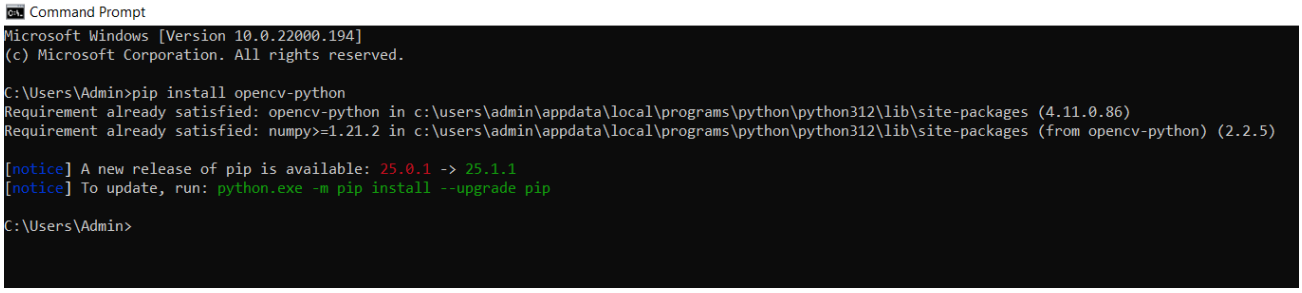
Practical No:- 01

Aim: Introduction to Open-CV.

Description: OpenCV (Open Source Computer Vision Library) is an open-source library used for real-time computer vision and machine learning tasks. It provides tools for image and video processing, including operations like object detection, image manipulation, and feature extraction. **OpenCV** supports multiple programming languages like Python, C++, and Java, and works across platforms like Windows, macOS, and Linux. It's widely used in fields like robotics, augmented reality, and security, offering efficient solutions for real-time visual applications.



1.To read image: Reading an image in OpenCV refers to loading an image from a file into memory so it can be processed or analyzed. This is done using the cv2.imread() function, which reads the image and stores it as a NumPy array. Once the image is loaded, you can manipulate it, convert it to different formats (like grayscale), and apply various image processing techniques.

Code:

```
import cv2
cv2.imread

myImg = cv2.imread("OIP.jpg")
cv2.imshow("Output", myImg)
import numpy as np
```

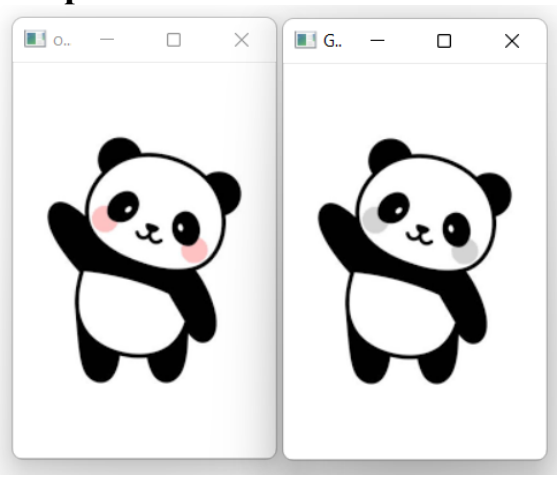
Output:

2.Converting to Grayscale image: Converting an image to grayscale in OpenCV reduces a color image to a single channel representing intensity (shades of gray). This is done using the `cv2.cvtColor()` function with the `cv2.COLOR_BGR2GRAY` flag. The result is a simplified image with pixel values ranging from 0 (black) to 255 (white).

Code:

```
import cv2
cv2.imread
import matplotlib.pyplot as plot

myImg = cv2.imread('img.png')

greyImg = cv2.cvtColor(myImg, cv2.COLOR_BGR2GRAY)

cv2.imshow("og image", myImg)
cv2.imshow("Gray image", greyImg)

img3 = cv2.imread("OIP.jpg", cv2.IMREAD_GRAYSCALE)
cv2.imshow("img3", img3)
```

Output:



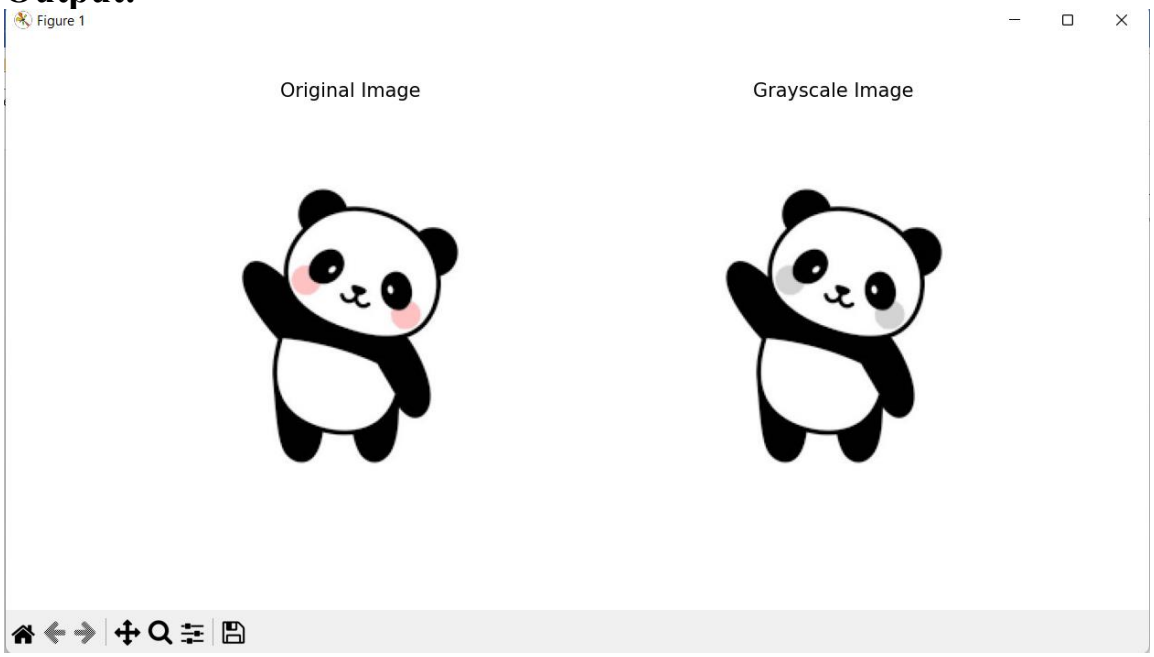
3.Subplot Images: Subplot images are used to display multiple stages of image processing side by side—like the original image, grayscale, edge detection, and contours. This helps visually compare results and track changes across steps.

Code:

```
import cv2
import matplotlib.pyplot as plt

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

#Load the original image
a = cv2.imread("img.png")

#Convert from BGR to RGB (OpenCV loads images in BGR format)
```

```
a = cv2.cvtColor(a, cv2.COLOR_BGR2RGB)

#Display the original image
plt.subplot(1, 2, 1)
plt.imshow(a)
plt.title("Original Image")
plt.axis('off')

#Load the grayscale image
b = cv2.imread("img.png", cv2.IMREAD_GRAYSCALE)

#Display the grayscale image
plt.subplot(1, 2, 2)
plt.imshow(b, cmap='gray')
plt.title("Grayscale Image")
plt.axis('off')

#Show the plot
plt.show()
```

Output:



4. Increasing Image, Decreasing Image, Image Rotation, Image Translation:

Increasing Image Size (Scaling Up): Increasing the size of an image involves enlarging its dimensions, making it more visible or suitable for certain analyses. This can be done by resizing the image, which adjusts its width and height based on specific scaling factors.

Decreasing Image Size (Scaling Down): Decreasing the size of an image reduces its dimensions, which helps in reducing the file size or making the image more manageable for processing, often by resizing it to smaller width and height values.

Image Rotation: Image rotation involves turning the image around a specific point, typically its center, by a certain angle. This transformation allows the image to be oriented in a new direction while maintaining its content.

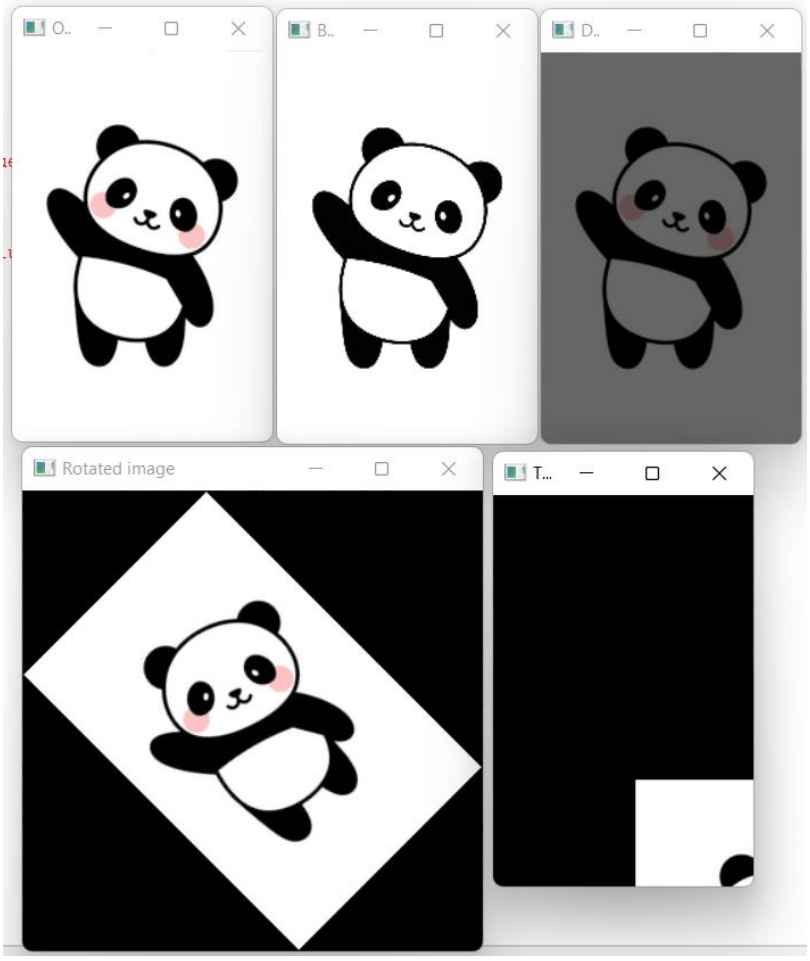
Image Translation: Image translation shifts the image horizontally or vertically without altering its orientation or size. It moves the image to a new position by applying a translation matrix that modifies its coordinates.

