MADRAS INSTITUTE OF TECHNOLOGY
ANNA UNIVERSITY

DEPARTMENT OF INFORMATION TECHNOLOGY
AD23402-COMPUTER VISION

RECORD

REGISTER NUMBER: 2023510057

NAME: REVAAN.J.R

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 **REVAAN.J.R**, Register Number **(2023510057)** Of **Four** Semester **B.Tech Artificial Intelligence and Data Science** in the **– Computer Vision Laboratory.**

Date: 01/05/2025

Course Instructor: **Ms M Hemalatha**

TABLE OF CONTENTS:

EX NO:1 DATE:09/01/2025	BASICS OF IMAGE PROCESSING

AIM:

To perform the basic operations in image processing like lenna.jpg and normal image

Lenna.jpg

Read the image:

Algorithm:

1. Import numpy and imread
2. Read the image from the path ("C:\Users\student\Downloads\lenna.jpg")
3. Store the image as a numpy array (image)
4. Perform further operations (optional) - Example: Convert to grayscale, resize, etc.

Code:

```
import numpy as np
from matplotlib.image import imread  image =
imread(r"C:\Users\student\Downloads\lenna.jpg") image
```

output:

```
array([[[236, 139, 122],
        [231, 136, 118],
        [231, 136, 118],
        ...,
        [226, 121, 126],
        [248, 149, 154],
        [243, 148, 152]],

       [[234, 137, 120],
        [229, 134, 116],
        [229, 134, 116],
        ...,
        [221, 120, 126],
        [241, 146, 152],
        [227, 136, 143]],

       [[231, 134, 117],
        [226, 131, 113],
        [226, 131, 111],
        ...,
        [191,  98, 106],
        [161,  74,  83],
        [135,  51,  64]],

       ...,

       [[ 85,  33,  55],
        [ 90,  29,  60],
        [ 96,  23,  68],
        ...,
        [114,  36,  62],
        [118,  40,  64],
        [121,  42,  64]],
```

```
[[ 84,  32,  54],
 [ 90,  29,  60],
 [ 96,  23,  68],
 ...,
 [129,  51,  75],
 [140,  61,  83],
 [146,  65,  84]],

[[ 86,  35,  54],
 [ 91,  30,  61],
 [ 97,  24,  69],
 ...,
 [140,  60,  85],
 [155,  73,  95],
 [161,  76,  95]]], dtype=uint8)
```

Algorithm:

1. Import cv2 and matplotlib.pyplot
2. Read the image from "C:\Users\student\Downloads\lenna.jpg"
3. Store the image in the variable "image"
4. Display the image using plt.imshow(image)
5. Turn off the axis with plt.axis('off')
6. Show the image with plt.show() **Code:** import cv2

import matplotlib.pyplot as plt image =

cv2.imread(r"C:\Users\student\Downloads\lenna.jpg")

plt.imshow(image) plt.axis('off') plt.show()

output:

Resize the image: By

setting values

Algorithm:

1. Import cv2 and matplotlib.pyplot
2. Read the image from the path (e.g., "C:\Users\student\Downloads\lenna.jpg")
3. Resize the image to (100, 100) using cv2.resize(image, (100, 100))
4. Display the resized image using plt.imshow(resized_image)
5. Turn off the axis with plt.axis('off')
6. Show the image with plt.show()
7. Print the resized image data using print(resized_image) **Code:** resized_image = cv2.resize(image, (100, 100)) # Resize to 500x500 pixels plt.imshow(resized_image) plt.axis('off') plt.show()

print(resized_image)

output:

```
[[[119 136 233]
  [117 135 230]
  [112 133 227]
  ...
  [ 86  82 194]
  [118 113 216]
  [152 145 239]]

 [[115 132 229]
  [111 131 226]
  [109 129 224]
  ...
  [104  99 198]
  [ 99  89 181]
  [ 61  47 131]]

 [[109 128 225]
  [108 128 223]
  [107 127 222]
  ...
  [104  91 174]
  [ 33  14  94]
  [ 51  25  99]]

 ...

 [[ 64  36  92]
  [ 80  35 109]
  [ 87  48 118]
  ...
  [ 59  31 107]
  [ 56  29 105]
  [ 51  25 100]]
```

```
[[ 59  32  88]
 [ 70  24  98]
 [ 72  34 103]
 ...
 [ 53  24 100]
 [ 59  33 110]
 [ 61  37 116]]

[[ 58  31  87]
 [ 67  23  97]
 [ 59  20  90]
 ...
 [ 57  27 102]
 [ 78  52 131]
 [ 91  70 152]]]
```

By dividing the size(height,width):

Algorithm:

1. Import cv2 and matplotlib.pyplot
2. Read the image from the path (e.g., "C:\Users\student\Downloads\lenna.jpg")
3. Resize the image to half of its original size:
    - width = image.shape[1] / 2
    - height = image.shape[0] / 2
    - resized_image = cv2.resize(image, (width, height))
4. Display the resized image using plt.imshow(resized_image)
5. Turn off the axis with plt.axis('off')
6. Show the image with plt.show()
7. Print the resized image data using print(resized_image)

Code: resized_image = cv2.resize(image, (int(image.shape[1] / 2),

int(image.shape[0]

/2)))

plt.imshow(resized_image)

plt.axis('off') plt.show()

print(resized_image)

Output:

```
[[[119 136 233]
  [117 135 230]
  [113 133 228]
  ...
  [ 86  84 195]
  [109 104 208]
  [152 145 239]]

 [[114 131 228]
  [110 130 225]
  [109 129 224]
  ...
  [104  99 197]
  [ 93  83 175]
  [ 49  34 117]]

 [[109 128 225]
  [109 129 224]
  [109 129 223]
  ...
  [ 95  81 164]
  [ 41  21  98]
  [ 55  26  98]]

 ...

 [[ 64  35  92]
  [ 88  42 116]
  [105  72 136]
  ...
  [ 66  37 113]
  [ 58  31 107]
  [ 53  27 102]]
```

```
[[ 64  35  92]
 [ 88  42 116]
 [105  72 136]
 ...
 [ 66  37 113]
 [ 58  31 107]
 [ 53  27 102]]

[[ 60  32  88]
 [ 73  27 101]
 [ 93  61 125]
 ...
 [ 55  26 102]
 [ 57  30 106]
 [ 57  33 111]]

[[ 58  31  88]
 [ 68  23  97]
 [ 82  51 115]
 ...
 [ 54  23  99]
 [ 73  47 125]
 [ 90  68 150]]]
```

Dimension of the image:

Algorithm:

1. Import necessary libraries (if not done already).
2. Read the image into image_rgb.
3. Extract the image's dimensions using image_rgb.shape:
    - height = image_rgb.shape[0]
    - width = image_rgb.shape[1] - channels = image_rgb.shape[2]
4. Print the height, width, and channels:
    - Print "height:", height
    - Print "width:", width

- Print "channel:", channels **Code:** height, width, channels = image_rgb.shape print("height:",height) print("width:",width) print("channel:",channel) **output:**

```
height: 183
width: 183
channel: 3
```

Converting BGR to RGB With inbuilt function:

Algorithm:

1. Import cv2 and matplotlib.pyplot.
2. Read the image using cv2.imread() into "image".
3. Convert the image from BGR to RGB using:
   image_rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
4. Display the RGB image using plt.imshow(image_rgb)
5. Turn off the axis with plt.axis('off')
6. Show the image with plt.show()
7. Print the RGB image data using print(image_rgb)

Code: image_rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB) plt.imshow(image_rgb) plt.axis('off') plt.show() print(image_rgb)

Output:

```
[[[236 139 122]
  [232 135 118]
  [231 136 118]
  ...
  [225 122 126]
  [248 149 154]
  [241 149 154]]

 [[234 137 120]
  [230 133 116]
  [229 134 116]
  ...
  [220 121 126]
  [241 146 154]
  [227 136 145]]

 [[231 134 117]
  [227 130 113]
  [226 131 111]
  ...
  [191  98 108]
  [161  73  85]
  [135  51  64]]

 ...

 [[ 86  32  56]
  [ 89  30  60]
  [ 96  23  68]
  ...
  [114  36  62]
  [118  40  64]
  [121  42  64]]
```

```
...

[[ 86  32  56]
 [ 89  30  60]
 [ 96  23  68]
 ...
 [114  36  62]
 [118  40  64]
 [121  42  64]]

[[ 85  31  55]
 [ 89  30  60]
 [ 96  23  68]
 ...
 [129  51  75]
 [140  61  83]
 [146  64  86]]

[[ 87  33  57]
 [ 90  31  61]
 [ 97  24  67]
 ...
 [140  60  85]
 [155  73  95]
 [161  76  97]]]
```

Without inbuilt function:

Algorithm:

1. Import cv2 and matplotlib.pyplot.
2. Read the image into "image" (in BGR format).
3. Convert BGR to RGB by rearranging the channels:
   - image_rgb = image[:, :, [2, 1, 0]]
4. Display the RGB image using plt.imshow(image_rgb)
5. Turn off the axis with plt.axis('off')
6. Show the image with plt.show()

7. Print the RGB image data using print(image_rgb) **Code:** import cv2 import matplotlib.pyplot as plt image_rgb = image[:, :, [2, 1, 0]]

plt.imshow(image_rgb)

plt.axis('off') plt.show()

print(image_rgb)

Output:

```
[[[236 139 122]
  [232 135 118]
  [231 136 118]
  ...
  [225 122 126]
  [248 149 154]
  [241 149 154]]

 [[234 137 120]
  [230 133 116]
  [229 134 116]
  ...
  [220 121 126]
  [241 146 154]
  [227 136 145]]

 [[231 134 117]
  [227 130 113]
  [226 131 111]
  ...
  [191  98 108]
  [161  73  85]
  [135  51  64]]

 ...

 [[ 86  32  56]
  [ 89  30  60]
  [ 96  23  68]
  ...
  [114  36  62]
  [118  40  64]
  [121  42  64]]
```

```
   ...
   [114  36  62]
   [118  40  64]
   [121  42  64]]

  [[ 85  31  55]
   [ 89  30  60]
   [ 96  23  68]
   ...
   [129  51  75]
   [140  61  83]
   [146  64  86]]

  [[ 87  33  57]
   [ 90  31  61]
   [ 97  24  67]
   ...
   [140  60  85]
   [155  73  95]
   [161  76  97]]]
```

Converting BGR to GRAY:

Algorithm:

1. Import cv2 and matplotlib.pyplot.
2. Read the image into "image" (in BGR format).
3. Convert the image to grayscale:

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

4. Display the grayscale image using plt.imshow(gray_image, cmap='gray')
5. Turn off the axis with plt.axis('off')
6. Show the image with plt.show() **Code:**

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

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

print(gray_image)
```

Output:

```
[[166 162 162 ... 153 179 177]
 [164 160 160 ... 151 175 164]
 [161 157 157 ... 127 101  78]
 ...
 [ 51  51  50 ...  62  66  68]
 [ 50  51  50 ...  77  87  91]
 [ 52  52  51 ...  87 100 104]]
```

Without inbuilt function:

Algorithm:

1. Import cv2, numpy, and matplotlib.pyplot.
2. Read the image into "image_rgb" (in RGB format).
3. Get the image dimensions (height, width, channels) from image_rgb.
4. Create a blank grayscale image of size (height, width).
5. Loop through each pixel in the image and apply the grayscale conversion formula:

- grayscale_value = 0.2989 * r + 0.5870 * g + 0.1140 * b

6. Store the grayscale value in the corresponding pixel of grayscale_image.
7. Display the grayscale image using plt.imshow(grayscale_image, cmap='gray').
8. Turn off the axis using plt.axis('off').
9. Show the image using plt.show().
10. Print the grayscale image data using print(grayscale_image).

Code: import cv2 import numpy as np

import matplotlib.pyplot as plt height,

width, channels = image_rgb.shape

grayscale_image = np.zeros((height, width), dtype=np.uint8) for

i in range(height): for j in range(width): r = image_rgb[i,

j, 0] g = image_rgb[i, j, 1] b = image_rgb[i, j, 2]

grayscale_value = int(0.2989 * r + 0.5870 * g + 0.1140 * b)

grayscale_image[i, j] = grayscale_value

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

plt.show() print(grayscale_image)

Output:

```
[[166 162 162 ... 153 179 177]
 [164 160 160 ... 151 175 164]
 [161 157 157 ... 126 100  77]
 ...
 [ 50  51  49 ...  62  66  68]
 [ 49  51  49 ...  77  87  91]
 [ 51  52  50 ...  86 100 103]]
```

Histogram equalization:

With inbuilt function:

Algorithm:

1. Import cv2 and matplotlib.pyplot.
2. Read the image into "image" (in BGR format).
3. Convert the image to grayscale using:

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

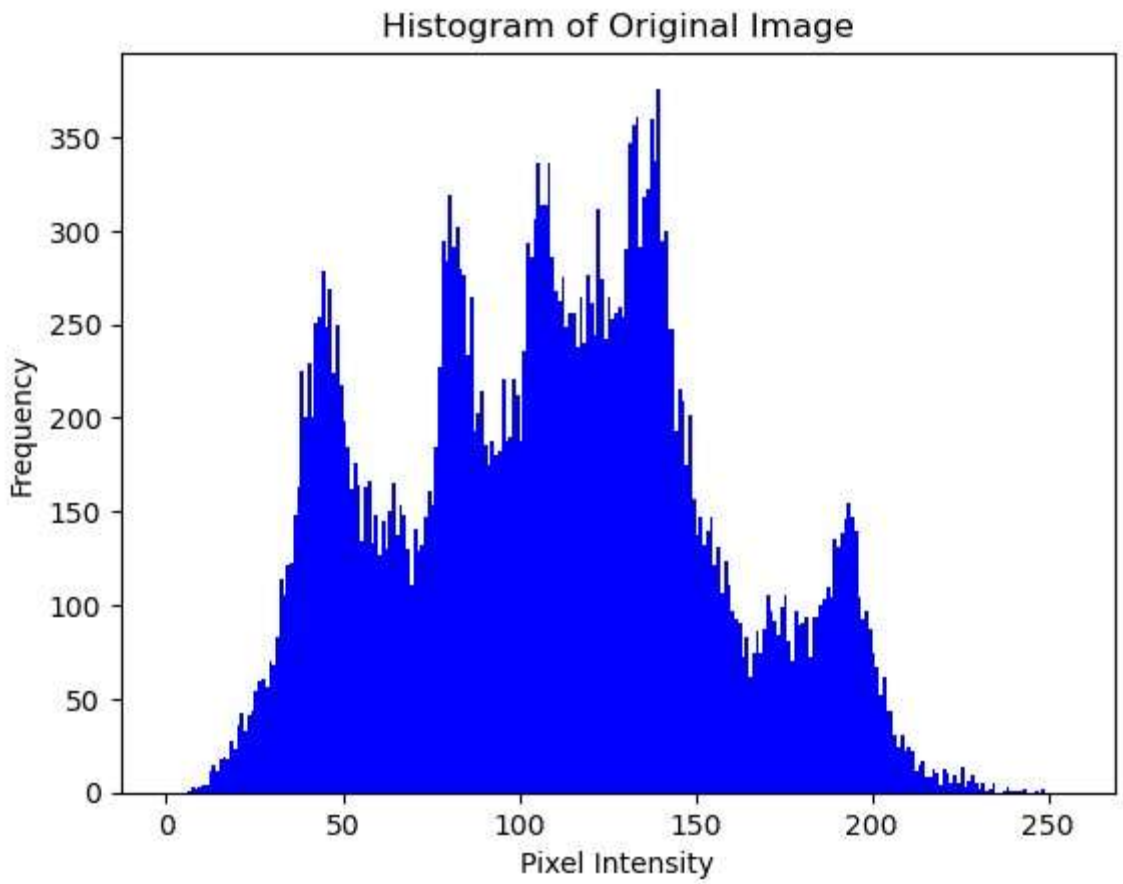
4. Apply histogram equalization to the grayscale image: equalized_image = cv2.equalizeHist(gray_image)

5. Display the original grayscale image using plt.imshow(gray_image, cmap='gray').
6. Plot the histogram of the original image using plt.hist(gray_image.flatten(), bins=256).
7. Display the equalized image using plt.imshow(equalized_image, cmap='gray').
8. Plot the histogram of the equalized image using plt.hist(equalized_image.flatten(), bins=256).

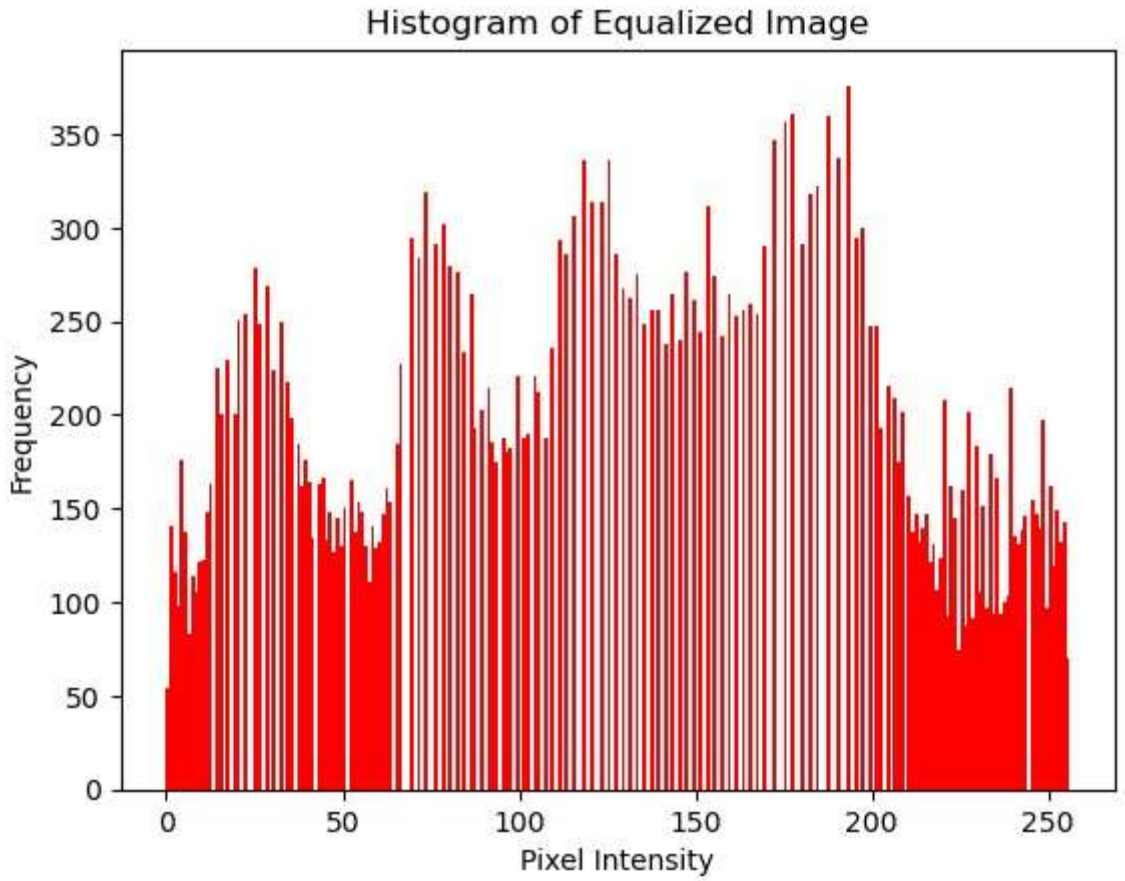
Code:

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

equalized_image = cv2.equalizeHist(gray_image)


plt.title('Original Grayscale Image') plt.imshow(gray_image,

cmap='gray')

plt.axis('off') plt.show()


plt.title('Histogram of Original Image') plt.hist(gray_image.flatten(),

bins=256, range=[0, 256], color='blue') plt.xlabel('Pixel Intensity')

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


plt.title('Histogram Equalized Image')

plt.imshow(equalized_image,

cmap='gray') plt.axis('off') plt.show()


plt.title('Histogram of Equalized Image')

plt.hist(equalized_image.flatten(), bins=256, range=[0, 256], color='red')

plt.xlabel('Pixel Intensity') plt.ylabel('Frequency') plt.show()
```

Output:



Histogram Equalized Image



Without inbuilt function:

Algorithm:

Code:

gray_image = 0.2989*image[:,:,2] + 0.5870*image[:,:,1] + 0.1140*image[:,:,0]

hist, bins = np.histogram(gray_image.flatten(), bins=256, range=[0, 256]) cdf = hist.cumsum()

cdf_normalized = cdf * 255 / cdf.max() equalized_image = np.interp(gray_image.flatten(), bins[:-1], cdf_normalized) equalized_image = equalized_image.reshape(gray_image.shape) plt.title('Original Grayscale Image') plt.imshow(gray_image, cmap='gray')

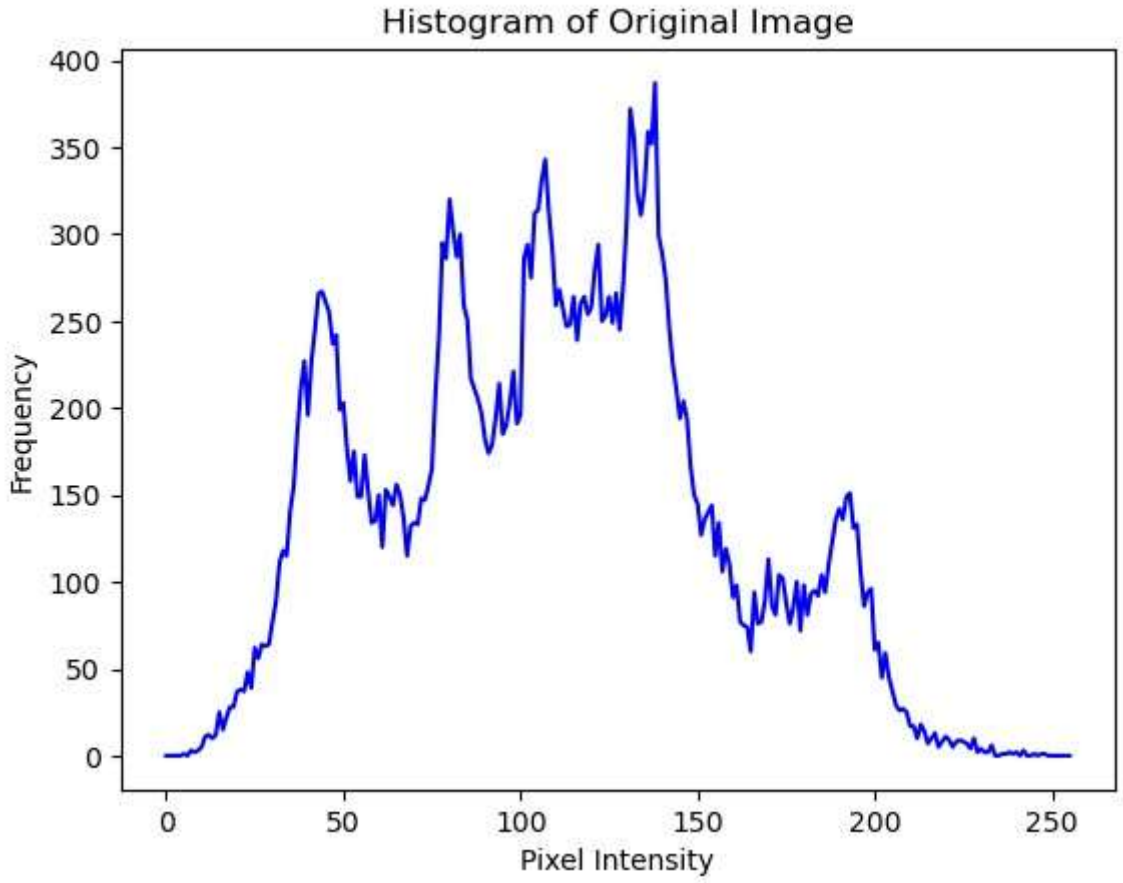
plt.axis('off') plt.show()

plt.title('Histogram of

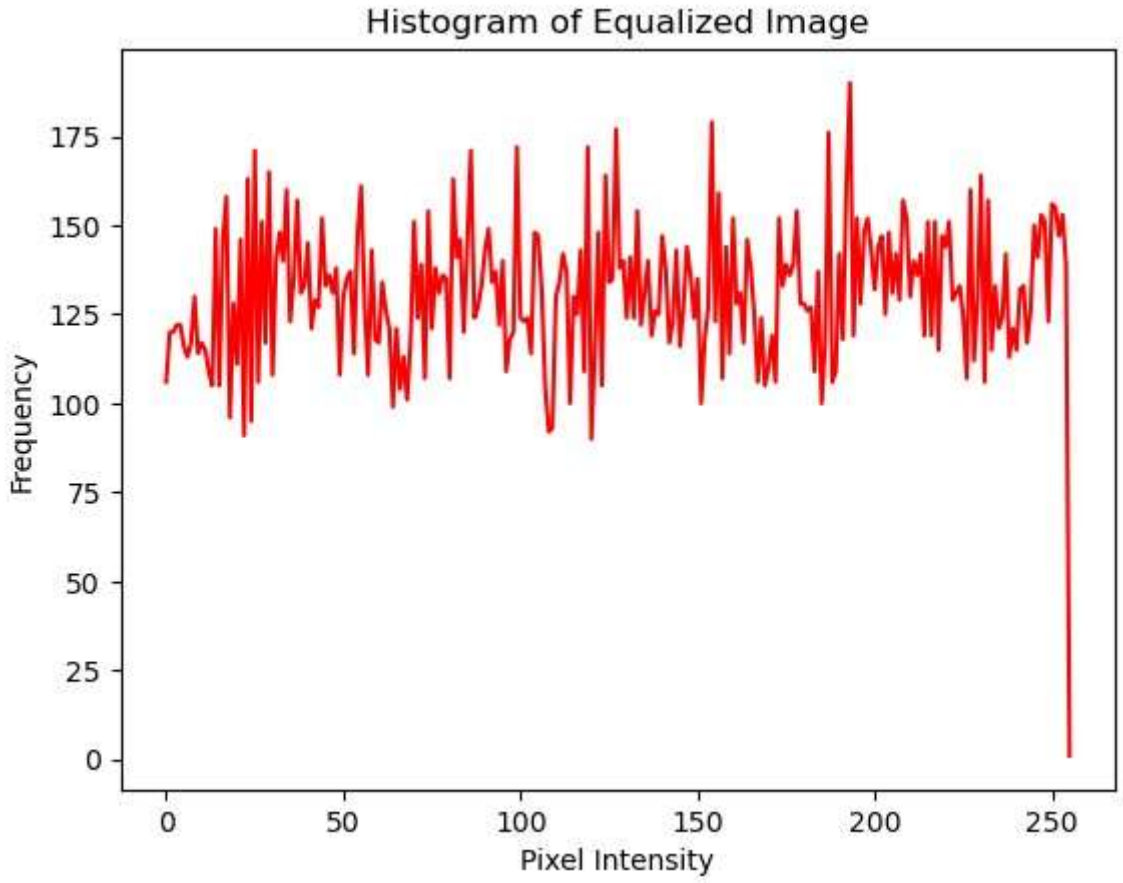
```
Original Image')
plt.plot(hist,
color='blue')
plt.xlabel('Pixel
Intensity')
plt.ylabel('Frequency')
plt.show()

plt.title('Histogram Equalized Image') plt.imshow(equalized_image,
cmap='gray')
plt.axis('off') plt.show()

equalized_hist, _ = np.histogram(equalized_image.flatten(), bins=256,
range=[0, 256]) plt.title('Histogram of Equalized Image') plt.plot(
equalized_hist, color='red') plt.xlabel('Pixel Intensity')
plt.ylabel('Frequency') plt.show()
```

Output:


Original Grayscale Image


Histogram of Original Image
Frequency
Pixel Intensity
0
50
100
150
200
250
300
350
400


Histogram Equalized Image


Histogram of Equalized Image
Frequency
175
150
125
100
75
50
25
0
0
50
100
150
200
250
Pixel Intensity

Read the normal image:

