frame, background, learning_rate)
        diff_frame = cv2.absdiff(gray_frame, cv2.convertScaleAbs(background))

        _, fg_mask = cv2.threshold(diff_frame, 25, 255, cv2.THRESH_BINARY)

        changed_pixels = cv2.countNonZero(fg_mask)
           if changed_pixels > threshold:
            print("Scene change detected!")
        original_frame = cv2.resize(original_frame, (500, 500))
        fg_mask_colored = cv2.cvtColor(fg_mask, cv2.COLOR_GRAY2BGR)
        combined = cv2.hconcat([original_frame, fg_mask_colored])
       cv2.imshow('Original Video and Foreground Mask', combined)
       if cv2.waitKey(30) & 0xFF == ord('q'):
            break
```

```
    cap.release()
    cv2.destroyAllWindows()
detect_scene_change(r"C:\Users\madhu\Downloads\video3.mp4")
```

OUTPUT:

SCENE CHANGE BY MOTION DETECTION:

ALGORITHM:

- **Start**
- **Load the video** from the given file path.
- **Check if the video opens successfully**, otherwise exit.
- **Initialize the background model** as None.
- **Process each frame** of the video:
  - Read the frame. If reading fails, exit.
  - Keep a **copy of the original frame** for display.
  - Convert the frame to **grayscale** and resize it to **500×500** pixels.
  - If the background model is **empty**, initialize it and continue.
  - Update the **background model** using **accumulateWeighted()**.
  - Compute the **absolute difference** between the current frame and the background.

- Apply **thresholding** to get the **foreground mask**.
- Count the **number of changed pixels** in the foreground mask.
- If the count **exceeds the threshold**, print "Scene change detected!".
- Display the **original frame and the foreground mask** side by side.

 Wait for user input (q) to exit.

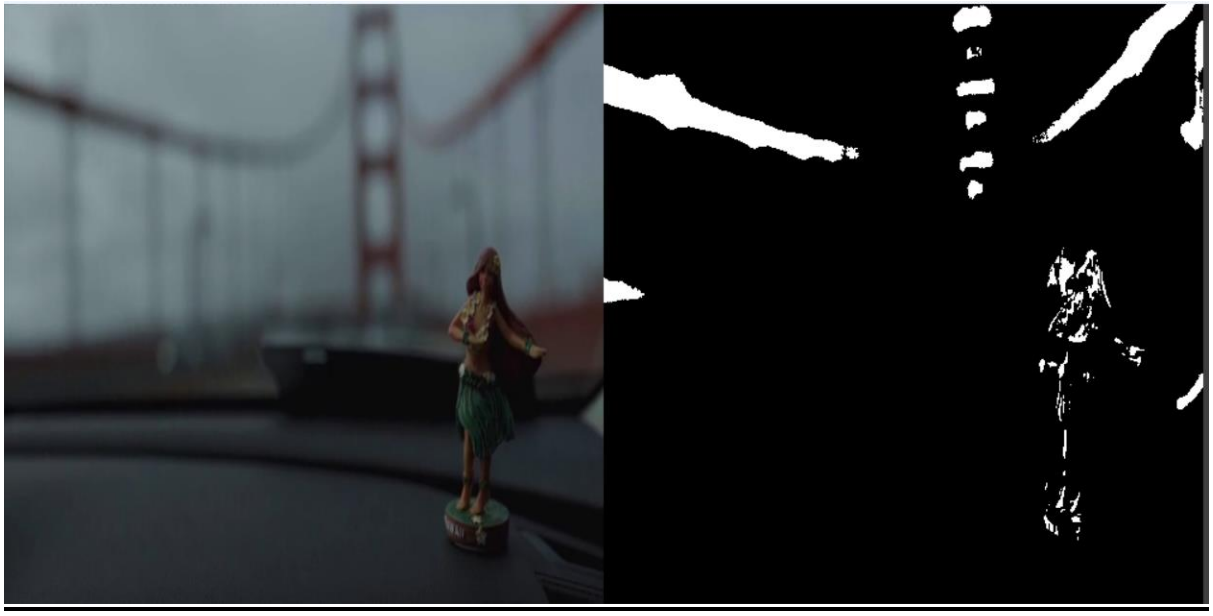
 Release resources and close the display window.

CODE:

```
import cv2
import numpy as np
def detect_scene_change_by_background_subtraction(video_path,
change_threshold=100000, learning_rate=0.01):
 cap = cv2.VideoCapture(video_path)
   if not cap.isOpened():
       print("Error: Unable to open video.")
       return
 background = None
 while True:
       ret, frame = cap.read()
       if not ret:
          break
original_frame = frame.copy()  # Keep a copy of the original frame
 gray_frame = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
       gray_frame = cv2.resize(gray_frame, (500, 500))

       if background is None:
          background = gray_frame.astype("float")
          continue
```

```
        cv2.accumulateWeighted(gray_frame, background, learning_rate)