Code:

```
import cv2
from scipy import ndimage
import numpy as np

#Load the image
image=cv2.imread("C:\\CV\\images.jpg");

#Increase Brightness
image_float = image.astype(float)

# Increase the brightness
brightness_factor = 2.0  # Increase or decrease this value as needed
brightened_image = image_float * brightness_factor

#Decrease the Brightness
brightness_factor1 = 0.4  # Increase or decrease this value as needed
dark_image = image_float * brightness_factor1

# Clip the pixel values to the valid range [0, 255]
brightened_image = np.clip(brightened_image, 0, 255)
dark_image = np.clip(dark_image, 0, 255)

# Convert the image back to unsigned 8-bit integers
brightened_image = brightened_image.astype(np.uint8)
dark_image = dark_image.astype(np.uint8)

# Image Rotation
image_rotate = ndimage.rotate(image,45)
```

```
#Image translation
height, width = image.shape[:2] #Store height and Weight
T = np.float32([[1, 0, 100], [0, 1, 200]])
img_trans = cv2.warpAffine(image, T, (width, height))

#Show images
cv2.imshow("Original Image",image)
cv2.imshow("Brightened image",brightened_image)
cv2.imshow("Dark image",dark_image)
cv2.imshow("Rotated image",image_rotate)
cv2.imshow("Translated image",img_trans)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Output:



Practical No:- 02

Aim: Histogram Equalization, Grayscale Conversion.

Description: **Histogram equalization** is a technique used to enhance the contrast of an image.It works by spreading out the most frequent intensity values across the full range of pixel values (0–255). First, it calculates the histogram (distribution) of pixel intensities. Then, it computes the cumulative distribution function (CDF). Each pixel's value is then remapped based on the CDF to achieve a more uniform histogram. This process improves visibility in images with low contrast or poor lighting. It's commonly used in medical imaging, satellite imagery, and photo enhancement.

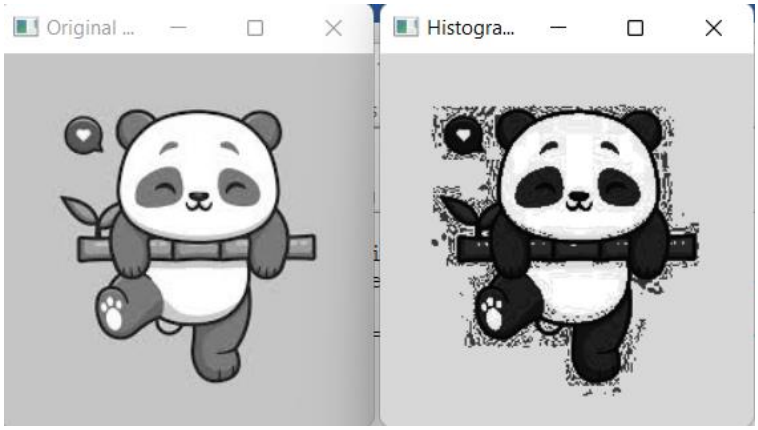
Grayscale conversion is the process of converting a color image into a black-and-white image using varying shades of gray. It removes the hue and saturation while retaining the luminance (brightness). This is typically done using a weighted formula:
Gray = 0.299 × R + 0.587 × G + 0.114 × B, which matches human visual perception. Green contributes the most to brightness, while blue contributes the least. The output is a single-channel image with pixel values from 0 (black) to 255 (white). Grayscale images are simpler and often used in pre-processing for computer vision tasks.

Code:

#Histogram Equalization

```
# Import required Libraries
import cv2
from scipy import ndimage
import numpy as np
#Load the image
image=cv2.imread("C:\\CV\\images.jpg")
#Convert to Grayscale
gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
#Perform Histogram Equalization
equalized_img = cv2.equalizeHist(gray)
cv2.imshow("Original Gray Image",gray)
cv2.imshow("Histogram Equalized Image",equalized_img)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Output:

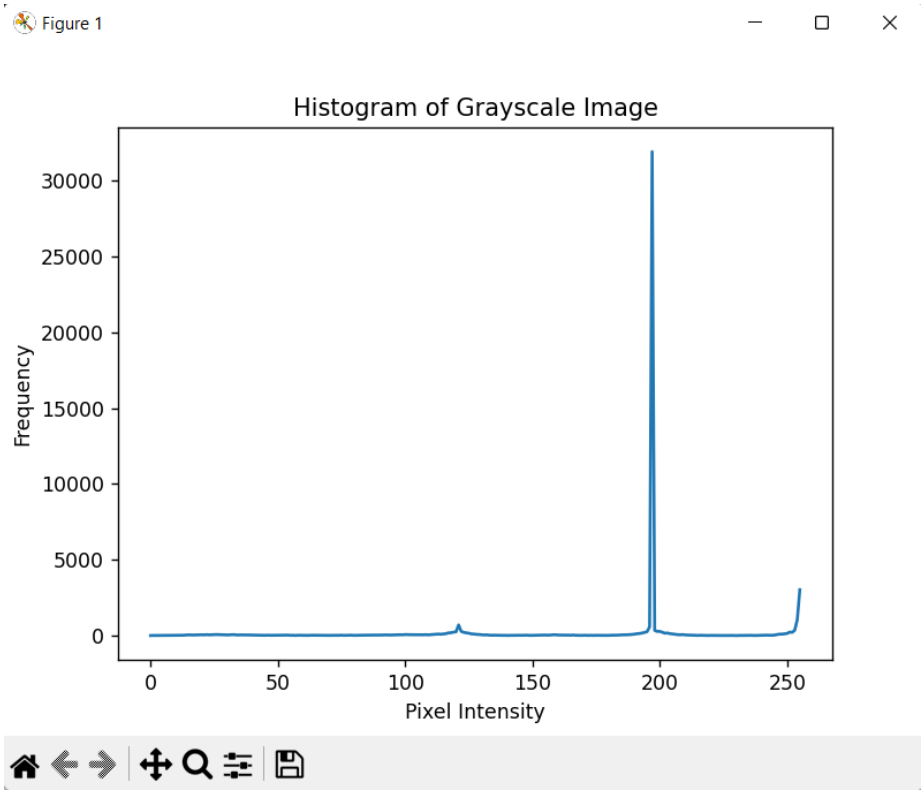


Histogram Plotting:

Code:

```
#Histogram plotting
import cv2
import matplotlib.pyplot as plt
# Step 2: Read the image
#img=cv2("C:\\CV\\images.jpg")
image_path = ("C:\\CV\\images.jpg")
img = cv2.imread(image_path)
# Step 3: Convert to grayscale (optional)
gray_img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
# Step 4: Calculate the histogram
hist = cv2.calcHist([gray_img], [0], None, [256], [0, 256])
# Step 5: Plot the histogram (optional)
plt.plot(hist)
plt.title('Histogram of Grayscale Image')
plt.xlabel('Pixel Intensity')
plt.ylabel('Frequency')
plt.show()
```

Output:



Practical No:- 03

Aim: Filtering Methods.

Description: **Filtering methods** in image processing are techniques used to modify or enhance an image by emphasizing certain features or reducing unwanted ones like noise. They involve applying a **filter (or kernel)**—a small matrix—to each pixel and its neighbors in the image. This process alters the pixel values based on a specific rule defined by the filter.

1. Low Filter: A **low-pass filter** is a type of image filtering method used to **reduce high-frequency content** such as noise, fine edges, or sharp transitions in an image.
It allows **low-frequency components** (like smooth areas and gradual changes) to pass through while **suppressing high-frequency components** (like sudden changes or noise).