Code:

```
import numpy as np from matplotlib.image import
imread  image =
imread(r"C:\Users\madhu\Downloads\C.jpg") image
```

output:

```
array([[[ 84, 173, 253],
        [ 84, 173, 253],
        [ 84, 173, 253],
        ...,
        [184, 211, 254],
        [184, 211, 254],
        [184, 211, 254]],

       [[ 84, 173, 253],
        [ 84, 173, 253],
        [ 84, 173, 253],
        ...,
        [184, 211, 254],
        [184, 211, 254],
        [184, 211, 254]],

       [[ 84, 173, 253],
        [ 84, 173, 253],
        [ 84, 173, 253],
        ...,
        [184, 211, 254],
        [184, 211, 254],
        [184, 211, 254]],

       ...,

       [[200, 179, 152],
        [200, 179, 152],
        [199, 178, 151],
        ...,
        [113,  84,  52],
        [155, 126,  94],
        [201, 172, 140]],
```

```
[[221, 200, 173],
 [221, 200, 173],
 [222, 201, 174],
 ...,
 [109,  80,  48],
 [138, 109,  77],
 [173, 144, 112]],

[[203, 182, 155],
 [204, 183, 156],
 [206, 185, 158],
 ...,
 [106,  77,  45],
 [121,  92,  60],
 [146, 117,  85]]], dtype=uint8)
```

Show the image:

Code: import cv2 import matplotlib.pyplot as plt

image =

imread(r"C:\Users\madhu\Downloads\C.jpg")

plt.imshow(image)

plt.axis('off')

plt.show() **output:**

Resize the image:

Code:

```
resized_image = cv2.resize(image, (100, 100))  # Resize to 500x500 pixels
plt.imshow(resized_image)
plt.axis('off')
plt.show()
```

output:

Dimensions of the image:

Code:

```
height, width, channels = image.shape
print("height:",height)
print("width:",width)
print("channel:",channels)
```

Output:

```
height: 4190
width: 7139
channel: 3
```

Converting BGR to RGB:

Code:

```
image_rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
plt.imshow(image_rgb)
plt.axis('off')
plt.show()
```

output:

Converting BGR to GRAY:

Code: gray_image = cv2.cvtColor(image,

cv2.COLOR_BGR2GRAY) plt.imshow(gray_image,

cmap='gray')

plt.axis('off')

plt.show() **Output:**

Without inbuilt function:

Code: import cv2 import numpy as np import matplotlib.pyplot as plt height, width, channels = image_rgb.shape grayscale_image = np.zeros((height, width), dtype=np.uint8) for i in range(height): for j in range(width): r = image_rgb[i, j, 0] g = image_rgb[i, j, 1] b = image_rgb[i, j, 2] grayscale_value = int(0.2989 * r + 0.5870 * g + 0.1140 * b) grayscale_image[i, j] = grayscale_value plt.imshow(grayscale_image, cmap='gray') plt.axis('off') plt.show() print(grayscale_image)

Output:

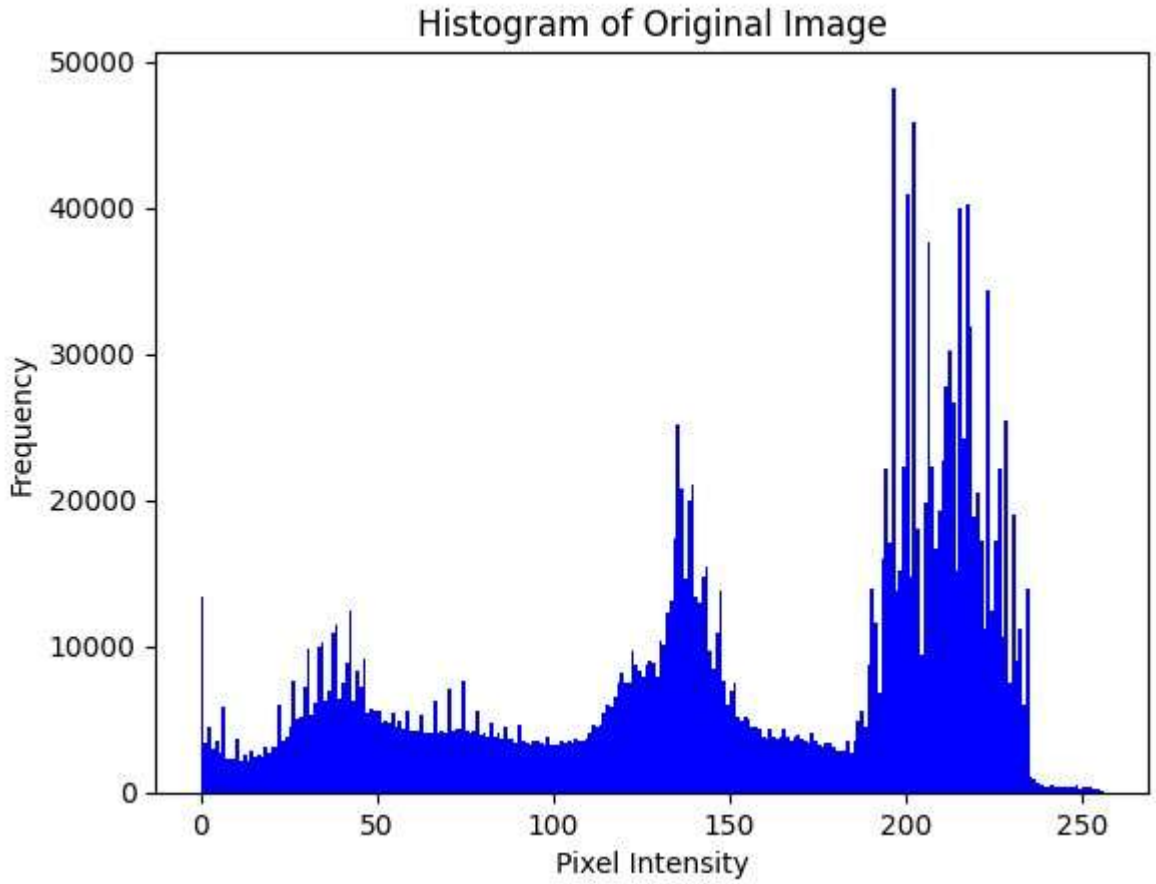
Histogram equalization:

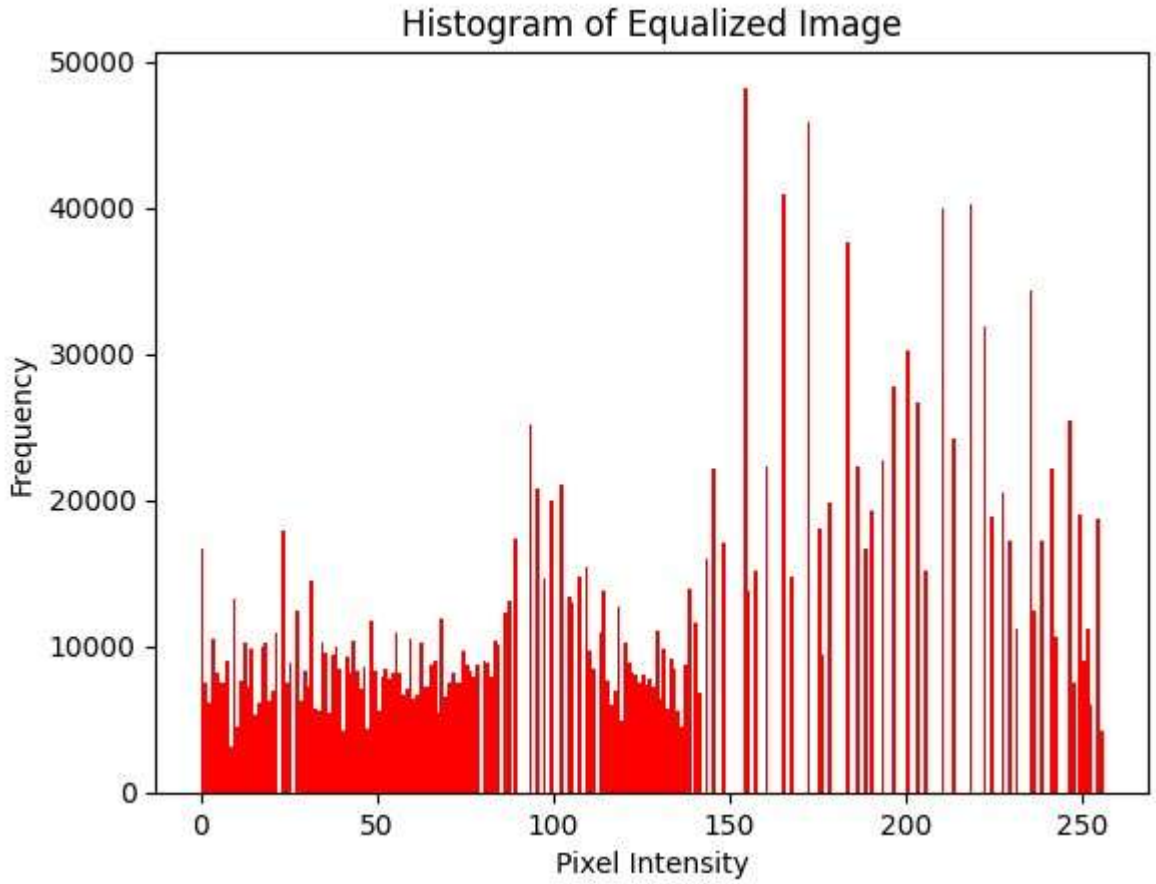
With inbuilt function:

Code: gray_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY) equalized_image = cv2.equalizeHist(gray_image) plt.title('Original Grayscale Image') plt.imshow(gray_image, cmap='gray') plt.axis('off') plt.show() plt.title('Histogram of Original Image') plt.hist(gray_image.flatten(), bins=256, range=[0, 256], color='blue') plt.xlabel('Pixel Intensity') plt.ylabel('Frequency') plt.show() plt.title('Histogram Equalized Image') plt.imshow(equalized_image, cmap='gray')

plt.axis('off') plt.show() plt.title('Histogram of Equalized Image') plt.hist(equalized_image.flatten(), bins=256, range=[0, 256], color='red') plt.xlabel('Pixel Intensity') plt.ylabel('Frequency') plt.show()

Output:


Histogram of Original Image
Frequency
Pixel Intensity
0
10000
20000
30000
40000
50000
0
50
100
150
200
250



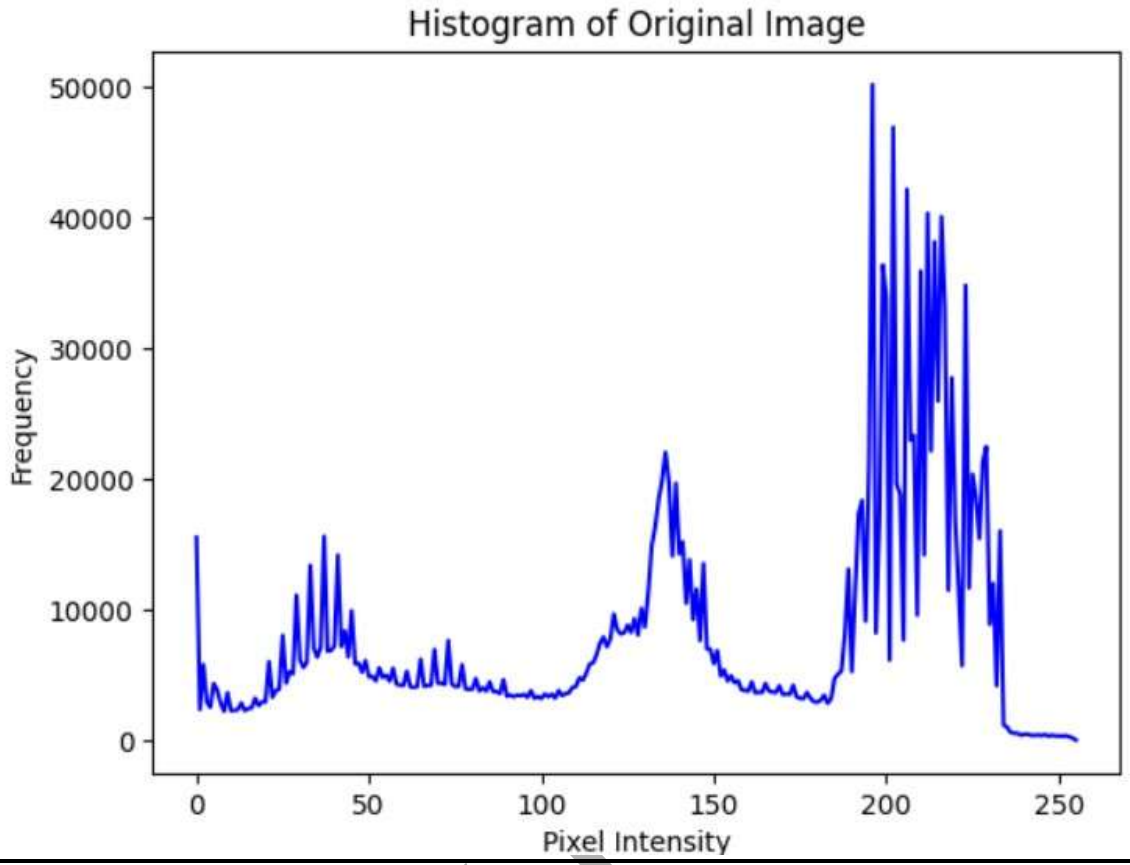
Without inbuilt function:

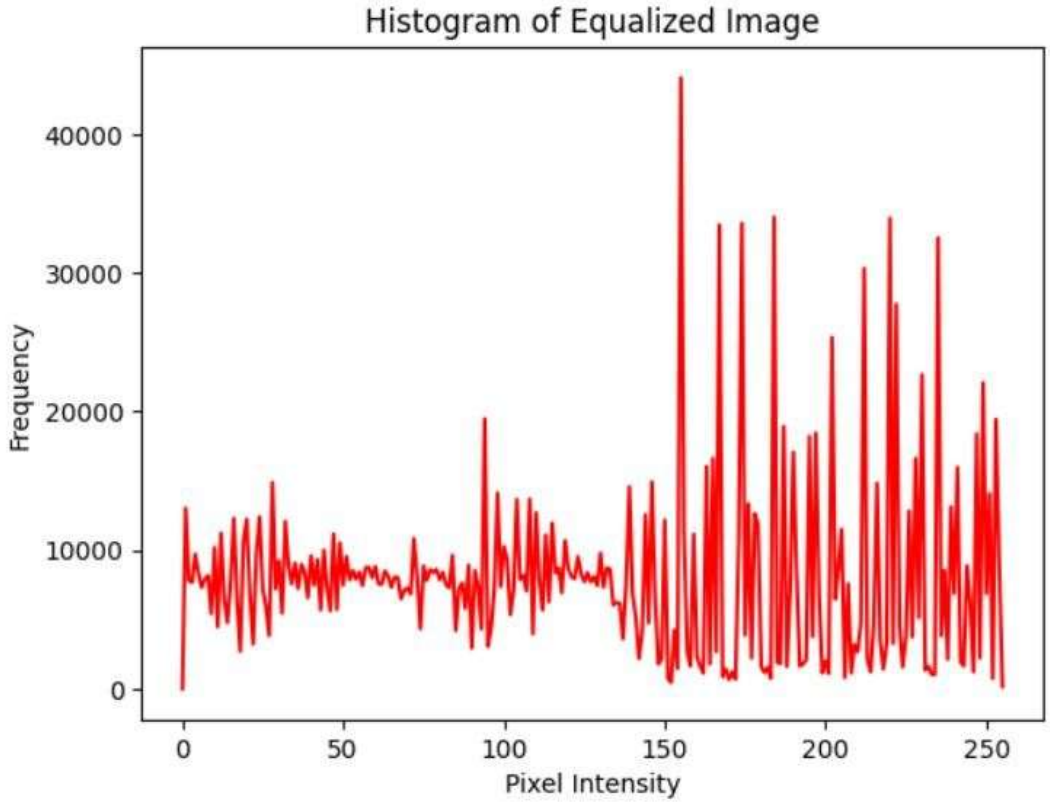
Code:

```
gray_image = 0.2989*image[:,:,2] + 0.5870*image[:,:,1] + 0.1140*image[:,:,0]
hist, bins = np.histogram(gray_image.flatten(), bins=256, range=[0, 256]) cdf
= hist.cumsum() cdf_normalized = cdf * 255 / cdf.max() equalized_image =
np.interp(gray_image.flatten(), bins[:-1], cdf_normalized) equalized_image =
equalized_image.reshape(gray_image.shape)


plt.title('Original Grayscale Image')
plt.imshow(gray_image,
cmap='gray') plt.axis('off') plt.show()
```

```
plt.title('Histogram of Original Image')
plt.plot(hist, color='blue')
plt.xlabel('Pixel Intensity')
plt.ylabel('Frequency') plt.show()

plt.title('Histogram Equalized Image') plt.imshow(equalized_image,
cmap='gray')
plt.axis('off') plt.show()
equalized_hist, _ = np.histogram(equalized_image.flatten(), bins=256,
range=[0, 256])

plt.title('Histogram of Equalized Image')
plt.plot( equalized_hist, color='red')
plt.xlabel('Pixel Intensity')
plt.ylabel('Frequency') plt.show()
```

Output:


Histogram of Original Image
Frequency
Pixel Intensity
50000
40000
30000
20000
10000
0
0
50
100
150
200
250



Functions:

Imread: The imread function reads an image from a specified file path and returns it as a NumPy array for further processing or display.

Resize: The resize function resizes an image to the specified width and height, returning a new image with the updated dimensions.

Shape: The shape attribute provides the dimensions of an image or array, such as its height, width, and the number of color channels.

Cvtcolor: The cvtColor function converts an image from one color space to another (e.g., BGR to RGB or BGR to Grayscale).

Imshow: The imshow function displays an image in a plot or window, allowing you to visualize it.

equalizeHist :The equalizeHist function improves the contrast of a grayscale image by redistributing the pixel intensity values to cover the full range. **np.histogram**: Compute the histogram of the grayscale image, showing the distribution of pixel intensities. **hist.cumsum**: Compute the cumulative sum of the histogram to create the cumulative distribution function (CDF).

cdf * 255 / cdf.max: Normalize the CDF to the full range of pixel intensities [0, 255]. **np.interp**: Interpolate the original pixel intensities to their new values based on the normalized CDF for histogram equalization.

INFERENCE:

By using many function we can done the basic operation read,resize,converting one form to another.

RESULT:

The basic operation in the lenna and normal image has been analysed.

EX NO:2 DATE:23/01/2025	COLOUR SPACE VISUALIZATION

AIM:

To perform colour spaces on images like CMY, HSV,YIQ etc..

Image:1 Read the image:

Algorithm:

1. Import cv2 and matplotlib.pyplot
2. Read the image from "C:\Users\student\Downloads\lenna.jpg"
3. Store the image in the variable "image"
4. Display the image using plt.imshow(image)
5. Turn off the axis with plt.axis('off')
6. Show the image with plt.show() **Code:** import cv2

import matplotlib.pyplot as plt image = cv2.imread(r"

C:\Users\madhu\Downloads\download.jpeg") plt.imshow(image)

plt.axis('off')

plt.show() **output:**

Image:2

Converting BGR to RGB Algorithm:

1. Import cv2 and matplotlib.pyplot.

2. Read the image using cv2.imread() into "image".

3. Convert the image from BGR to RGB using:

   image_rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

4. Display the RGB image using plt.imshow(image_rgb)

5. Turn off the axis with plt.axis('off')

6. Show the image with plt.show()

7. Print the RGB image data using print(image_rgb) **Code:** image_rgb =

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

plt.axis('off')

plt.show()

print(image_rgb)

Output:

Image:2

Visualization of RGB on separate colour space:

Algorithm:

1. **Copy original image**:
   red_plane = image_rgb.copy(), green_plane = image_rgb.copy(),
   blue_plane = image_rgb.copy()
2. **Modify red plane**: red_plane[:, :, 1] = 0, red_plane[:, :, 2] = 0
3. **Modify green plane**: green_plane[:, :, 0] = 0, green_plane[:, :, 2] = 0
4. **Modify blue plane**:
   blue_plane[:, :, 0] = 0, blue_plane[:, :, 1] = 0
5. **Plot images**: plt.subplot(2, 2, 1), plt.imshow(image_rgb),
   plt.title("Original Image"), plt.axis('off')
   plt.subplot(2, 2, 2), plt.imshow(red_plane), plt.title("Red Plane"),
   plt.axis('off')

plt.subplot(2, 2, 3), plt.imshow(green_plane), plt.title("Green Plane"), plt.axis('off')
plt.subplot(2, 2, 4), plt.imshow(blue_plane), plt.title("Blue Plane"), plt.axis('off') plt.show()

Code:

```
red_plane = image_rgb.copy()
red_plane[:, :, 0] = 0
red_plane[:, :, 1] = 0
green_plane = image_rgb.copy()
green_plane[:, :, 0] = 0
green_plane[:, :, 2] = 0
blue_plane = image_rgb.copy()
blue_plane[:, :, 1] = 0
blue_plane[:, :, 2] = 0
plt.subplot(2, 2, 1)
plt.imshow(image_rgb)
plt.title("Original Image")
plt.axis('off')

plt.subplot(2, 2, 2)
plt.imshow(red_plane)
plt.title("Red Plane")
plt.axis('off')
```

```
plt.subplot(2, 2, 3)
plt.imshow(green_plane
) plt.title("Green Plane")
plt.axis('off')
plt.subplot(2, 2, 4)
plt.imshow(blue_plane)
plt.title("Blue Plane")
plt.axis('off') plt.show()
```

output:

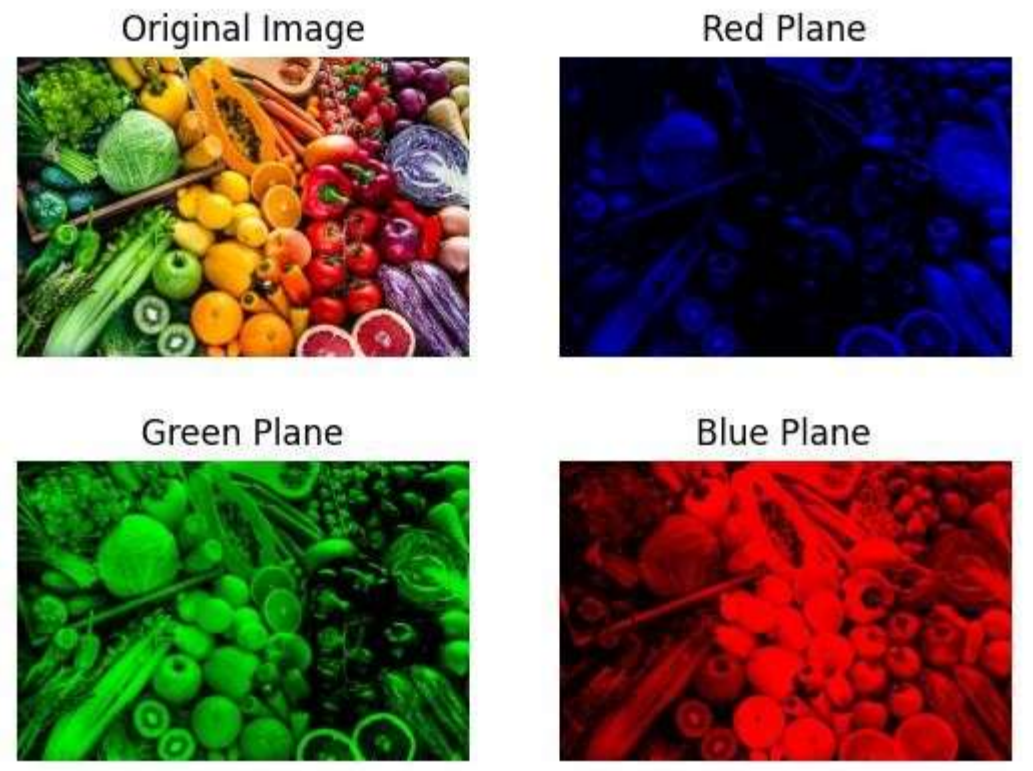
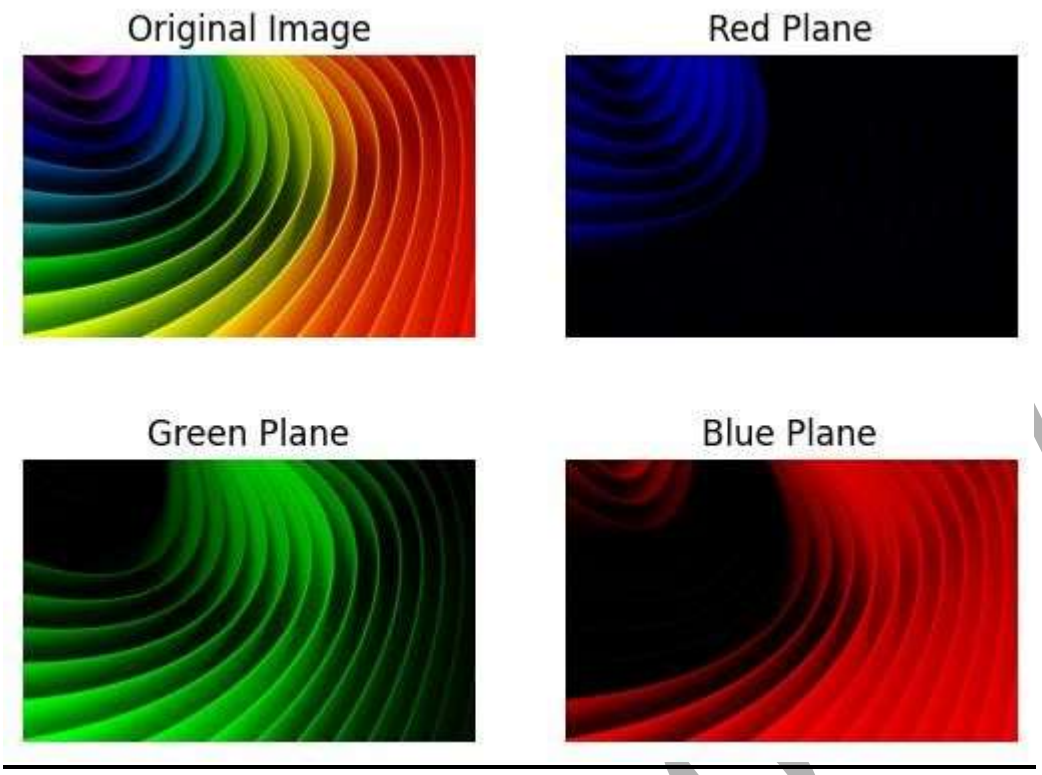


Image:2



Conversion of RGB to a CMY image:

Algorithm:

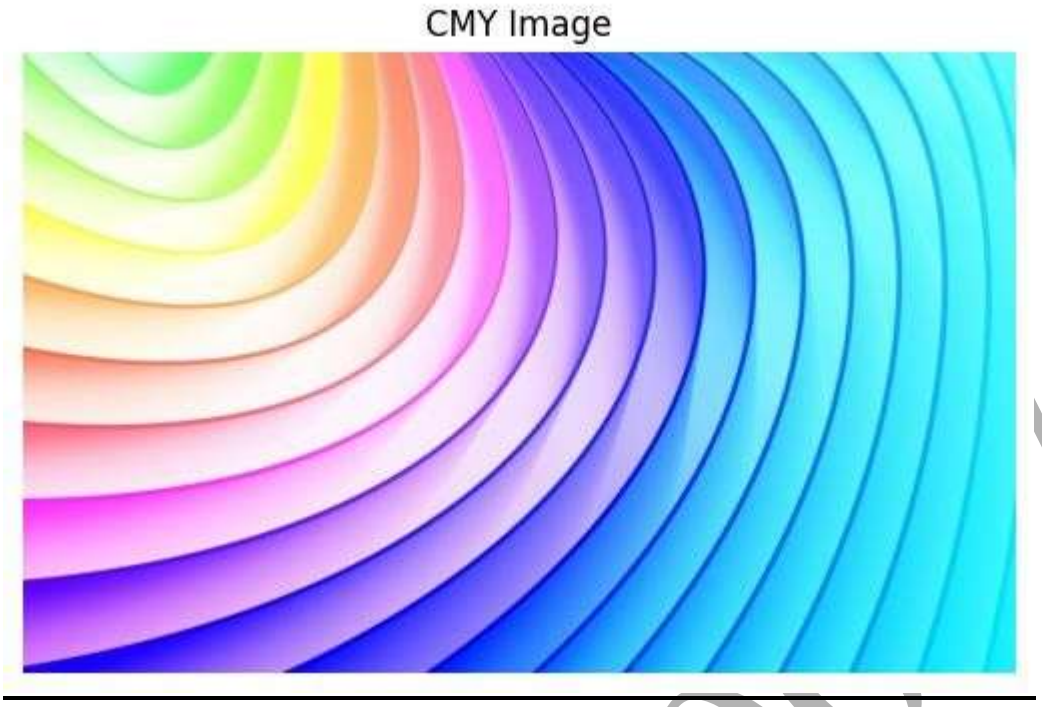
1. **Normalize the image**: image_rgb_normalized = image_rgb / 255.0
2. **Convert to CMY**: cmy_image = 1 - image_rgb_normalized
3. **Scale back to [0, 255] and convert type**: cmy_image = (cmy_image * 255).astype('uint8')
4. **Display the image**: plt.imshow(cmy_image), plt.axis('off'), plt.title("CMY Image"), plt.show() **Code:**

```
image_rgb_normalized = image_rgb / 255.0

cmy_image = 1 - image_rgb_normalized

cmy_image = (cmy_image * 255).astype('uint8')

plt.imshow(cmy_image)
```

```
plt.axis('off')
plt.title("CMY Image")
plt.show()
```

output:

Image:2



Conversion of RGB to a CMYK extension image:

Algorithm:

1. **Normalize the image**:
   image_rgb_normalized = image_rgb / 255.0
   Converts the RGB values to the range [0, 1].

