The report for assignment1 of Machine Vision

Wenbin Lin

The assignment is divided into two parts. The first part is about image loading and basic processing. The second part is image analysis using peppers.png. In my assignment, I used opencv-python and matplotlib in python as the tools. In the first part the picture of Lena Forsen was used.

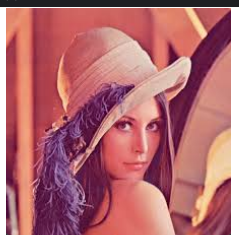
PART A: Image Loading and Basic Processing

Q1: Image Loading and Conversion

1. Load an image using Python or MATLAB. Display the image and provide the code.

In this issue, cv2.imread() and cv2.imshow() function were used to read the image and show the image. The code and the picture displayed were the following:

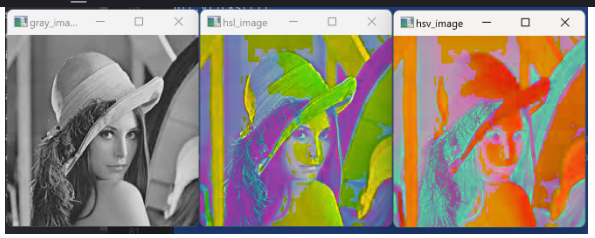
```
import cv2

# Load the image
image = cv2.imread('girl.jpg')
# Display the image
cv2.imshow('beautiful girl', image)
# Wait for a key press and close the window
cv2.waitKey(0)
cv2.destroyAllWindows()
```



2. Convert the image to grayscale and HSL/HSV color spaces. Display both converted images.

In this issue, the function cv2.cvtColor() was used for the conversion. Two parameters were passed to the function, one is the image data and the other is conversion type, such as cv2.COLOR_BGR2GRAY, cv2.COLOR_BGR2HLS, cv2.COLOR_BGR2HSV and so on. And the cv2.imshow() function was used for display the pictures. The code and the picture displayed were the following:

```
# Convert to grayscale
gray_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
# Convert to HSL color space
hsl_image = cv2.cvtColor(image, cv2.COLOR_BGR2HLS)
# Convert to HSV color space
hsv_image = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)
cv2.imshow("gray_image",gray_image)
cv2.imshow("hsl_image",hsl_image)
cv2.imshow("hsv_image",hsv_image)
```

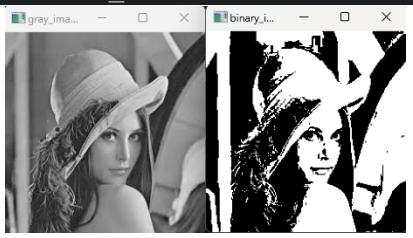


3. Binarize the grayscale image using a threshold based on intensity or hue. Display the binarized image and explain your threshold selection.

To binarize the gray image, cv2.threshold() function was used to the conversion. And

for the threshold value selection, a threshold of 127 is a good starting point for evenly lit images with clear foreground-background separation. For darker images, a lower threshold (e.g., 100) might be needed. For brighter images, a higher threshold (e.g., 150) might be more effective. In this issue, a threshold value 127 based on intensity was used to binarize the grayscale image for the image was on normal lighting condition. The intensity value larger than 127 would be translated into 1, other will be 0. The code and the result was show in the following:

```
# Apply binary thresholding
threshold_value = 127
# Change this value based on brightness
_, binary_image = cv2.threshold(gray_image, threshold_value, 255,
cv2.THRESH_BINARY)
cv2.imshow("gray_image",gray_image)
cv2.imshow("binary_image",binary_image)
```



Q2: Geometric Transformations

1. Perform a translation on the image using a translation matrix with tx = 50 and ty=30. Display the translated image.

In this issue we use the cv2.warpAffine() function to do the translation and use the matplotlib to display the two images. In this case, to display the two images, we use matplotlib and we should change the bgr type data of imread to rbg type of date. The code and the two pictures were the following:

```
# Load the image
image = cv2.imread("girl.jpg")

# Convert from BGR to RGB for correct color display in Matplotlib
image_rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

# Get image dimensions
height, width = image.shape[:2]

# Define the translation matrix (tx = 50, ty = 30)
tx, ty = 50, 30
translation_matrix = np.float32([[1, 0, tx], [0, 1, ty]])

# Apply the translation using warpAffine
translated_image = cv2.warpAffine(image_rgb, translation_matrix,
(width + tx, height + ty))

# Display the original and translated images
fig, axes = plt.subplots(1, 2, figsize=(10, 5))

# Show the original image
axes[0].imshow(image_rgb)
axes[0].set_title("Original Image")
axes[0].axis("off")


# Show the translated image
axes[1].imshow(translated_image)
axes[1].set_title("Translated Image (Right 50px, Down 30px)")
axes[1].axis("off")
```