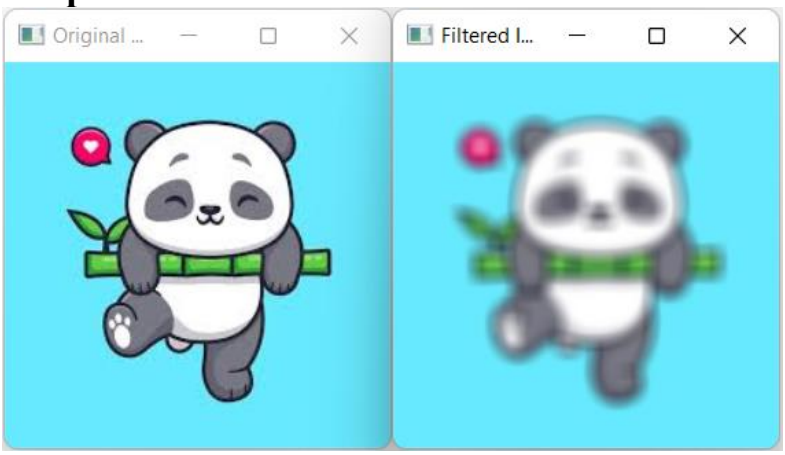
This filter is commonly used for:

- **Smoothing** or **blurring** images
- **Noise reduction**
- **Pre-processing** before edge detection or segmentation.

Code:

```
import cv2
import numpy as np
def apply_average_filter(image_path, kernel_size):

    # Read the image
    image = cv2.imread(image_path)

    # Apply the average filter
    filtered_image = cv2.blur(image, (kernel_size, kernel_size))
    return filtered_image
if __name__ == "__main__":

    # Input image path and kernel size
    input_image_path = ("C:\\CV\\images.jpg")
    kernel_size = 10  # You can adjust the kernel size as needed.

    # Apply the average filter
    filtered_image = apply_average_filter(input_image_path, kernel_size)

   # Display the original and filtered images
   cv2.imshow("Original Image", cv2.imread(input_image_path))
   cv2.imshow("Filtered Image", filtered_image)

  # Wait for a key press and then close the windows
   cv2.waitKey(0)
   cv2.destroyAllWindows()
```

Output:



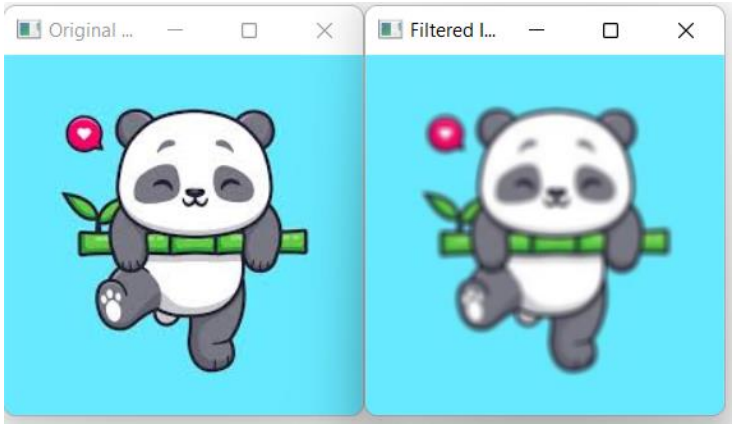
2.Guassian Filter: A **Gaussian filter** is a type of low-pass filter used in image processing to smooth or blur images. It works by applying a **Gaussian function** to the pixels in an image, where pixels near the center of the filter's kernel are weighted more heavily than those further away. This results in a **smoothing effect** that reduces noise and detail in the image. The filter works by convolving the image with a Gaussian kernel, which is a matrix that resembles the bell curve (Gaussian distribution). This kernel is applied to each pixel and its neighbours, where the weight decreases with distance from the centre pixel.

Code:

```
import cv2

def apply_weighted_average_filter(image_path, kernel_size, sigmaX):
    # Read the image
    image = cv2.imread(image_path)

    # Apply the weighted average filter (Gaussian blur)
    filtered_image = cv2.GaussianBlur(image, (kernel_size, kernel_size), sigmaX)

    return filtered_image

if __name__ == "__main__":
    # Input image path, kernel size, and sigmaX (standard deviation in X direction)
    input_image_path = ("C:\\CV\\images.jpg")
    kernel_size = 5  # You can adjust the kernel size as needed.
    sigmaX = 4  # You can adjust the sigmaX value for the Gaussian kernel.

    # Apply the weighted average filter
    filtered_image = apply_weighted_average_filter(input_image_path, kernel_size, sigmaX)

    # Display the original and filtered images
    cv2.imshow("Original Image", cv2.imread(input_image_path))
    cv2.imshow("Filtered Image", filtered_image)

    # Wait for a key press and then close the windows
```

```
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Output:

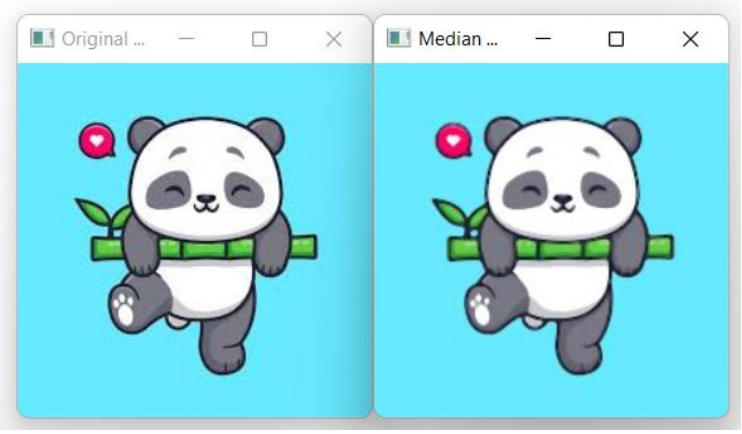


3.Median Filter: A **median filter** is a non-linear image filter used to remove **salt-and-pepper noise** by replacing each pixel with the **median** value of its surrounding pixels. It works by sorting the pixel values in a defined neighbourhood (e.g., 3x3 or 5x5) and replacing the centre pixel with the median of the sorted list. This process reduces noise while preserving edges better than linear filters. Median filters are particularly effective for noise removal without causing excessive blurring of image details. It's widely used in pre-processing steps for image enhancement.

Code:

```
import cv2
import numpy as np

# Load the image
image_path = ("C:\\CV\\images.jpg")
original_image  = cv2.imread(image_path)

# Apply median filter with specified kernel size
kernel_size = 3  # Adjust this value to change the filter size
median_filtered_image = cv2.medianBlur(original_image,  kernel_size)

# Display the original and filtered images
cv2.imshow('Original Image', original_image)
cv2.imshow('Median Filtered Image', median_filtered_image)

# Wait for a key press and close the windows
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Output:



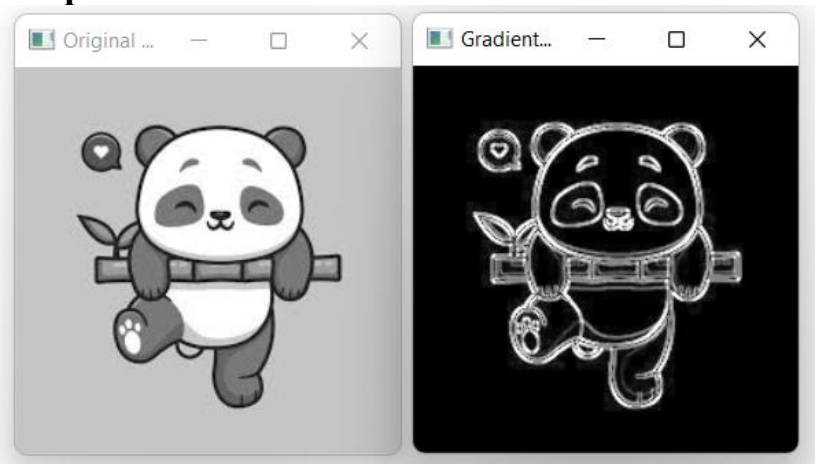
4.Sobel Filter: The **Sobel filter** is an edge detection technique used in image processing to highlight edges by measuring intensity changes. It uses two 3×3 convolution kernels: one for detecting **horizontal** edges (Gx) and one for **vertical** edges (Gy). These filters approximate the gradient of the image intensity, emphasizing regions where pixel values change sharply. The final edge strength is often calculated using: **Gradient = $\sqrt{(Gx^2 + Gy^2)}$**
Sobel filtering is commonly used to detect object boundaries and structural features in images.

Code:

```
import cv2
import numpy as np

# Load the image
image_path = ("C:\\CV\\images.jpg")
original_image = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)

# Check if the image was loaded successfully
if original_image is None:
    print("Error loading the image.")
else:
    # Apply Sobel filter in the x and y directions
    sobel_x = cv2.Sobel(original_image, cv2.CV_64F, 1, 0, ksize=3)
    sobel_y = cv2.Sobel(original_image, cv2.CV_64F, 0, 1, ksize=3)

    # Convert the results to absolute values and then to 8-bit
    sobel_x = cv2.convertScaleAbs(sobel_x)
    sobel_y = cv2.convertScaleAbs(sobel_y)

    # Combine the Sobel images to get the gradient magnitude
    gradient_magnitude = cv2.addWeighted(sobel_x, 0.5, sobel_y, 0.5, 0)

    # Display the original image and the gradient magnitude
    cv2.imshow('Original Image', original_image)
    cv2.imshow('Gradient Magnitude', gradient_magnitude)
```

```
    # Wait for a key press and close the windows
    cv2.waitKey(0)
    cv2.destroyAllWindows()
```

Output:



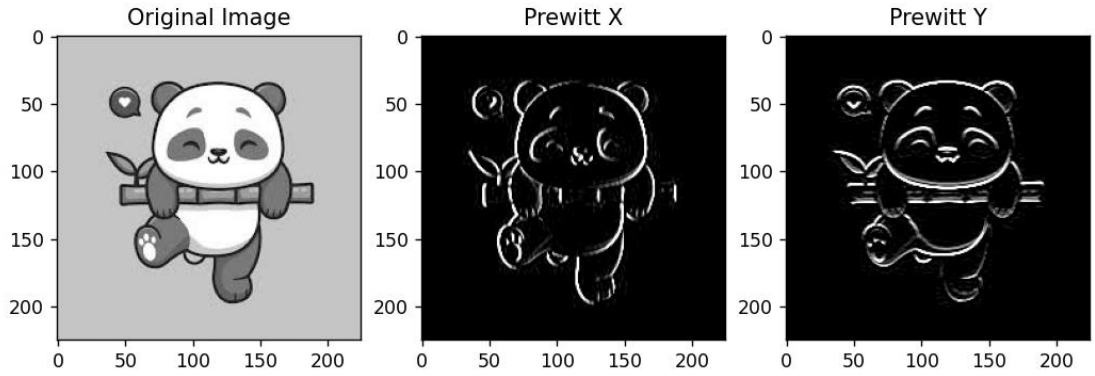
5.Prewitt Filter: The **Prewitt filter** is an edge detection method used to identify **horizontal** and **vertical** edges in an image. It uses two 3×3 convolution kernels: one for detecting changes in the **x-direction** and one in the **y-direction**. Like the Sobel filter, it estimates the **gradient** of the image intensity but with **simpler kernel weights**. The edge strength is calculated as: **Gradient = $\sqrt{(Gx^2 + Gy^2)}$**
Prewitt is effective for basic edge detection, especially when speed is more important than precision.

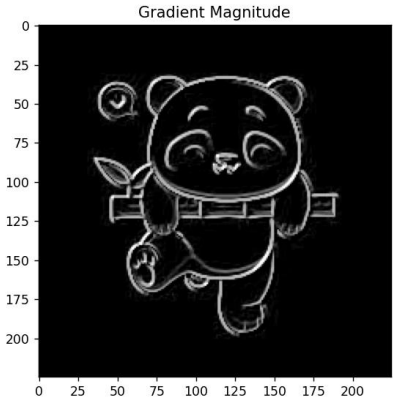
Code:

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

def apply_prewitt_filter(image_path):
    # Read the image in grayscale
    img = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)
    if img is None:
        print("Error: Unable to load image.")
        return
    # Define Prewitt operator kernels
    prewitt_x = np.array([[1, 0, -1], [1, 0, -1], [1, 0, -1]], dtype=np.float32)
    prewitt_y = np.array([[1, 1, 1], [0, 0, 0], [-1, -1, -1]], dtype=np.float32)
    # Apply filters
    grad_x = cv2.filter2D(img, -1, prewitt_x)
    grad_y = cv2.filter2D(img, -1, prewitt_y)
    # Combine gradients
    gradient_magnitude = cv2.magnitude(grad_x.astype(np.float32),
grad_y.astype(np.float32))
```

```
    gradient_magnitude = cv2.normalize(gradient_magnitude, None, 0, 255,
cv2.NORM_MINMAX).astype(np.uint8)
    # Display results
    plt.figure(figsize=(10, 5))
    plt.subplot(1, 3, 1), plt.imshow(img, cmap='gray'), plt.title('Original Image')
    plt.subplot(1, 3, 2), plt.imshow(grad_x, cmap='gray'), plt.title('Prewitt X')
    plt.subplot(1, 3, 3), plt.imshow(grad_y, cmap='gray'), plt.title('Prewitt Y')
    plt.figure(figsize=(5, 5))
    plt.imshow(gradient_magnitude, cmap='gray'), plt.title('Gradient Magnitude')
    plt.show()

# Example usage
apply_prewitt_filter("C:\\CV\\images.jpg")
```

Output:



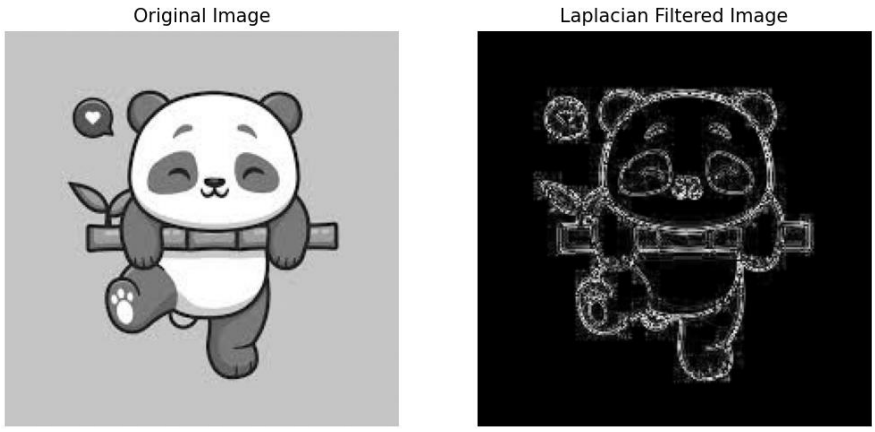


6.Laplacian Filter: The **Laplacian filter** is an edge detection technique that uses second-order derivatives to find areas of rapid intensity change in an image. Unlike gradient-based filters, it detects edges in **all directions simultaneously**. It works by measuring the rate at which the intensity changes around a pixel, highlighting regions with sudden brightness shifts. The filter enhances fine details and edges but is also **sensitive to noise**, so it's often used after smoothing. It's commonly used in applications requiring **sharp edge enhancement** and feature detection.

Code:

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

# Read the image in grayscale
image = cv2.imread("C:\\CV\\images.jpg",  cv2.IMREAD_GRAYSCALE)

# Apply the Laplacian filter
laplacian = cv2.Laplacian(image, cv2.CV_64F)

# Convert the result to uint8
laplacian = np.uint8(np.absolute(laplacian))

# Display the original and filtered images
plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.title("Original Image")
plt.imshow(image, cmap='gray')
plt.axis('off')

plt.subplot(1, 2, 2)
plt.title("Laplacian Filtered Image")
plt.imshow(laplacian, cmap='gray')
plt.axis('off')

plt.show()
```

Output:



Practical No:- 04

Aim: Binary Image Generation.

Description: **Binary image generation** is the process of converting a grayscale or color image into a **two-tone image** consisting of only black and white pixels. This is typically done using **thresholding**, where each pixel is compared to a specified intensity value (threshold). Pixels above the threshold are set to white (1), and those below are set to black (0).
It simplifies the image and is useful for tasks like object detection, segmentation, and OCR. Binary images reduce complexity and are ideal for analyzing shapes and regions.

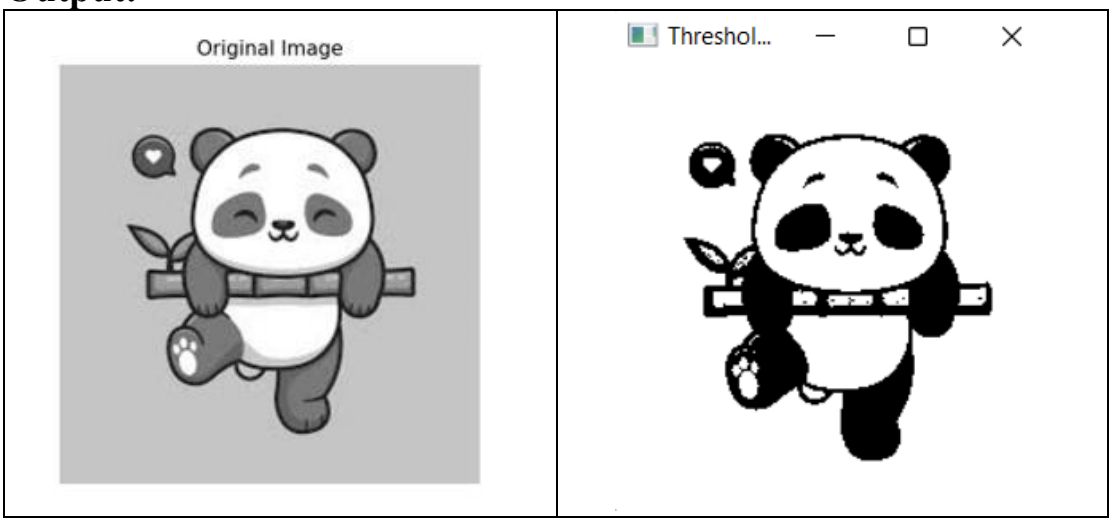
Converting rgb image to binary image using thresholding method:

Converting an RGB image to a binary image using thresholding involves two main steps. First, the RGB image is converted to **grayscale** to remove color information and retain brightness. Then, a **threshold value** is applied: pixels with intensity above the threshold are set to white (1), and those below are set to black (0). This creates a binary image that highlights objects or regions of interest. It's a simple and effective method for image segmentation and pattern recognition tasks.

Code:

```
import cv2
import numpy as np

# Load the grayscale image
image = cv2.imread("C:\\CV\\images.jpg",  cv2.IMREAD_GRAYSCALE)

# Apply thresholding
threshold_value = 150  # You can adjust this value
_, thresholded_image = cv2.threshold(image, threshold_value, 255, cv2.THRESH_BINARY)

# Display the thresholded image
cv2.imshow('Thresholded Image', thresholded_image)
cv2.waitKey(0)
cv2.destroyAllWindows()

Total = np.sum(image)
W,h = image.shape
Avg = int (Total/(W*h))
```

Output:



Practical No:- 05

Aim: Color Model Conversion.