2. **Calculate CMY channels**:
   C = 1 - image_rgb_normalized[:, :, 0]
   M = 1 - image_rgb_normalized[:, :, 1]
   Y = 1 - image_rgb_normalized[:, :, 2]
   Converts the RGB values to Cyan, Magenta, and Yellow by subtracting each channel value from 1.

3. **Calculate Key (Black) channel**:
   K = np.min([C, M, Y])
   Finds the minimum value among the C, M, and Y channels to determine the black (K) channel (though it's not used in the image display here).

4. **Display the CMY image**:
   plt.imshow(np.stack([C, M, Y], axis=-1))

Stacks the C, M, and Y channels together to form a CMY image and displays it.

5. **Remove axis and add title**: plt.axis('off') plt.title("CMY Image")
   Removes axis labels and adds a title to the plot.
6. **Show the plot**: plt.show()
   Displays the CMY image.

Code:

```
import numpy as np

image_rgb_normalized = image_rgb / 255.0

C = 1 - image_rgb_normalized[:, :, 0]

M = 1 - image_rgb_normalized[:, :, 1]

Y = 1 - image_rgb_normalized[:, :, 2] K

= np.min([C, M, Y])

plt.imshow(np.stack([C, M, Y], axis=-1))

plt.axis('off')

plt.title("CMY

Image") plt.show()
```

Output:

CMY Image

CMY Image

Conversion of an RGB image to HSV image:

Algorithm:

1. **Convert RGB to HSV**: image_hsv = cv2.cvtColor(image, cv2.COLOR_RGB2HSV)

Converts the RGB image to HSV (Hue, Saturation, Value) color space using OpenCV.

2. **Display the RGB image**:

plt.subplot(1, 2, 1), plt.imshow(image_rgb), plt.title("RGB Image"), plt.axis('off')

Displays the original RGB image in the first subplot.

3. **Display the HSV image**:

plt.subplot(1, 2, 2), plt.imshow(image_hsv), plt.title("HSV Image"), plt.axis('off')

Displays the converted HSV image in the second subplot.

4. **Show the plot**:

plt.show()

Displays both images side by side.

Code: image_hsv = cv2.cvtColor(image,

cv2.COLOR_RGB2HSV) plt.subplot(1, 2, 1)

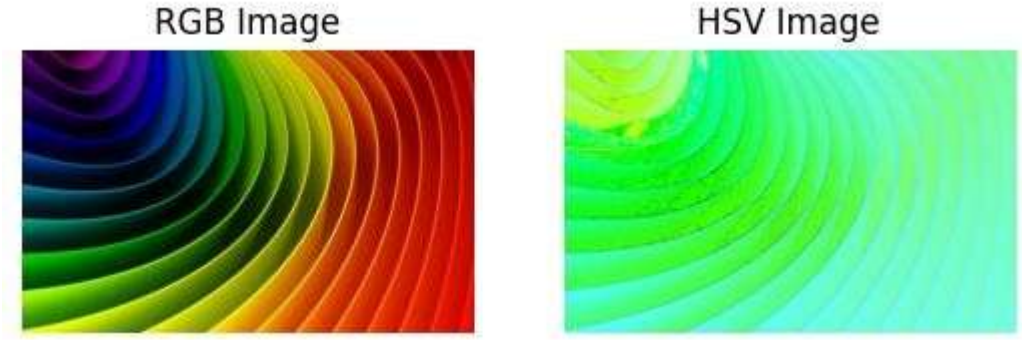
plt.imshow(image_rgb) plt.title("RGB Image") plt.axis('off')

plt.subplot(1, 2, 2) plt.imshow(image_hsv) plt.title("HSV

Image") plt.axis('off') plt.show()

Output:

Image:2



To visualize the HSV image as a separate channel:

Algorithm:

1. **Split the HSV channels**:
hue, saturation, value = cv2.split(image_hsv)
Splits the HSV image into three separate channels: Hue, Saturation, and Value.

2. **Create the plot for the channels**: plt.figure(figsize=(12, 4))
Sets up a figure with a specified size for displaying the channels.

3. **Display the Hue channel**:
plt.subplot(1, 3, 1), plt.imshow(hue, cmap='hsv'), plt.title("Hue Channel"), plt.axis('off')
Displays the Hue channel using the 'hsv' colormap.

4. **Display the Saturation channel**:
plt.subplot(1, 3, 2), plt.imshow(saturation, cmap='gray'), plt.title("Saturation Channel"), plt.axis('off')
Displays the Saturation channel in grayscale.

5. **Display the Value channel**:
plt.subplot(1, 3, 3), plt.imshow(value, cmap='gray'), plt.title("Value Channel"), plt.axis('off')
Displays the Value channel in grayscale.

6. **Show the plot**:
plt.show()
Displays all three channels side by side.

7. **Get and print HSV values at a specific pixel**: pixel_hsv = image_hsv[y, x]
print(f"HSV values at pixel ({x}, {y}): Hue = {pixel_hsv[0]}, Saturation = {pixel_hsv[1]}, Value = {pixel_hsv[2]}")
Retrieves the HSV values of the pixel at coordinates (100, 100) and prints them out. **Code:**

```
hue, saturation, value =
cv2.split(image_hsv) plt.figure(figsize=(12,
4)) plt.subplot(1, 3, 1) plt.imshow(hue,
cmap='hsv') plt.title("Hue Channel")
plt.axis('off') plt.subplot(1, 3, 2)
plt.imshow(saturation,cmap='gray')
plt.title("Saturation Channel") plt.axis('off')
plt.subplot(1, 3, 3) plt.imshow(value,
cmap='gray') plt.title("Value Channel")
plt.axis('off') plt.show()

x, y = 100, 100   pixel_hsv
= image_hsv[y, x]
print(f"HSV values at pixel ({x}, {y}): Hue = {pixel_hsv[0]}, Saturation
= {pixel_hsv[1]}, Value = {pixel_hsv[2]}")
```

Output:


```
HSV values at pixel (100, 100): Hue = 60, Saturation = 255, Value = 4
```


```
HSV values at pixel (100, 100): Hue = 158, Saturation = 237, Value = 140
```

To visualize the HSV image by creating an empty image:

Algorithm:

1. **Create a blank HSV image**: image_hsv = np.zeros((height, width, 3), dtype=np.uint8)

Creates an image of size 100x100 with all pixel values initialized to zero (black) in the HSV color space.

2. **Set the Hue, Saturation, and Value for the first image**: image_hsv[:, :, 0] = 30 image_hsv[:, :, 1] = 255 image_hsv[:, :, 2] = 255

Sets the Hue to 30, Saturation to 255, and Value to 255 (a bright orange color in HSV space).

3. **Convert the HSV image to RGB**:

image_bgr = cv2.cvtColor(image_hsv, cv2.COLOR_HSV2RGB)

Converts the HSV image to an RGB image (OpenCV uses BGR by default).

4. **Display the first image**: plt.imshow(image_bgr), plt.title("Hue=30, Saturation=255, Value=255"), plt.axis('off'), plt.show()

Displays the first image with Hue = 30, Saturation = 255, and Value = 255 (bright orange).

5. **Set the Hue, Saturation, and Value for the second image**:

image_hsv[:, :, 0] = 120
image_hsv[:, :, 1] = 128
image_hsv[:, :, 2] = 255
Sets the Hue to 120, Saturation to 128, and Value to 255 (a greenish color).

6. **Convert and display the second image**:

image_bgr = cv2.cvtColor(image_hsv, cv2.COLOR_HSV2RGB)
Converts the second HSV image to RGB and displays it.

7. **Set the Hue, Saturation, and Value for the third image**: image_hsv[:, :, 0] = 0 image_hsv[:, :, 1] = 255 image_hsv[:, :, 2] = 50

Sets the Hue to 0 (red), Saturation to 255 (fully saturated), and Value to 50 (dark red).

8. **Convert and display the third image**: image_bgr = cv2.cvtColor(image_hsv, cv2.COLOR_HSV2RGB) Converts the third HSV image to RGB and displays it.

Code:

```
import numpy as np

height, width = 50, 50

image_hsv = np.zeros((height, width, 3), dtype=np.uint8)

image_hsv[:, :, 0] = 30   image_hsv[:, :, 1] = 255   image_hsv[:,

:, 2] = 255   image_bgr = cv2.cvtColor(image_hsv,

cv2.COLOR_HSV2RGB) plt.figure(figsize=(5, 5))

plt.imshow(image_bgr) plt.title("Hue=30, Saturation=255,

Value=255 ") plt.axis('off') plt.show() image_hsv[:, :, 0] = 120

image_hsv[:, :, 1] = 128   image_hsv[:, :, 2] = 255   image_bgr =

cv2.cvtColor(image_hsv, cv2.COLOR_HSV2RGB)

plt.figure(figsize=(5, 5)) plt.imshow(image_bgr)

plt.title("Hue=120, Saturation=128, Value=255 ") plt.axis('off')
```

plt.show() image_hsv[:, :, 0] = 0 image_hsv[:, :, 1] = 255

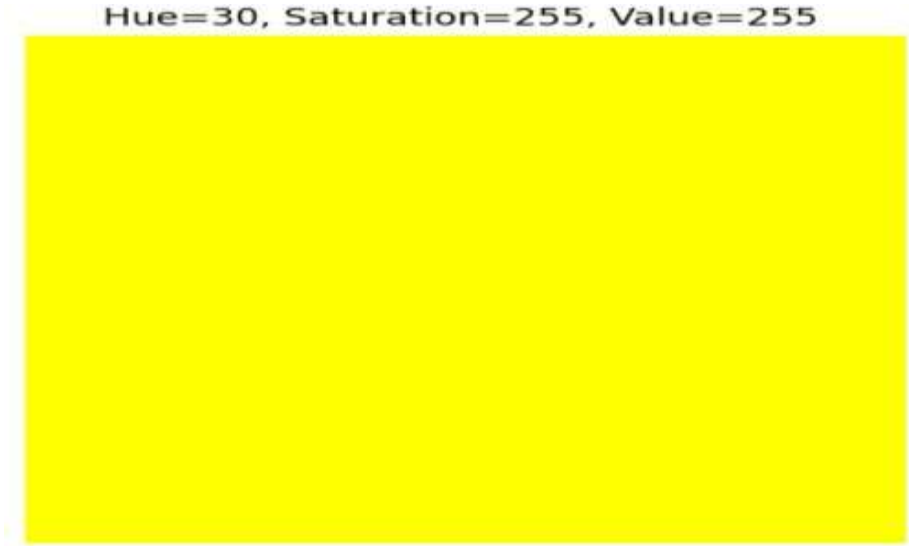
image_hsv[:, :, 2] = 50

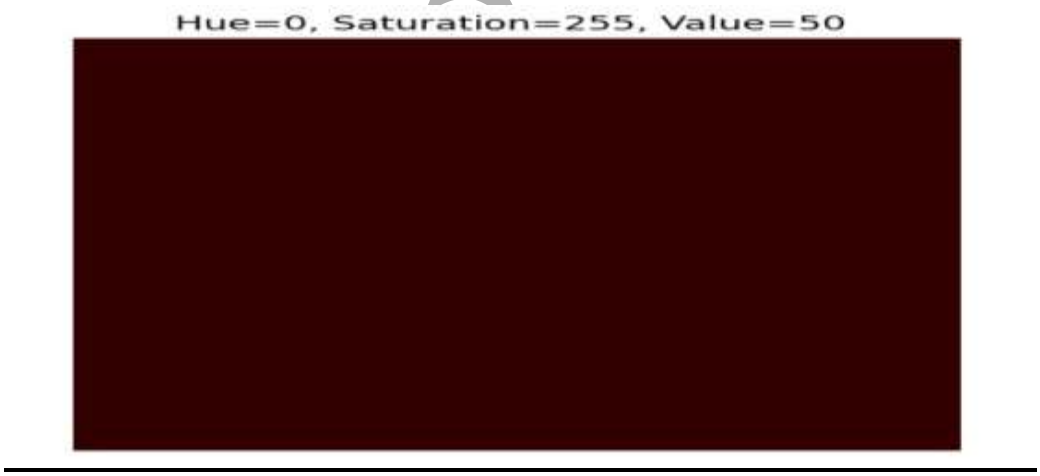
image_bgr = cv2.cvtColor(image_hsv,

cv2.COLOR_HSV2RGB) plt.figure(figsize=(5, 5))

plt.imshow(image_bgr) plt.title("Hue=0, Saturation=255,

Value=50 ") plt.axis('off') plt.show() **Output:**





By doubling the value of saturation:

Algorithm:

1. **Convert BGR to HSV**:

image_hsv = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)
Converts the image from BGR to HSV.

2. **Split HSV channels**:

hue, saturation1, value = cv2.split(image_hsv)

Splits the image into Hue, Saturation, and Value channels.

3. **Store original Saturation**: saturation_before = saturation1.copy() Saves the original Saturation channel.

4. **Double Saturation**:

image_hsv[:, :, 1] = cv2.min(image_hsv[:, :, 1] * 2, 255)
Doubles the Saturation, clamping values at 255.

5. **Split again after modification**: hue, saturation2, value = cv2.split(image_hsv) Splits the updated image into channels again.

6. **Store modified Saturation**:

saturation_after = saturation2.copy()
Saves the modified Saturation channel.

7. **Convert back to RGB**: image_rgb_modified = cv2.cvtColor(image_hsv, cv2.COLOR_HSV2RGB) Converts the updated HSV image back to RGB.

8. **Plot the images and channels**: Displays:

   - Original image
   - Image with doubled saturation
   - Hue, Saturation (before and after), and Value channels.

9. **Show plot**: plt.show()

Displays all the plots.

Code: image_hsv = cv2.cvtColor(image,

cv2.COLOR_BGR2HSV) hue, saturation1, value =

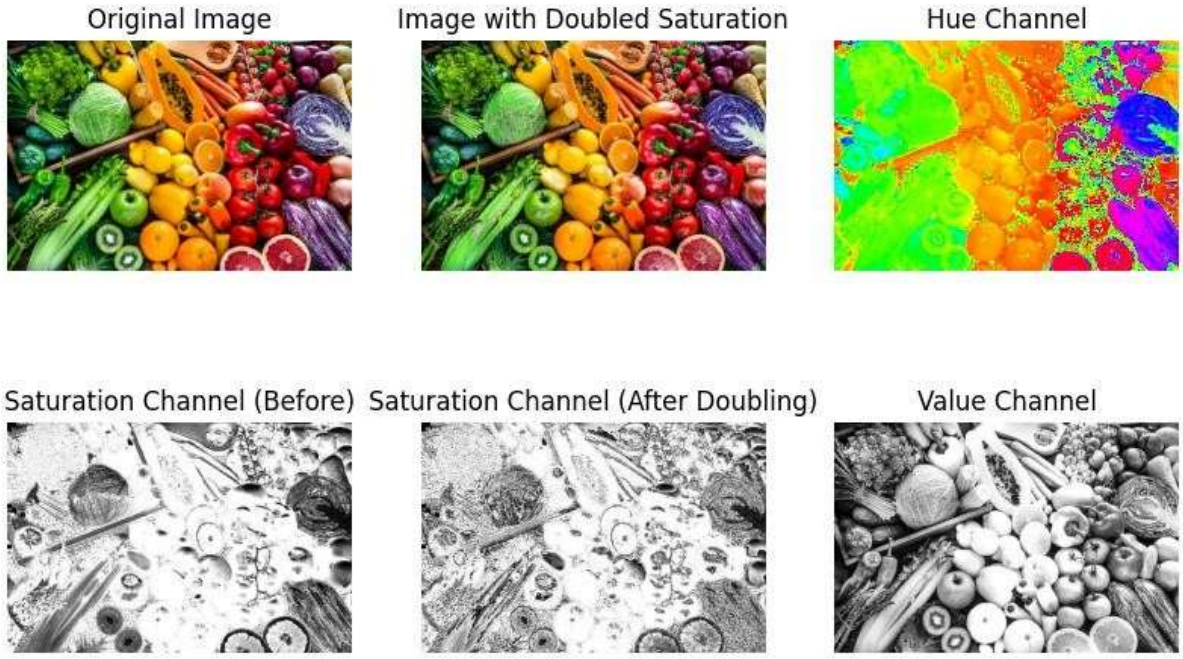
cv2.split(image_hsv) saturation_before = saturation1.copy()

image_hsv[:, :, 1] = cv2.min(image_hsv[:, :, 1] * 2, 255)

```
hue,saturation2,value = cv2.split(image_hsv)
saturation_after = saturation2.copy()

image_rgb_modified = cv2.cvtColor(image_hsv,
cv2.COLOR_HSV2RGB) plt.figure(figsize=(10, 6)) plt.subplot(2, 3, 1)
plt.imshow(cv2.cvtColor(image,
cv2.COLOR_BGR2RGB)) plt.title("Original Image")
plt.axis('off') plt.subplot(2, 3, 2)
plt.imshow(image_rgb_modified) plt.title("Image with
Doubled Saturation") plt.axis('off') plt.subplot(2, 3, 3)
plt.imshow(hue, cmap='hsv') plt.title("Hue Channel")
plt.axis('off') plt.subplot(2, 3, 4)
plt.imshow(saturation_before, cmap='gray')
plt.title("Saturation Channel (Before)")
plt.axis('off') plt.subplot(2, 3, 5)
plt.imshow(saturation_after, cmap='gray')
plt.title("Saturation Channel (After
Doubling)") plt.axis('off') plt.subplot(2, 3, 6)
plt.imshow(value, cmap='gray') plt.title("Value
Channel") plt.axis('off') plt.show()
x, y = 100, 100   pixel_hue = hue[y, x]
pixel_saturation_before = saturation_before[y, x]
pixel_saturation_after = saturation_after[y, x] pixel_value =
value[y, x] print(f"HSV values at pixel ({x}, {y}):")
print(f"Hue: {pixel_hue}") print(f"Saturation (Before):
```

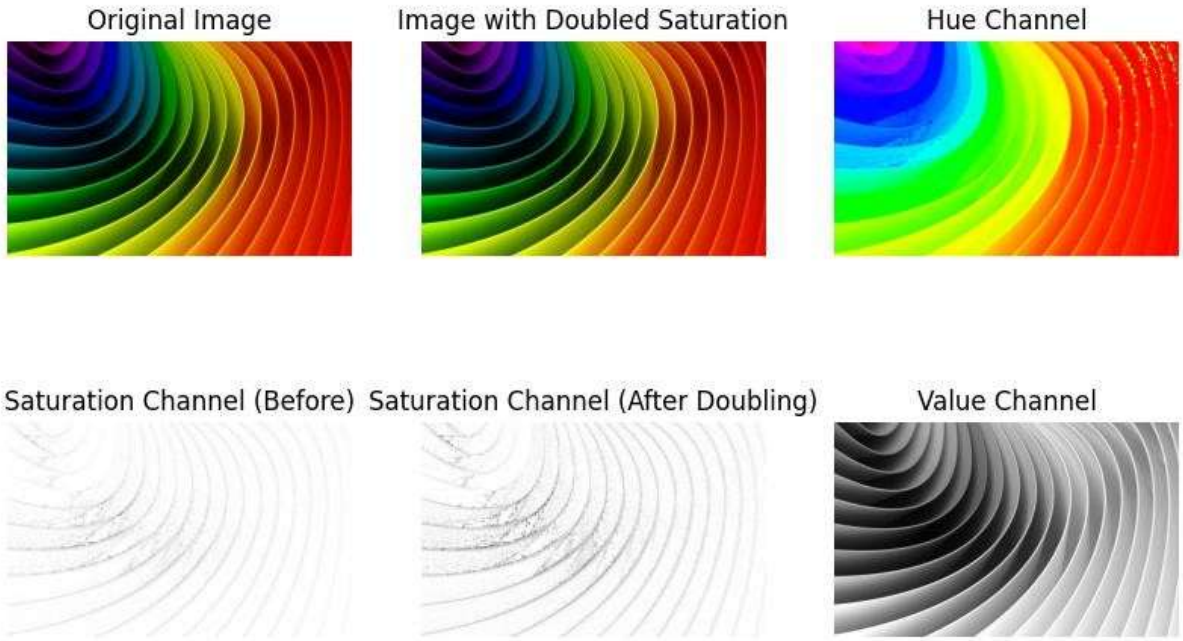
{pixel_saturation_before}") print(f"Saturation (After Doubling): {pixel_saturation_after}") print(f"Value: {pixel_value}")

Output:




```
HSV values at pixel (100, 100):
Hue: 60
Saturation (Before): 255
Saturation (After Doubling): 254
Value: 4
```

Image:2




```
HSV values at pixel (100, 100):
Hue: 142
Saturation (Before): 237
Saturation (After Doubling): 218
Value: 140
```

By shifted to -0.2:

Algorithm:

1. **Convert image from BGR to HSV**:

image_hsv = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)
Converts the BGR image to HSV color space.

2. **Split HSV into channels**:

hue, saturation, value = cv2.split(image_hsv)
Splits the HSV image into Hue, Saturation, and Value channels.

3. **Calculate hue shift**: hue_shift = -0.2 * 179

Computes the shift amount for the Hue channel (by -0.2 times the maximum Hue value of 179).

4. **Apply hue shift**:

hue_after = (hue + hue_shift) % 180

Shifts the Hue channel and ensures it stays within the valid range [0, 180] by using modulo.

5. **Merge modified channels**:

image_hsv_shifted = cv2.merge([hue_after.astype(np.uint8), saturation, value])
Merges the shifted Hue with the original Saturation and Value channels.

6. **Convert back to RGB**: image_rgb_shifted = cv2.cvtColor(image_hsv_shifted, cv2.COLOR_HSV2RGB)

Converts the modified HSV image back to RGB.

7. **Plot the original and shifted images**:

Displays the original image and the image with the hue shifted by -0.2.
plt.subplot(1, 2, 1), plt.imshow(cv2.cvtColor(image, cv2.COLOR_BGR2RGB))
plt.subplot(1, 2, 2), plt.imshow(image_rgb_shifted)

8. **Print original and shifted Hue values at pixel (100, 100)**:

original_hue = hue[y, x], shifted_hue = hue_after[y, x]
Prints the Hue values at pixel (100, 100) before and after the shift.

9. **Plot original and shifted Hue channels**:

Displays the Hue channel before and after the shift using the 'hsv' colormap. plt.subplot(1, 2, 1), plt.imshow(hue, cmap='hsv') plt.subplot(1, 2, 2), plt.imshow(hue_after, cmap='hsv')

10. **Show the plots**:

plt.show()
Displays all the plots.

Code: image_hsv = cv2.cvtColor(image,

cv2.COLOR_BGR2HSV)

hue, saturation, value = cv2.split(image_hsv)

hue_shift = -0.2 * 179

hue_after = (hue + hue_shift) % 180

image_hsv_shifted = cv2.merge([hue_after.astype(np.uint8), saturation, value])

image_rgb_shifted = cv2.cvtColor(image_hsv_shifted, cv2.COLOR_HSV2RGB)

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

```
plt.subplot(1, 2, 1)
plt.imshow(cv2.cvtColor(image,
cv2.COLOR_BGR2RGB))
plt.title("Original Image")
plt.axis('off') plt.subplot(1, 2, 2)
plt.imshow(image_rgb_shifted)
plt.title("Image with Hue Shifted by -
0.2") plt.axis('off') plt.show() x, y =
100, 100   original_hue = hue[y, x]
shifted_hue = hue_after[y, x]
print(f"Original Hue value at pixel ({x}, {y}):
{original_hue}") print(f"Shifted Hue value at pixel ({x}, {y}):
{shifted_hue}") plt.figure(figsize=(12, 6)) plt.subplot(1, 2, 1)
plt.imshow(hue, cmap='hsv') plt.title("Original Hue Channel")
plt.axis('off') plt.subplot(1, 2, 2) plt.imshow(hue_after,
cmap='hsv') plt.title("Shifted Hue Channel") plt.axis('off')
plt.show()
```

Output:


```
Original Hue value at pixel (100, 100): 60
Shifted Hue value at pixel (100, 100): 24.199999999999996
```

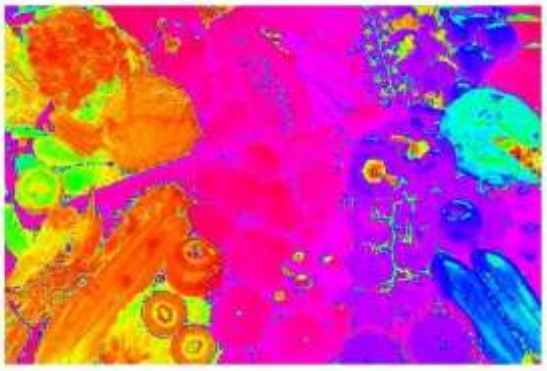
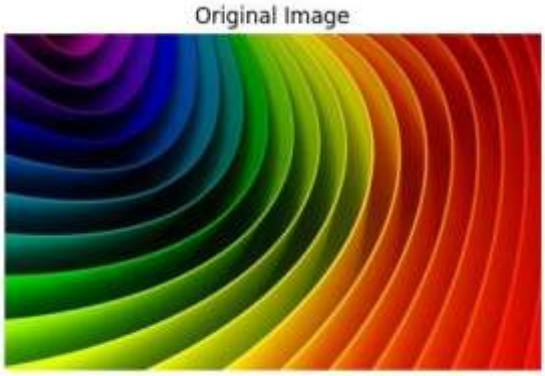
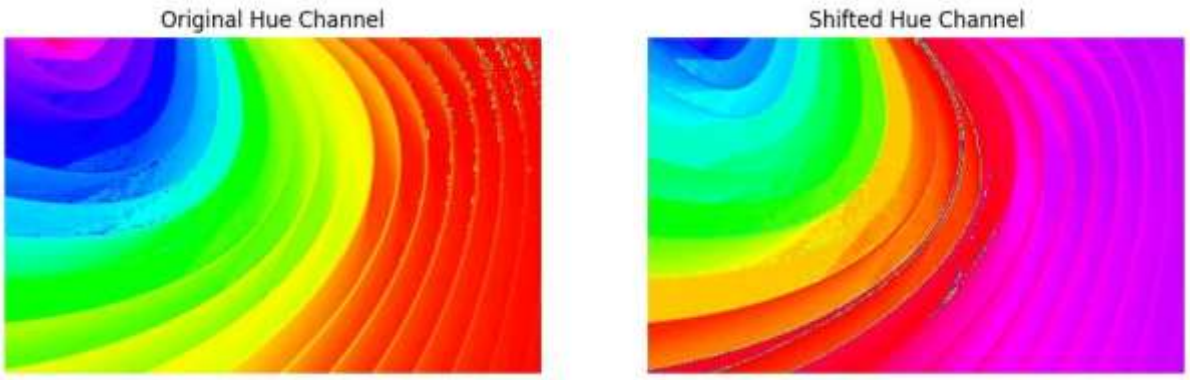


Image:2




```
Original Hue value at pixel (100, 100): 142
Shifted Hue value at pixel (100, 100): 106.19999999999999
```



By dividing the value :

Algorithm:

1. **Convert to HSV**:

image_hsv = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)
Converts the image from BGR to HSV.

