adani | Institute of Digital Technology Management

Submission

of

Image Processing Using Python-Lab Manual

By

Prathmesh B. Talhande (20231043)

PGDM 2023-25

In Partial Fulfillment of the Requirements for the

Post Graduate Diploma in Management (BDA)

At

Adani Institute of Digital Technology Management

(AICTE Approved Two-Year Full-Time Programme)

Plot No. 225, Opp. Maharaj Hotel Lane, Jamiyatpura Road,

S G Highway, PO: Jamiyatpura Gandhinagar - 382423, Gujarat.

Index	*Page*

1. Format change

Objective:

Python code to convert image in different file format as given following

1) JPEG - Joint Photographic Expert Group

2) PNG - Portable Network Graphics

3) TIFF - Tag Image File Format

4) GIF - Graphics Interchange Format

Input:

- **Conversion of original image to JPEG, PNG, TIFF**

```python
import imageio.v3 as iio
import matplotlib.pyplot as plt
from PIL import Image

# Load an image using imageio
def load_image(file_path):
    """
    Load an image from the specified file path using imageio.

    Parameters:
    - file_path (str): The path to the image file to be loaded.

    Returns:
    - image (numpy.ndarray): The loaded image array.
    """
    image = iio.imread(file_path)
    return image

def display_image(image):
    """
    Display an image using matplotlib.

    Parameters:
    - image (numpy.ndarray): The image array to be displayed.
    """
    plt.imshow(image)
    plt.axis('off')  # Hide the axes
    plt.show()
```

```python
# Save image using Pillow (convert numpy array to PIL Image)
def save_image(image, output_path, format):
    """
    Save an image in a specified format to the given output path.

    Parameters:
    - image (numpy.ndarray): The image array to be saved.
    - output_path (str): The path where the image will be saved.
    - format (str): The format to save the image in (e.g., 'JPEG', 'PNG', 'TIFF').
    """
    pil_image = Image.fromarray(image)
    pil_image.save(output_path, format=format)

# Main function to demonstrate the process
def main():
    # File paths
    input_file_path = r"C:\Users\roari\Downloads\Mountain.jpg"  # Path to the input image
    output_file_path_jpeg = r"C:\Users\roari\Downloads\Mountain_converted.jpg"  # Path to save the JPEG version
    output_file_path_png = r"C:\Users\roari\Downloads\Mountain_converted.png"   # Path to save the PNG version
    output_file_path_tiff = r"C:\Users\roari\Downloads\Mountain_converted.tiff" # Path to save the TIFF version

    # Load the image
    image = load_image(input_file_path)

    # Display the image
    display_image(image)

    # Save the image in different formats
    save_image(image, output_file_path_jpeg, 'JPEG')
    save_image(image, output_file_path_png, 'PNG')
    save_image(image, output_file_path_tiff, 'TIFF')

# Call the main function directly
main()
```

```python
from PIL import Image
import matplotlib.pyplot as plt

def display_image(file_path, title):
    """
    Display an image from the specified file path.

    Parameters:
    - file_path (str): The path to the image file to be displayed.
    - title (str): The title of the plot.
    """
    image = Image.open(file_path)
    plt.imshow(image)
    plt.title(title)
    plt.axis('off')
    plt.show()

# Display the converted images
display_image(r"C:\Users\roari\Downloads\Mountain_converted.jpg", 'JPEG Format')
display_image(r"C:\Users\roari\Downloads\Mountain_converted.png", 'PNG Format')
display_image(r"C:\Users\roari\Downloads\Mountain_converted.tiff", 'TIFF Format')
```

- **Conversion of original image to GIF**

```python
from PIL import Image

# Load the JPG image
jpg_image_path = r"C:\Users\roari\Downloads\Bird 6.jpg"
gif_image_path = r"C:\Users\roari\Downloads\Bird 6.gif"

# Open the JPG image
with Image.open(jpg_image_path) as img:
    # Convert to GIF and save it
    img.convert('RGB').save(gif_image_path, 'GIF')

# Open and display the GIF image
with Image.open(gif_image_path) as gif_img:
    gif_img.show()
```

```python
from IPython.display import Image, display

# Paths to the images
jpg_image_path = r"C:\Users\roari\Downloads\Bird 6.jpg"
gif_image_path = r"C:\Users\roari\Downloads\Bird 6.gif"

# Display the original JPG image
print("Original JPG Image:")
display(Image(filename=jpg_image_path))

# Display the converted GIF image
print("Converted GIF Image:")
display(Image(filename=gif_image_path))
```

Output

- **Original image**

- **Converted Images:**

JPEG Format

PNG Format

TIFF Format

- **Conversion of original image to GIF**

Original JPG Image:

Converted GIF Image:

Result:

Successfully converted all images from their original format to multiple file formats like JPEG, PNG, TIFF and GIF.

2. Crop, resize and rotate

Objective:

To perform Cropping and resizing and rotating operation on image and making interpretation based on that.

Input:

- **Performing Cropping and resizing on image**

```
from PIL import Image
import matplotlib.pyplot as plt
import os

# Define the paths
input_image_path = r"C:\Users\roari\Downloads\bird.jpg"

# Load the original image
image = Image.open(input_image_path)

# Print original image dimensions
width, height = image.size
print(f"Original image dimensions: {width} x {height}")

# Define cropping coordinates
left = 50
top = 50
right = min(width, 600)
bottom = min(height, 400)

# Check if cropping coordinates are within image dimensions
if left < 0 or top < 0 or right > width or bottom > height or left >= right or top >= bottom:
    raise ValueError("Cropping coordinates are out of the image bounds or invalid.")

# Crop the original image
cropped_image = image.crop((left, top, right, bottom))

# Resize the original image
resized_image = image.resize((200, 200))

# Plot the original, cropped, and resized images
fig, axes = plt.subplots(1, 3, figsize=(15, 5))
fig.suptitle('Original, Cropped, and Resized Images', fontsize=20)
```

```
# Original Image
axes[0].imshow(image)
axes[0].set_title('Original Image')
axes[0].axis('off')

# Cropped Image
axes[1].imshow(cropped_image)
axes[1].set_title('Cropped Image')
axes[1].axis('off')

# Resized Image
axes[2].imshow(resized_image)
axes[2].set_title('Resized Image')
axes[2].axis('off')

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

Cropping and resizing histogram

```
# Convert images to grayscale for boxplot analysis
original_gray = image.convert("L")
cropped_gray = cropped_image.convert("L")
resized_gray = resized_image.convert("L")

# Convert images to arrays
original_array = list(original_gray.getdata())
cropped_array = list(cropped_gray.getdata())
resized_array = list(resized_gray.getdata())

# Plot the boxplots
fig, ax = plt.subplots(figsize=(10, 6))
ax.boxplot([original_array, cropped_array, resized_array], labels=["Original", "Cropped", "Resized"])
ax.set_title("Boxplot of Pixel Intensities")
ax.set_ylabel("Pixel Intensity")