Description: **Color model conversion** is the process of changing an image from one color representation to another, allowing for different ways of manipulating and analysing colors. common color models include **RGB** (Red, Green, Blue), **HSV** (Hue, Saturation, Value), **CMYK** (Cyan, Magenta, Yellow, Key), and **Lab** (Lightness, a, b). Conversion between these models is done using mathematical formulas that map color values from one space to another. This process is useful in image processing tasks like **color correction**, **enhancement**, and **segmentation**. Each model offers distinct advantages depending on the application, such as easier color manipulation or better human perception alignment.

Code:

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

# Load the image
image = cv2.imread("C:\\CV\\images.jpg")

# Convert to HSV (Hue= 0,360, Saturation=crominence,0 to 100%, Value=luminence,0
t0 255)
hsv_image = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)

# Convert to LAB (CIELAB) color space b=+blue to -green, a = red to yellow
lab_image = cv2.cvtColor(image, cv2.COLOR_BGR2LAB)


# Convert to YCbCr y=intensity
ycbcr_image = cv2.cvtColor(image, cv2.COLOR_BGR2YCrCb)

# Convert to CMYK=cyan,magenta,yellow,black
#cmyk_image = cv2.cvtColor(rgb_image, cv2.COLOR_BGR2CMYK)
# Convert to YIQ =intensity,
yiq_image = np.zeros_like(image, dtype=np.float32)
yiq_image[:,:,0] = 0.299 * image[:,:,2] + 0.587 * image[:,:,1] + 0.114 * image[:,:,0]
yiq_image[:,:,1] = 0.596 * image[:,:,2] - 0.274 * image[:,:,1] - 0.322 * image[:,:,0]
yiq_image[:,:,2] = 0.211 * image[:,:,2] - 0.523 * image[:,:,1] + 0.312 * image[:,:,0]
yiq_image = np.clip(yiq_image, 0, 255).astype(np.uint8)

# Display the original and converted images
plt.figure(figsize=(10, 8))
plt.subplot(4, 2, 1)
plt.imshow(cv2.cvtColor(image, cv2.COLOR_BGR2RGB))
plt.title('Original Image')
plt.axis("off")
```

```
plt.subplot(4, 2, 2)
plt.imshow(cv2.cvtColor(hsv_image, cv2.COLOR_BGR2RGB))
plt.title('HSV Image')
plt.axis("off")

plt.subplot(4, 2, 3)
plt.imshow(cv2.cvtColor(image, cv2.COLOR_BGR2RGB))
plt.title('Original Image')
plt.axis("off")

plt.subplot(4, 2, 4)
plt.imshow(cv2.cvtColor(lab_image, cv2.COLOR_BGR2RGB))
plt.title('LAB Image')
plt.axis("off")

plt.subplot(4, 2, 5)
plt.imshow(cv2.cvtColor(image, cv2.COLOR_BGR2RGB))
plt.title('Original Image')
plt.axis("off")

plt.subplot(4, 2, 6)
plt.imshow(cv2.cvtColor(ycbcr_image, cv2.COLOR_BGR2RGB))
plt.title('YCbCr Image')
plt.axis("off")

plt.subplot(4, 2, 7)
plt.imshow(cv2.cvtColor(image, cv2.COLOR_BGR2RGB))
plt.title('Original Image')
plt.axis("off")

plt.subplot(4, 2, 8)
plt.imshow(cv2.cvtColor(yiq_image, cv2.COLOR_BGR2RGB))
plt.title('YIQ Image')
plt.axis("off")
plt.show()
```

Output:

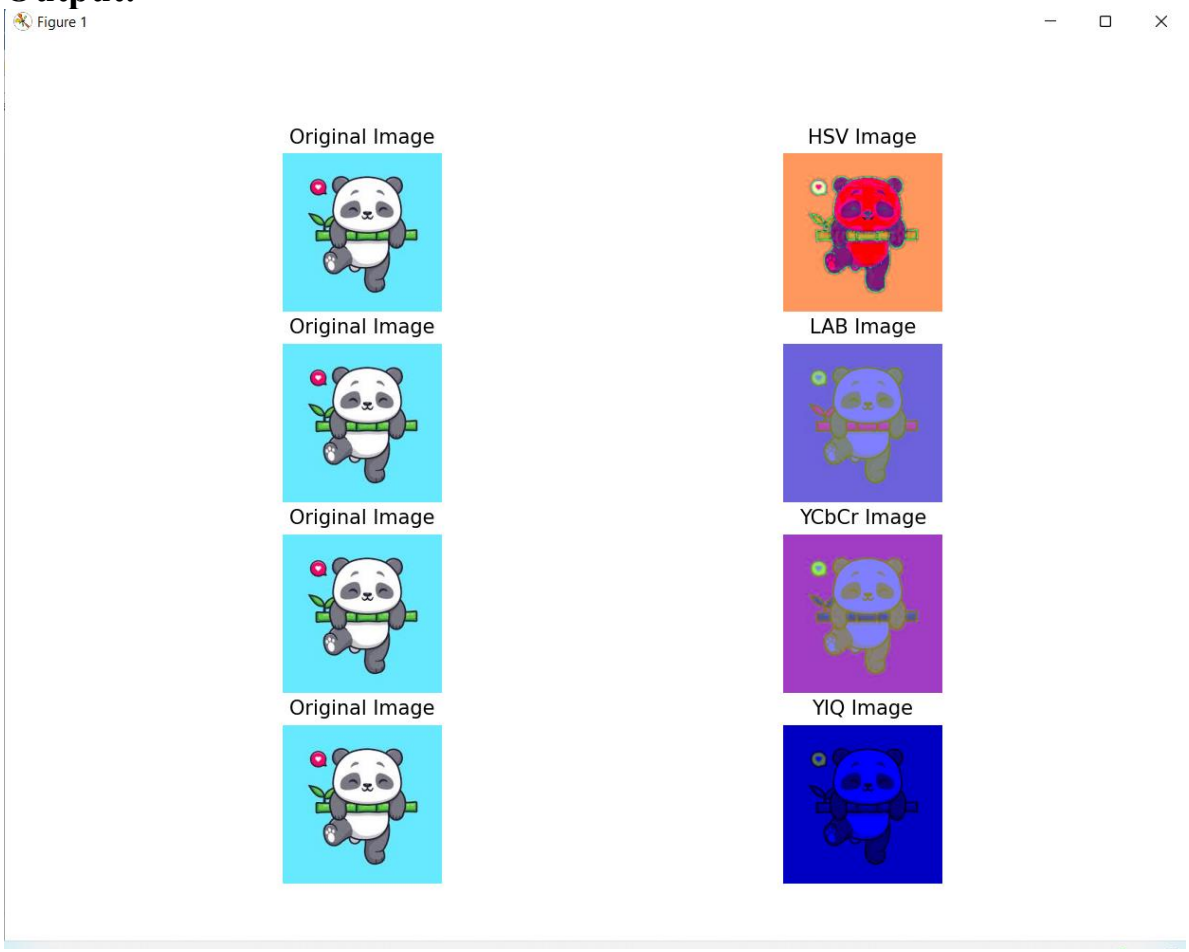


Figure 1
Original Image
HSV Image
Original Image
LAB Image
Original Image
YCbCr Image
Original Image
YIQ Image

Practical No:- 06

Aim: Morphological Operation.

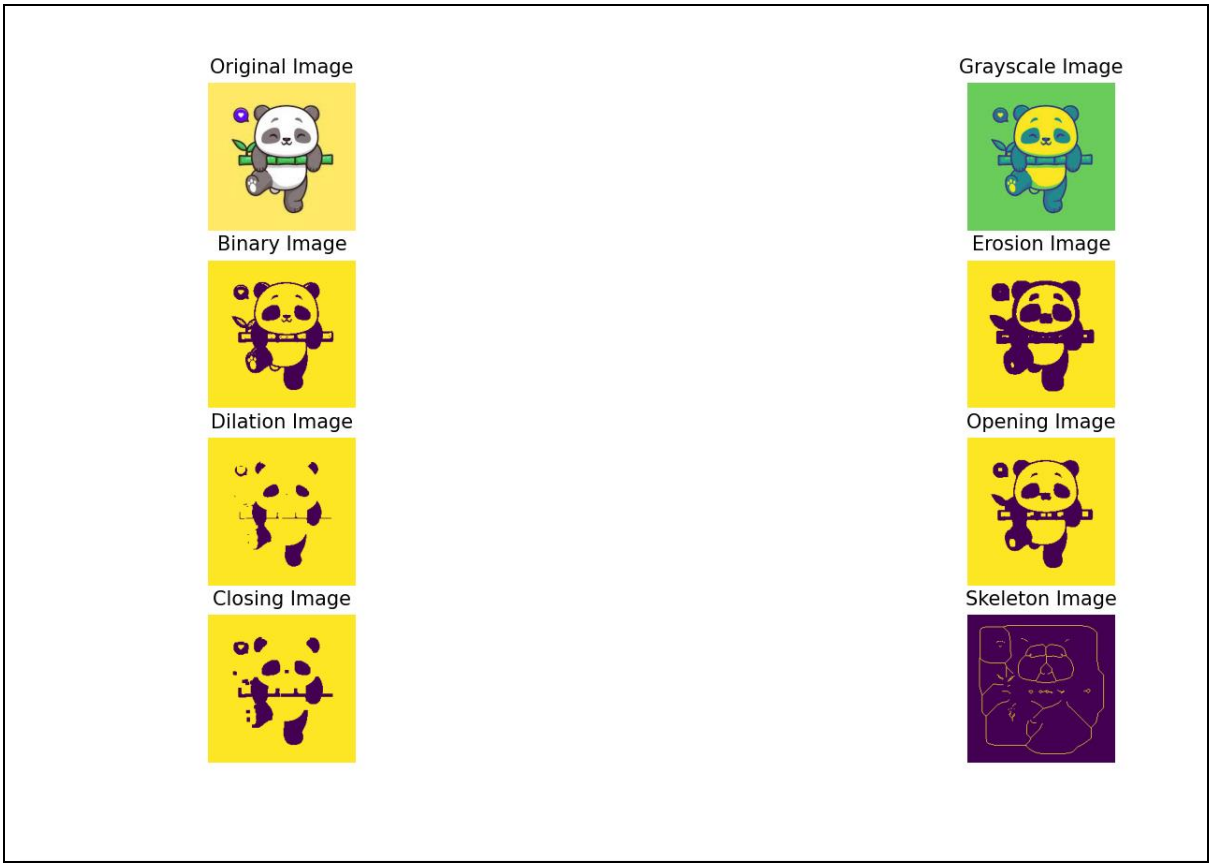
Description: **Morphological operations** are a set of image processing techniques used to analyze and manipulate the structure or shape of objects in a binary image. These operations work by applying a **structuring element** (a small matrix) to an image to modify its shape. Common operations include **erosion**, which reduces object size, and **dilation**, which enlarges objects. Other operations like **opening** and **closing** combine erosion and dilation to remove noise or fill gaps.Morphological operations are widely used for tasks such as **object extraction**, **noise removal**, and **shape analysis** in computer vision.

Code:

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
from skimage.morphology  import skeletonize

# Read the image in grayscale
og_image = cv2.imread(r"D:\\CV\\images.png")

image = cv2.imread(r"D:\\CV\\images.png",  cv2.IMREAD_GRAYSCALE)

# Apply a binary thresholding operation
threshold_value = 150
_, binary_image = cv2.threshold(image, threshold_value, 255, cv2.THRESH_BINARY)

# Create a kernel for morphological operations
kernel = np.ones((5, 5), np.uint8)

# Apply morphological operations
erosion = cv2.erode(binary_image, kernel, iterations=1)
dilation = cv2.dilate(binary_image, kernel, iterations=1)
opening = cv2.dilate(erosion, kernel, iterations=1)
closing = cv2.erode(dilation, kernel, iterations=1)

# Normalize the binary image (0 or 1 values) for skeletonization
binary_norm = binary_image // 255  # Convert to 0 and 1
skeleton = skeletonize(binary_norm)   # Apply thinning

# Convert skeleton back to uint8 (255 for "1" and 0 for "0")
skeleton = (skeleton * 255).astype(np.uint8)

# Display the images
# cv2.imshow("Original Image", image)
# cv2.imshow("Grey Image", image)
# cv2.imshow("Binary Image", binary_image)
# cv2.imshow("Erosion Image", erosion)
# cv2.imshow("Dilation Image", dilation)
```

```
# cv2.imshow("Opening Image", opening)
# cv2.imshow("Closing Image", closing)
# cv2.imshow("Skeleton Image", skeleton)

plt.subplot(4, 2, 1)
plt.title("Original Image")
plt.imshow(og_image)
plt.axis('off')

plt.subplot(4, 2, 2)
plt.title("Grayscale Image")
plt.imshow(image)
plt.axis('off')

plt.subplot(4, 2, 3)
plt.title("Binary Image")
plt.imshow(binary_image)
plt.axis('off')

plt.subplot(4, 2, 4)
plt.title("Erosion Image")
plt.imshow(erosion)
plt.axis('off')

plt.subplot(4, 2, 5)
plt.title("Dilation Image")
plt.imshow(dilation)
plt.axis('off')

plt.subplot(4, 2, 6)
plt.title("Opening Image")
plt.imshow(opening)
plt.axis('off')

plt.subplot(4, 2, 7)
plt.title("Closing Image")
plt.imshow(closing)
plt.axis('off')

plt.subplot(4, 2, 8)
plt.title("Skeleton Image")
plt.imshow(skeleton)
plt.axis('off')

plt.show()

cv2.waitKey(0)
cv2.destroyAllWindows()
```