2. **Extract the channels**: hue, saturation, value = cv2.split(image_hsv)

Splits the HSV image into Hue, Saturation, and Value channels.

3. **Store the original Value channel**:

value_before = value.copy()
Saves the original Value channel.

4. **Modify the Value channel**: image_hsv[:, :, 2] = cv2.min(image_hsv[:, :, 2] / 2, 255) Reduces the Value channel by half, making the image darker.

5. **Get the modified Value channel**:

hue, saturation, value2 = cv2.split(image_hsv)

Splits the modified image to get the updated Value channel.

6. **Convert back to RGB**:

image_rgb_modified = cv2.cvtColor(image_hsv, cv2.COLOR_HSV2RGB)
Converts the updated HSV image back to RGB.

7. **Plot the images and channels**:

- Displays the original image, modified image, Hue, Saturation, and Value channels.

8. **Print pixel values**:

Prints the Hue, Saturation, and Value values at pixel (100, 100) before and after modifying the Value channel.

Code: image_hsv = cv2.cvtColor(image,

cv2.COLOR_BGR2HSV) hue, saturation, value =

cv2.split(image_hsv) value_before = value.copy()

image_hsv[:, :, 2] = cv2.min(image_hsv[:, :, 2] / 2, 255) hue, saturation,

value2 = cv2.split(image_hsv) value_after = value2.copy()

image_rgb_modified = cv2.cvtColor(image_hsv,

cv2.COLOR_HSV2RGB) plt.figure(figsize=(15, 8)) plt.subplot(2, 3, 1)

plt.imshow(cv2.cvtColor(image,

cv2.COLOR_BGR2RGB)) plt.title("Original Image")

plt.axis('off') plt.subplot(2, 3, 2)

plt.imshow(image_rgb_modified) plt.title("Image with

Divided Value") plt.axis('off') plt.subplot(2, 3, 3)

plt.imshow(hue, cmap='hsv') plt.title("Hue Channel")

plt.axis('off') plt.subplot(2, 3, 4) plt.imshow(saturation,

cmap='gray') plt.title("Saturation Channel (Original)")

plt.axis('off') plt.subplot(2, 3, 5) plt.imshow(value,

cmap='gray')

plt.title("value Channel ")

plt.axis('off')

plt.subplot(2, 3, 5)

plt.imshow(value_after, cmap='gray') plt.title("Value

Channel (After Dividing)") plt.axis('off') plt.show() x, y =

100, 100 pixel_hue = hue[y, x] pixel_saturation_before =

saturation_before[y, x] pixel_value_before = value_before[y,

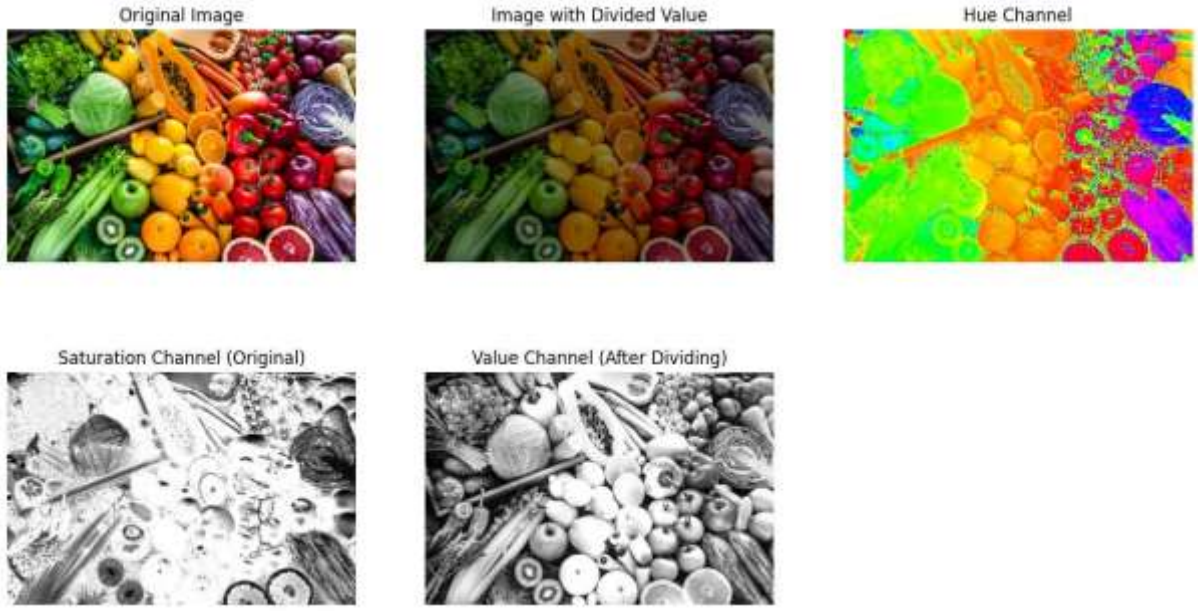
x] pixel_value_after = value_after[y, x] print(f"HSV values

at pixel ({x}, {y}):") print(f"Hue: {pixel_hue}")

print(f"Saturation (Original): {pixel_saturation_before}")

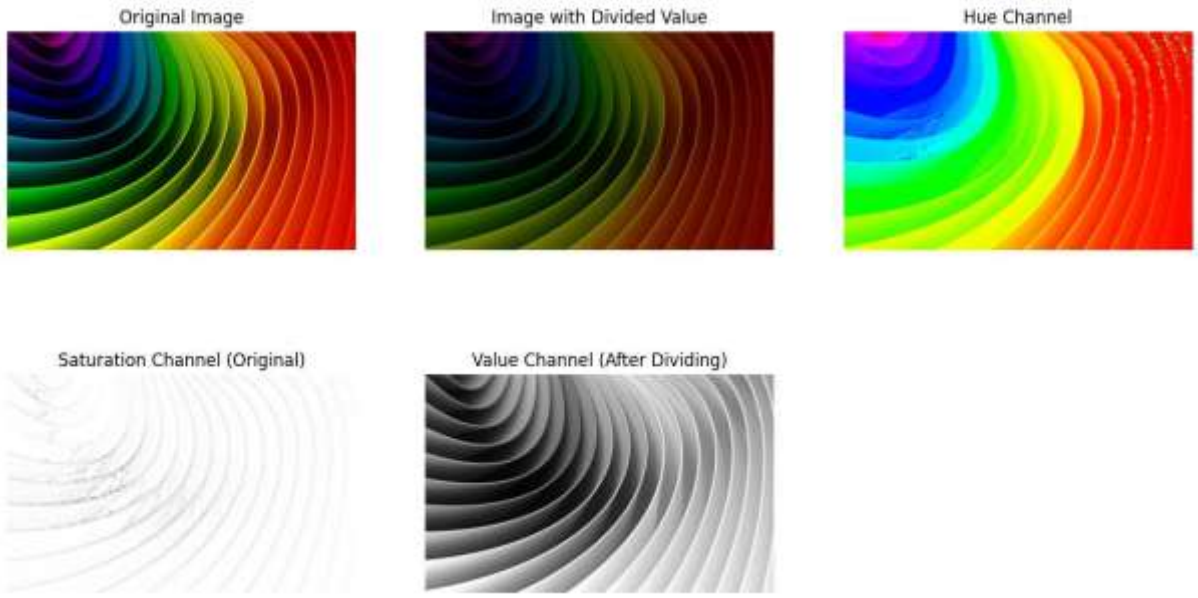
print(f"Value (Before): {pixel_value_before}") print(f"Value

(After Doubling): {pixel_value_after}")Output: **output:**




```
HSV values at pixel (100, 100):
Hue: 60
Saturation (Original): 255
Value (Before): 4
Value (After Doubling): 2
```

Image:2




```
HSV values at pixel (100, 100):
Hue: 142
Saturation (Original): 237
Value (Before): 140
Value (After Doubling): 70
```

By incrementing all the values by 10:

Algorithm:

1. **Increment Value**: increment_value = 10
   You decide to increase the Hue, Saturation, and Value by 10.
2. **Convert to HSV**:
   image_hsv = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)
   Convert the image from BGR to HSV color space.
3. **Split HSV into channels**:
   hue, saturation, value = cv2.split(image_hsv)
   Split the image into three parts: Hue, Saturation, and Value.
4. **Store original Saturation and Value**: saturation_before = saturation.copy(), value_before = value.copy() Save the original Saturation and Value for comparison later.
5. **Increase Hue, Saturation, and Value**:
   hue = cv2.add(hue, increment_value), saturation = cv2.add(saturation, increment_value), value = cv2.add(value,

increment_value) Increase each of the Hue, Saturation, and Value channels by 10.

6. **Clamp values to valid ranges**: hue = cv2.min(hue, 179), saturation = cv2.min(saturation, 255), value = cv2.min(value, 255)
   Make sure the values don't go above the valid range (Hue up to 179, others up to 255).

7. **Merge the channels back**:
   image_hsv = cv2.merge([hue, saturation, value]) Merge the modified channels back into one HSV image.

8. **Convert back to RGB**: image_rgb_modified = cv2.cvtColor(image_hsv, cv2.COLOR_HSV2RGB)
   Convert the modified HSV image back to RGB for visualization.

9. **Show the images**:
   Plot the original image, modified image, and the individual channels (Hue, Saturation, and Value).

10. **Print pixel values**:
    Show the HSV values at pixel (100, 100) before and after the increment.

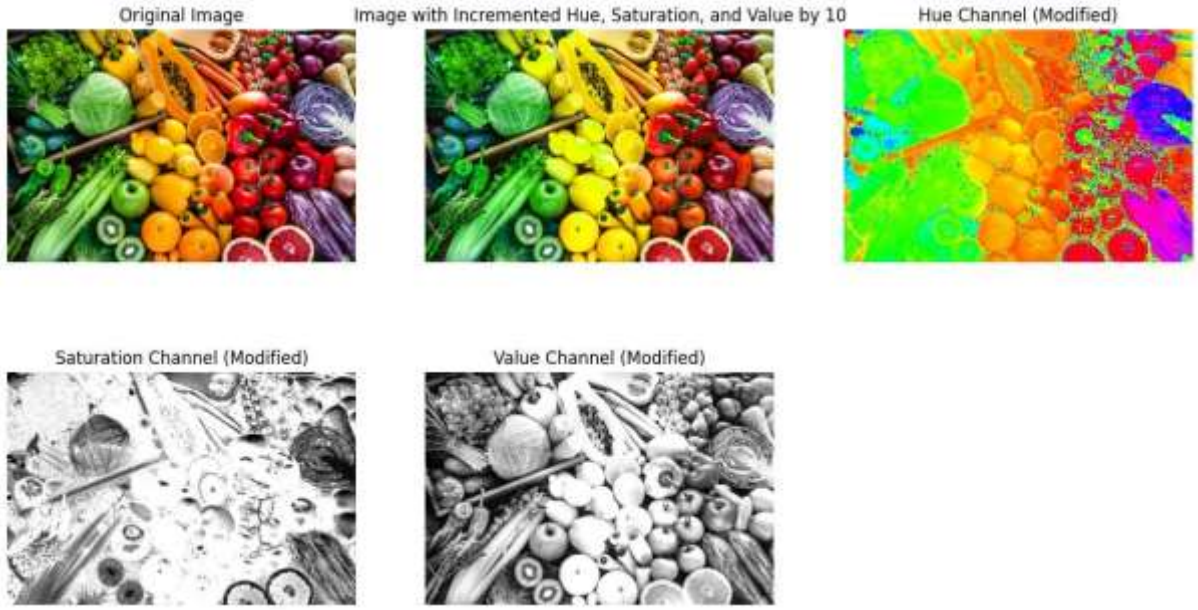
Code:

```
increment_value = 10

image_hsv = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)

hue, saturation, value = cv2.split(image_hsv)

saturation_before = saturation.copy()

value_before = value.copy()


hue = cv2.add(hue, increment_value) saturation = cv2.add(saturation,

increment_value)   value = cv2.add(value, increment_value)   hue =

cv2.min(hue, 179)   saturation = cv2.min(saturation, 255) value =

cv2.min(value, 255) image_hsv = cv2.merge([hue, saturation, value])
```

```
image_rgb_modified = cv2.cvtColor(image_hsv,
cv2.COLOR_HSV2RGB) plt.figure(figsize=(15, 8)) plt.subplot(2, 3, 1)
plt.imshow(cv2.cvtColor(image, cv2.COLOR_BGR2RGB))
plt.title("Original Image")
plt.axis('off') plt.subplot(2, 3, 2)
plt.imshow(image_rgb_modified
)
plt.title(f"Image with Incremented Hue, Saturation, and Value by
{increment_value}")
plt.axis('off')
plt.subplot(2, 3, 3)
plt.imshow(hue, cmap='hsv')
plt.title("Hue Channel (Modified)")
plt.axis('off') plt.subplot(2, 3, 4)
plt.imshow(saturation, cmap='gray')
plt.title("Saturation Channel
(Modified)") plt.axis('off') plt.subplot(2,
3, 5) plt.imshow(value, cmap='gray')
plt.title("Value Channel (Modified)")
plt.axis('off') plt.show() x, y = 100, 100
pixel_hue = hue[y, x]
pixel_saturation_before =
saturation_before[y, x]
pixel_saturation_after = saturation[y, x]
pixel_value_before = value_before[y, x]
```

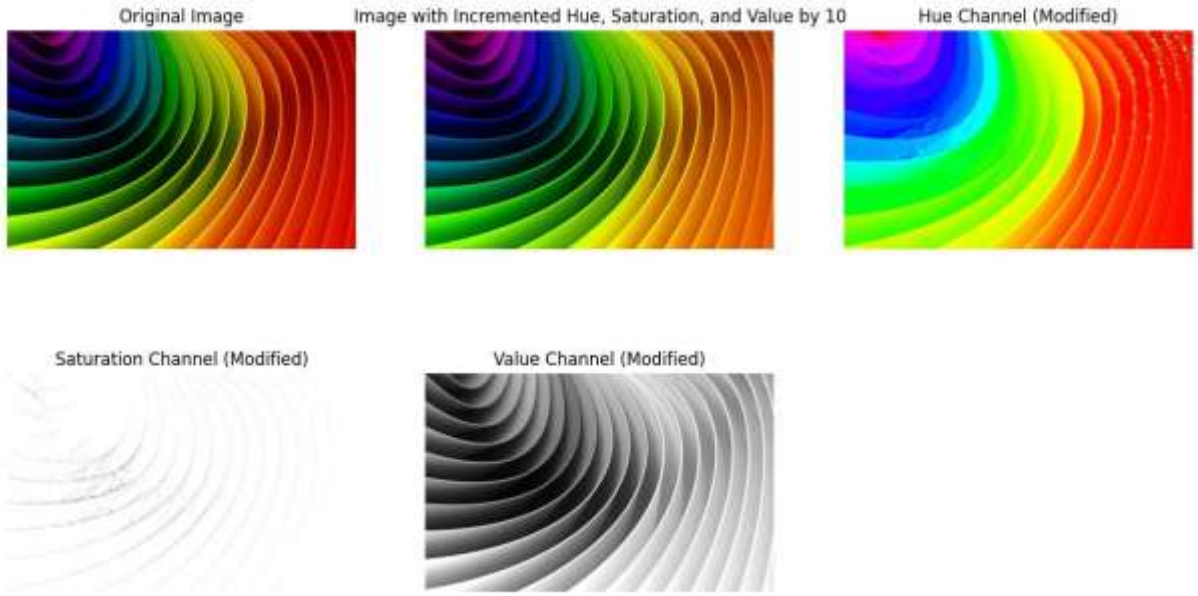
```
pixel_value_after = value[y, x]
print(f"HSV values at pixel ({x}, {y}):")
print(f"Hue (After Increment):
{pixel_hue}") print(f"Saturation
(Original): {pixel_saturation_before}")
print(f"Saturation (After Increment):
{pixel_saturation_after}") print(f"Value
(Before Increment):
{pixel_value_before}") print(f"Value
(After Increment): {pixel_value_after}")
```

Output:




```
HSV values at pixel (100, 100):
Hue (After Increment): 70
Saturation (Original): 255
Saturation (After Increment): 255
Value (Before Increment): 4
Value (After Increment): 14
```

Image:2




```
HSV values at pixel (100, 100):
Hue (After Increment): 152
Saturation (Original): 237
Saturation (After Increment): 247
Value (Before Increment): 140
Value (After Increment): 150
```

Conversion of RGB to YIQ image:

Algorithm:

1. **Normalize the image to [0, 1] range**: image_rgb = image_rgb / 255.0

This scales the pixel values from the [0, 255] range to [0, 1].

2. **Define the RGB to YIQ transformation matrix**:

transformation_matrix = np.array([...])

This matrix is used to convert the image from the RGB color space to YIQ.

3. **Reshape the image**: reshaped_image = image_rgb.reshape(-1, 3)

Flattens the image into a 2D array where each row is a pixel with RGB values.

4. **Apply the transformation**: yiq_image = reshaped_image.dot(transformation_matrix.T)

Multiplies each pixel's RGB values by the transformation matrix to convert it to YIQ.

5. **Reshape back to original image shape**:

yiq_image = yiq_image.reshape(image_rgb.shape)

Reshapes the transformed pixels back to the original image dimensions.

6. **Clip the values and convert to [0, 255] range**: yiq_image = np.clip(yiq_image * 255, 0, 255).astype(np.uint8)

Scales the YIQ image back to the [0, 255] range and converts the values to integers.

7. **Plot the original and converted images**:

- **Original image**:
  plt.subplot(1, 2, 1), plt.imshow(image_rgb), plt.title("Original RGB Image")
- **Converted YIQ image**:
  plt.subplot(1, 2, 2), plt.imshow(yiq_image), plt.title("Converted YIQ Image") The images are displayed side by side.

Code:

```
image_rgb = image_rgb / 255.0

transformation_matrix = np.array([
      [0.299,  0.587,  0.114],
      [0.5957, -0.2746, -0.3213],
      [0.2115, -0.5227,  0.3113]
  ])
 reshaped_image = image_rgb.reshape(-1, 3)  yiq_image =
reshaped_image.dot(transformation_matrix.T)  yiq_image =
yiq_image.reshape(image_rgb.shape) yiq_image =
np.clip(yiq_image * 255, 0, 255).astype(np.uint8)
plt.subplot(1, 2, 1) plt.imshow(image_rgb) plt.title("Original
```

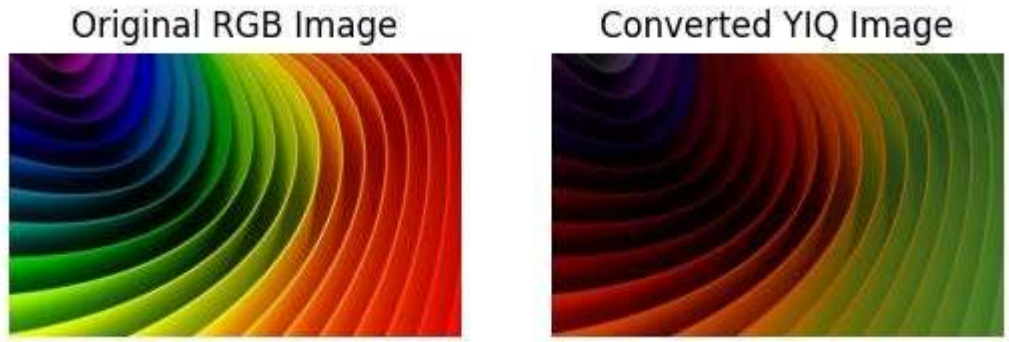
RGB Image") plt.axis('off') plt.subplot(1, 2, 2)

plt.imshow(yiq_image) plt.title("Converted YIQ Image")

plt.axis('off') plt.show()

Output:

Image:2



Visualizing the separate channel:

Algorithm:

1. **Extract Y, I, and Q channels from the YIQ image**:
    - Y = yiq_image[:, :, 0] — Extracts the Y (luminance) channel. ◦ I = yiq_image[:, :, 1] — Extracts the I (in-phase) channel.
    - Q = yiq_image[:, :, 2] — Extracts the Q (quadrature) channel.

2. **Adjust I and Q channels**:
    - I = np.clip(I + 128, 0, 255) — Shifts the I channel by 128 to make its values positive and clips it to the [0, 255] range. ◦ Q = np.clip(Q + 128, 0, 255) — Does the same for the Q channel.
3. **Display the channels**:
    - plt.subplot(1, 3, 1), plt.imshow(Y, cmap='gray') — Displays the Y channel in grayscale. ◦ plt.subplot(1, 3, 2), plt.imshow(I, cmap='gray') — Displays the I channel in grayscale. ◦ plt.subplot(1, 3, 3), plt.imshow(Q, cmap='gray') — Displays the Q channel in grayscale.
4. **Show the plots**: plt.show() — Displays the three channel images side by side.

Code:

```
Y = yiq_image[:, :, 0]  # Y channel
I = yiq_image[:, :, 1]  # I channel
Q = yiq_image[:, :, 2]  # Q channel
I = np.clip(I + 128, 0, 255)
Q = np.clip(Q + 128, 0, 255)  # Shift Q channel to positive values
plt.subplot(1, 3, 1)  plt.imshow(Y, cmap='gray')  plt.title("Y Channel")  plt.axis('off')
plt.subplot(1, 3, 2)
plt.imshow(I, cmap='gray')
plt.title("I Channel")
plt.axis('off')
# Display the Q channel
plt.subplot(1, 3, 3)
```

```
    plt.imshow(Q,
cmap='gray')     plt.title("Q
Channel")     plt.axis('off')
plt.show()
```

Output:

RESULT:

The colour spaces on images are analysed.

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().

CODE:

```
import cv2
import matplotlib.pyplot as plt image =
cv2.imread(r"C:\Users\revaa\Downloads\ben 10.jpg")
plt.imshow(image)
plt.axis('off')
plt.show()
```

OUTPUT:

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:

image = cv2.cvtColor(image,

cv2.COLOR_BGR2RGB) plt.imshow(image_rgb)

plt.axis('off') plt.show()

OUTPUT:

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

height, width = image.shape[:2] print(height) print(width)

CODE:

```
tx, ty = 200, 200   x, y = np.meshgrid(np.arange(width),
np.arange(height)) flat_x = x.flatten() flat_y =
y.flatten() ones = np.ones_like(flat_x) original_coords =
np.stack([flat_x, flat_y, ones], axis=0)
translation_matrix = np.array([[1, 0, tx], [0, 1, ty], [0, 0,
1]])
new_coords = translation_matrix @ original_coords
translated_image = np.zeros_like(image)
valid_idx = (new_x >= 0) & (new_x < width) & (new_y >= 0) & (new_y <
height)
translated_image[new_y[valid_idx], new_x[valid_idx]]
= image[flat_y[valid_idx], flat_x[valid_idx]]
plt.imshow(translated_image) plt.title("Matrix-Based
Translated Image") 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:

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

theta = 45    theta_rad = np.deg2rad(theta)

center_x, center_y = width // 2, height //

2
```

```
x, y = np.meshgrid(np.arange(width), np.arange(height))
flat_x = x.flatten() flat_y = y.flatten() centered_x = flat_x -
center_x centered_y = flat_y - center_y rotation_matrix =
np.array([[np.cos(theta_rad), -np.sin(theta_rad)],
                    [np.sin(theta_rad), np.cos(theta_rad)]])
rotated_coords = rotation_matrix @ np.vstack([centered_x,
centered_y]) rotated_x = rotated_coords[0, :] + center_x rotated_y =
rotated_coords[1, :] + center_y rotated_x =
np.round(rotated_x).astype(int) rotated_y =
np.round(rotated_y).astype(int) rotated_image = np.zeros_like(image)

valid_idx = (rotated_x >= 0) & (rotated_x < width) & (rotated_y >= 0) &
(rotated_y < height)
rotated_image[rotated_y[valid_idx], rotated_x[valid_idx]]
= image[flat_y[valid_idx], flat_x[valid_idx]]
plt.imshow(rotated_image) plt.title(f"Rotated Image by
{theta} degrees") 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:

```
image =
cv2.imread(r"C:\Users\student\Downloads\SMURF.png") height,
width, channels = image.shape scale_x = 1.5  scale_y = 1.5
y, x = np.indices((height, width))   scaled_x =
scale_x * (x - width // 2) + width // 2   scaled_y
= scale_y * (y - height // 2) + height // 2
scaled_x = np.round(scaled_x).astype(int)
scaled_y = np.round(scaled_y).astype(int)
valid_idx = (scaled_x >= 0) & (scaled_x < width) & (scaled_y >= 0) &
(scaled_y < height)
scaled_image = np.zeros_like(image)
scaled_image[scaled_y[valid_idx], scaled_x[valid_idx]] =
image[y[valid_idx], x[valid_idx]] plt.imshow(scaled_image)
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:

```
height, width = image.shape[:2] sh_x = 0.5
sh_y = 0.5
y, x = np.indices((height, width))
sheared_x = x + sh_x * y sheared_y = y +
sh_y * x sheared_x =
np.round(sheared_x).astype(int) sheared_y
= np.round(sheared_y).astype(int)
valid_idx = (sheared_x >= 0) & (sheared_x < width) & (sheared_y >= 0) &
(sheared_y < height)
sheared_image = np.zeros_like(image)
sheared_image[sheared_y[valid_idx], sheared_x[valid_idx]]
= image[y[valid_idx], x[valid_idx]]
plt.imshow(sheared_image)
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:

image = cv2.imread(r"C:\Users\revaa\Downloads\ben 10.jpg") height, width = image.shape[:2] y, x = np.indices((height, width)) reflected_x = width - 1 - x reflected_x = np.clip(reflected_x, 0, width - 1)

```
reflected_image = np.zeros_like(image) reflected_image[y,
reflected_x] = image[y, x] plt.imshow(reflected_image)
plt.axis('off')
plt.show()
```

OUTPUT:

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
import matplotlib.pyplot as plt image =
cv2.imread(r"C:\Users\revaa\Downloads\iron man.jpg")
plt.imshow(image)
plt.axis('off')
print(image.shape)
plt.show()
```

OUTPUT:

RGB CONVERSION :

```
image= cv2.cvtColor(image,
cv2.COLOR_BGR2RGB) plt.imshow(image)
plt.axis("off")
```

OUTPUT:

DFT WITH ORIGINAL IMAGE:

ALGORITHM:

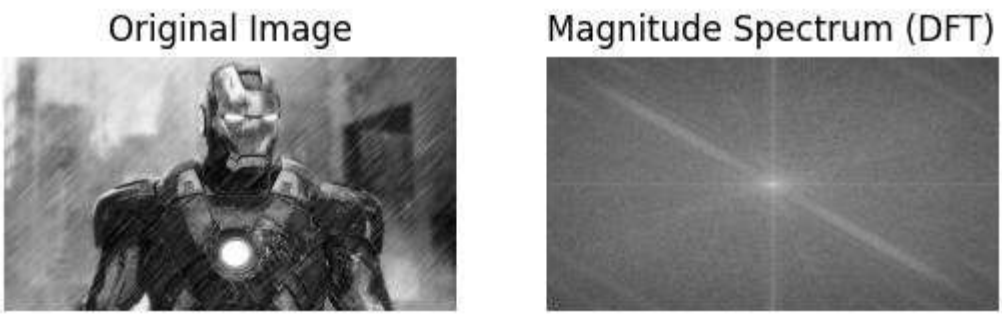
1. Load the grayscale image.
2. Apply DFT on the image to get frequency components.