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

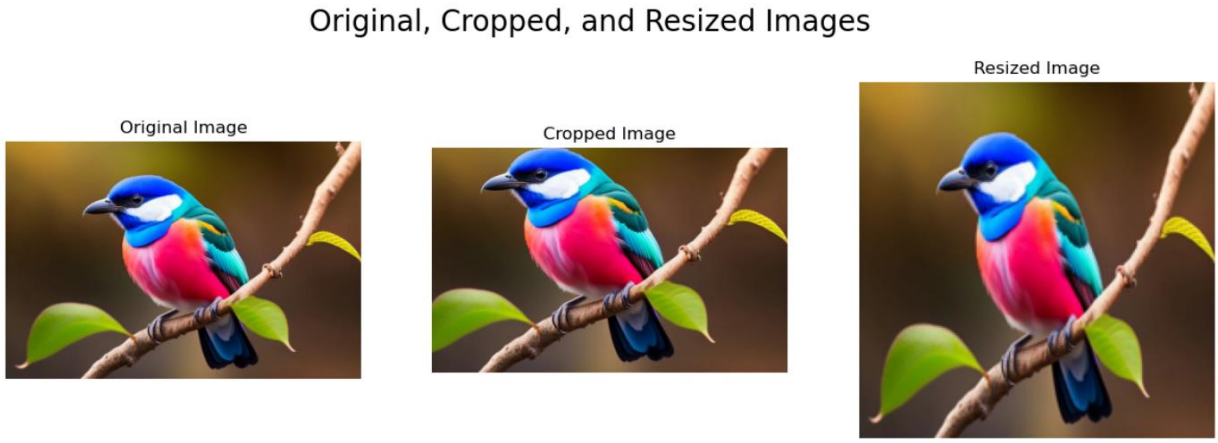
- **Rotating Image**

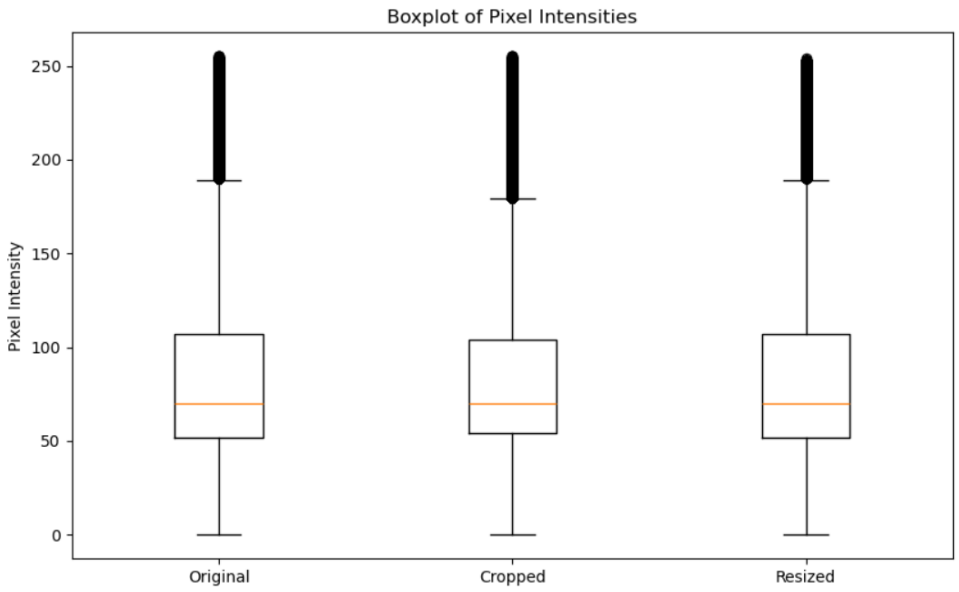
```
from PIL import Image
import matplotlib.pyplot as plt
import os

# Define the paths
input_image_path = r"C:\Users\roari\Downloads\bird.jpg"

# Load the original image
image = Image.open(input_image_path)

# Print original image dimensions
width, height = image.size
print(f"Original image dimensions: {width} x {height}")

# Rotate the image by different angles
rotated_45 = image.rotate(45)
rotated_90 = image.rotate(90)
rotated_180 = image.rotate(180)
rotated_360 = image.rotate(360)

# Plot the original and rotated images
fig, axes = plt.subplots(1, 5, figsize=(20, 5))
fig.suptitle('Original and Rotated Images', fontsize=20)

# Original Image
axes[0].imshow(image)
axes[0].set_title('Original Image')
axes[0].axis('off')

# Rotated by 45 degrees
axes[1].imshow(rotated_45)
axes[1].set_title('Rotated 45°')
axes[1].axis('off')
```

```
# Rotated by 90 degrees
axes[2].imshow(rotated_90)
axes[2].set_title('Rotated 90°')
axes[2].axis('off')

# Rotated by 180 degrees
axes[3].imshow(rotated_180)
axes[3].set_title('Rotated 180°')
axes[3].axis('off')

# Rotated by 360 degrees
axes[4].imshow(rotated_360)
axes[4].set_title('Rotated 360°')
axes[4].axis('off')

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

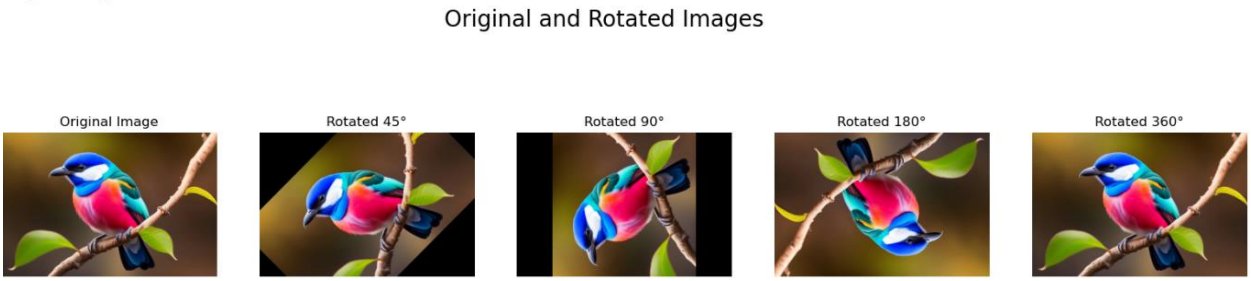
Output:

- **Performing Cropping and resizing on image**





- **Rotating Image**



Result:

- **Cropping and resizing on image**

Key observations:

Median pixel intensity: The median pixel intensity appears to be similar across all three operations, suggesting that the overall brightness of the images is not significantly affected by cropping or resizing.

IQR (Interquartile Range): The IQR, which represents the spread of the middle 50% of the data, is also similar across the three operations, indicating that the distribution of pixel intensities is relatively consistent.

Outliers: There are some outliers present in all three groups, as indicated by the individual points beyond the whiskers. These outliers are higher in cropped image while lower in original and resized image.

- **Rotating Image**

Image rotation does not significantly affect the overall brightness or the distribution of pixel intensities, It just rotate the images at particular angle.

3. Histogram equalization

Objective:

The primary objective of **Histogram Equalization** is to enhance the contrast of an image by redistributing the intensity values more evenly across the entire range of the grayscale or color spectrum.

Input:

- **Histogram Equalization code:**

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

# Define the paths
input_image_path = r"C:\Users\roari\Downloads\bird.jpg"
output_folder_path = r"C:\Users\roari\Downloads\output"

# Create the output folder if it doesn't exist
os.makedirs(output_folder_path, exist_ok=True)

# Load the original image
image = Image.open(input_image_path)

# Print original image dimensions
width, height = image.size
print(f"Original image dimensions: {width} x {height}")

# Crop the original image
left, top, right, bottom = 50, 50, min(width, 600), min(height, 400)
cropped_image = image.crop((left, top, right, bottom))
cropped_image_path = os.path.join(output_folder_path, "cropped_bird.jpg")
cropped_image.save(cropped_image_path)
print(f"Cropped image saved at: {cropped_image_path}")

# Resize the original image
resized_image = image.resize((200, 200))
resized_image_path = os.path.join(output_folder_path, "resized_bird.jpg")
resized_image.save(resized_image_path)
print(f"Resized image saved at: {resized_image_path}")
```

```
# Phase 2: Analysis

# Reload the image for analysis
image = Image.open(input_image_path)

# Split the image into its respective Red, Green, and Blue channels
red_channel, green_channel, blue_channel = image.split()

# Calculate histograms for each channel
red_histogram = red_channel.histogram()
green_histogram = green_channel.histogram()
blue_histogram = blue_channel.histogram()

# Perform histogram equalization on the RGB image
equalized_image = ImageOps.equalize(image)
equalized_red_channel, equalized_green_channel, equalized_blue_channel = equalized_image.split()

# Calculate histograms for each channel of the equalized image
equalized_red_histogram = equalized_red_channel.histogram()
equalized_green_histogram = equalized_green_channel.histogram()
equalized_blue_histogram = equalized_blue_channel.histogram()

# Plot histograms and images
fig, axes = plt.subplots(3, 3, figsize=(24, 18))
fig.suptitle('Image Analysis: Histograms and Images', fontsize=24)

# Original Channel Histograms
axes[0, 0].plot(red_histogram, color='red')
axes[0, 0].set_title('Original Red Histogram', fontsize=16)
axes[0, 0].set_xlabel('Pixel Value', fontsize=14)
axes[0, 0].set_ylabel('Frequency', fontsize=14)
```

```
axes[0, 1].plot(green_histogram, color='green')
axes[0, 1].set_title('Original Green Histogram', fontsize=16)
axes[0, 1].set_xlabel('Pixel Value', fontsize=14)
axes[0, 1].set_ylabel('Frequency', fontsize=14)

axes[0, 2].plot(blue_histogram, color='blue')
axes[0, 2].set_title('Original Blue Histogram', fontsize=16)
axes[0, 2].set_xlabel('Pixel Value', fontsize=14)
axes[0, 2].set_ylabel('Frequency', fontsize=14)

# Equalized Channel Histograms
axes[1, 0].plot(equalized_red_histogram, color='red')
axes[1, 0].set_title('Equalized Red Histogram', fontsize=16)
axes[1, 0].set_xlabel('Pixel Value', fontsize=14)
axes[1, 0].set_ylabel('Frequency', fontsize=14)

axes[1, 1].plot(equalized_green_histogram, color='green')
axes[1, 1].set_title('Equalized Green Histogram', fontsize=16)
axes[1, 1].set_xlabel('Pixel Value', fontsize=14)
axes[1, 1].set_ylabel('Frequency', fontsize=14)

axes[1, 2].plot(equalized_blue_histogram, color='blue')
axes[1, 2].set_title('Equalized Blue Histogram', fontsize=16)
axes[1, 2].set_xlabel('Pixel Value', fontsize=14)
axes[1, 2].set_ylabel('Frequency', fontsize=14)

# Original and Equalized Images
axes[2, 0].imshow(image)
axes[2, 0].set_title('Original Image')
axes[2, 0].axis('off')
```