Output:


Original Image
Grayscale Image
Binary Image
Erosion Image
Dilation Image
Opening Image
Closing Image
Skeleton Image

Practical No:- 07

Aim: Shift Invariant Fourier Transform (SIFT).

Description: The **Shift Invariant Fourier Transform (SIFT)** is a technique used in image processing to analyze and represent images in the frequency domain while maintaining **shift invariance**. This means that the transform remains unchanged even if the image is shifted in space, making it robust to translation. SIFT works by decomposing an image into sinusoidal components, allowing the identification of patterns, textures, and features at various scales. It is particularly useful in **image recognition** and **feature matching** tasks, as it provides a stable representation that is not affected by shifts in position. SIFT is commonly used in applications like **object detection**, **image stitching**, and **feature extraction**.

Code:

```
import cv2

# Load two images
image1 = cv2.imread('Test1.jpg')
image2 = cv2.imread('Test2.jpg')

# Convert images to grayscale
gray1 = cv2.cvtColor(image1, cv2.COLOR_BGR2GRAY)
gray2 = cv2.cvtColor(image2, cv2.COLOR_BGR2GRAY)

# Initialize SIFT detector
sift = cv2.SIFT_create()

# Detect keypoints and compute descriptors for both images
keypoints1, descriptors1 = sift.detectAndCompute(gray1, None)
keypoints2, descriptors2 = sift.detectAndCompute(gray2, None)

# Initialize a Brute Force Matcher
bf = cv2.BFMatcher()

# Match descriptors between the two images
matches = bf.knnMatch(descriptors1, descriptors2, k=2)

# Apply ratio test to filter good matches
good_matches = []
for m, n in matches:
    if m.distance < 0.5 * n.distance:
        good_matches.append(m)

# Draw matches
matched_image = cv2.drawMatches(image1, keypoints1, image2, keypoints2, good_matches,
None, flags=cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)
```

```
# Display the matched image
cv2.imshow('Key Point Matches', matched_image)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Output:

Practical No:- 08

Aim: Image Stitching.

Description: **Image stitching** is the process of combining multiple images with overlapping fields of view to create a single panoramic image or a wider field of view. The process typically involves four main steps:

1. **Feature detection**: Identifying key points or features in each image (often using methods like SIFT, SURF, or ORB).
2. **Feature matching**: Finding corresponding features between the images.
3. **Image alignment**: Using the matched features to align the images, often by applying geometric transformations (such as homography).
4. **Blending**: Merging the aligned images smoothly to avoid seams and ensure continuity in the final output.
   Image stitching is widely used in creating panoramas, 360-degree images, and large scene capture in fields like photography and computer vision.

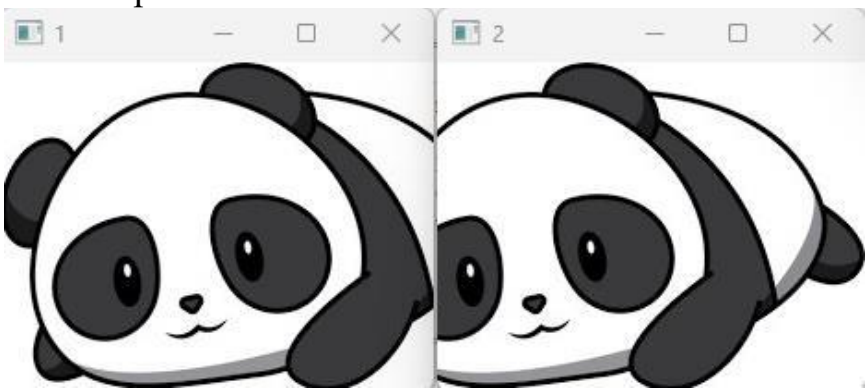
Code:

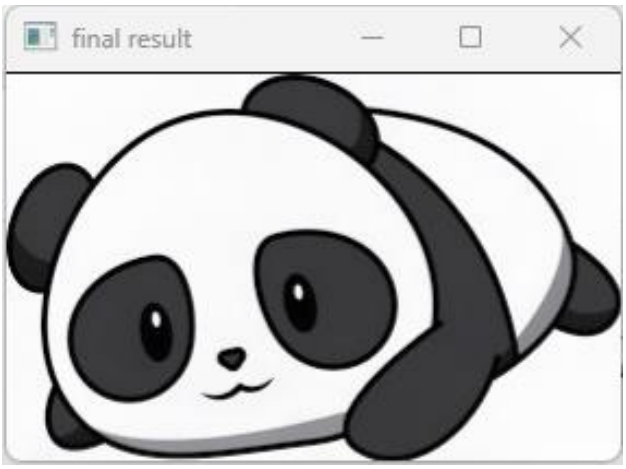
```
import cv2
image_paths=['Test1.jpg','Test2.jpg']

# initialized a list of images
imgs = []
for i in range(len(image_paths)):
    imgs.append(cv2.imread(image_paths[i]))
    #imgs[i]=cv2.resize(imgs[i],(0,0),fx=0.4,fy=0.4)