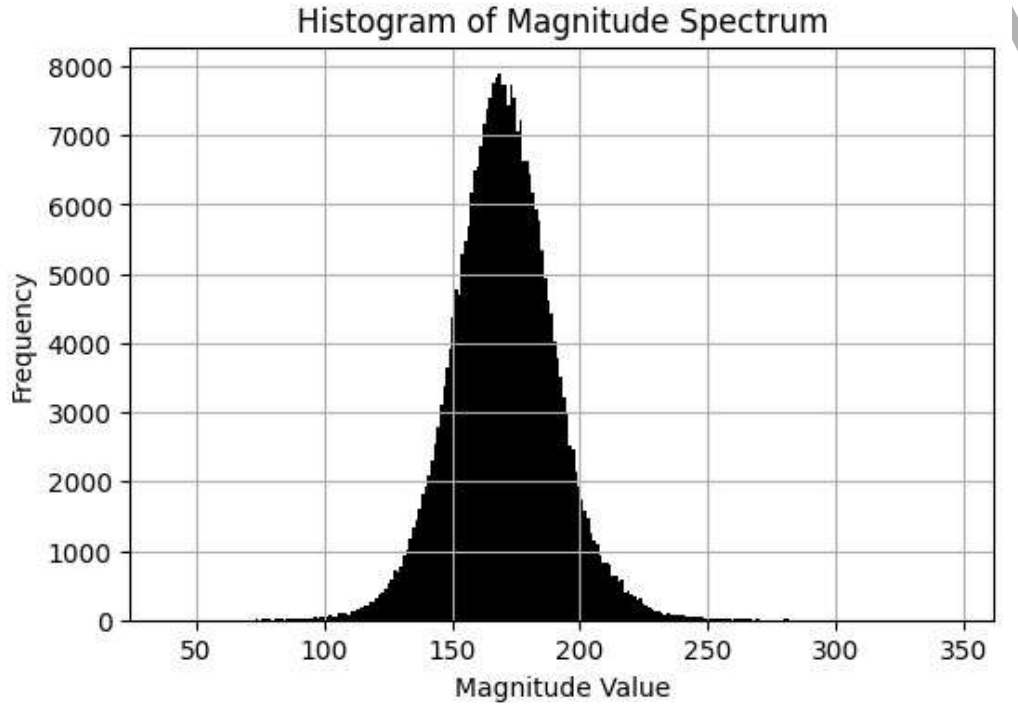
3. Shift the DFT result to center the zero-frequency component.
4. Calculate the magnitude of the DFT components.
5. Display the original and magnitude spectrum images side by side.
6. Plot the histogram of the magnitude spectrum to show frequency distribution.

CODE:

```
import numpy as np
image = cv2.imread(r"C:\Users\revaa\Downloads\iron man.jpg",
cv2.IMREAD_GRAYSCALE)
dft = cv2.dft(np.float32(image), flags=cv2.DFT_COMPLEX_OUTPUT)
dft_shift = np.fft.fftshift(dft)
magnitude_spectrum = 20 * np.log(cv2.magnitude(dft_shift[:, :, 0], dft_shift[:,
:,
1]) + 1) plt.subplot(1, 2, 1) plt.imshow(image, cmap="gray")
plt.title("Original Image") plt.axis("off") plt.subplot(1, 2, 2)
plt.imshow(magnitude_spectrum, cmap="gray")
plt.title("Magnitude Spectrum (DFT)") plt.axis("off")
plt.show() plt.figure(figsize=(6, 4))
plt.hist(magnitude_spectrum.ravel(), bins=256,
color='black') plt.title("Histogram of Magnitude Spectrum")
plt.xlabel("Magnitude Value") plt.ylabel("Frequency")
plt.grid(True) plt.show()
```

OUTPUT:





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.

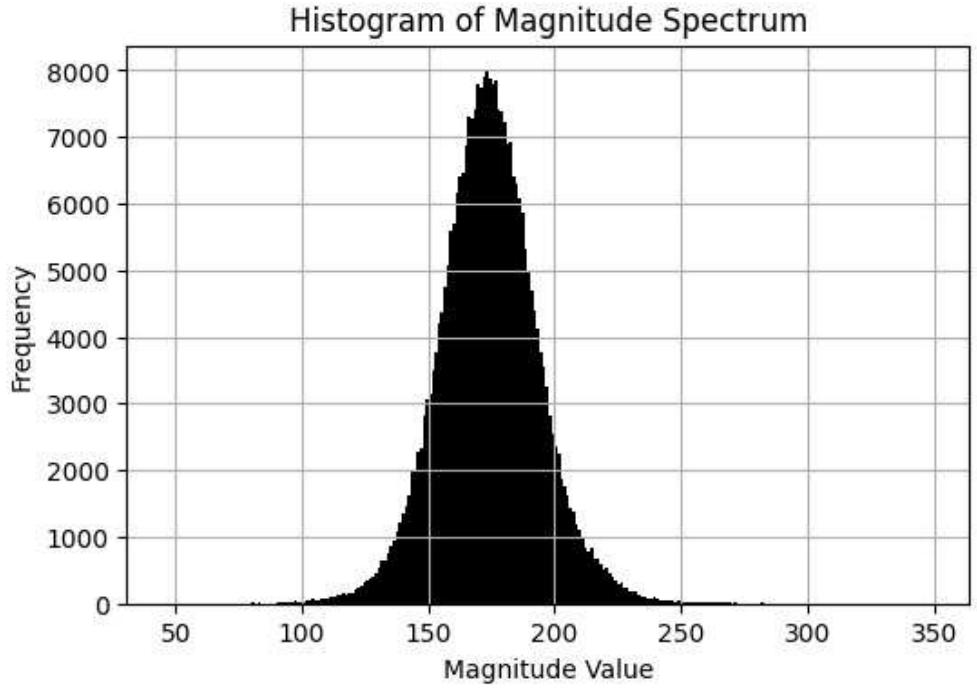
CODE:

```
image = cv2.imread(r"C:\Users\revaa\Downloads\iron man.jpg",
cv2.IMREAD_GRAYSCALE)

equalized_image = cv2.equalizeHist(image)

plt.figure(figsize=(12, 6)) plt.subplot(1, 3,

1) plt.imshow(image, cmap="gray")

plt.title("Original Image") plt.axis("off")

plt.subplot(1, 3, 2)

plt.imshow(equalized_image,

cmap="gray") plt.title("Histogram

Equalized Image") plt.axis("off")

dft = cv2.dft(np.float32(equalized_image),
flags=cv2.DFT_COMPLEX_OUTPUT)

dft_shift = np.fft.fftshift(dft)  # Shift zero frequency to the center

magnitude_spectrum = 20 * np.log(cv2.magnitude(dft_shift[:, :, 0], dft_shift[:,
:,
1]) + 1) plt.subplot(1, 3, 3) plt.imshow(magnitude_spectrum,

cmap="gray") plt.title("Magnitude Spectrum (DFT) of

Equalized Image") plt.axis("off") plt.show()

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

plt.hist(magnitude_spectrum.ravel(), bins=256,

color='black') plt.title("Histogram of Magnitude Spectrum")
```

plt.xlabel("Magnitude Value") plt.ylabel("Frequency")

plt.grid(True) 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

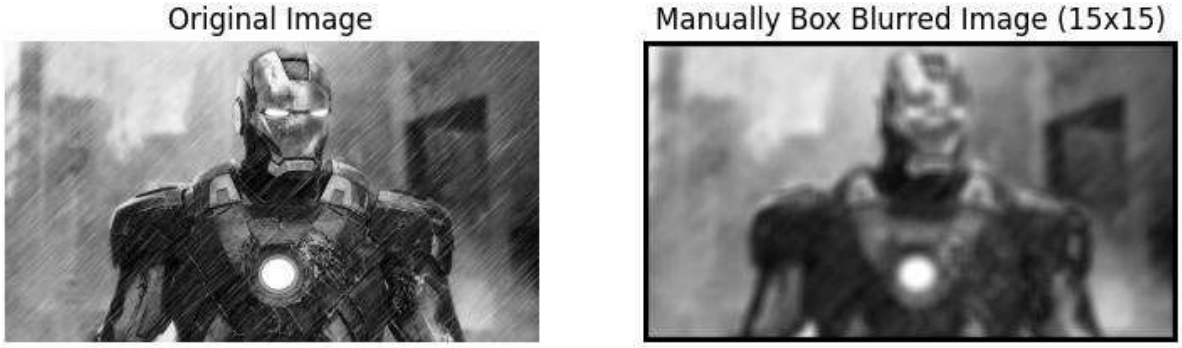
- Show the original image alongside the blurred result for comparison.

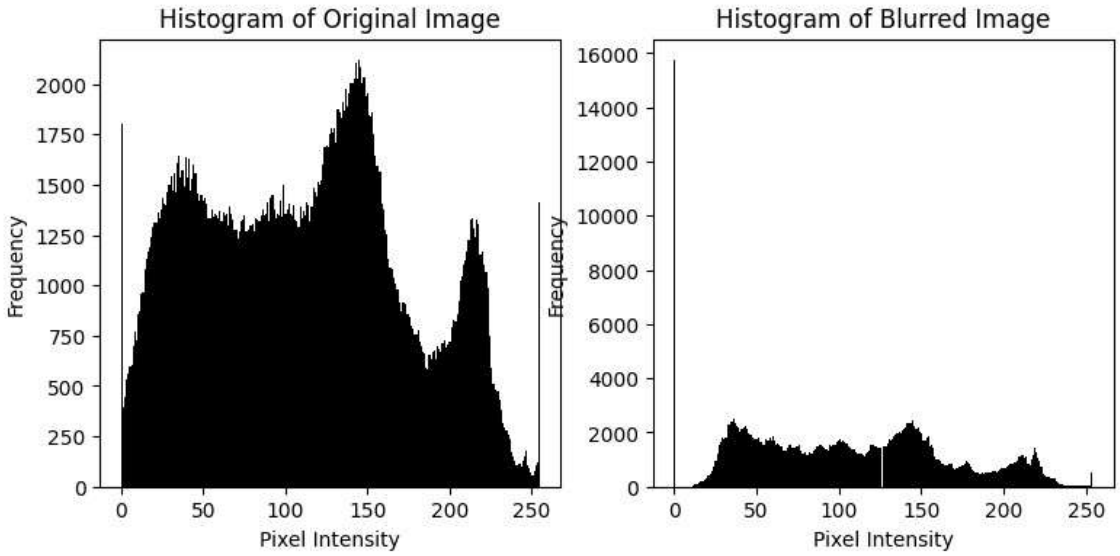
CODE:

```
image = cv2.imread(r"C:\Users\revaa\Downloads\iron man.jpg",
cv2.IMREAD_GRAYSCALE)

height, width = image.shape

blurred_image =

np.zeros_like(image) kernel =

np.ones((15, 15)) / 225   for i in

range(7, height - 7):       for j in

range(7, width - 7):

      region = image[i - 7:i + 8, j - 7:j + 8]

blurred_image[i, j] = np.sum(region * kernel)
```

```
blurred_image = np.uint8(blurred_image)
plt.figure(figsize=(9,4)) plt.subplot(1, 2, 1)

plt.imshow(image, cmap="gray") plt.title("Original
Image") plt.axis("off") plt.subplot(1, 2, 2)
plt.imshow(blurred_image, cmap="gray")
plt.title("Manually Box Blurred Image (15x15)")
plt.axis("off") plt.show() plt.figure(figsize=(9, 4))
plt.subplot(1, 2, 1) plt.hist(image.ravel(), bins=256,
color='black') plt.title("Histogram of Original Image")
plt.xlabel("Pixel Intensity") plt.ylabel("Frequency")
plt.subplot(1, 2, 2) plt.hist(blurred_image.ravel(),
bins=256, color='black') plt.title("Histogram of
Blurred Image") plt.xlabel("Pixel Intensity")
plt.ylabel("Frequency") plt.show()
```

OUTPUT:





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.

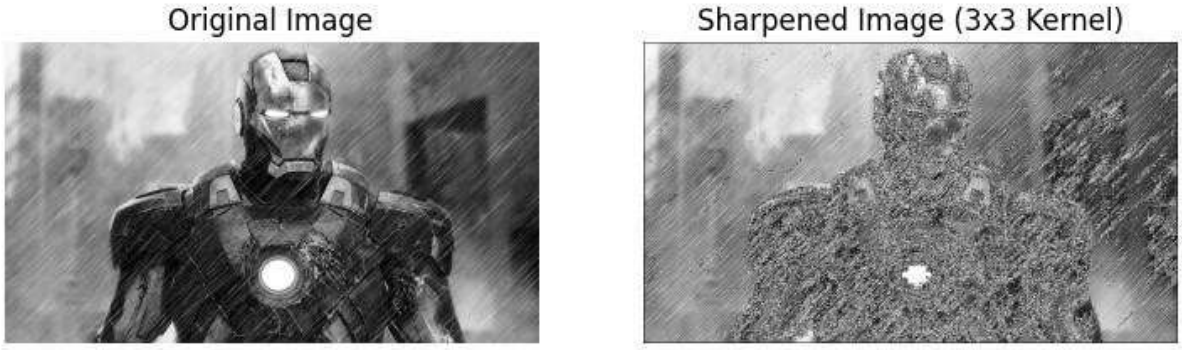
5. **Update the Pixel Value**:
    - The result from applying the kernel (sum of products) will be the new value for the current pixel in the sharpened image.
6. **Edge Handling**:
    - To avoid going out of bounds, skip processing the borders of the image (pixels on the first and last row/column).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

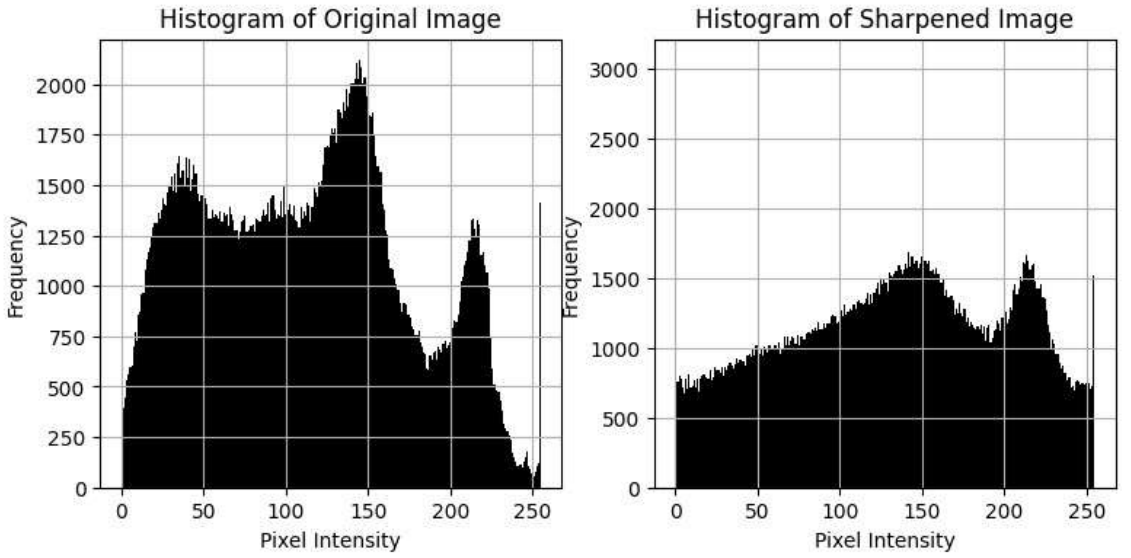
CODE:

```
image = cv2.imread(r"C:\Users\revaa\Downloads\iron man.jpg",
cv2.IMREAD_GRAYSCALE)

height, width = image.shape

sharpened_image = np.zeros_like(image)

sharpening_kernel = np.array([

   [ 0, -1,  0],

   [-1,  5, -1],

   [ 0, -1,  0]

]) for i in range(1, height -

1):       for j in range(1,

width - 1):

      region = image[i - 1:i + 2, j - 1:j + 2]

      sharpened_image[i, j] = np.sum(region * sharpening_kernel)


sharpened_image =

np.uint8(sharpened_image)

plt.figure(figsize=(9, 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(sharpened_image, cmap="gray")
plt.title("Sharpened Image (3x3 Kernel)")
plt.axis("off") plt.show() plt.figure(figsize=(9,
4)) plt.subplot(1, 2, 1) plt.hist(image.ravel(),
bins=256, color='black') plt.title("Histogram of
Original Image") plt.xlabel("Pixel Intensity")
plt.ylabel("Frequency") plt.grid(True)
plt.subplot(1, 2, 2)
plt.hist(sharpened_image.ravel(), bins=256,
color='black') plt.title("Histogram of
Sharpened Image") plt.xlabel("Pixel
Intensity") plt.ylabel("Frequency")
plt.grid(True) 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.

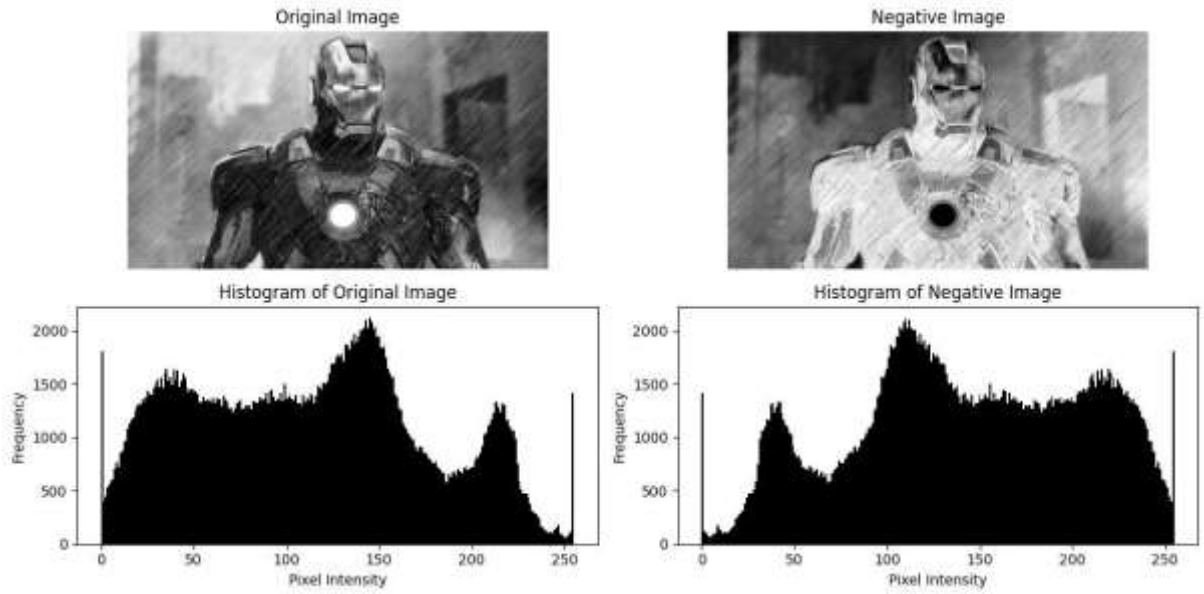
CODE:

```
image = cv2.imread(r"C:\Users\revaa\Downloads\iron man.jpg",
cv2.IMREAD_GRAYSCALE)

negative_image = 255 - image

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

plt.subplot(2, 2, 1)

plt.imshow(image,

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(image.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:

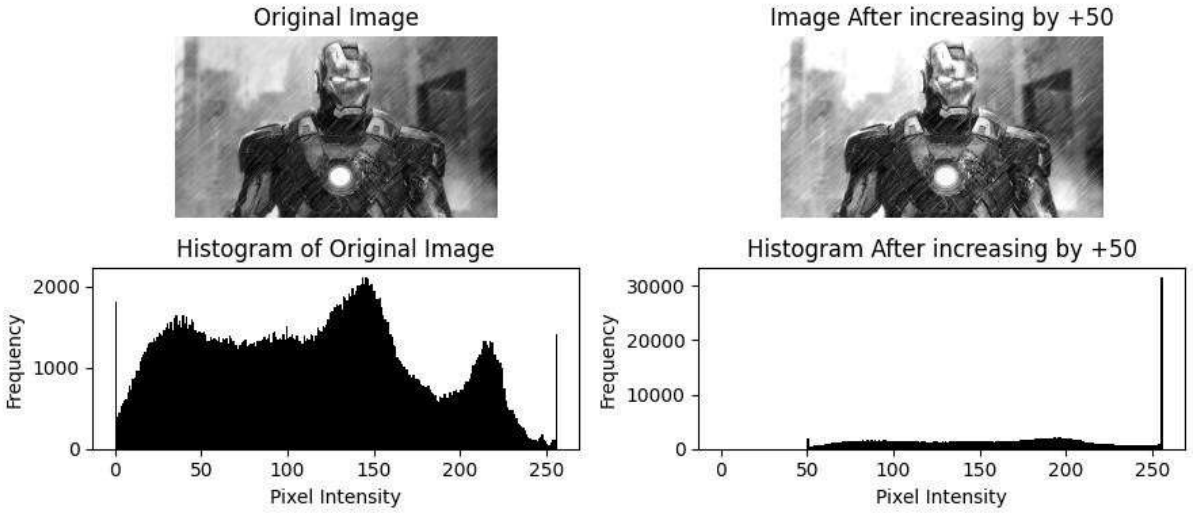
- The original image and the adjusted image (brighter) are displayed side by side in a 2x2 grid.
- The histograms of both images are also plotted. Histograms show the distribution of pixel intensities for both the original and the adjusted images.

CODE:

```
image = cv2.imread(r"C:\Users\revaa\Downloads\iron man.jpg",
cv2.IMREAD_GRAYSCALE)

i2 = cv2.add(image, +50)

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

plt.subplot(221)

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

plt.axis('off')

plt.title("Original Image")

plt.subplot(222)

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

plt.axis('off')

plt.title("Image After increasing by +50")

plt.subplot(223)

plt.hist(image.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 increasing by +50")

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.

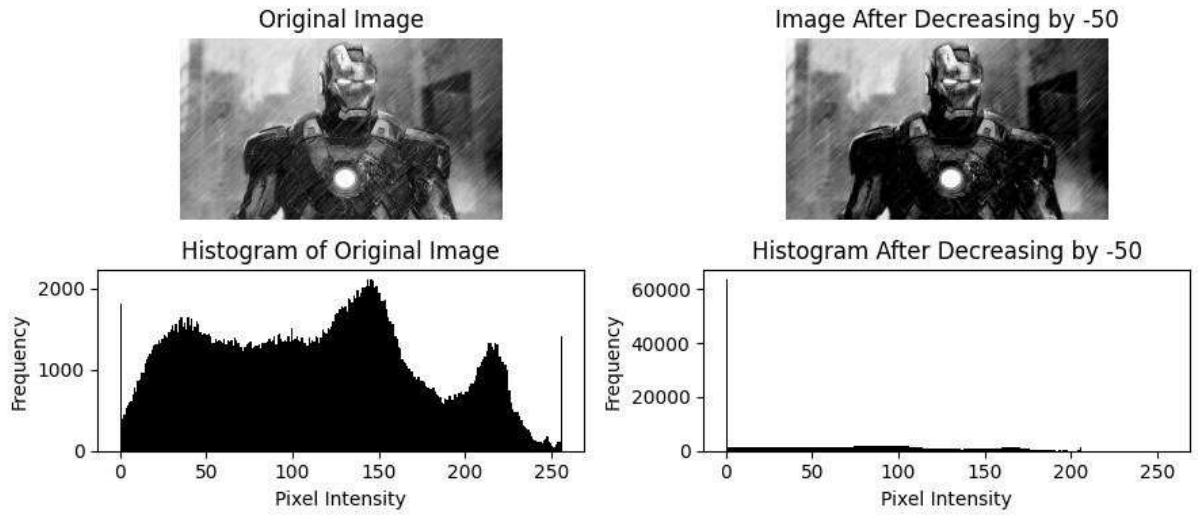
Plot histograms:

- Plot the histogram of pixel intensities for the original image.
- Plot the histogram of pixel intensities for the modified (darker) image.

CODE:

```
image = cv2.imread(r"C:\Users\revaa\Downloads\iron man.jpg",
cv2.IMREAD_GRAYSCALE)
i2 = cv2.add(image, -50)
plt.figure(figsize=(9, 4))
plt.subplot(221)
plt.imshow(image, 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 -50") plt.subplot(223)
plt.hist(image.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 -
50") 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 contraststretched 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

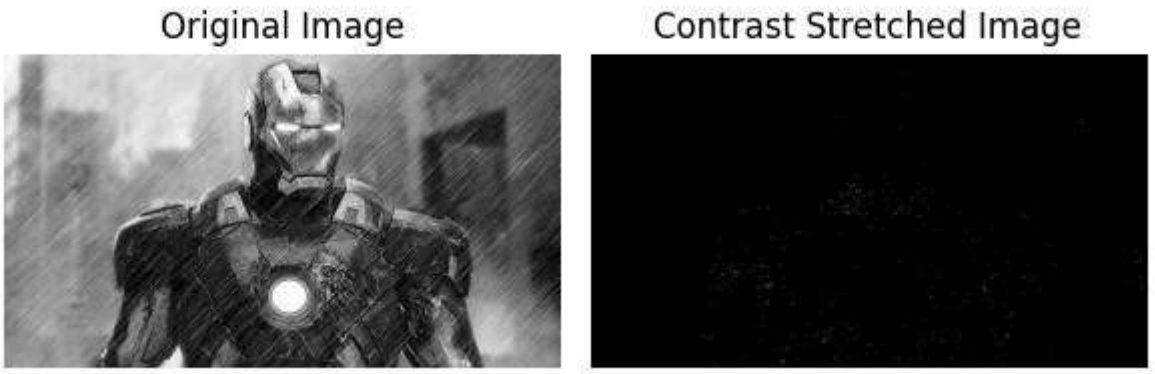
CODE:

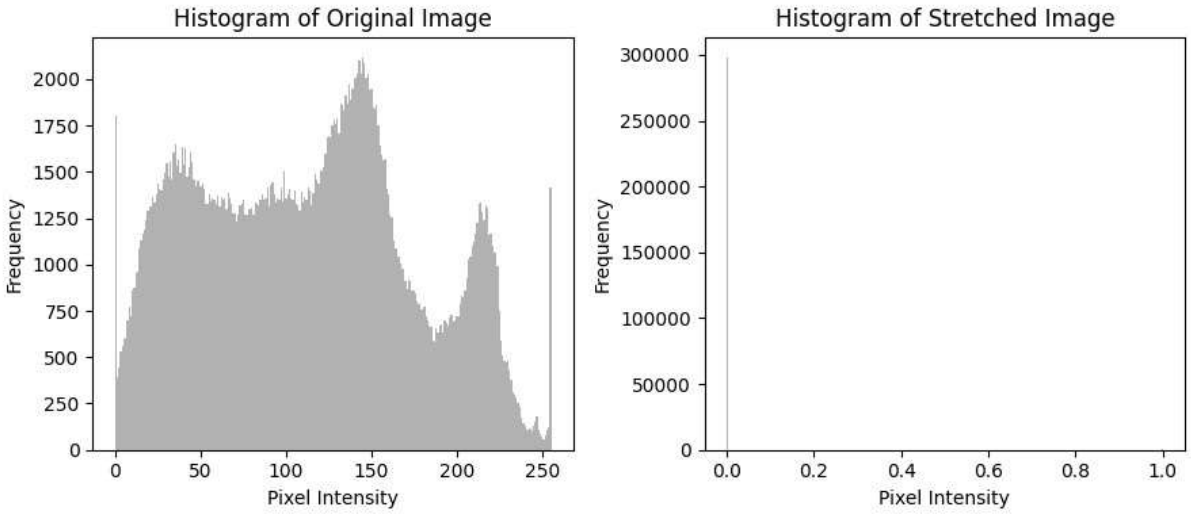
```
gray_image = cv2.imread(r"C:\Users\revaa\Downloads\iron man.jpg",
cv2.IMREAD_GRAYSCALE)

I_min = np.min(gray_image)

I_max = np.max(gray_image)

stretched_image = np.clip(((gray_image - I_min) * 255) / (I_max - I_min), 0,
255).astype(np.uint8) plt.figure(figsize=(6, 6)) plt.subplot(1, 2,

1) plt.imshow(gray_image, 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(gray_image.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:

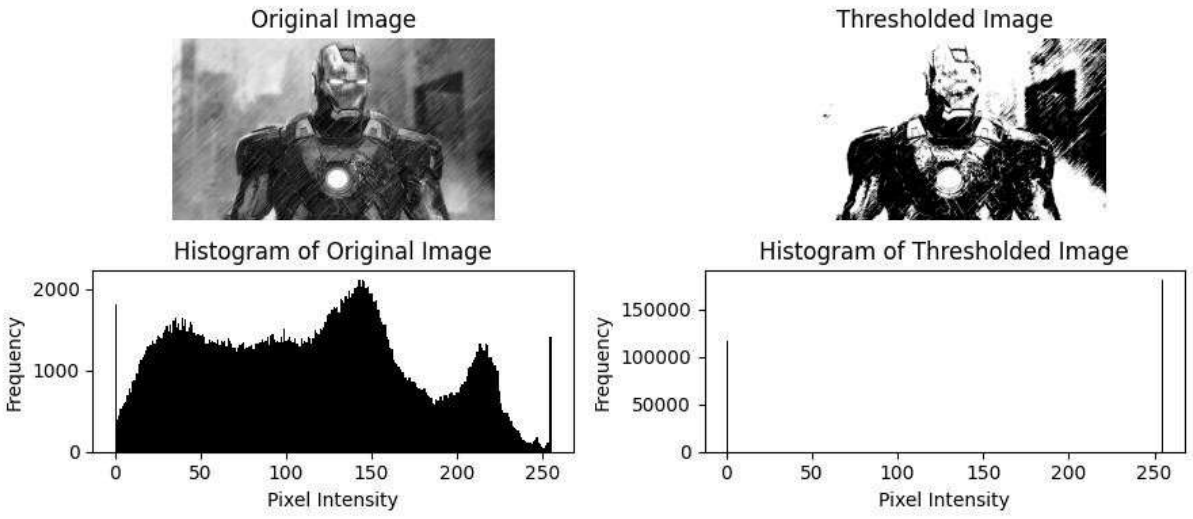
- Histograms for both the **original image** and the **thresholded image** are plotted: ○ The **original histogram** will show the distribution of pixel intensities, which could span the entire range from 0 to 255. ○ The **thresholded histogram** will show a binary distribution, with two distinct peaks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

CODE:

```
image_path =  r"C:\Users\revaa\Downloads\iron man.jpg"
image = cv2.imread(image_path,
cv2.IMREAD_GRAYSCALE) val, t = cv2.threshold(image, 90,
255, cv2.THRESH_BINARY) plt.figure(figsize=(9, 4))
plt.subplot(2, 2, 1) plt.imshow(image, 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(image.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().