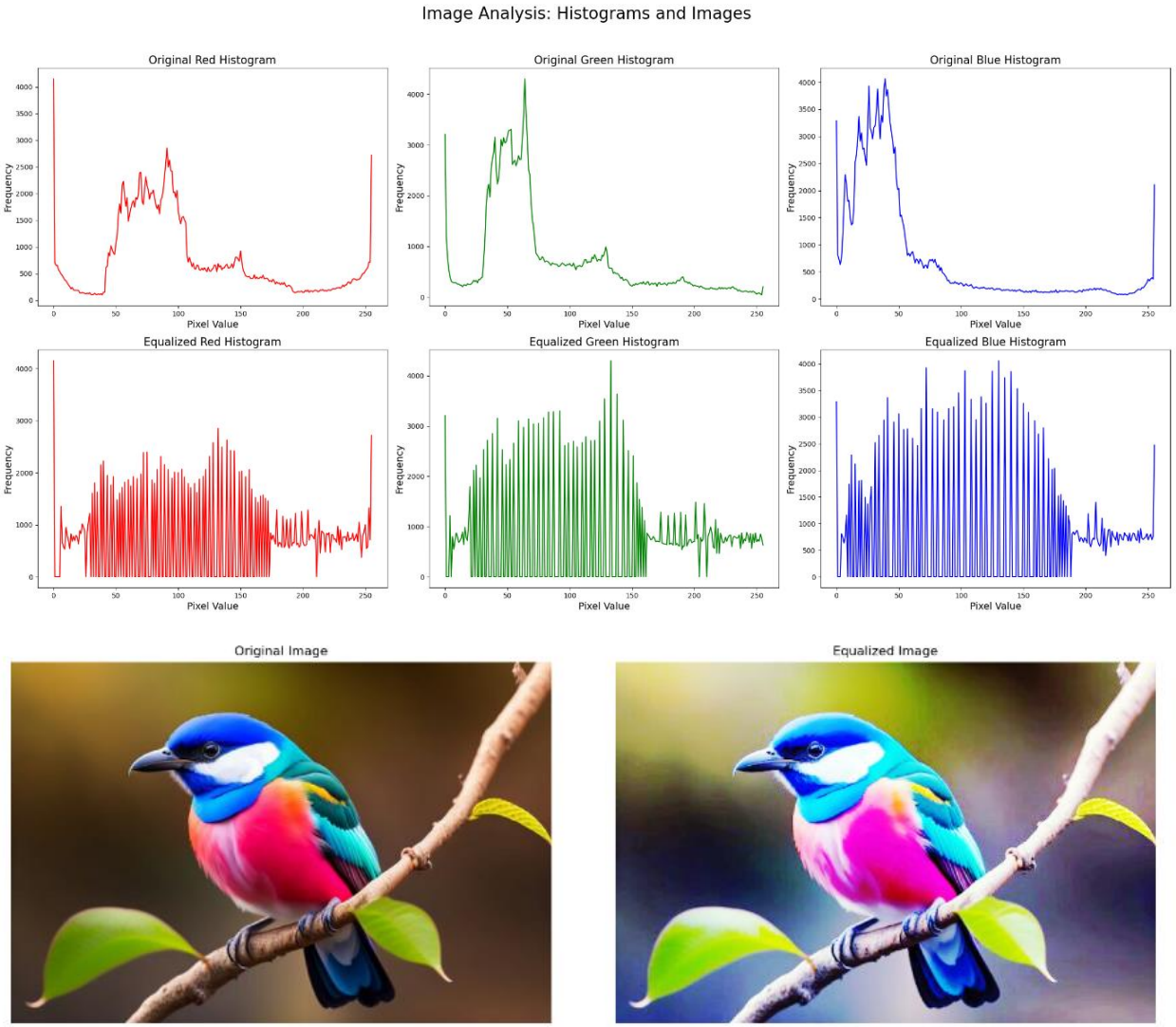
```
axes[2, 1].imshow(equalized_image)
axes[2, 1].set_title('Equalized Image')
axes[2, 1].axis('off')

# Hide empty box
axes[2, 2].axis('off')

# Adjust layout to prevent overlapping
plt.tight_layout(rect=[0, 0, 1, 0.96])
plt.show()
```

- **Boxplot of Equalized image:**

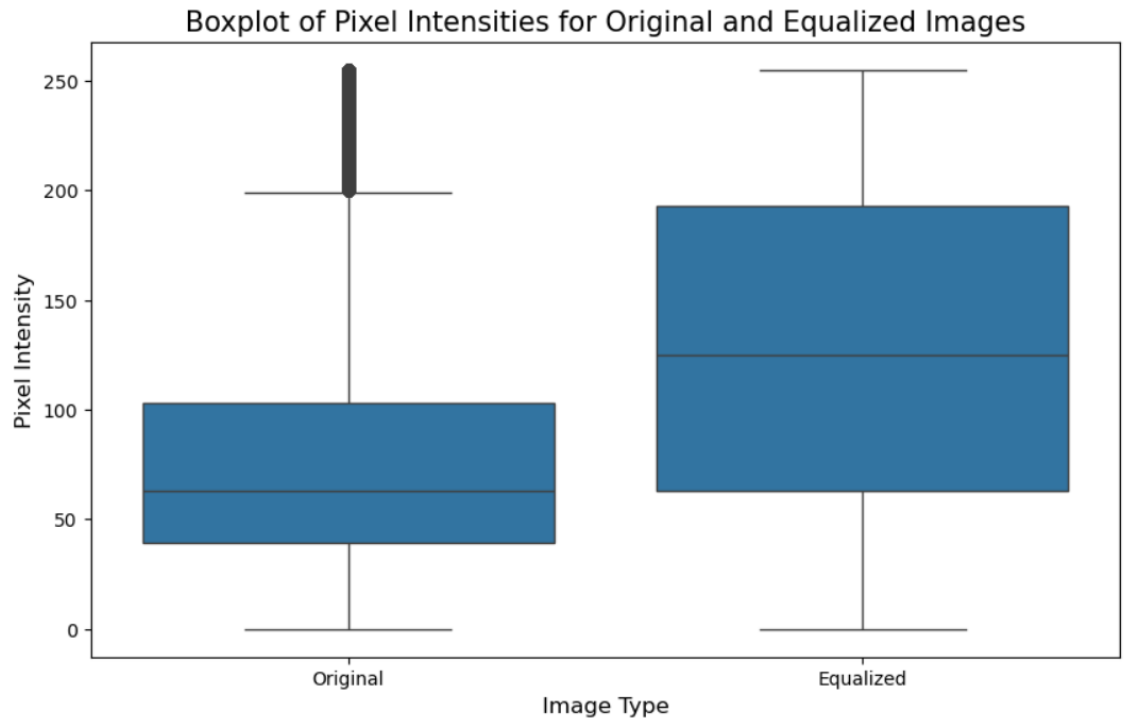
```
import numpy as np
import seaborn as sns
import pandas as pd

# Convert the original and equalized images to NumPy arrays
original_array = np.array(image).flatten()
equalized_array = np.array(equalized_image).flatten()

# Create a DataFrame for easier plotting with Seaborn
image_data = pd.DataFrame({
    'Original': original_array,
    'Equalized': equalized_array
})

# Prepare data for plotting by melting the DataFrame
melted_data = pd.melt(image_data, var_name='Image Type', value_name='Pixel Intensity')

# Plot boxplot for pixel intensities
plt.figure(figsize=(10, 6))
sns.boxplot(x='Image Type', y='Pixel Intensity', data=melted_data)
plt.title('Boxplot of Pixel Intensities for Original and Equalized Images', fontsize=15)
plt.xlabel('Image Type', fontsize=12)
plt.ylabel('Pixel Intensity', fontsize=12)
plt.show()
```

Output:

- **Histogram Equalization code:**



- **Boxplot of Equalized image:**



Result:

Key observations:

Median pixel intensity: The median pixel intensity is higher for equalized images compared to original images. This indicates that equalization has increased the overall brightness of the images.

IQR (Interquartile Range): The IQR, which represents the spread of the middle 50% of the data, is larger for equalized images. This suggests that the distribution of pixel intensities is more spread out after equalization.

Outliers: There are fewer outliers present in equalized images compared to original images. This indicates that equalization has reduced the number of extreme pixel values or noise.

Distribution shape: The box plots suggest that the distribution of pixel intensities is slightly skewed to the right for both original and equalized images, but the skewness is reduced after equalization.

4. Filters (gaussian, median, uniform, high pass filter)

Objective:

The objective of applying various image filters is to enhance specific features of an image while reducing noise or unwanted details, depending on the filter type.

Input:

- **Performing different types of filter operation on images**

```
# Import the necessary libraries
from skimage import io, img_as_float
from skimage.filters import gaussian, median
from skimage.morphology import disk
from scipy.ndimage import uniform_filter
import matplotlib.pyplot as plt

# Load an example image from the specified path
image_path = r"C:\Users\roari\Downloads\bird.jpg"
image = img_as_float(io.imread(image_path, as_gray=True))

"""
Load the image from the specified path and convert it to a float representation.
The image is loaded as a grayscale image using the 'as_gray=True' parameter.
"""

# Apply Gaussian filter (Low-Pass Filter)
gaussian_filtered = gaussian(image, sigma=1)

"""
Apply a Gaussian filter to the image.
The Gaussian filter is a low-pass filter that smooths the image by reducing noise and details.
'sigma' controls the extent of the blurring; a larger sigma means more blurring.
"""

# Apply Median filter
median_filtered = median(image, disk(3))

"""
Apply a Median filter to the image.
The Median filter is a non-linear filter used for noise reduction, particularly effective at removing salt-and-pepper noise.
'disk(3)' specifies the size of the neighborhood for the median calculation (a disk of radius 3).
"""
```

```
# Apply Uniform filter (Low-Pass Filter)
uniform_filtered = uniform_filter(image, size=3)

"""
Apply a Uniform filter to the image.
The Uniform filter, also known as a box filter or average filter, smooths the image by averaging the pixels in a specified neighborhood.
'size=3' specifies the size of the square neighborhood.
"""

# Apply High-Pass filter (original - blurred)
blurred_image = gaussian(image, sigma=2)
high_pass_filtered = image - blurred_image

"""
Apply a High-Pass filter to the image by subtracting a blurred version of the image from the original.
The High-Pass filter enhances the edges and details by highlighting the differences between the original image and its blurred version.
'sigma=2' is used for the Gaussian blur, which determines the level of smoothing before subtraction.
"""

# Display the results in a 2x3 grid
fig, axes = plt.subplots(2, 3, figsize=(20, 15))
ax = axes.ravel()

"""
Set up the plotting area with 2 rows and 3 columns of subplots using Matplotlib.
'figsize=(20, 15)' sets the size of the entire figure.
'axes.ravel()' flattens the 2D array of axes into a 1D array for easier indexing.
"""

# Display original image
ax[0].imshow(image, cmap='gray')
ax[0].set_title('Original Image')
```

```
"""
Display the original image in the first subplot.
'cmap='gray'' ensures the image is displayed in grayscale.
'set_title' sets the title of the subplot.
"""

# Display Gaussian filtered image
ax[1].imshow(gaussian_filtered, cmap='gray')
ax[1].set_title('Gaussian Filtered')

"""
Display the Gaussian filtered image in the second subplot.
"""

# Display Median filtered image
ax[2].imshow(median_filtered, cmap='gray')
ax[2].set_title('Median Filtered')

"""
Display the Median filtered image in the third subplot.
"""

# Display Uniform filtered image
ax[3].imshow(uniform_filtered, cmap='gray')
ax[3].set_title('Uniform Filtered')

"""
Display the Uniform filtered image in the fourth subplot.
"""

# Display High-Pass filtered image
ax[4].imshow(high_pass_filtered, cmap='gray')
ax[4].set_title('High-Pass Filtered')
```