# showing the original pictures
cv2.imshow('1',imgs[0])
cv2.imshow('2',imgs[1])
stitchy=cv2.Stitcher.create()
(dummy,output)=stitchy.stitch(imgs)
if dummy != cv2.STITCHER_OK:
  # checking if the stitching procedure is successful
    print("stitching ain't successful")
else:
    print('Your Panorama is ready!!!')
# final output
cv2.imshow('final result',output)
cv2.waitKey(0)
```

Output:

Original Image:

The Output:





Practical No:- 09

Aim: 2D to 3D conversion.

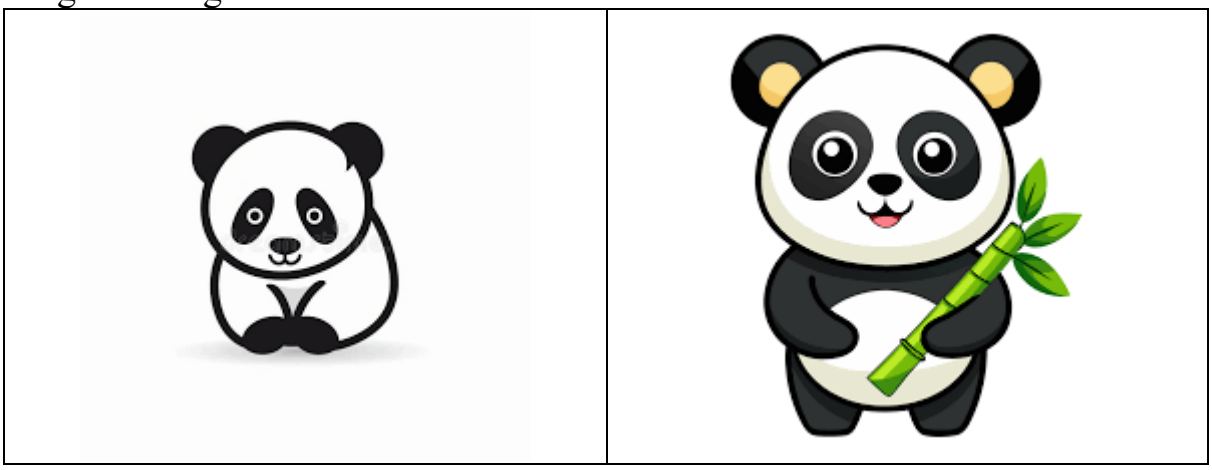
Description: **2D to 3D conversion** is the process of transforming a two-dimensional image or video into a three-dimensional representation. This is achieved by adding depth to the original 2D content, often simulating a 3D perspective, making it appear as though it has volume and depth. Techniques for conversion include **depth mapping**, where a depth map is used to assign depth values to different areas of the 2D image, and **stereoscopic methods**, where two slightly offset images are generated to simulate 3D viewing. 2D to 3D conversion is widely used in **film production**, **virtual reality**, and **video games** to create immersive experiences from flat images. It can also be performed for 3D printing purposes, where a flat design is converted into a model suitable for printing.

Code:

```
from PIL import Image
import numpy as np

def shift_image(img, depth_img, shift_amount=10):
    # Ensure base image has alpha
    img = img.convert("RGBA")
    data = np.array(img)

    # Ensure depth image is grayscale (for single value)
    depth_img = depth_img.convert("L")
    depth_data = np.array(depth_img)
    deltas = ((depth_data / 255.0) * float(shift_amount)).astype(int)
    # This creates the transparent resulting image.
    # For now, we're dealing with pixel data.
    shifted_data = np.zeros_like(data)

    height, width, _ = data.shape

    for y, row in enumerate(deltas):
        for x, dx in enumerate(row):
            if x + dx < width and x + dx >= 0:
                shifted_data[y, x + dx] = data[y, x]

    # Convert the pixel data to an image.
    shifted_image = Image.fromarray(shifted_data.astype(np.uint8))

    return shifted_image

img = Image.open("C:\\CV\\2D_1.png")
depth_img = Image.open("C:\\CV\\2D_2.png")
shifted_img = shift_image(img, depth_img, shift_amount=10)
shifted_img.show()
```

Output:

Original Image



Converted Image:

Practical No:- 10

Aim: Object Detection.

Description: **Object detection** is a computer vision task that involves identifying and locating objects within an image or video. It not only detects what objects are present but also where they are located, typically using **bounding boxes** around the objects. The process combines two main tasks: **classification** (identifying the object) and **localization** (finding the object's position). Object detection is widely used in applications like **autonomous vehicles**, **security systems**, and **medical imaging**. Modern methods typically rely on deep learning techniques, such as **YOLO**, **Faster R-CNN**, and **SSD**, for improved accuracy and speed.

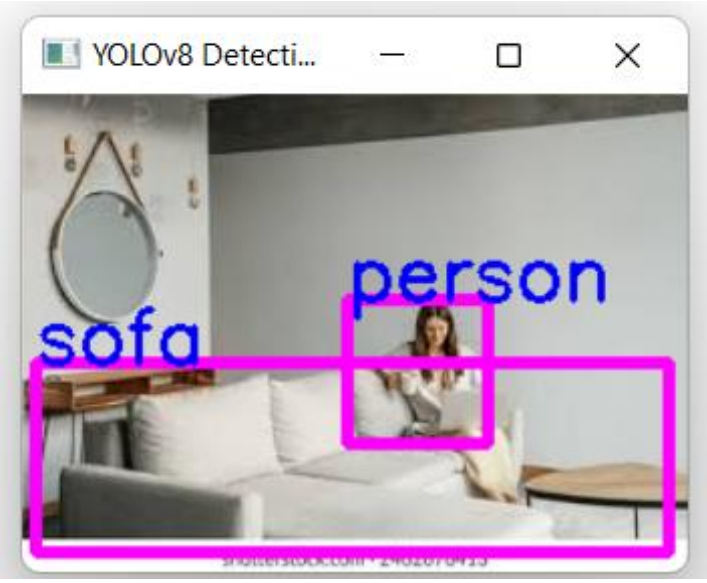
Code:

```
from ultralytics import YOLO
import cv2
import math

# Load YOLOv8 model
model = YOLO("yolov8n.pt")

# Load image
img = cv2.imread('sample2.jpg')

# Class names for COCO dataset
classNames = [ "person", "bicycle", "car", "motorbike", "aeroplane", "bus", "train", "truck",
"boat", "traffic light", "fire hydrant", "stop sign", "parking meter", "bench", "bird", "cat",
 "dog", "horse", "sheep", "cow", "elephant", "bear", "zebra", "giraffe", "backpack",
"umbrella", "handbag", "tie", "suitcase", "frisbee", "skis", "snowboard", "sports ball", "kite",
"baseball bat", "baseball glove", "skateboard", "surfboard", "tennis racket",
 "bottle", "wine glass", "cup", "fork", "knife", "spoon", "bowl", "banana", "apple",
"sandwich", "orange", "broccoli", "carrot", "hot dog", "pizza", "donut", "cake", "chair",
 "sofa", "pottedplant", "bed", "diningtable", "toilet", "tvmonitor", "laptop", "mouse",
  "remote", "keyboard", "cell phone", "microwave", "oven", "toaster", "sink", "refrigerator",
 "book", "clock", "vase", "scissors", "teddy bear", "hair drier", "toothbrush"
]

# Run model inference
results = model(img, stream=True)

# Process results
for r in results:
  boxes = r.boxes
    for box in boxes:
      x1, y1, x2, y2 = box.xyxy[0]
      x1, y1, x2, y2 = int(x1), int(y1), int(x2), int(y2)

       # Draw rectangle
           cv2.rectangle(img, (x1, y1), (x2, y2), (255, 0, 255), 3)
```

```
        # Confidence score
        confidence = math.ceil((box.conf[0] * 100)) / 100
        print("Confidence --->", confidence)

        # Class name
        cls = int(box.cls[0])
        print("Class name -->", classNames[cls])

        # Put label on image
        org = [x1, y1]
        font = cv2.FONT_HERSHEY_SIMPLEX
        fontScale = 1
        color = (255, 0, 0)
        thickness = 2
        cv2.putText(img, classNames[cls], org, font, fontScale, color, thickness)

# Display the result
cv2.imshow("YOLOv8 Detection", img)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Output:

```
========================== RESTART: C:\CV\prac 10.py ==========================

0: 480x640 1 person, 1 couch, 122.6ms
Confidence ---> 0.66
Class name --> person
Confidence ---> 0.42
Class name --> sofa
Speed: 5.9ms preprocess, 122.6ms inference, 2.2ms postprocess per image at shape (1, 3, 480, 640)
```



Practical No:- 11

Aim: Camera Calibration.

Description: **Camera calibration** is the process of determining a camera's internal parameters (like focal length, optical center, and lens distortion) and external orientation in space. It typically involves capturing images of a known pattern (like a checkerboard) from different angles. Feature points on the pattern are detected in each image. These are used to compute the relationship between 3D world points and 2D image points. The result is a mathematical model that corrects distortions and improves measurement accuracy. Calibration is crucial in fields like robotics, AR, and 3D vision.