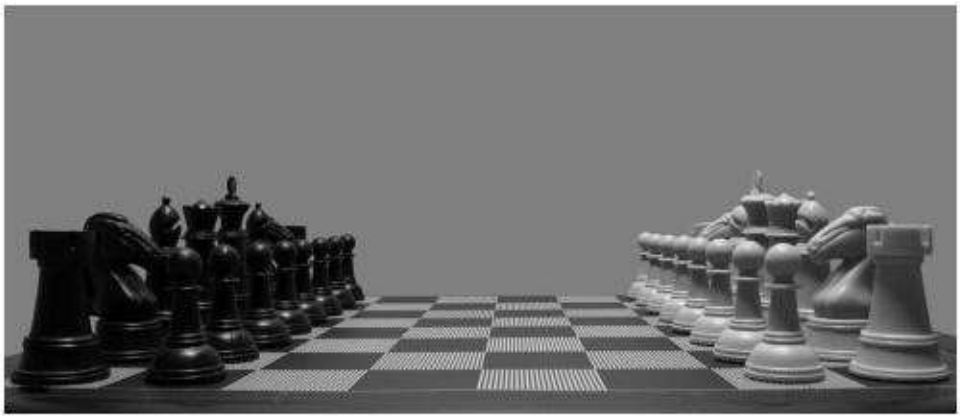
Remove axes around the image to make it cleaner with plt.axis('off').

Display the image with plt.show().

CODE:

```
import matplotlib.pyplot as plt import numpy as np import
cv2 image = cv2.imread(r"C:\Users\revaa\Downloads\chess
1.jpg")  plt.imshow(image)
plt.axis('off')
plt.show()
```

OUTPUT:



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:

mean = 0 std_dev = 30 gaussian_noise =

np.random.normal(mean, std_dev, image.shape) noisy_image =

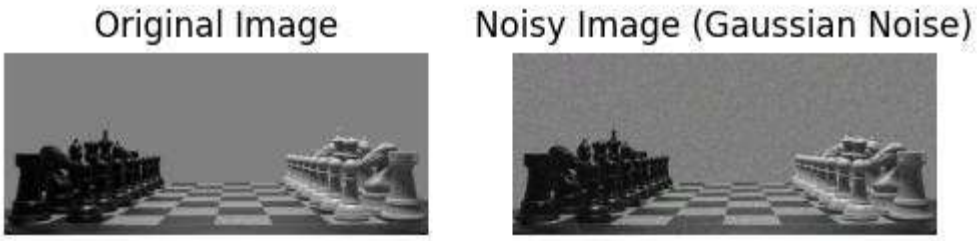
image.astype(np.float32) + gaussian_noise noisy_image =

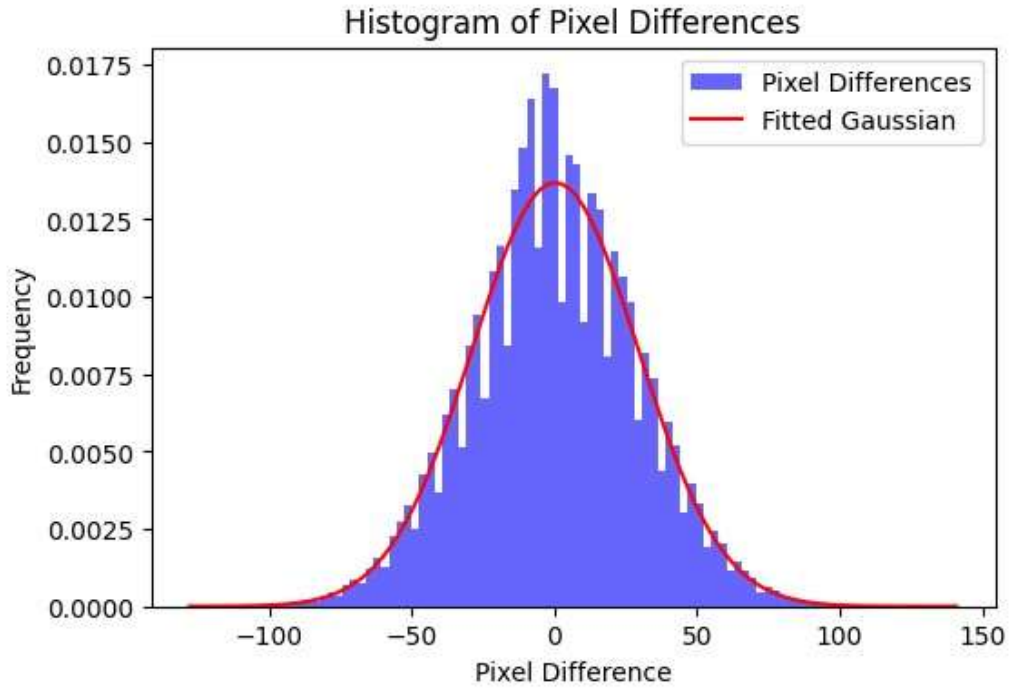
np.clip(noisy_image, 0, 255).astype(np.uint8)

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 (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()
```

OUTPUT:

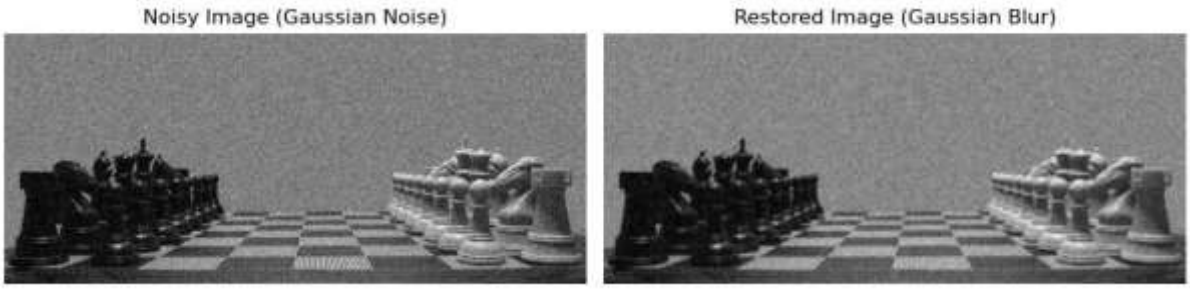




RESTORED IMAGE (GAUSSIAN BLUR FILTER):

```
restored_image = cv2.GaussianBlur(noisy_image, (5, 5),
0) plt.figure(figsize=(10, 5)) plt.subplot(1, 2, 1)
plt.imshow(noisy_image, cmap='gray') plt.title("Noisy
Image (Gaussian Noise)") plt.axis("off") plt.subplot(1, 2,
2) plt.imshow(restored_image, cmap='gray')
plt.title("Restored Image (Gaussian Blur)") plt.axis("off")
plt.tight_layout() 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.

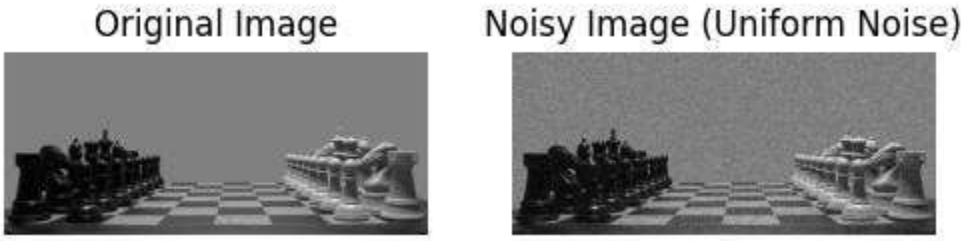
Apply Filters:

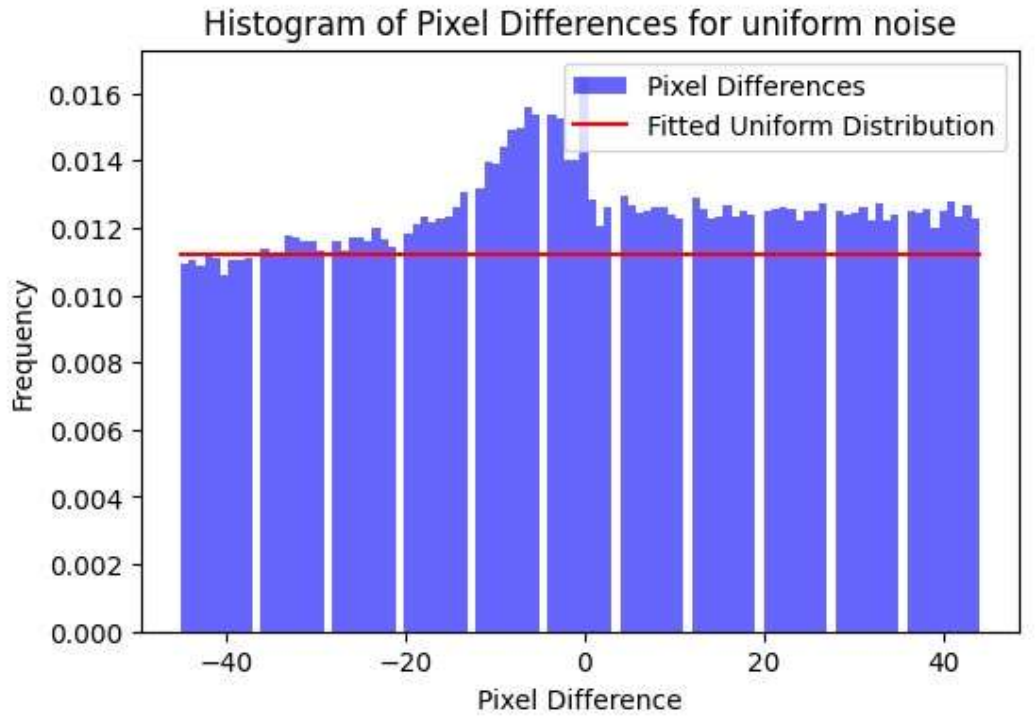
- We use two **filters** to remove the noise:
  - **Min Filter**: Looks at each pixel's neighbors and keeps the smallest value. This helps remove bright spots (like noise).
  - **Max Filter**: Looks at each pixel's neighbors and keeps the largest value. This helps remove dark spots.

Display Filtered Images:

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

CODE:





RESTORED IMAGE FOR UNIFORM NOISE(MIN MAX FILTER)

CODE :

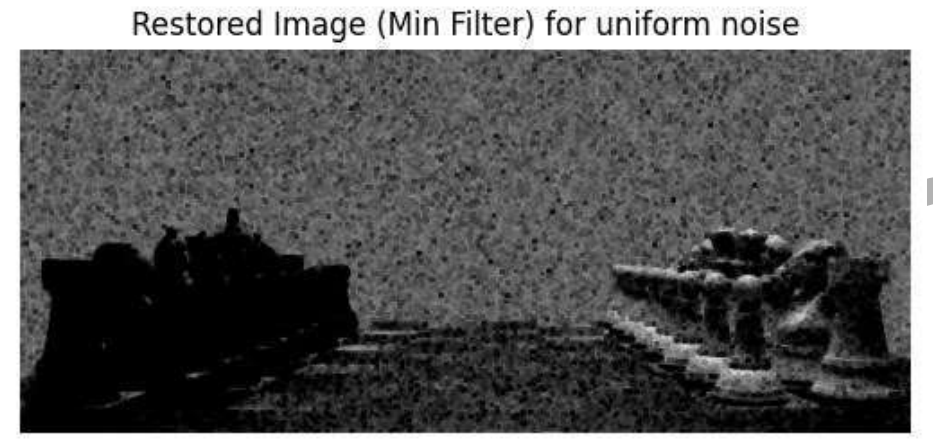
```
image = cv2.imread("chess 1.jpg",
cv2.IMREAD_GRAYSCALE) sobel_x = cv2.Sobel(image,
cv2.CV_64F, 1, 0, ksize=3)   sobel_y = cv2.Sobel(image,
cv2.CV_64F, 0, 1, ksize=3) sobel_magnitude =
cv2.magnitude(sobel_x, sobel_y) sobel_magnitude =
np.uint8(np.absolute(sobel_magnitude)) restored_image =
```

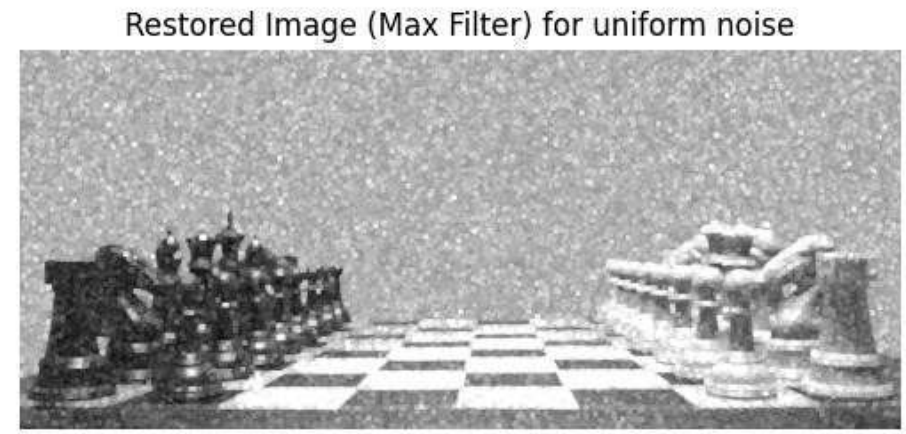
cv2.medianBlur(sobel_magnitude, 5)

plt.imshow(restored_image, cmap='gray') plt.title("Restored

Image (Median Filter)") plt.axis('off') plt.show()

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 saltand-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:

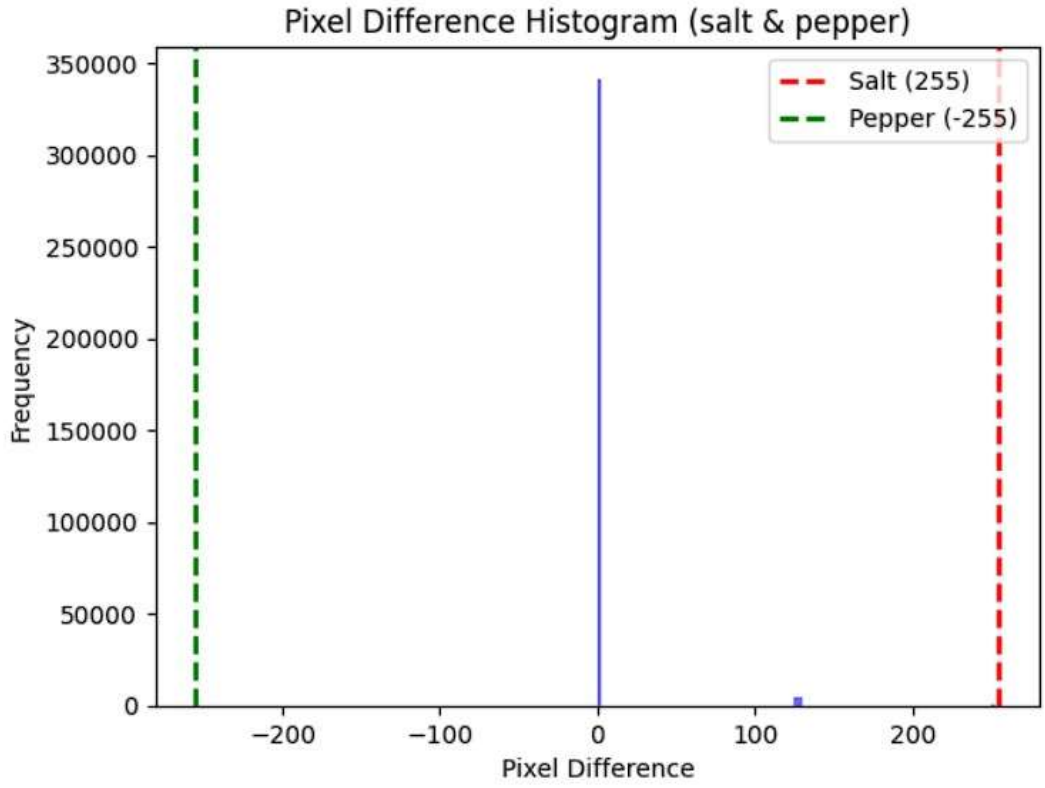
- Finally, the **restored image** after applying the median filter is displayed, showing how the noise is reduced.

CODE:

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

image = cv2.imread("chess 1.jpg",

cv2.IMREAD_GRAYSCALE) salt_prob, pepper_prob = 0.02,

0.02 noisy_image = image.copy()

noisy_image[np.random.rand(*image.shape) < salt_prob] = 255

noisy_image[np.random.rand(*image.shape) < pepper_prob] = 0

plt.figure(figsize=(10, 5)) 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(salt & pepper"), plt.axis("off")
plt.show()

diff = noisy_image - image
plt.hist(diff.ravel(), bins=100, color='blue', alpha=0.6)
plt.axvline(255, color='red', linestyle='dashed', linewidth=2, label='Salt (255)')
plt.axvline(-255, color='green', linestyle='dashed', linewidth=2, label='Pepper (-255)') plt.legend() plt.title("Pixel Difference
Histogram (salt & pepper)") plt.xlabel("Pixel
Difference") plt.ylabel("Frequency") plt.show()
```

OUTPUT:

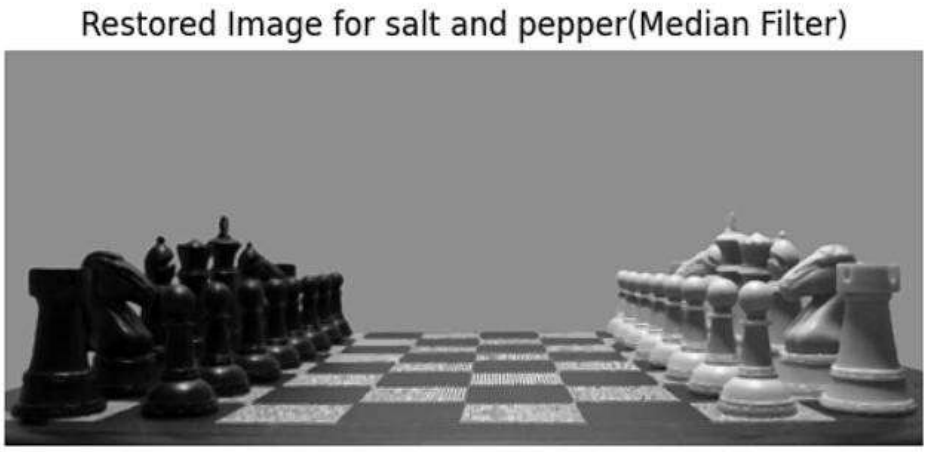


RESTORED IMAGE FOR SALT & PEPPER (MEDIAN FILTER):

CODE:

```
restored_image = cv2.medianBlur(noisy_image, 5)  #
Corrected plt.imshow(restored_image, cmap='gray')
plt.title("Restored Image for salt and pepper(Median Filter)")
plt.axis('off') plt.show()
```

OUTPUT:



EDGE DETECTION-SOBERT(MANUAL)

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:

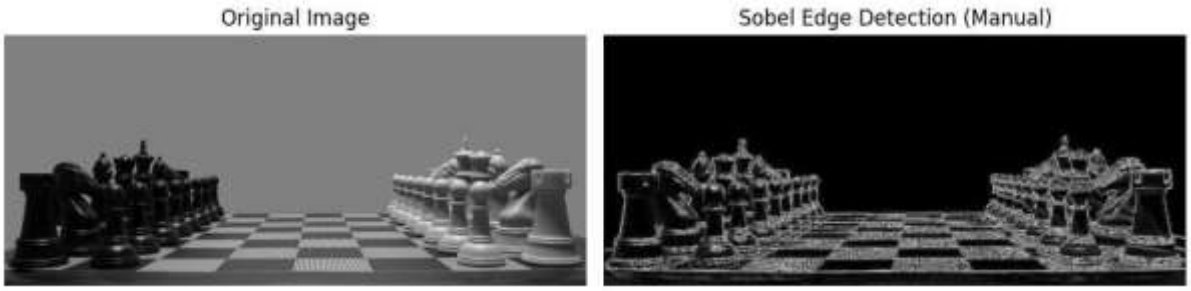
- First, it displays the **original grayscale image**.
- Then it displays the **Sobel edge-detected image** to show the result of edge detection.

CODE:

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

image = cv2.imread("chess 1.jpg",

cv2.IMREAD_GRAYSCALE) height, width = image.shape

sobel_x = np.array([[-1, 0, 1],

                    [-2, 0, 2],

[-1, 0, 1]]) sobel_y =

np.array([[-1, -2, -1],

                    [0, 0, 0],

                    [1, 2, 1]])

gradient_x = np.zeros_like(image, dtype=np.float32)

gradient_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]

grad_x = np.sum(region * sobel_x)

grad_y = np.sum(region * sobel_y)
```

```
gradient_x[i, j] = grad_x        gradient_y[i, j]
= grad_y
gradient_magnitude = np.sqrt(gradient_x**2 + gradient_y**2)
gradient_magnitude =
np.uint8(np.absolute(gradient_magnitude))
plt.figure(figsize=(10, 5)) plt.subplot(1, 2, 1) plt.imshow(image,
cmap='gray') plt.title("Original Image") plt.axis('off')
plt.subplot(1, 2, 2) plt.imshow(gradient_magnitude,
cmap='gray') plt.title("Sobel Edge Detection (Manual)")
plt.axis('off') plt.tight_layout() 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()

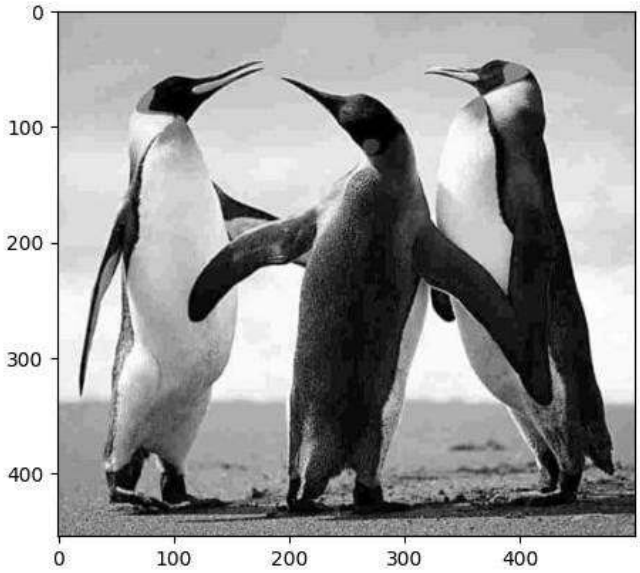
CODE:

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

image = cv2.imread(R"C:\Users\revaa\Downloads\penguin.jpg")

plt.imshow(image) plt.show()

OUTPUT:



ROBERT CROSS DETECTION:

ALGORITHM:

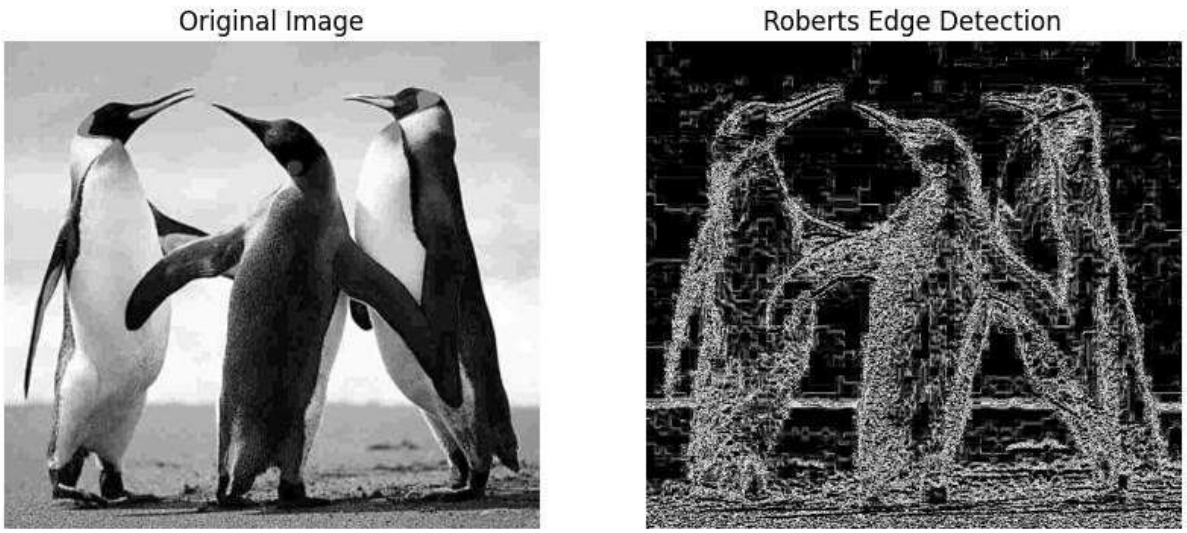
1. **Import necessary libraries** (cv2, numpy, matplotlib.pyplot).
2. **Read the image in grayscale** using cv2.imread().
3. **Define Roberts Cross operator kernels** (roberts_x and roberts_y).
4. **Apply convolution** using cv2.filter2D() to get edge responses (edges_x and edges_y).
5. **Compute edge 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()

CODE:

```
image = cv2.imread(r"C:\Users\revaa\Downloads\penguin.jpg",
cv2.IMREAD_GRAYSCALE)

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

roberts_y = np.array([[0, 1], [-1, 0]] ) edges_x
```

```
= cv2.filter2D(image, -1, roberts_x) edges_y
= cv2.filter2D(image, -1, roberts_y) edges =
np.sqrt(edges_x**2 + edges_y**2) edges =
np.uint8(edges) plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1) plt.imshow(image,
cmap="gray") plt.title("Original Image")
plt.axis("off") plt.subplot(1, 2, 2)
plt.imshow(edges, cmap="gray")
plt.title("Roberts Edge Detection")
plt.axis("off") plt.show()
```

OUTPUT:



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:

```
image = cv2.imread(r"C:\Users\revaa\Downloads\penguin.jpg",
cv2.IMREAD_GRAYSCALE)

prewitt_x = np.array([[-1, 0, 1],

                      [-1, 0, 1],

[-1, 0, 1]])

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

                      [0, 0, 0],

[1, 1, 1]])

edges_x = cv2.filter2D(image, -1, prewitt_x)

edges_y = cv2.filter2D(image, -1, prewitt_y)

edges = np.sqrt(edges_x**2 + edges_y**2)

plt.figure(figsize=(10, 5)) plt.subplot(1, 3, 1)

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

plt.title("Prewitt - X Direction") plt.axis("off")


plt.subplot(1, 3, 2)

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

```
plt.title("Prewitt - Y Direction")
plt.axis("off") plt.subplot(1, 3, 3)
plt.imshow(edges, cmap="gray")
plt.title("Prewitt - Combined")
plt.axis("off") plt.show()
```

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().