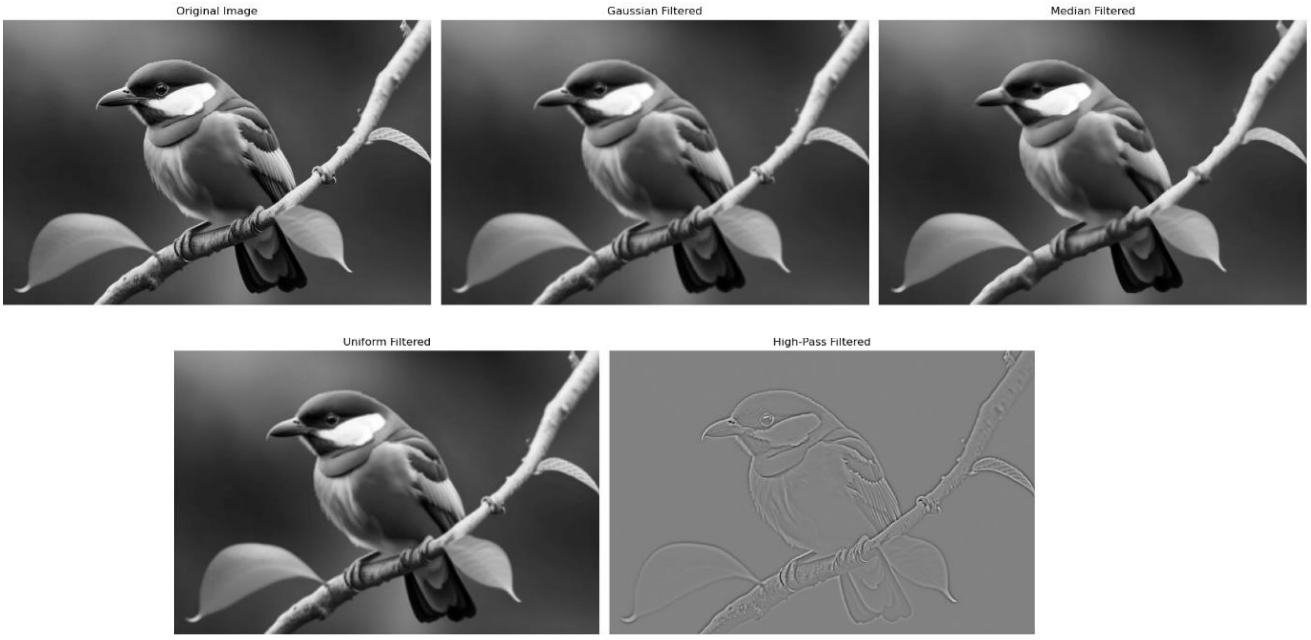
```
# Display High-Pass filtered image
ax[4].imshow(high_pass_filtered, cmap='gray')
ax[4].set_title('High-Pass Filtered')

"""
Display the High-Pass filtered image in the fifth subplot.
"""

# Remove the axis from the 6th subplot (empty)
fig.delaxes(axes[1, 2])

"""
Remove the axis from the sixth subplot since it is not being used.
'delaxes' is used to remove a subplot from the figure.
"""

for a in ax:
    a.axis('off')

"""
Remove the axis ticks and labels from all subplots for a cleaner look.
"""

plt.tight_layout()

"""
Adjust the subplot parameters to give specified padding between plots and display the figure.
'tight_layout()' automatically adjusts subplot parameters to give specified padding.
'show()' renders and displays the plot.
"""
plt.show()
```

- **Boxplot for filtered Images:**

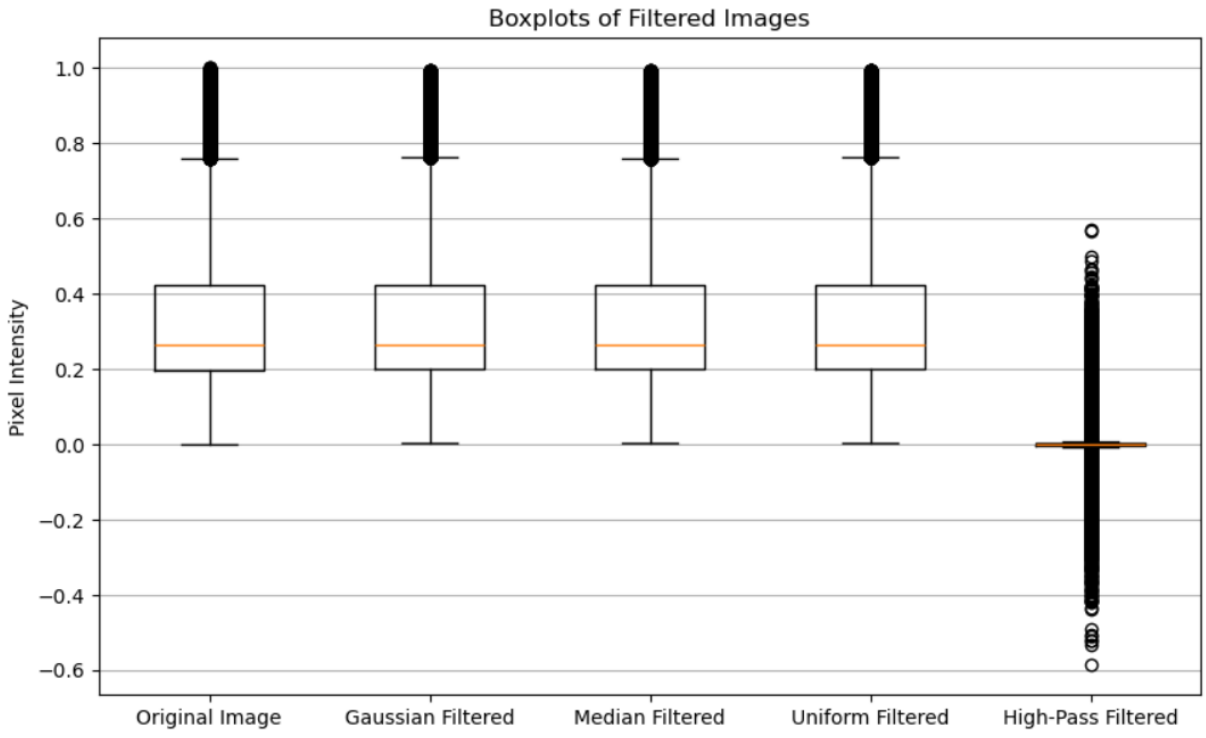
```
# Create a smaller boxplot for each image
plt.figure(figsize=(10, 6))  # Adjust the figure size
plt.boxplot([img.ravel() for img in images], labels=titles, widths=0.5)  # Set the width of the boxplots

"""
Create a boxplot for each image in the 'images' list.
'ravel()' flattens the 2D image arrays into 1D arrays for boxplotting.
'labels' specifies the titles for each boxplot.
'widths' adjusts the width of the boxplots for a smaller appearance.
"""

plt.title('Boxplots of Filtered Images')
plt.ylabel('Pixel Intensity')
plt.grid(axis='y')

"""
Set the title for the boxplot figure.
'label' sets the label for the y-axis.
'grid' adds a grid along the y-axis for better visibility.
"""

plt.show()

"""
Display the boxplot figure.
"""
```

Output:

- **Performing different types of filter operation on images**



- **Boxplot for filtered Images:**



Result:

Outliers: There are more outliers present in the high-pass filtered image compared to the other filters. This suggests that high-pass filtering might introduce more extreme pixel values.

Distribution shape: The box plots suggest that the distribution of pixel intensities is slightly skewed to the right for all images, with the exception of the high-pass filtered image, which appears to have a more symmetric distribution.