        diff_frame = cv2.absdiff(gray_frame, cv2.convertScaleAbs(background))
 _, fg_mask = cv2.threshold(diff_frame, 25, 255, cv2.THRESH_BINARY) changed_pixels
= cv2.countNonZero(fg_mask)
 if changed_pixels > change_threshold:
            print("Scene change detected!")
        original_frame = cv2.resize(original_frame, (500, 500))
        fg_mask_colored = cv2.cvtColor(fg_mask, cv2.COLOR_GRAY2BGR)
        combined = cv2.hconcat([original_frame, fg_mask_colored])
    cv2.imshow('Original Video and Foreground Mask', combined)
   if cv2.waitKey(30) & 0xFF == ord('q'):
            break
    cap.release()
    cv2.destroyAllWindows()
detect_scene_change_by_background_subtraction(r"C:\Users\madhu\Downloads\video4.
mp4")
```

OUTPUT:



RESULT:

We have performed background subtraction in video for motion detection.

EXERCISE NO:12	LUCAS KANNADE ALGORITHM
DATE:3/4/2025	

AIM:

To perform the lucas kannade algorithm between two frame differences

and in video.

ALGORITHM:

1. Frame Handling:

   Always check if frames are loaded correctly; stop the program if not.

   Resize frames to the same size if necessary.

   Convert frames to grayscale for efficient processing.

2. Feature Detection:

   Use Shi-Tomasi corner detection to find good features (edges, corners) to track.

   If no features are found, exit the program to avoid tracking errors.

3. Lucas-Kanade Optical Flow:

   Use it to track how feature points move from one frame to the next.

   Define optical flow parameters (e.g., window size, pyramid levels, termination criteria).

4. Tracking and Visualization:

   Filter out invalid points (those that weren't tracked properly).

   Use arrows or lines to visualize movement from original to new positions.

   Update the reference frame and points after each iteration.

5. Video Looping:

   Continuously read frames from the video.

   Perform tracking and visualization for each frame.

   Allow exit from the loop with a key press (like 'q').

6. Output and Display:

Display the processed frames using Matplotlib or OpenCV's imshow.

Use a mask image to draw motion vectors without altering the original frame.

7. Clean Exit:

Release video resources and close all OpenCV windows at the end.

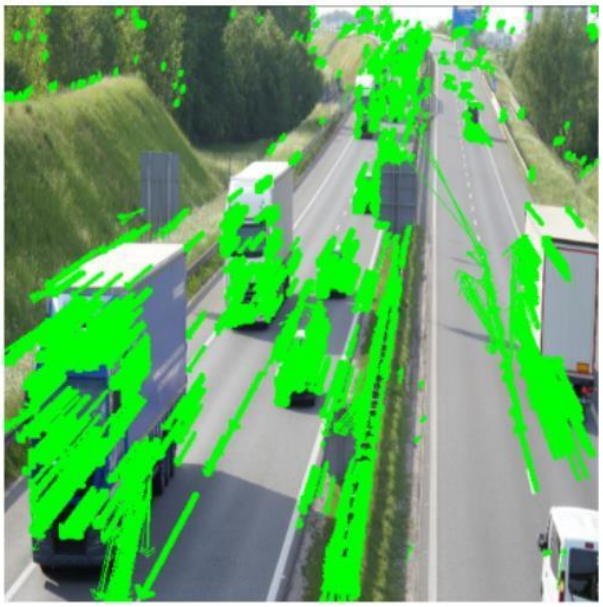
CODE:

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
def Lucas_kanade(frame1,frame2_vis):
    if frame1 is None or frame2_vis is None:
        raise ValueError("One or both frames are empty!")

    frame1=cv2.resize(frame1,(1000,800))
    frame2_vis=cv2.resize(frame2_vis,(1000,800))
    frame2_vis = cv2.resize(frame2_vis, (frame1.shape[1], frame1.shape[0]))
    frame1=cv2.cvtColor(frame1,cv2.COLOR_BGR2GRAY) if
len(frame1.shape) == 3 else frame1
    frame2=cv2.cvtColor(frame2_vis,cv2.COLOR_BGR2GRAY) if
len(frame2_vis.shape) == 3 else frame2
    dst1=cv2.cornerHarris(frame1, blockSize=2, ksize=3, k=0.04)
    dst1=cv2.dilate(dst1,None)
    thresh=0.01*dst1.max()
    features=np.argwhere(dst1>thresh)
    features=np.flip(features,axis=1)
    features=np.float32(features).reshape(-1,1,2)
    if features is None or len(features) == 0:
        raise ValueError("No keypoints detected!")
```

```
    new_features, status, err = cv2.calcOpticalFlowPyrLK(frame1, frame2,
features, None)
    new_features = new_features[status == 1]
    features = features[status == 1]
    motion_threshold = .5
    max_motion = 50
    for i, (new, old) in enumerate(zip(new_features, features)):
        if status[i]:
            x_new, y_new = new.ravel()
            x_old, y_old = old.ravel()
            displacement = np.linalg.norm(new - old)
            if motion_threshold < displacement < max_motion:

                cv2.arrowedLine(frame2_vis, (int(x_old), int(y_old)), (int(x_new),
int(y_new)), (0, 255, 0),1, tipLength=0.1)
    return frame2_vis,frame2
cap=cv2.VideoCapture(r"C:\Users\admin\Downloads\vecteezy_car-and-truck-
traffic-on-the-highway-in-europe-poland_7957364.mp4")
ret,frame1=cap.read()
frame1=cv2.cvtColor(frame1,cv2.COLOR_BGR2GRAY)
while cap.isOpened():
    ret,frame2=cap.read()
    if not ret:
        print("Image not loadded")
        break
    frame2_vis,frame2_gray=Lucas_kanade(frame1,frame2)
    cv2.imshow('Optical Flow',frame2_vis)
    if cv2.waitKey(30) & 0xFF == 27:
```

```
        break
    frame1=frame2_gray
cap.release()
cv2.destroyAllWindows()
im1=cv2.imread(r"C:\Users\admin\Pictures\Screenshots\Screenshot 2025-04-03 112035.png")
im2=cv2.imread(r"C:\Users\admin\Pictures\Screenshots\Screenshot 2025-04-03 112053.png")
vis,frame2=Lucas_kanade(im1,im2)
vis=cv2.cvtColor(vis,cv2.COLOR_BGR2RGB)
plt.imshow(vis)
plt.axis('off')
```

OUTPUT:



RESULT:

The lucas kannade algorithm is performed successfully.