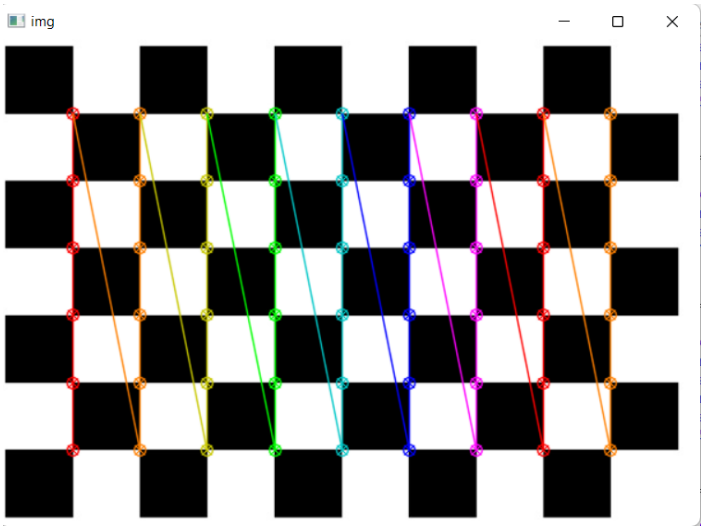
Code:

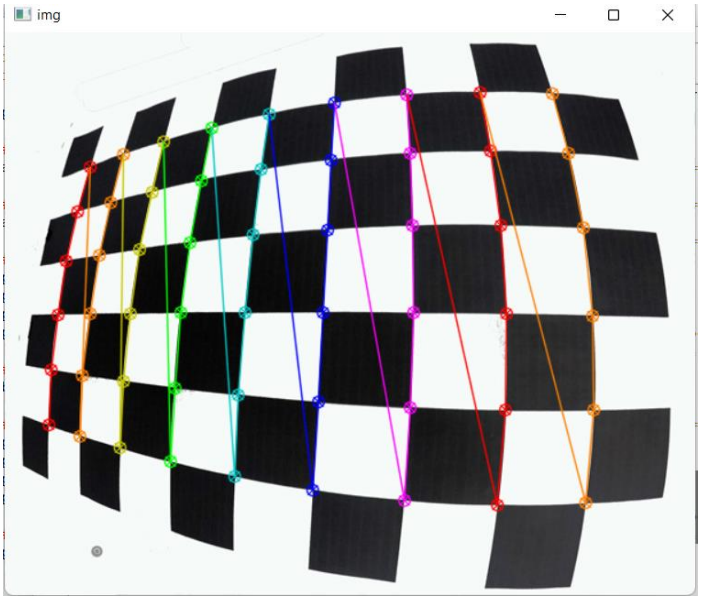
```
# Import required modules
import cv2
import numpy as np
import os
import glob

# Define the dimensions of checkerboard
CHECKERBOARD = (6, 9)

# stop the iteration when specified
# accuracy, epsilon, is reached or
# specified number of iterations are completed.
criteria = (cv2.TERM_CRITERIA_EPS +
            cv2.TERM_CRITERIA_MAX_ITER, 30, 0.001)

 # Vector for 3D points
threedpoints = []

 # Vector for 2D points
twodpoints = []

 # 3D points real world coordinates
objectp3d = np.zeros((1, CHECKERBOARD[0]
                      * CHECKERBOARD[1],
                      3), np.float32)
objectp3d[0, :, :2] = np.mgrid[0:CHECKERBOARD[0],
                               0:CHECKERBOARD[1]].T.reshape(-1, 2)
prev_img_shape = None

 # Extracting path of individual image stored
# in a given directory. Since no path is
# specified, it will take current directory
# jpg files alone
images = glob.glob('CC2.jpg')
```

```
for filename in images:
    image = cv2.imread(filename)
    grayColor = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

    # Find the chess board corners
    # If desired number of corners are
    # found in the image then ret = true
    ret, corners = cv2.findChessboardCorners(
                        grayColor, CHECKERBOARD,
                        cv2.CALIB_CB_ADAPTIVE_THRESH
                        + cv2.CALIB_CB_FAST_CHECK +
                        cv2.CALIB_CB_NORMALIZE_IMAGE)

    # If desired number of corners can be detected then,
    # refine the pixel coordinates and display
    # them on the images of checker board
    if ret == True:
        threedpoints.append(objectp3d)

        # Refining pixel coordinates
        # for given 2d points.
        corners2 = cv2.cornerSubPix(
            grayColor, corners, (11, 11), (-1, -1), criteria)

        twodpoints.append(corners2)

        # Draw and display the corners
        image = cv2.drawChessboardCorners(image, CHECKERBOARD, corners2, ret)

    cv2.imshow('img', image)
    cv2.waitKey(0)

cv2.destroyAllWindows()
 h, w = image.shape[:2]

 # Perform camera calibration by
# passing the value of above found out 3D points (threedpoints)
# and its corresponding pixel coordinates of the
# detected corners (twodpoints)
ret, matrix, distortion, r_vecs, t_vecs = cv2.calibrateCamera(
    threedpoints, twodpoints, grayColor.shape[::-1], None, None)

 # Displaying required output
print(" Camera matrix:")
print(matrix)
print("\n Distortion coefficient:")
print(distortion)
print("\n Rotation Vectors:")
print(r_vecs)
```

Output:

```
================== RESTART: C:/CV prac11/Camera Calibration.py ==================
 Camera matrix:
[[3.21592475e+03 0.00000000e+00 2.46243536e+02]
 [0.00000000e+00 3.21383001e+03 2.14013030e+02]
 [0.00000000e+00 0.00000000e+00 1.00000000e+00]]

 Distortion coefficient:
[[ 5.47441729e-02 -1.46319246e+01 -7.40585612e-04  4.91520754e-03
  -3.73215663e-01]]

 Rotation Vectors:
(array([[ 0.00422907],
       [-0.00454793],
       [-1.57078195]]),)
```





Camera Matrix: The camera matrix contains the intrinsic parameters that define how the camera captures and projects 3D scenes onto a 2D image. It includes the focal lengths (fx, fy), which in your case are around 3215 pixels, and the optical center (cx, cy), located at approximately (246.24, 214.01). These values describe the camera's internal geometry and are crucial for accurately modeling perspective. The matrix assumes no skew between image axes. This setup is used in image rectification, 3D reconstruction, and visual measurement.

Distortion Coefficient: These coefficients describe how the camera lens distorts the image. The first three values (k1, k2, k3) account for radial distortion, which bends straight lines, while the last two (p1, p2) represent tangential distortion due to lens misalignment. In your case, the distortion is dominated by a strong radial component ($k2 \approx -14.63$), suggesting noticeable curvature at the image edges. Tangential distortion is minimal. These values are applied to correct the raw image and obtain an undistorted view.

Rotation Vector: The rotation vector encodes the orientation of the camera relative to the calibration pattern using axis-angle representation. It can be converted into a 3×3 rotation matrix. Your vector indicates a rotation of about -1.57 radians (≈ -90°) around the Z-axis, showing the camera was rotated sideways during capture. This rotation helps position the camera correctly in 3D space. It's vital for determining the camera pose and transforming points between coordinate systems.

Practical No:- 12

Aim: Image Colorization.

Description: **Image colorization** is the process of adding color to grayscale (black-and-white) images using either manual techniques, algorithms, or AI-powered tools. The goal is to recreate a realistic or stylized color version of an image that originally lacked color information.
Applications:

- Restoration of historical photographs and films
- Enhancement of medical imaging
- Artistic reinterpretation
- Virtual reality and gaming
- Forensic analysis

Code:

```
import cv2
import numpy as np
import os

# Paths to load the model
DIR = r"C:\CV prac 12"
PROTOTXT = os.path.join(DIR, r"colorization_deploy_v2.prototxt")
POINTS = os.path.join(DIR, r"pts_in_hull.npy")
MODEL = os.path.join(DIR, r"colorization_release_v2.caffemodel")


# Load the Model
print("Load model")
net = cv2.dnn.readNetFromCaffe(PROTOTXT, MODEL)
pts = np.load(POINTS)

# Load centers for ab channel quantization used for rebalancing.
class8 = net.getLayerId("class8_ab")
conv8 = net.getLayerId("conv8_313_rh")
pts = pts.transpose().reshape(2, 313, 1, 1)
net.getLayer(class8).blobs = [pts.astype("float32")]
net.getLayer(conv8).blobs = [np.full([1, 313], 2.606, dtype="float32")]

# Load the input image
image = cv2.imread("image.jpg")
scaled = image.astype("float32") / 255.0
lab = cv2.cvtColor(scaled, cv2.COLOR_BGR2LAB)

resized = cv2.resize(lab, (224, 224))
L = cv2.split(resized)[0]
L -= 50

print("Colorizing the image")
```

```
net.setInput(cv2.dnn.blobFromImage(L))
ab = net.forward()[0, :, :, :].transpose((1, 2, 0))

ab = cv2.resize(ab, (image.shape[1], image.shape[0]))

L = cv2.split(lab)[0]
colorized = np.concatenate((L[:, :, np.newaxis], ab), axis=2)

colorized = cv2.cvtColor(colorized, cv2.COLOR_LAB2BGR)
colorized = np.clip(colorized, 0, 1)

colorized = (255 * colorized).astype("uint8")

cv2.imshow("Original", image)
cv2.imshow("Colorized", colorized)
cv2.waitKey(0)
```

Output:

```
================= RESTART: C:/CV prac 12/Image Colorization.py =================
Load model
Colorizing the image
```