5. Noise reduction techniques

Objective:

The objective of applying **noise reduction techniques** using **Gaussian** and **Median filters** is to improve the **visual quality** of images by **minimizing noise** while preserving important image details

Input:

- **Noise Reduction using gaussian and median filter**

```
# Import necessary libraries
import numpy as np
import matplotlib.pyplot as plt
from skimage import io, img_as_float
from skimage.filters import gaussian, median
from skimage.morphology import disk

# Load an example grayscale image
image_path = r"C:\Users\roari\Downloads\Bird 9.jpg"
image = io.imread(image_path, as_gray=True)

# Convert image to float representation (values between 0 and 1)
image = img_as_float(image)

# Introduce synthetic noise (optional, for demonstration purposes)
noisy_image = image + 0.2 * np.random.standard_normal(image.shape)
noisy_image = np.clip(noisy_image, 0, 1)  # Clip values to ensure they're between 0 and 1

# Apply Gaussian filter
gaussian_filtered = gaussian(noisy_image, sigma=1)

# Apply Median filter
median_filtered = median(noisy_image, disk(3))

# Plot the original, noisy, and filtered images
plt.figure(figsize=(15, 10))
```

```
plt.subplot(2, 2, 1)
plt.title('Original Image')
plt.imshow(image, cmap='gray')
plt.axis('off')

plt.subplot(2, 2, 2)
plt.title('Noisy Image')
plt.imshow(noisy_image, cmap='gray')
plt.axis('off')

plt.subplot(2, 2, 3)
plt.title('Gaussian Filtered Image')
plt.imshow(gaussian_filtered, cmap='gray')
plt.axis('off')

plt.subplot(2, 2, 4)
plt.title('Median Filtered Image')
plt.imshow(median_filtered, cmap='gray')
plt.axis('off')

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

- **boxplot comparison of the original, noisy, and filtered images**

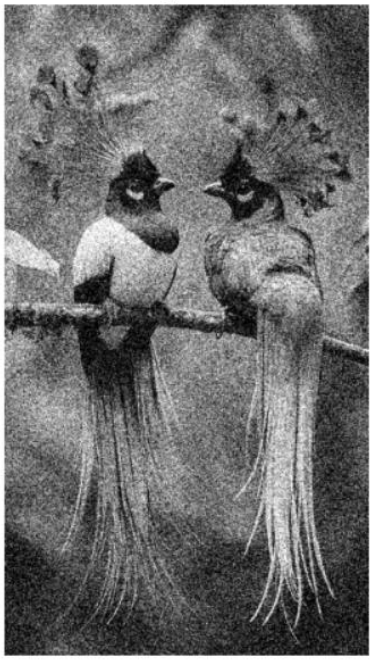
```
# Create a boxplot comparison of the original, noisy, and filtered images
plt.figure(figsize=(10, 6))

# Gather the pixel values from each image
data = [
    image.flatten(),
    noisy_image.flatten(),
    gaussian_filtered.flatten(),
    median_filtered.flatten()
]

# Create boxplot
plt.boxplot(data, labels=['Original', 'Noisy', 'Gaussian Filtered', 'Median Filtered'])
plt.title('Boxplot Comparison of Image Intensities')
plt.ylabel('Pixel Intensity')
plt.grid()
plt.show()
```

Output:

- **Noise Reduction using gaussian and median filter**

Original Image

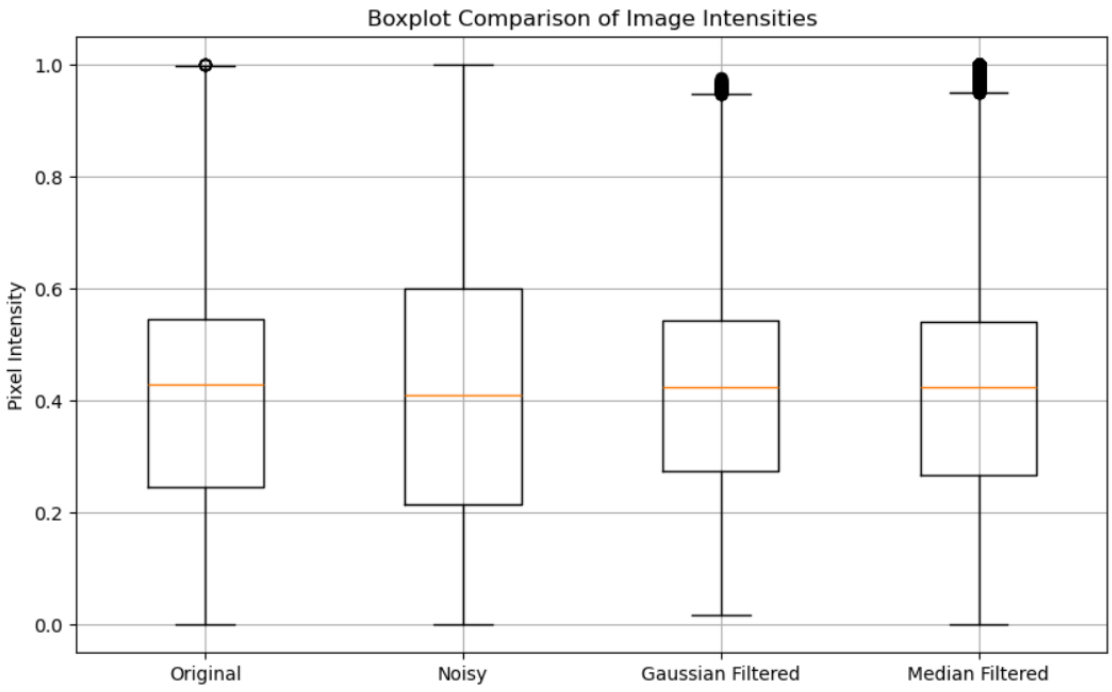
Noisy Image



Gaussian Filtered Image

Median Filtered Image

- **boxplot comparison of the original, noisy, and filtered images**



Result:

IQR (Interquartile Range): The IQR, which represents the spread of the middle 50% of the data, is larger for the noisy image compared to the other images. This indicates that noise addition has increased the variability of pixel intensities.

Gaussian filtering and median filtering can help to reduce the impact of noise, as evidenced by the smaller IQR.

6. Morphological

Erosion

Objective:

The objective of **erosion** in image processing is to reduce the size of objects and remove small, isolated foreground elements, such as noise or minor imperfections, from a binary or grayscale image.

Input:

- **Performing erosion operation:**

```
# Import necessary libraries
import numpy as np
import matplotlib.pyplot as plt
from skimage import io, img_as_float
from skimage.morphology import erosion, disk

# Load an example image from the specified path
image_path = r"C:\Users\roari\Downloads\Man 1.jpg"
image = img_as_float(io.imread(image_path, as_gray=True))

# Binary thresholding:
#Converts the grayscale image into a binary image where pixel values greater than 0.5 are set to True (or 1), and others are set to False (or 0).
#erosion are typically performed on binary images
binary_image = image > 0.5

# Define a structuring element
selem = disk(5)

# Perform erosion
eroded_image = erosion(binary_image, selem)

# Plot the original and eroded images
plt.figure(figsize=(10, 5))
```

```
# Perform erosion
eroded_image = erosion(binary_image, selem)

# Plot the original and eroded images
plt.figure(figsize=(10, 5))

plt.subplot(1, 2, 1)
plt.title('Original Image')
plt.imshow(binary_image, cmap='gray')
plt.axis('off')

plt.subplot(1, 2, 2)
plt.title('Eroded Image')
plt.imshow(eroded_image, cmap='gray')
plt.axis('off')

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

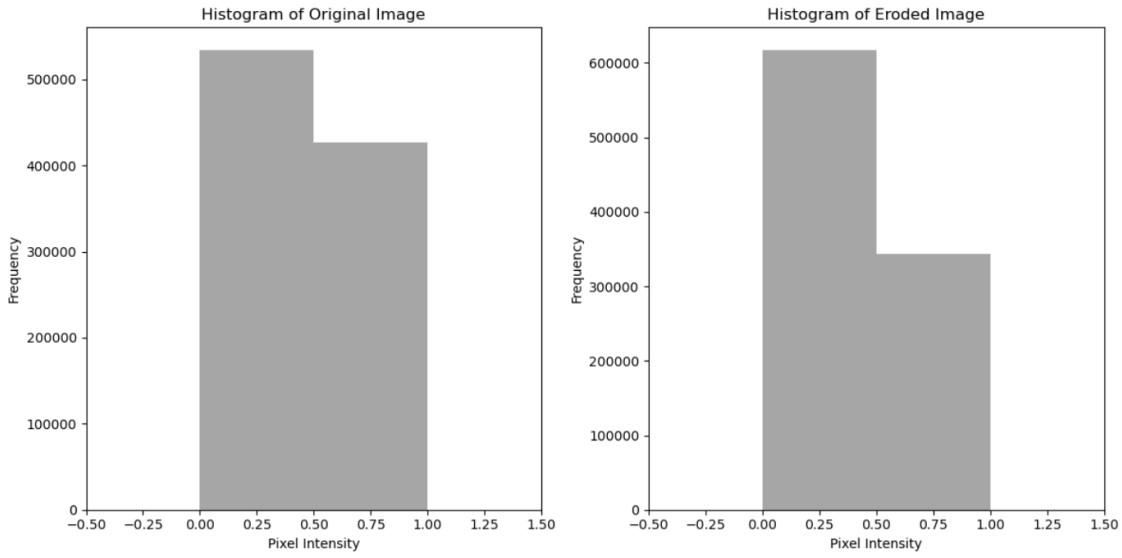
- **Histogram evaluation**

```
# Plot histograms for both images
plt.figure(figsize=(12, 6))

# Histogram for the original binary image (converted to integers)
plt.subplot(1, 2, 1)
plt.title('Histogram of Original Image')
plt.hist(binary_image.astype(int).ravel(), bins=2, color='gray', alpha=0.7)
plt.xlim(-0.5, 1.5)  # Limiting x-axis for binary image (0 to 1)
plt.xlabel('Pixel Intensity')
plt.ylabel('Frequency')

# Histogram for the eroded image (also converted to integers)
plt.subplot(1, 2, 2)
plt.title('Histogram of Eroded Image')
plt.hist(eroded_image.astype(int).ravel(), bins=2, color='gray', alpha=0.7)
plt.xlim(-0.5, 1.5)  # Limiting x-axis for binary image (0 to 1)
plt.xlabel('Pixel Intensity')
plt.ylabel('Frequency')

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

Output:

- **Performing erosion operation:**

- **Histogram evaluation**



Result:

0: Represents the darkest pixel intensity (black). (Low intensity)

1: Represents the brightest pixel intensity (white). (High intensity)

Erosion decreases the brightness intensity of the bright regions (foreground) and can make dark regions appear more dominant.

The frequency of pixels with intensities between 0.5 and 1 is significantly lower for the eroded image compared to the original image. This suggests that erosion has reduced the number of pixels with high intensity values. Thus Morphological erosion decreses the brightness.

Dilation

Objective:

The objective of **dilation** in image processing is to grow or expand the boundaries of objects in a binary or grayscale image. This process adds pixels to the edges of objects based on the shape and size of a structuring element.

Input:

- **Performing Image dilation**