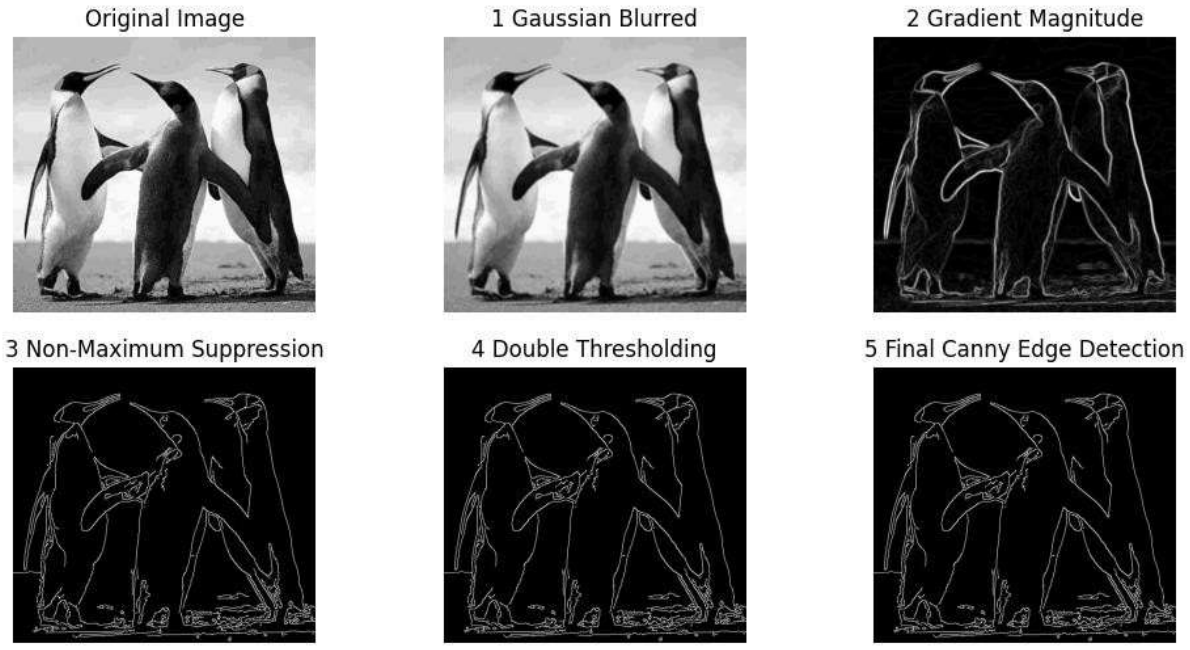
Show final Canny edge-detected image using plt.show().

CODE:

```
image = cv2.imread(r"C:\Users\revaa\Downloads\penguin.jpg",
cv2.IMREAD_GRAYSCALE)

blurred = cv2.GaussianBlur(image, (5, 5), 1.4)   sobel_x

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

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

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

gradient_magnitude = np.uint8(gradient_magnitude * 255 /

np.max(gradient_magnitude)) gradient_direction =

np.arctan2(sobel_y, sobel_x)   edges_nms =

cv2.Canny(blurred, 50, 150)

strong_edges = (edges_nms >= 150).astype(np.uint8) * 255

weak_edges = ((edges_nms >= 50) & (edges_nms < 150)).astype(np.uint8) *
255

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

plt.subplot(2, 3, 1)

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

plt.title('Original Image')

plt.axis("off") plt.subplot(2, 3, 2)

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

plt.title('1 Gaussian Blurred')

plt.axis("off") plt.subplot(2, 3, 3)

plt.imshow(gradient_magnitude, cmap='gray') plt.title('2

Gradient Magnitude')

plt.axis("off") plt.subplot(2, 3, 4)

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

```
plt.title('3 Non-Maximum Suppression')
plt.axis("off") plt.subplot(2, 3, 5)
plt.imshow(strong_edges + weak_edges, cmap='gray')
plt.title('4 Double Thresholding')
plt.axis("off") plt.subplot(2, 3, 6)
plt.imshow(edges_nms, cmap='gray')
plt.title('5 Final Canny Edge Detection')
plt.axis("off") plt.show()
```

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.

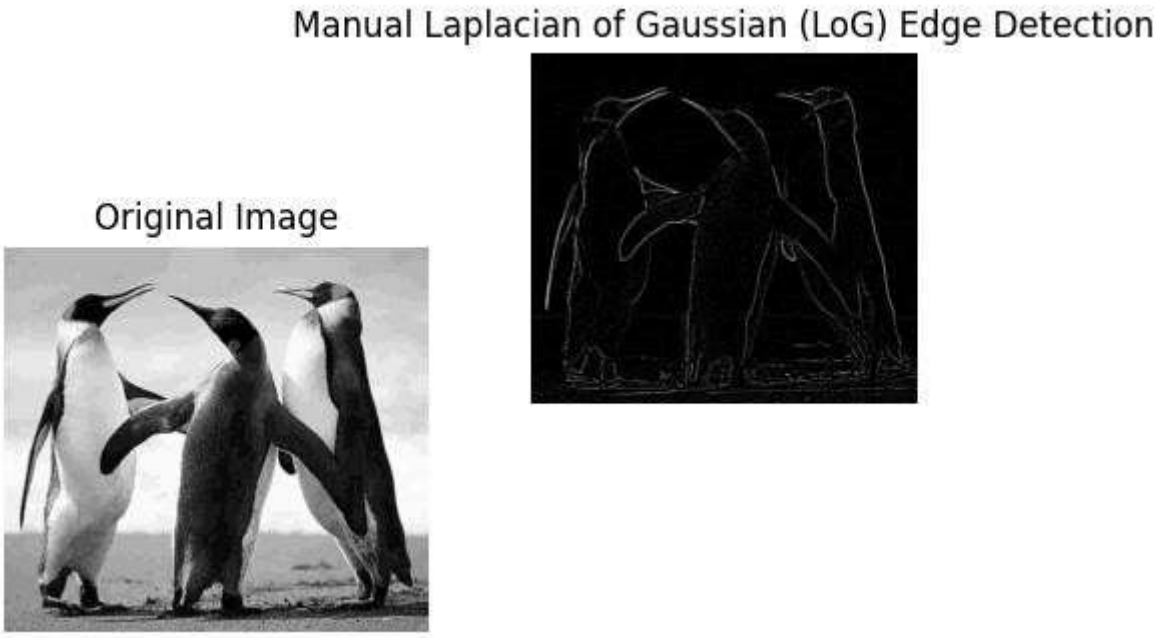
4. **Define the Laplacian kernel** for edge detection.
5. **Apply convolution** using cv2.filter2D() to detect edges.
6. **Convert edge values to absolute and normalize** for better visualization.
7. **Display the original and edge-detected images** using plt.subplot() and plt.imshow().
8. **Show the final results** using plt.show().

CODE:

1: image =

```
cv2.imread(r"C:\Users\revaa\Downloads\penguin.jpg",
cv2.IMREAD_GRAYSCALE)
gaussian_blur = cv2.GaussianBlur(image, (5, 5), 1.4)  laplacian_kernel
= np.array([[0,  1,  0],
                    [1, -4,  1],
[0,  1,  0]])
log_edges = cv2.filter2D(gaussian_blur, -1, laplacian_kernel)
log_edges = np.abs(log_edges)   log_edges = (log_edges /
np.max(log_edges) * 255).astype(np.uint8)   plt.figure(figsize=(6,
5)) plt.subplot(1, 2, 1) plt.imshow(image, cmap="gray")
plt.title("Original Image") plt.axis("off") plt.subplot(2, 2, 2)

plt.imshow(log_edges, cmap="gray") plt.title("Manual
Laplacian of Gaussian (LoG) Edge Detection") plt.axis("off")
plt.show()
```

OUTPUT:



2:

ALGORITHM:

1. **Import necessary libraries** (cv2, numpy, matplotlib.pyplot, scipy.ndimage).
2. **Read the image in grayscale** using cv2.imread().
3. **Convert image to float** for precise calculations.
4. **Apply Gaussian Blur** using gaussian_filter() to smooth the image.
5. **Compute first-order derivatives (gradients)** in both X (dx) and Y (dy) directions.
6. **Compute second-order derivatives** (dxx, dyy) using gradient values.
7. **Compute Laplacian of Gaussian (LoG)** by summing dxx and dyy.
8. **Normalize the result** to 0-255 and convert to 8-bit format.
9. **Display the original image, Gaussian-blurred image, and LoG edgedetected image** using plt.subplot().
10. **Show the final results** using plt.show()

CODE:

```
from scipy.ndimage import gaussian_filter

import cv2 import numpy as np import

matplotlib.pyplot as plt

image = cv2.imread(r"C:\Users\revaa\Downloads\penguin.jpg",
cv2.IMREAD_GRAYSCALE)

image = image.astype(np.float32)   sigma = 1.0  #

Standard deviation for Gaussian smoothing smoothed =

gaussian_filter(image, sigma=sigma)   dx =

np.zeros_like(smoothed) dy = np.zeros_like(smoothed)

dx[:, 1:-1] = (smoothed[:, 2:] - smoothed[:, :-2]) / 2

dy[1:-1, :] = (smoothed[2:, :] - smoothed[:-2, :]) / 2

dxx = np.zeros_like(smoothed) dyy =

np.zeros_like(smoothed)

dxx[:, 1:-1] = (dx[:, 2:] - dx[:, :-2]) / 2   dyy[1:-

1, :] = (dy[2:, :] - dy[:-2, :]) / 2   log = dxx +

dyy   log = np.abs(log)   log = (log /

np.max(log) * 255).astype(np.uint8)

plt.figure(figsize=(10, 6)) plt.subplot(1, 3, 1)

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

plt.title("Original Image")

plt.axis("off") plt.subplot(1,

3, 2)

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

plt.title(f"Gaussian Blurred (σ={sigma})")
```

```
plt.axis("off") plt.subplot(1, 3, 3)
plt.imshow(log, cmap='gray')
plt.title("Laplacian of Gaussian (LoG)")
plt.axis("off") plt.show()
```

OUTPUT:

DOG:

ALGORITHM:

Load the Image

- Read the input image in grayscale using OpenCV (cv2.imread).

Apply Gaussian Blur with Two Different Standard Deviations

- Use scipy.ndimage.gaussian_filter to apply Gaussian blur with standard deviation σ1 (e.g., sigma1 = 1).
- Apply another Gaussian blur with a different standard deviation σ2 (e.g., sigma2 = 2).

Compute the Difference of Gaussian (DoG)

- Subtract the two blurred images:
    - DoG = Blurred Image (σ1) - Blurred Image (σ2).

Display the Results

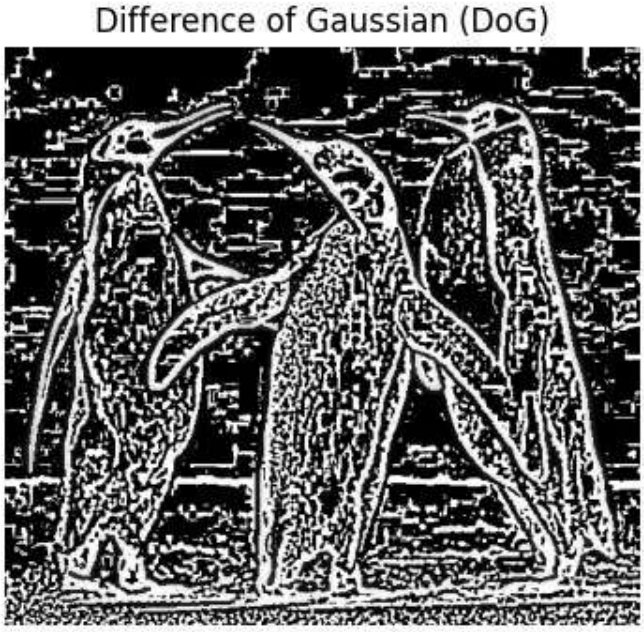
- Show the original image.

- Show the two blurred images (σ1 and σ2).
- Show the DoG image using matplotlib.

CODE:

```
from scipy.ndimage
import gaussian_filter
image = cv2.imread(r"C:\Users\revaa\Downloads\penguin.jpg",
cv2.IMREAD_GRAYSCALE)
sigma1 = 1   sigma2 = 2  blurred1 =
gaussian_filter(image, sigma=sigma1)   blurred2
= gaussian_filter(image, sigma=sigma2)   dog =
blurred1 - blurred2   plt.figure(figsize=(6, 6))
plt.subplot(1, 3, 1) plt.imshow(image,
cmap='gray') plt.title("Original Image")
plt.axis("off") plt.subplot(1, 3, 2)
plt.imshow(blurred1, cmap='gray')
plt.title(f"Gaussian Blur (σ={sigma1})")
plt.axis("off") plt.subplot(1, 3, 3)
plt.imshow(blurred2, cmap='gray')
plt.title(f"Gaussian Blur (σ={sigma2})")
plt.axis("off") plt.figure(figsize=(6, 4))
plt.imshow(dog, cmap='gray')
plt.title("Difference of Gaussian (DoG)")
plt.axis("off") plt.show()
```

OUTPUT:



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

EXERCISE NO: 7	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.

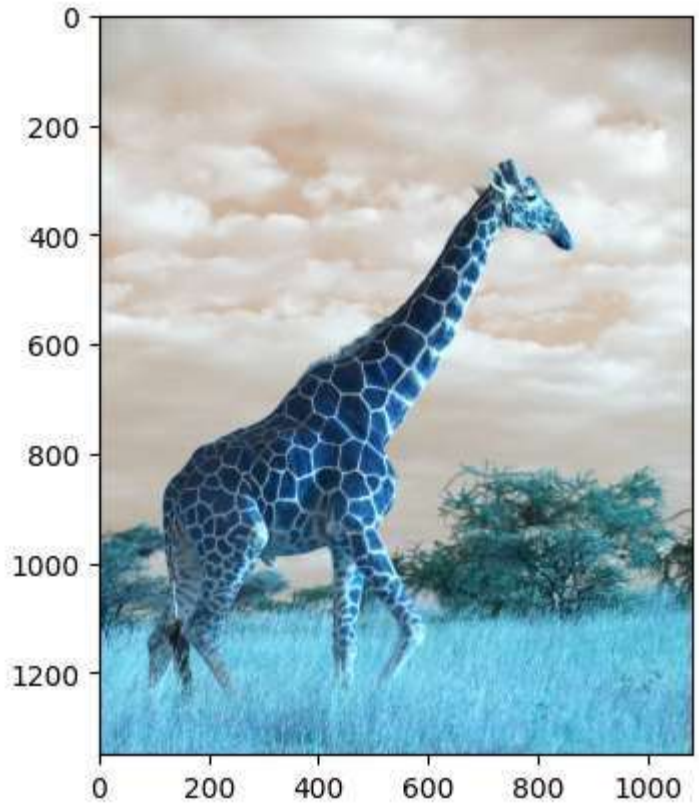
   - Import matplotlib.pyplot for displaying the image.
3. **Read the image**
   - Use cv2.imread("image_path") to load the image from the specified file path.
4. **Display the image using Matplotlib** ○ Use

   plt.imshow(image) to display the image.

   - Use plt.show() to render the output.
5. **End CODE:**

```
import cv2
import matplotlib.pyplot as plt image =
cv2.imread(r"C:\Users\madhu\Downloads\giraffe.jpg")
plt.imshow(image) plt.show()
```

OUTPUT:

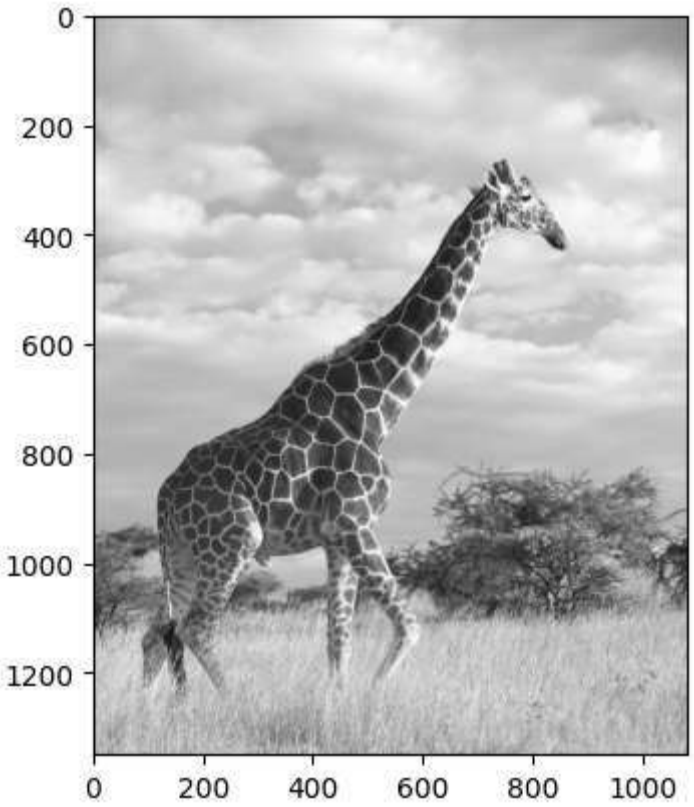


CONVERT INTO GRAY SCALE:

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

plt.imshow(gray_image, cmap='gray') plt.show()

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.

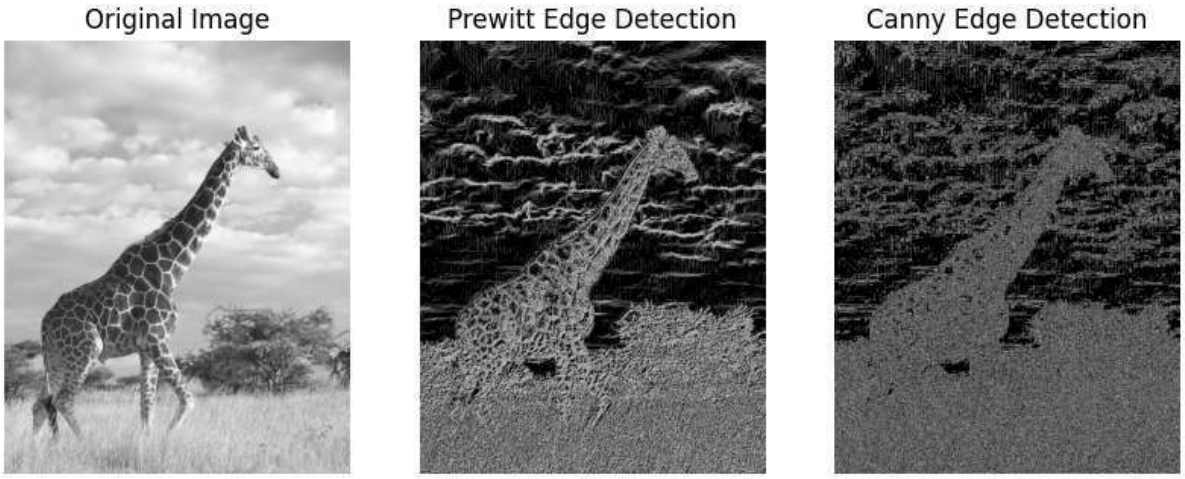
– **End**

CODE:

```
import cv2 import numpy as

np import matplotlib.pyplot as

plt

image = cv2.imread(r"C:\Users\madhu\Downloads\giraffe.jpg",
cv2.IMREAD_GRAYSCALE)

prewitt_kernel_x = np.array([[ -1, 0, 1],

                    [ -1, 0, 1],

                    [ -1, 0, 1]])


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

                    [ 0,  0,  0],

                    [ 1,  1,  1]])

gradient_x = cv2.filter2D(image, -1, prewitt_kernel_x) gradient_y =

cv2.filter2D(image, -1, prewitt_kernel_y) gradient_magnitude =

np.sqrt(gradient_x**2 + gradient_y**2) gradient_magnitude =

np.uint8(np.clip(gradient_magnitude, 0, 255))

normalized_gradient_magnitude = cv2.normalize(gradient_magnitude, None, 0,
255, cv2.NORM_MINMAX)

canny_edges = cv2.Canny(normalized_gradient_magnitude, 50, 150)  # Adjust

thresholds as necessary plt.figure(figsize=(10, 7)) plt.subplot(1, 3, 1)

plt.imshow(image, cmap='gray') plt.title('Original Image') plt.axis('off')


plt.subplot(1, 3, 2)
```

```
plt.imshow(gradient_magnitude, cmap='gray')
plt.title('Prewitt Edge Detection') plt.axis('off')
plt.subplot(1, 3, 3)
plt.imshow(canny_edges, cmap='gray')
plt.title('Canny Edge Detection')
plt.axis('off') plt.show()
```

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 8bit 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**

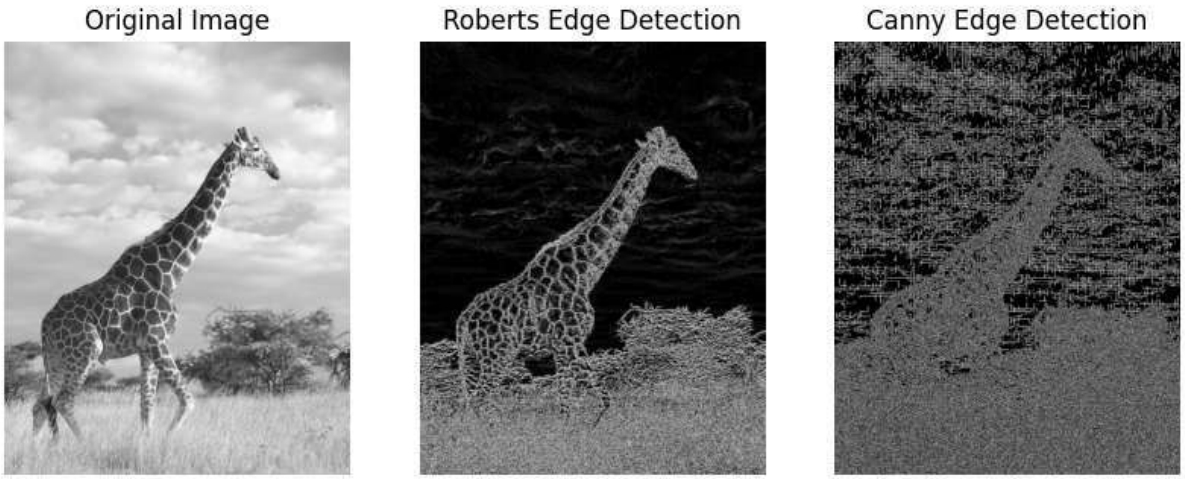
CODE:

```
import cv2 import numpy as

np import matplotlib.pyplot as

plt

image = cv2.imread(r"C:\Users\madhu\Downloads\giraffe.jpg", cv2.IMREAD_GRAYSCALE)

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


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

gradient_x = cv2.filter2D(image, -1, roberts_kernel_x) gradient_y

= cv2.filter2D(image, -1, roberts_kernel_y)


gradient_magnitude = np.sqrt(gradient_x**2 + gradient_y**2)
```

```
# Normalize the magnitude to range [0, 255] for display gradient_magnitude
= np.uint8(np.clip(gradient_magnitude, 0, 255)) # Normalize the gradient
magnitude for better visualization

normalized_gradient_magnitude = cv2.normalize(gradient_magnitude, None, 0,
255, cv2.NORM_MINMAX)


# Apply Canny edge detection on the normalized gradient
canny_edges = cv2.Canny(normalized_gradient_magnitude, 50, 150)  # Adjust
thresholds as necessary


# Plotting the results plt.figure(figsize=(10,
7))


# Original image plt.subplot(1,
3, 1) plt.imshow(image,
cmap='gray') plt.title('Original
Image') plt.axis('off')
# Roberts edge detection result plt.subplot(1,
3, 2)
plt.imshow(gradient_magnitude, cmap='gray')
plt.title('Roberts Edge Detection') plt.axis('off')
# Canny edge detection result plt.subplot(1,
3, 3)
plt.imshow(canny_edges, cmap='gray')
plt.title('Canny Edge Detection') plt.axis('off')
```

plt.show()

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.

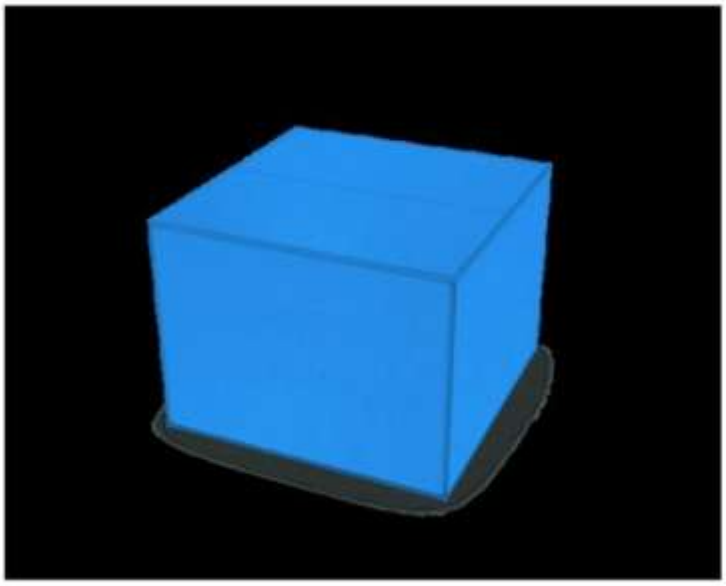
Display the Image

- 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 image =
cv2.imread(r"C:\Users\student\Desktop\sportsshop\exp4\oo.jpg")
plt.imshow(image) plt.show()
```

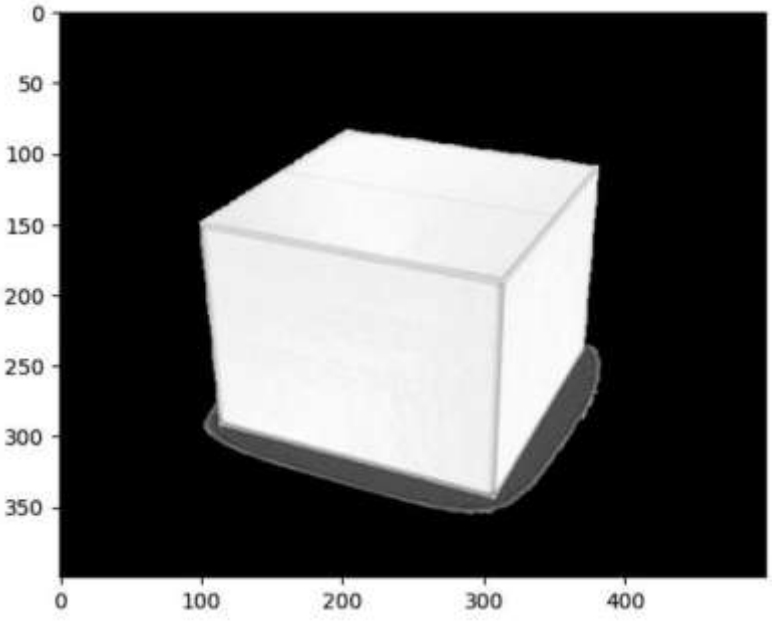
OUTPUT:



CONVERT INTO GRAYSCALE:

```
gray_image = cv2.cvtColor(image,
cv2.COLOR_BGR2GRAY) plt.imshow(gray_image,
cmap='gray') plt.show()
```

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 Green** ○ Identify the strong corner responses (dst > 0.01 * dst.max()).