```
# Import necessary libraries
import numpy as np
import matplotlib.pyplot as plt
from skimage import io, img_as_float
from skimage.morphology import dilation, disk

# Load an example image from the specified path
image_path = r"C:\Users\roari\Downloads\Man 1.jpg"
image = img_as_float(io.imread(image_path, as_gray=True))

# Apply binary thresholding for demonstration purposes
binary_image = image > 0.5

# Define a structuring element
selem = disk(5)

# Perform dilation
dilated_image = dilation(binary_image, selem)

# Plot the original and dilated images
plt.figure(figsize=(10, 5))
```

```
plt.subplot(1, 2, 1)
plt.title('Original Image')
plt.imshow(binary_image, cmap='gray')
plt.axis('off')

plt.subplot(1, 2, 2)
plt.title('Dilated Image')
plt.imshow(dilated_image, cmap='gray')
plt.axis('off')

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

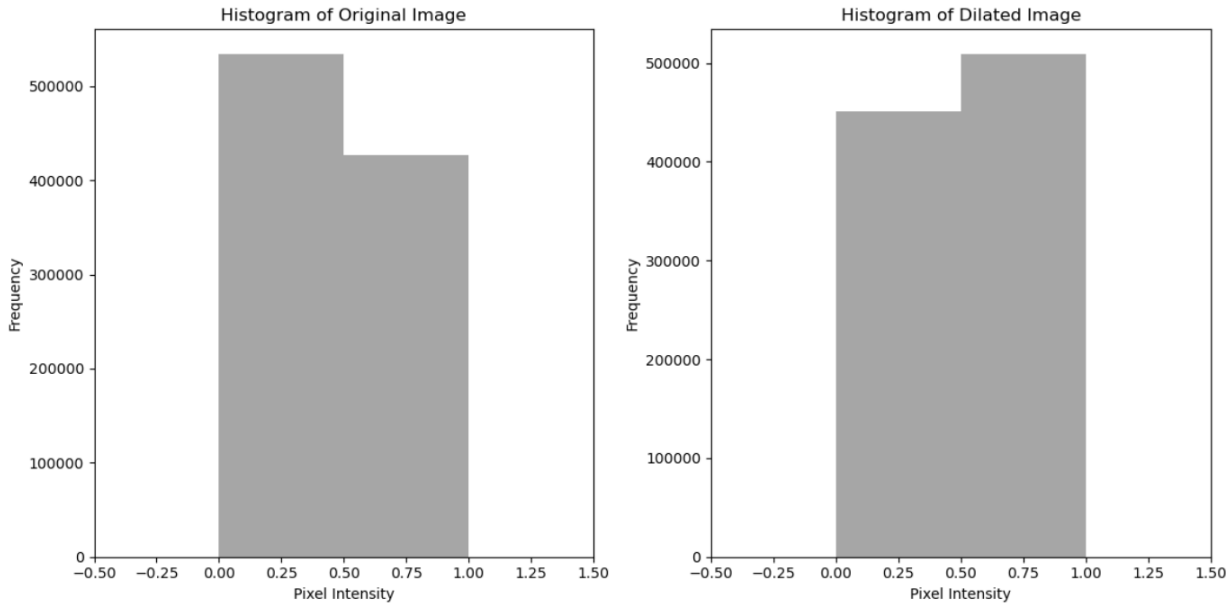
- **Histogram for pixel intensity comparison**

```
# Plot histograms for both images
plt.figure(figsize=(12, 6))

# Histogram for the original binary image (converted to integers)
plt.subplot(1, 2, 1)
plt.title('Histogram of Original Image')
plt.hist(binary_image.astype(int).ravel(), bins=2, color='gray', alpha=0.7)
plt.xlim(-0.5, 1.5)  # Limiting x-axis for binary image (0 to 1)
plt.xlabel('Pixel Intensity')
plt.ylabel('Frequency')

# Histogram for the dilated image (also converted to integers)
plt.subplot(1, 2, 2)
plt.title('Histogram of Dilated Image')
plt.hist(dilated_image.astype(int).ravel(), bins=2, color='gray', alpha=0.7)
plt.xlim(-0.5, 1.5)  # Limiting x-axis for binary image (0 to 1)
plt.xlabel('Pixel Intensity')
plt.ylabel('Frequency')

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

Output:

- **Performing Image dilation**

- **Histogram for pixel intensity comparison**



Result:

0: Represents the darkest pixel intensity (black). (Low intensity)

1: Represents the brightest pixel intensity (white). (High intensity)

Dilation increases the pixel intensity of the bright regions (High intensity) (foreground) in a binary image, making them larger and more prominent.

This suggests that dilation has reduced the number of pixels with lower intensity values.

Opening and closing

Objective:

Opening and **Closing** are fundamental morphological operations that serve to refine the structure of objects in an image by removing noise and smoothing boundaries.

Input:

- **Performing Opening and closing operation:**

```
# Import necessary libraries
import numpy as np
import matplotlib.pyplot as plt
from skimage import io, img_as_float
from skimage.morphology import opening, closing, disk

# Load an example image from the specified path
image_path = r"C:\Users\roari\Downloads\Man 1.jpg"
image = img_as_float(io.imread(image_path, as_gray=True))

# Apply binary thresholding for demonstration purposes
binary_image = image > 0.5

# Define a structuring element
selem = disk(5)

# Perform opening
opened_image = opening(binary_image, selem)

# Perform closing
closed_image = closing(binary_image, selem)

# Plot the original, opened, and closed images
plt.figure(figsize=(15, 10))
```

```
plt.subplot(1, 3, 1)
plt.title('Original Image')
plt.imshow(binary_image, cmap='gray')
plt.axis('off')

plt.subplot(1, 3, 2)
plt.title('Opened Image')
plt.imshow(opened_image, cmap='gray')
plt.axis('off')

plt.subplot(1, 3, 3)
plt.title('Closed Image')
plt.imshow(closed_image, cmap='gray')
plt.axis('off')