- Set the corresponding pixels in the color image to **green (0, 255, 0)**.

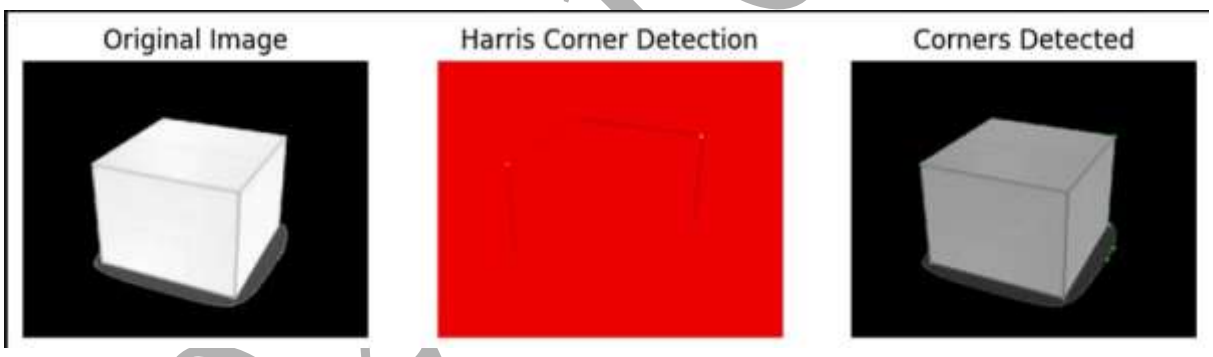
8. **Display the Results Using Matplotlib** ○ Create a figure with **3 subplots**:
   - **Original grayscale image**.
   - **Harris response heatmap** (hot colormap).
   - **Image with detected corners** (converted to RGB for correct display in Matplotlib).
9. **Show the Plots** ○ Use plt.show() to display the results.

CODE:

```
import cv2 import numpy as
np import matplotlib.pyplot
as plt
image = cv2.imread(r"C:\Users\student\Desktop\sportsshop\exp4\oo.jpg",
cv2.IMREAD_GRAYSCALE)
blurred_image = cv2.GaussianBlur(image, (5, 5),
0) dst = cv2.cornerHarris(blurred_image, 2, 3,
0.04) dst = cv2.dilate(dst, None)
image_color = cv2.cvtColor(image,
cv2.COLOR_GRAY2BGR) image_color[dst > 0.01 *
dst.max()] = [0, 255, 0]   plt.figure(figsize=(10, 7))
plt.subplot(1, 3, 1) plt.imshow(image, cmap='gray')
plt.title('Original Image') plt.axis('off') plt.subplot(1, 3, 2)
```

```
plt.imshow(dst, cmap='hot')   plt.title('Harris Corner
Detection')

plt.axis('off')
plt.subplot(1, 3, 3)
plt.imshow(cv2.cvtColor(image_color,
cv2.COLOR_BGR2RGB))   plt.title('Corners Detected')
plt.axis('off')


plt.show()
```

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

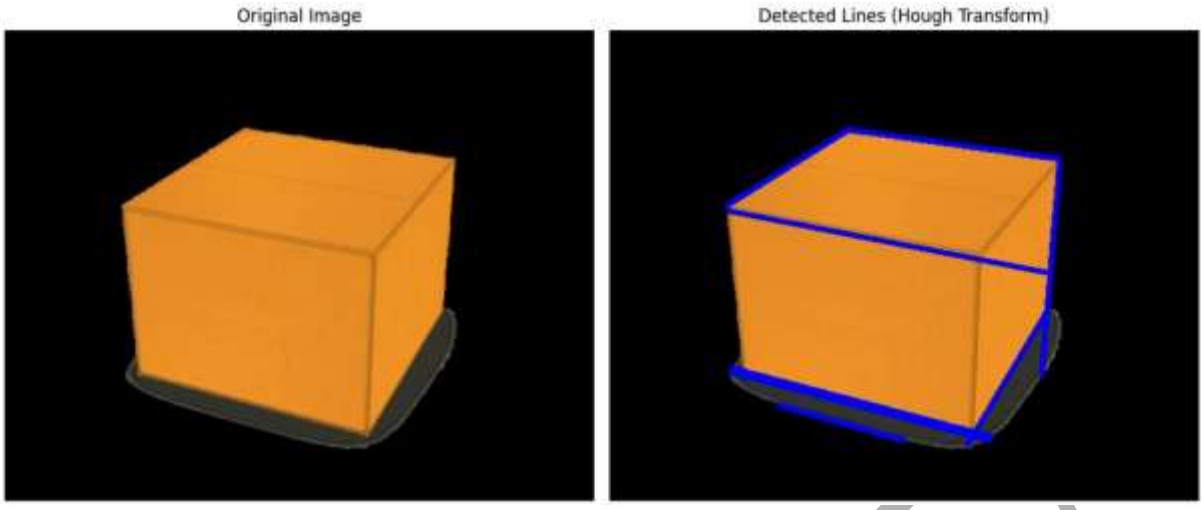
   the layout.

   - Use plt.show() to display the images.

CODE:

```
import numpy as np
import cv2
import matplotlib.pyplot as plt
img = cv2.imread(r"C:\Users\student\Desktop\sportsshop\exp4\oo.jpg",
cv2.IMREAD_COLOR)
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
edges = cv2.Canny(gray, 50, 200)

lines = cv2.HoughLinesP(edges, 1, np.pi/180, 80,
minLineLength=15, maxLineGap=250) img_with_lines =
img.copy()   for line in lines:     x1, y1, x2, y2 = line[0]
   cv2.line(img_with_lines, (x1, y1), (x2, y2), (255, 0, 0),
3)  plt.figure(figsize=(12, 6)) plt.subplot(1, 2, 1)
plt.imshow(cv2.cvtColor(img,
cv2.COLOR_BGR2RGB))   plt.title('Original Image')
plt.axis('off') plt.subplot(1, 2, 2)
plt.imshow(cv2.cvtColor(img_with_lines,
cv2.COLOR_BGR2RGB))          plt.title('Detected   Lines   (Hough
Transform)') plt.axis('off') plt.tight_layout() plt.show()
```

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 o Use cv2.SIFT_create() to create a SIFT object. o Detect keypoints and compute descriptors. o If descriptors are None, return a zero vector. o Otherwise, compute the mean of descriptors to get a fixed-length feature vector.
2. HOG Feature Extraction o 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). o Extract SIFT, HOG, and GLOH features. o 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 ₀ If a cluster

   has less than 2 keypoints, skip it.

2. Use FLANN-based Feature Matching o

Convert descriptors to float32 type.

o Use FLANN KNN Matcher to find good matches (Lowe's ratio test).

3. Filter Good Matches o Keep matches where distance(m) < 0.5

* distance(n). o Ensure len(good) >

MIN_MATCH_COUNT to proceed.

Step 9: Find Homography & Transform Points

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

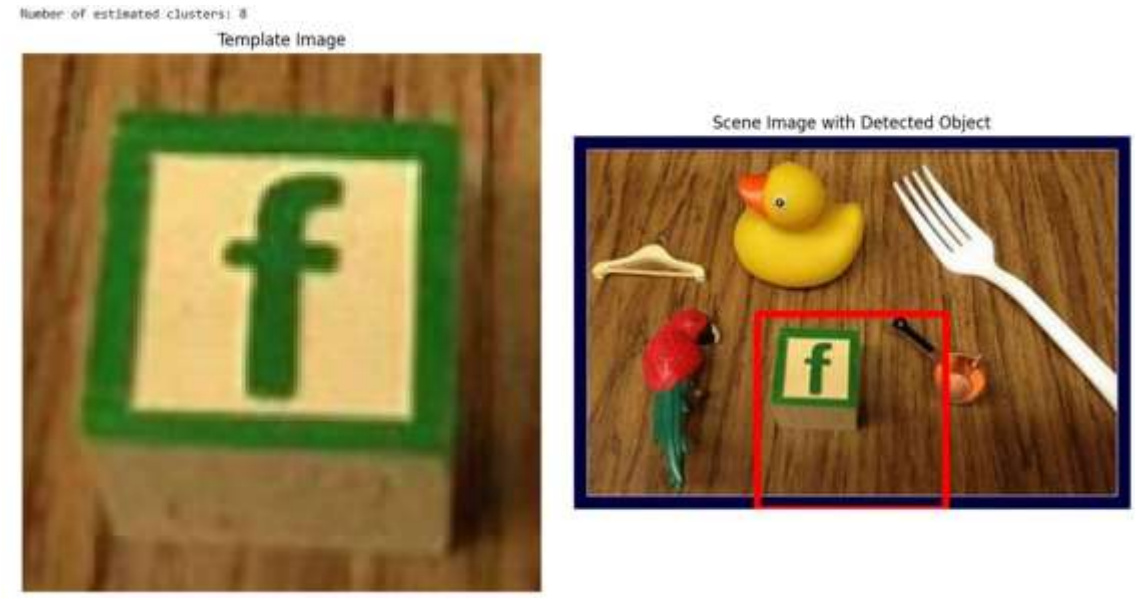
CODE:

```
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
202503-17 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 o Load the image using OpenCV (cv2.imread). o Convert it from BGR (default OpenCV format) to RGB for proper visualization. o 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 o 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 o Replace each pixel with its corresponding cluster center. o Reshape the processed pixel array back to the original image dimensions. o 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(r"C:\Users\student\Downloads\home.jfif")
image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
pixels = image.reshape((-1, 3))   pixels = np.float32(pixels)   K = 3
np.random.seed(42)  random_indices =
np.random.choice(len(pixels), K, replace=False) centers =
pixels[random_indices] max_iters = 100   for _ in
range(max_iters)
   distances = np.linalg.norm(pixels[:, np.newaxis] - centers, axis=2)  #
Euclidean distance
   labels = np.argmin(distances, axis=1)  # Assign each pixel to the closest
center
   new_centers = np.array([pixels[labels == k].mean(axis=0) if np.any(labels
== k) else centers[k] for k in range(K)])     if np.allclose(centers, new_centers,
atol=1e-3):
       break
centers = new_centers  # Update centers segmented_pixels
= centers[labels]
segmented_image = segmented_pixels.reshape(image.shape)  # Reshape back
to image segmented_image = np.uint8(segmented_image)
```

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(f"Manual K-Means Segmentation (K={K})") plt.axis("off") plt.show()

OUTPUT:

B

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:

```
import numpy as np import matplotlib.pyplot as plt from PIL import Image
# Using PIL to open the image image =
Image.open(r"C:\Users\student\Downloads\home.jfif").convert("L")
image_np = np.array(image)   threshold = 100  binary_image =
np.where(image_np > threshold, 255, 0)   plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1) plt.imshow(image_np, cmap="gray") plt.title("Original
Grayscale Image") plt.axis("off") plt.subplot(1, 2, 2)
plt.imshow(binary_image, cmap="gray") plt.title(f"Thresholded Image
(T={threshold})") 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. o 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:

```
import cv2 import numpy as np

import matplotlib.pyplot as plt

image = cv2.imread(r"C:\Users\student\Downloads\home.jfif")  # Replace with your image path
```

```
image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)  # Convert BGR to
RGB for correct display mask = np.zeros(image.shape[:2], np.uint8)
bgd_model = np.zeros((1, 65), np.float64) fgd_model = np.zeros((1, 65),
np.float64) rect = (50, 50, image.shape[1] - 100, image.shape[0] - 100)  #
Adjust as needed
cv2.grabCut(image, mask, rect, bgd_model, fgd_model, 5,
cv2.GC_INIT_WITH_RECT)
mask_final = np.where((mask == 2) | (mask == 0), 0, 1).astype("uint8")
segmented_image = image * mask_final[:, :, np.newaxis]
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 (GrabCut)")
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: o sp = 20: Defines the spatial window radius for filtering. o 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:

```
import cv2 import numpy as np
import matplotlib.pyplot as plt
image = cv2.imread(r"C:\Users\student\Downloads\home.jfif")  # Replace with your image path
image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)  # Convert BGR
to RGB for correct display 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. o Check if it is already visited. o 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:

```
import numpy as np import
matplotlib.pyplot as plt from PIL
import Image
image_path = r"C:\Users\student\Downloads\home.jfif"  # Use raw string
for Windows paths image = Image.open(image_path).convert("L")  #
Convert to grayscale image_np = np.array(image) seed_x, seed_y = 150,
120 # Change as needed threshold = 50 segmented =
np.zeros_like(image_np, dtype=np.uint8) stack = [(seed_x, seed_y)] visited
= set()  # To track visited pixels height, width = image_np.shape
neighbors = [(-1, 0), (1, 0), (0, -1), (0, 1), (-1, -1), (-1, 1), (1, -1), (1, 1)]
while stack:

   x, y = stack.pop()     if (x, y) in visited:          continue     visited.add((x,
y))     if abs(int(image_np[x, y]) - int(image_np[seed_x, seed_y])) <
threshold:
        segmented[x, y] = 255  # Mark as foreground
```

```
        # Add neighbors to the stack        for dx, dy in neighbors:            nx,
ny = x + dx, y + dy            if 0 <= nx < height and 0 <= ny < width and
(nx, ny) not in visited:
                stack.append((nx, ny))
plt.figure(figsize=(10, 5)) plt.subplot(1, 2, 1)
plt.imshow(image_np, cmap="gray")
plt.title("Original Image") plt.axis("off")
plt.subplot(1, 2, 2) plt.imshow(segmented,
cmap="gray") plt.title("Segmented Image
(Region Growing)") plt.axis("off") plt.show()
```

OUTPUT:

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.

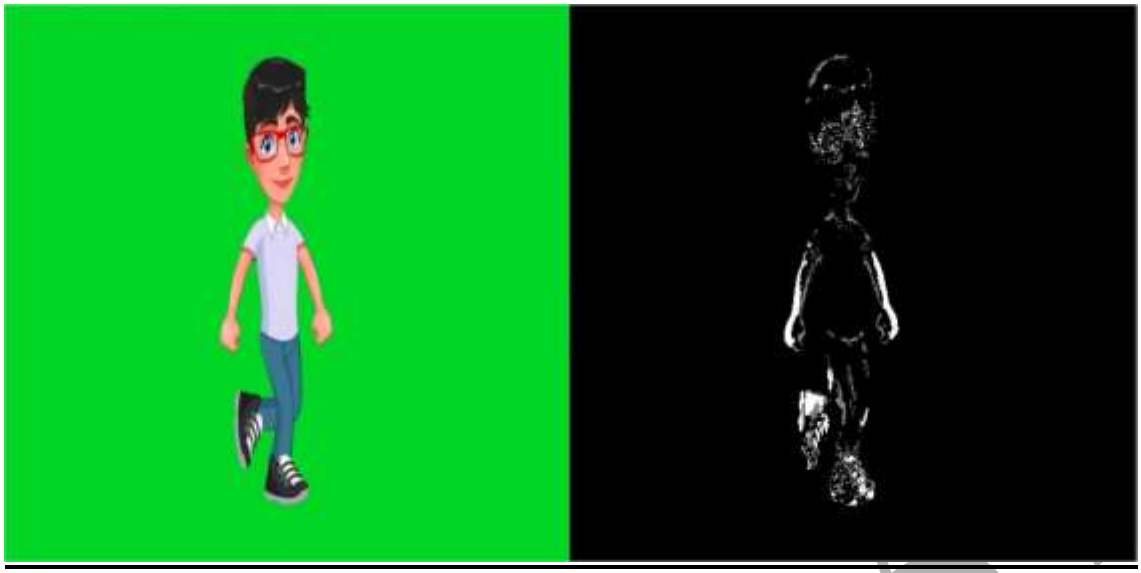
ALGORITHM:

- 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.

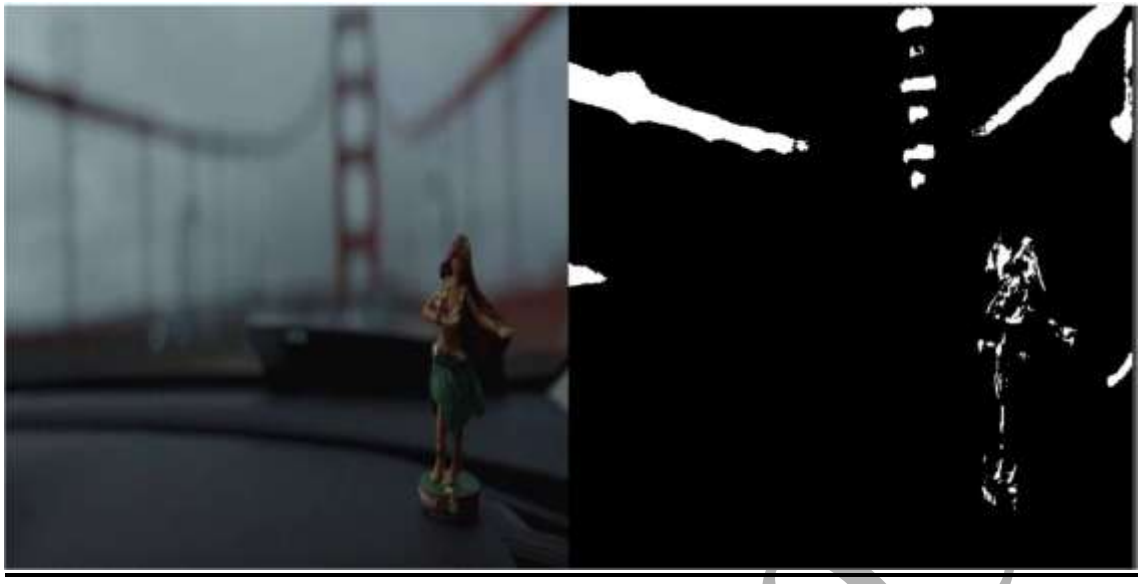
- Count the number of changed pixels in the foreground mask.
- If the count exceeds the threshold, print "Scene change detected!".
- Display the original frame and the foreground mask side by side.
- Wait for user input (q) to exit.
- Release resources and close the display window.
- End

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\Download
s\video4.mp4")
```

OUTPUT:



RESULT:

We have performed background subtraction in video for motion detection.

EXERCISE NO: 12	LUCAS KANNADE ALGORITHM
DATE: 03/04/2025	

AIM:

To perform the lucas kannade algorithm between two frame differences and in video.

TWO FRAME DIFFERENCES:

ALGORITHM:

- **Import Tools**

Use OpenCV to read video and Matplotlib to show images.

- **Open the Video File**

Give the path of the video and open it using OpenCV.

- **Read Two Frames**

Read the first and second frames from the video.

- **Stop Reading Video**

Release the video file after reading.

- **Check if Frames are Read**

If both frames are read properly, go to the next step.

- **Convert Colors**

Convert frames from BGR to RGB so that Matplotlib can display them correctly.

- **Display Frames**

Show both frames side by side using Matplotlib.

- **Error Handling**

If frames are not read, show an error message.

CODE:

```
import cv2
import matplotlib.pyplot as plt video_path =
r"C:\Users\madhu\Downloads\video4.mp4" cap =
cv2.VideoCapture(video_path) ret1, frame1 = cap.read()
ret2, frame2 = cap.read() cap.release() if ret1 and ret2:
   frame1 = cv2.cvtColor(frame1, cv2.COLOR_BGR2RGB)
frame2 = cv2.cvtColor(frame2, cv2.COLOR_BGR2RGB)
   plt.figure(figsize=(10, 5))     plt.subplot(1, 2, 1)
plt.imshow(frame1)     plt.title("Frame 1")
plt.axis("off")     plt.subplot(1, 2, 2)
plt.imshow(frame2)     plt.title("Frame 2")
plt.axis("off")     plt.show() else:     print("Error:
Could not read frames from video.")
```

OUTPUT:

ALGORITHM:

- **Check if the frames are loaded**

If not, stop the program.

- **Make both frames the same size**

Resize if needed.

- **Convert frames to black and white**

(Grayscale – easier to work with).

- **Find important points in the first frame**

Like corners or edges to track.

- **Check if we found points**

If not, stop the program.

- **Track where those points moved in the second frame**

Use Lucas-Kanade method.

- **Keep only the points that moved properly**
- **Draw arrows (lines) showing movement from old to new point**
- **Show the result using Matplotlib**
- **If tracking didn't work, show an error**

CODE:

```
import cv2 import numpy as np import
matplotlib.pyplot as plt if frame1 is None
or frame2 is None:    print("Error: Could
not load frames.")
   exit()
frame2 = cv2.resize(frame2, (frame1.shape[1], frame1.shape[0]))
gray1 = cv2.cvtColor(frame1, cv2.COLOR_BGR2GRAY) gray2 =
cv2.cvtColor(frame2, cv2.COLOR_BGR2GRAY)
feature_params = dict(maxCorners=100, qualityLevel=0.3, minDistance=7,
blockSize=7)
```

```
p0 = cv2.goodFeaturesToTrack(gray1, mask=None, **feature_params) if
p0 is None:     print("No good features found in frame1.")
   exit()
lk_params = dict(winSize=(15, 15), maxLevel=2,
                 criteria=(cv2.TERM_CRITERIA_EPS |
cv2.TERM_CRITERIA_COUNT, 10, 0.03))
p1, st, err = cv2.calcOpticalFlowPyrLK(gray1, gray2, p0, None, **lk_params)
if p1 is not None and st is not None:     good_new = p1[st == 1]     good_old =
p0[st == 1]     for i, (new, old) in enumerate(zip(good_new, good_old)):
       a, b = new.ravel()         c, d = old.ravel()         frame2 = cv2.line(frame2,
(int(a), int(b)), (int(c), int(d)), (0, 255, 0), 2)         frame2 =
cv2.circle(frame2, (int(a), int(b)), 5, (0, 0, 255), -1)

   frame2_rgb = cv2.cvtColor(frame2, cv2.COLOR_BGR2RGB)
   plt.figure(figsize=(8, 6))
plt.imshow(frame2_rgb)     plt.title("Lucas-
Kanade Optical Flow")     plt.axis("off")  # Hide
axes     plt.show() else:     print("Optical flow
could not be calculated.")
```

OUTPUT:

LUCAS FOR VIDEO:

ALGORITHM:

- **Import Libraries**

Use OpenCV and NumPy.

- **Open the Video**

Load the video file from the given path.

- **Check if Video Opened**

If it didn't open, show an error and stop.

- **Read the First Frame**

Capture the first frame from the video and convert it to grayscale.

- **Detect Features to Track**

Use Shi-Tomasi method to find good points (like corners) in the first frame.

– **Set Parameters for Lucas-Kanade Optical Flow**

Define how the tracking will be done (window size, levels, etc.).

– **Create a Mask Image**

This will be used to draw motion lines. –

Loop Over the Video Frames

- Read the next frame.
- Convert it to grayscale.
- Track how the points from the first frame moved using Lucas-Kanade.
- Draw lines and dots to show movement.
- Show the output frame.
- Update the previous frame and points for the next loop.
- Exit if 'q' is pressed.

– **Release Resources**

Close the video and all OpenCV windows.

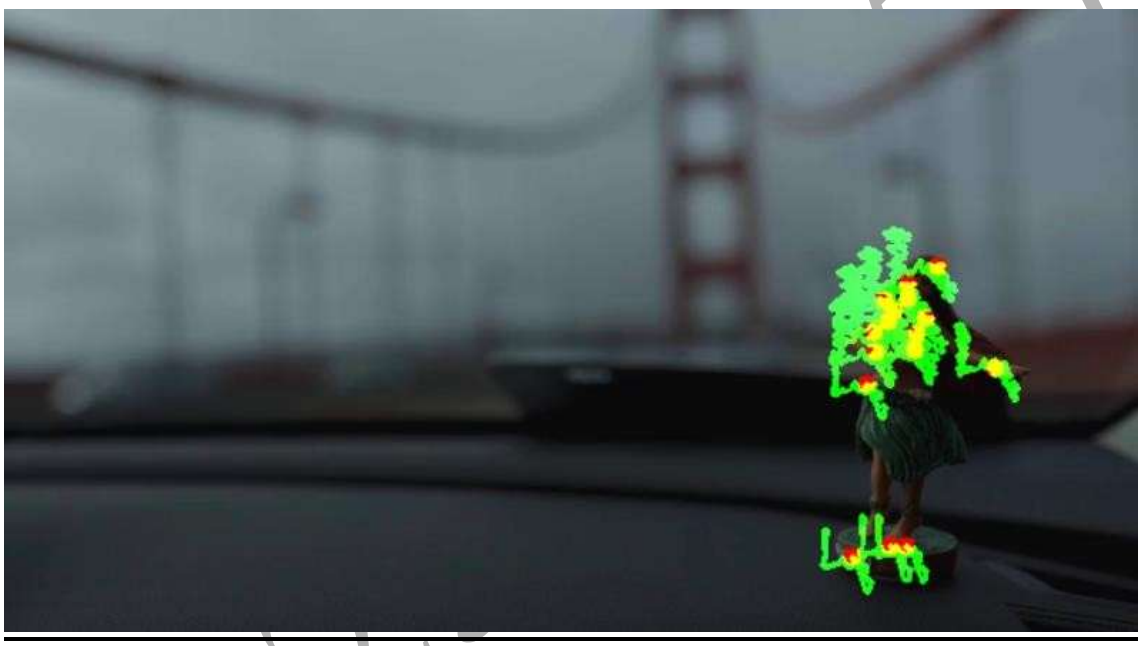
CODE:

```
import cv2 import numpy as np video_path =
r"C:\Users\madhu\Downloads\video4.mp4" cap =
cv2.VideoCapture(video_path) if not cap.isOpened():
print("Error: Could not open video.")
    exit()
```

```
ret, old_frame = cap.read() if not ret:

print("Error: Could not read first frame.")

exit()

old_gray = cv2.cvtColor(old_frame, cv2.COLOR_BGR2GRAY)

feature_params = dict(maxCorners=200, qualityLevel=0.3, minDistance=7,
blockSize=7)

p0 = cv2.goodFeaturesToTrack(old_gray, mask=None, **feature_params)

lk_params = dict(winSize=(15, 15), maxLevel=2,

            criteria=(cv2.TERM_CRITERIA_EPS |

cv2.TERM_CRITERIA_COUNT, 10, 0.03)) mask

= np.zeros_like(old_frame) while cap.isOpened():

ret, new_frame = cap.read()

   if not ret:

break

   new_gray = cv2.cvtColor(new_frame, cv2.COLOR_BGR2GRAY)

   p1, st, err = cv2.calcOpticalFlowPyrLK(old_gray, new_gray, p0, None,
**lk_params)

   if p1 is not None and st is not None:

      good_new = p1[st == 1]        good_old = p0[st == 1]

for i, (new, old) in enumerate(zip(good_new, good_old)):

         a, b = new.ravel()           c, d = old.ravel()           mask =

cv2.line(mask, (int(a), int(b)), (int(c), int(d)), (0, 255, 0), 2)

new_frame = cv2.circle(new_frame, (int(a), int(b)), 5, (0, 0, 255), -1)

output = cv2.add(new_frame, mask)        cv2.imshow('Optical Flow - Lucas
```

```
Kanade', output)        old_gray = new_gray.copy()        p0 =
good_new.reshape(-1, 1, 2)        if cv2.waitKey(50) & 0xFF == ord('q'):
            break
else:
        break cap.release()
cv2.destroyAllWindows()
```

OUTPUT:



RESULT:

The lucas kannade algorithm is performed successfully.