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

- **Evaluating histogram:**

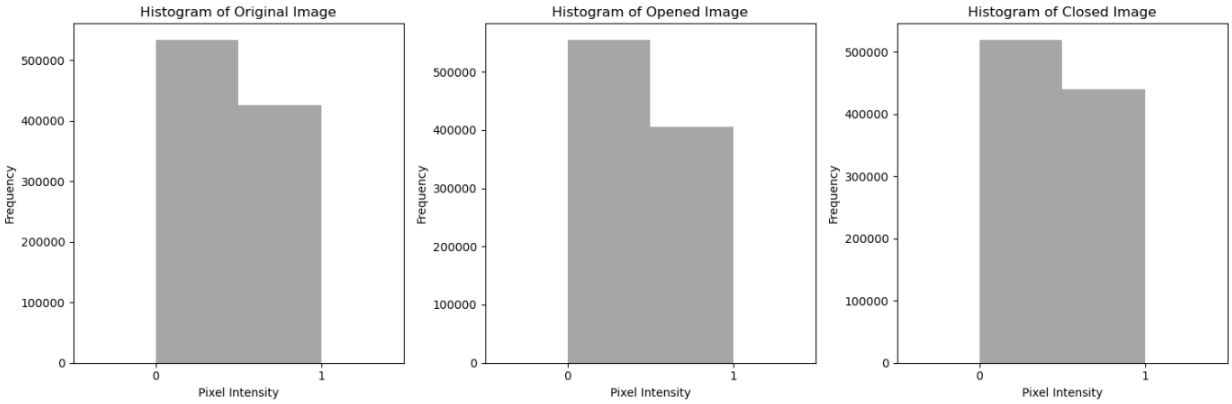
```python
# Import necessary libraries
import numpy as np
import matplotlib.pyplot as plt
from skimage import io, img_as_float
from skimage.morphology import opening, closing, disk

# Load an example image from the specified path
image_path = r"C:\Users\roari\Downloads\Man 1.jpg"
image = img_as_float(io.imread(image_path, as_gray=True))

# Apply binary thresholding for demonstration purposes
binary_image = image > 0.5

# Define a structuring element
selem = disk(5)

# Perform opening
opened_image = opening(binary_image, selem)

# Perform closing
closed_image = closing(binary_image, selem)

# Plot histograms for the original, opened, and closed images
plt.figure(figsize=(15, 5))
```

```python
# Histogram for the original image
plt.subplot(1, 3, 1)
plt.title('Histogram of Original Image')
plt.hist(binary_image.astype(int).ravel(), bins=2, color='gray', alpha=0.7)
plt.xlim(-0.5, 1.5)  # Limiting x-axis for binary image (0 to 1)
plt.xticks([0, 1])   # Show ticks for binary values
plt.xlabel('Pixel Intensity')
plt.ylabel('Frequency')

# Histogram for the opened image
plt.subplot(1, 3, 2)
plt.title('Histogram of Opened Image')
plt.hist(opened_image.astype(int).ravel(), bins=2, color='gray', alpha=0.7)
plt.xlim(-0.5, 1.5)  # Limiting x-axis for binary image (0 to 1)
plt.xticks([0, 1])   # Show ticks for binary values
plt.xlabel('Pixel Intensity')
plt.ylabel('Frequency')

# Histogram for the closed image
plt.subplot(1, 3, 3)
plt.title('Histogram of Closed Image')
plt.hist(closed_image.astype(int).ravel(), bins=2, color='gray', alpha=0.7)
plt.xlim(-0.5, 1.5)  # Limiting x-axis for binary image (0 to 1)
plt.xticks([0, 1])   # Show ticks for binary values
plt.xlabel('Pixel Intensity')
plt.ylabel('Frequency')

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

Output:

Performing Opening and closing operation:

- **Evaluating histogram:**



Result:

Opening operation:- Reduce high intensity and Increase low intensity.

Closing operation:- Reduce low intensity and Increase high intensity.

7. Edge detection

Objective:

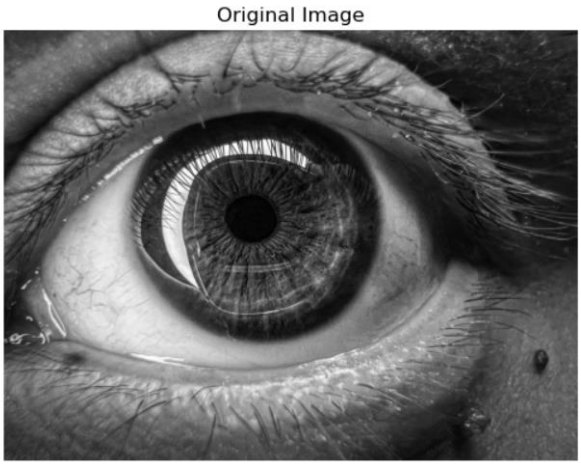
The **Sobel edge detection** method is to identify the edges within an image.

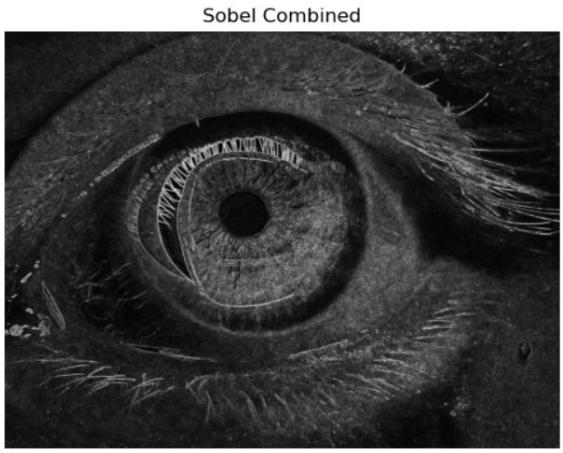
Input:

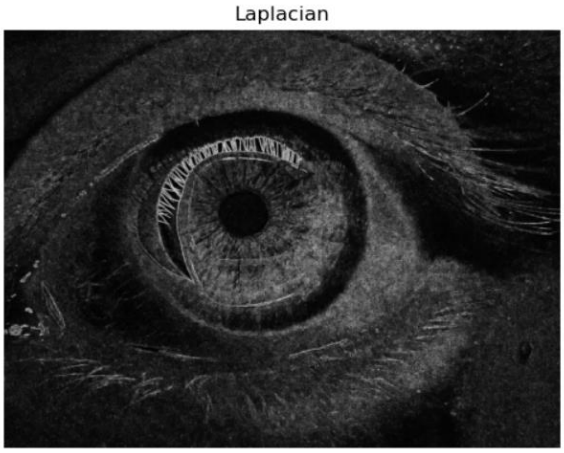
```
# Load the image
image_path = r"C:\Users\roari\Downloads\sobel 2.jpeg"
image = cv2.imread(image_path)

# Check if the image was loaded successfully
if image is None:
    print("Error: Image not found. Please check the file path.")
else:
    # Step 1: Convert the image to grayscale
    gray_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

    # Step 2: Apply Sobel edge detection
    sobel_x = cv2.Sobel(gray_image, cv2.CV_64F, 1, 0, ksize=3)  # Sobel in X direction
    sobel_y = cv2.Sobel(gray_image, cv2.CV_64F, 0, 1, ksize=3)  # Sobel in Y direction
    sobel_combined = cv2.magnitude(sobel_x, sobel_y)  # Combine the X and Y gradients

    # Step 3: Apply Laplacian edge detection
    laplacian = cv2.Laplacian(gray_image, cv2.CV_64F, ksize=3)

    # Convert the results to a displayable format
    sobel_x = cv2.convertScaleAbs(sobel_x)
    sobel_y = cv2.convertScaleAbs(sobel_y)
    sobel_combined = cv2.convertScaleAbs(sobel_combined)
    laplacian = cv2.convertScaleAbs(laplacian)

    # Step 4: Plot the results
    plt.figure(figsize=(12, 8))

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

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

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

# Sobel Combined
plt.subplot(2, 2, 4)
plt.imshow(sobel_combined, cmap='gray')
plt.title('Sobel Combined')
plt.axis('off')

# Display the Sobel results
plt.tight_layout()
plt.show()

# Step 5: Plot the Laplacian result
plt.figure(figsize=(6, 6))
plt.imshow(laplacian, cmap='gray')
plt.title('Laplacian')
plt.axis('off')

# Show the Laplacian result
plt.show()
```

Output:







Result:

Key observations:

Sobel X: The Sobel X filter highlights edges that are oriented vertically. This is evident in the enhanced edges along the iris and the pupil boundaries.

Sobel Y: The Sobel Y filter highlights edges that are oriented horizontally. This is evident in the enhanced edges along the upper and lower eyelids.

Sobel Combined: The combined Sobel filter, which is the sum of the Sobel X and Sobel Y filters, highlights edges in both horizontal and vertical directions. This provides a more comprehensive representation of the edges in the image.

Inference:

The Sobel edge detection operator is effective at detecting edges in the image of the human eye.

The different orientations of the Sobel filters allow for the detection of edges in various directions.

8. LBP

Objective:

Using LBP to captures local texture information by comparing each pixel with its surrounding neighbors to different textures in the image.

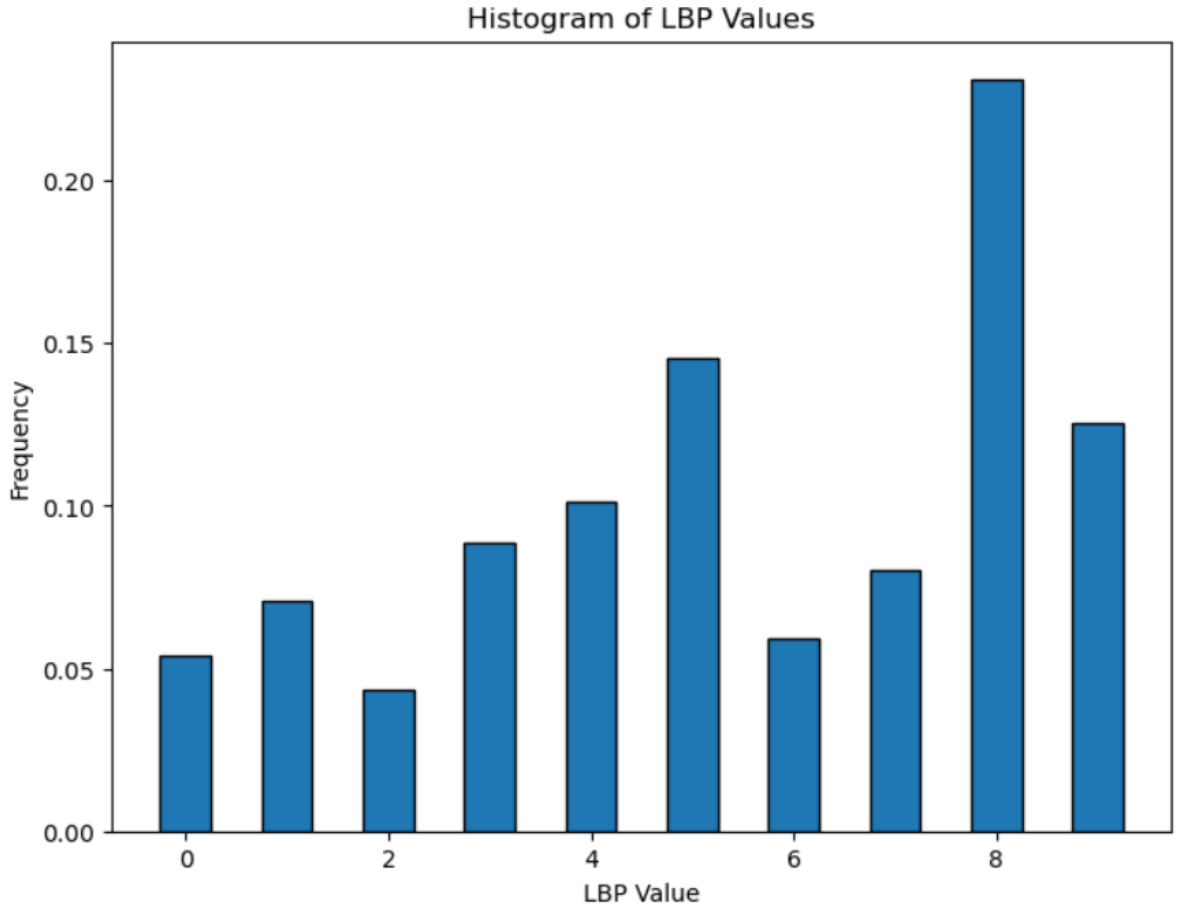
Input:

```python
import cv2
import numpy as np
import matplotlib.pyplot as plt
from skimage.feature import local_binary_pattern
from skimage import io
from skimage.color import rgb2gray

# Parameters for LBP
radius = 1  # Radius of the circle (default 1)
n_points = 8 * radius  # Number of points to consider in the LBP (8 neighbors)

# Load the image from the specified path (use raw string or double backslashes)
image_path = r"C:\Users\roari\Downloads\Bird 7.jpg"
image = io.imread(image_path)

# Convert image to grayscale
gray_image = rgb2gray(image)

# Apply Local Binary Pattern (LBP)
lbp = local_binary_pattern(gray_image, n_points, radius, method='uniform')

# Plot the original, grayscale, and LBP image
plt.figure(figsize=(18, 6))

# Display the original image
plt.subplot(1, 3, 1)
plt.imshow(image)
plt.title('Original Image')
plt.axis('off')
```

```
# Display the grayscale image
plt.subplot(1, 3, 2)
plt.imshow(gray_image, cmap='gray')
plt.title('Grayscale Image')
plt.axis('off')

# Display the LBP image
plt.subplot(1, 3, 3)
plt.imshow(lbp, cmap='gray')
plt.title('LBP Image')
plt.axis('off')

plt.show()

# Calculate the histogram of LBP values
n_bins = int(lbp.max() + 1)
hist, bins = np.histogram(lbp.ravel(), bins=n_bins, range=(0, n_bins), density=True)

# Print unique LBP patterns and their frequencies
unique_patterns, counts = np.unique(lbp, return_counts=True)
print("LBP Patterns and their Counts:")
for pattern, count in zip(unique_patterns, counts):
    print(f"Pattern: {pattern:.0f}, Count: {count}")

# Plot LBP histogram
plt.figure(figsize=(8, 6))
plt.bar(bins[:-1], hist, width=0.5, edgecolor='black')
plt.title('Histogram of LBP Values')
plt.xlabel('LBP Value')
plt.ylabel('Frequency')
plt.show()
```

Output:

```
LBP Patterns and their Counts:
Pattern: 0, Count: 4390
Pattern: 1, Count: 5732
Pattern: 2, Count: 3535
Pattern: 3, Count: 7171
Pattern: 4, Count: 8196
Pattern: 5, Count: 11764
Pattern: 6, Count: 4804
Pattern: 7, Count: 6484
Pattern: 8, Count: 18723
Pattern: 9, Count: 10149
```



Result:

LBP Image: The LBP image is a texture-based representation of the original image, where each pixel is assigned a unique LBP code based on the intensity differences between the pixel and its neighbors.

LBP Patterns and their Counts:

The table have provided lists the 10 possible LBP patterns and their corresponding counts in the image. These counts represent the number of pixels in the image that match each particular LBP pattern.

Inference:

LBP is a powerful technique for extracting texture information from images.
The LBP patterns and their counts provide valuable insights into the texture characteristics of the image.
The dominant LBP patterns in the image suggest that the bird's feathers have a relatively uniform texture with some variations in orientation and intensity.

9. Huffman

Objective:

The objective of applying Huffman encoding is to compress data efficiently by reducing the number of bits needed to represent frequently occurring symbols while using longer codes for less frequent symbols.

Input:

```python
import numpy as np
import matplotlib.pyplot as plt
from skimage import io, img_as_ubyte
from collections import Counter
import heapq

# Step 1: Load the image and convert it to grayscale
image_path = r"C:\Users\roari\Downloads\Bird 9.jpg"
image = io.imread(image_path, as_gray=True)
image = img_as_ubyte(image)  # Convert image to 8-bit unsigned integers

# Flatten the image (convert it to a 1D array of pixel values)
pixels = image.flatten()

# Step 2: Calculate the frequency of each pixel value
frequencies = Counter(pixels)

# Step 3: Build the Huffman Tree
class HuffmanNode:
    def __init__(self, frequency, symbol=None, left=None, right=None):
        self.symbol = symbol
        self.frequency = frequency
        self.left = left
        self.right = right

    def __lt__(self, other):
        return self.frequency < other.frequency
```

```
def build_huffman_tree(frequencies):
    heap = [HuffmanNode(freq, sym) for sym, freq in frequencies.items()]
    heapq.heapify(heap)

    while len(heap) > 1:
        left = heapq.heappop(heap)
        right = heapq.heappop(heap)
        merged = HuffmanNode(left.frequency + right.frequency, left=left, right=right)
        heapq.heappush(heap, merged)

    return heap[0]

# Step 4: Generate Huffman codes
def generate_huffman_codes(node, code='', huffman_code={}):
    if node is None:
        return
    if node.symbol is not None:  # It's a leaf node
        huffman_code[node.symbol] = code
    generate_huffman_codes(node.left, code + '0', huffman_code)
    generate_huffman_codes(node.right, code + '1', huffman_code)
    return huffman_code

# Build Huffman tree and generate codes
huffman_tree = build_huffman_tree(frequencies)
huffman_codes = generate_huffman_codes(huffman_tree)

# Step 5: Encode the image using Huffman codes
encoded_image = ''.join([huffman_codes[pixel] for pixel in pixels])
```

```
# Step 6: Decode the binary string back to the original image
def decode_huffman(encoded_data, huffman_tree, total_pixels):
    decoded_pixels = []
    node = huffman_tree
    for bit in encoded_data:
        node = node.left if bit == '0' else node.right
        if node.symbol is not None:
            decoded_pixels.append(node.symbol)
            node = huffman_tree
        if len(decoded_pixels) == total_pixels:
            break
    return np.array(decoded_pixels)

# Decode the image
decoded_pixels = decode_huffman(encoded_image, huffman_tree, len(pixels))
decoded_image = np.reshape(decoded_pixels, image.shape)
```

```python
# Step 7: Display original and decoded 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('Decoded Image')
plt.imshow(decoded_image, cmap='gray')
plt.axis('off')

plt.tight_layout()
plt.show()

# Check if the images are identical
if np.array_equal(image, decoded_image):
    print("The original and decoded images are identical!")
else:
    print("The images are not identical.")
```

Output:


```
The original and decoded images are identical!
```

Result:

Original Image: This is the original image in its uncompressed form. In the context of Huffman coding, it represents the data that needs to be compressed.
Decoded Image: This is the image that is reconstructed after applying Huffman decoding to the compressed data. In this case, the decoded image appears identical to the original image, indicating that Huffman coding has successfully compressed and decompressed the data without any loss of information.

10. Run length encoding

Objective:

The objective of performing Run-Length Encoding (RLE) is to compress data by reducing the size of sequences of repeated symbols or values.

Input:

```
import numpy as np
import matplotlib.pyplot as plt
from skimage import io, img_as_ubyte

# Step 1: Load the image and convert it to grayscale
image_path = r"C:\Users\roari\Downloads\Bird 11.jpg"
image = io.imread(image_path, as_gray=True)
image = img_as_ubyte(image)  # Convert image to 8-bit unsigned integers

# Step 2: Flatten the image into a 1D array of pixel values
pixels = image.flatten()

# Step 3: Run Length Encoding
def run_length_encoding(pixels):
    encoded = []
    count = 1
    previous_pixel = pixels[0]

    for pixel in pixels[1:]:
        if pixel == previous_pixel:
            count += 1
        else:
            encoded.append((previous_pixel, count))
            previous_pixel = pixel
            count = 1

    # Add the last run
    encoded.append((previous_pixel, count))

    return encoded
```

```python
# Step 4: Decode the Run Length Encoded data
def run_length_decoding(encoded):
    decoded = []
    for pixel, count in encoded:
        decoded.extend([pixel] * count)
    return np.array(decoded)

# Encode the image using RLE
encoded_image = run_length_encoding(pixels)

# Decode the RLE back to the original image
decoded_pixels = run_length_decoding(encoded_image)
decoded_image = np.reshape(decoded_pixels, image.shape)

# Step 5: Display original and decoded 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('Decoded Image (RLE)')
plt.imshow(decoded_image, cmap='gray')
plt.axis('off')

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

Output:

Result:

RLE is a simple and effective compression technique for images that contain long runs of pixels with the same intensity value. In many cases, RLE can achieve significant compression ratios without noticeable loss of quality but yes there is certain change in pixel values as some pixels which are lesser than threshold taken inside the long run while encoding which actually reduces the bits required but in decoding they were not able to set back to position where they were before had actually.