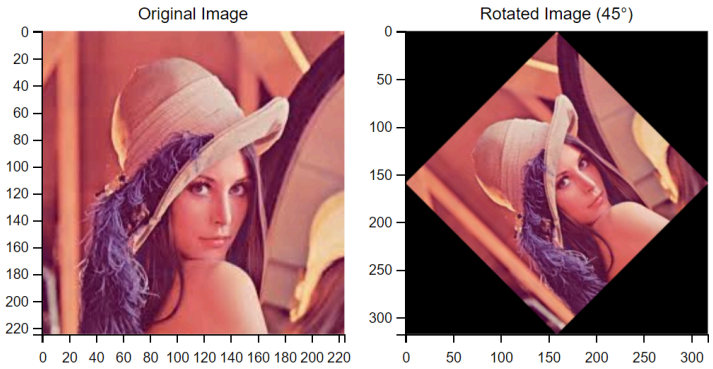
```
# Display both images
plt.show(block=False)
# waiting for the use to press the enter key to close the plot window
input("close the plot window if in pycharm and Press Enter to continue
... ")
# close the plot window
plt.close()
```

2. Define a rotation matrix for rotating the image by an angle of 45 degrees. Rotate the image and display the result.

In this issue, the center of the image should be calculated first. cv2.getRotationMatrix2D was used to calculate the rotation_matrix. Then the new rotated image's boundary was calculated. The function cv2.warpAffine() was used for the transformation. Then the two images were displayed by the matplotlib library. The code and the result were the following:

```
# Load the image
image = cv2.imread("girl.jpg")

# Convert from BGR to RGB (for correct color display in Matplotlib)
image_rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

# Get image dimensions
height, width = image.shape[:2]

# Define the center of rotation (center of the image)
center = (width // 2, height // 2)

# Define the rotation matrix (angle = 45 degrees, scale = 1)
angle = 45
scale = 1.0
rotation_matrix = cv2.getRotationMatrix2D(center, angle, scale)

# Compute the new bounding dimensions of the rotated image
cos_val = abs(rotation_matrix[0, 0])
sin_val = abs(rotation_matrix[0, 1])
new_width = int((height * sin_val) + (width * cos_val))
new_height = int((height * cos_val) + (width * sin_val))

# Adjust the rotation matrix to take into account the translation
rotation_matrix[0, 2] += (new_width / 2) - center[0]
rotation_matrix[1, 2] += (new_height / 2) - center[1]

# Apply the rotation using warpAffine
rotated_image = cv2.warpAffine(image_rgb, rotation_matrix, (new_width,
new_height))

# Display the original and rotated images side by side
```

```
fig, axes = plt.subplots(1, 2, figsize=(10, 5))

# Show the original image
axes[0].imshow(image_rgb)
axes[0].set_title("Original Image")
axes[0].axis("off")

# Show the rotated image
axes[1].imshow(rotated_image)
axes[1].set_title("Rotated Image (45°)")
axes[1].axis("off")

# Display both images
plt.show(block=False)
# waiting for the use to press the enter key to close the plot window
input("close the plot window if in pycharm and Press Enter to continue
... ")
# close the plot window
plt.close()
```



Q3: Smoothing Filters and Edge Detection

1. Apply a mean filter to the input image. Change the kernel size and observe its effect on the image.

A mean filter is a type of low-pass filter that reduces image noise and blurs the image. The filter replaces each pixel with the mean (average) of its neighboring pixels within a given kernel size. In this issue we apply the mean filter with different kernel sizes using cv2.blur(). The result was displayed in the following picture:

```
# Load the image
image = cv2.imread("girl.jpg")

# Convert from BGR to RGB for correct color display in Matplotlib
image_rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

# Apply mean filters with different kernel sizes
kernel_sizes = [(3, 3), (7, 7), (15, 15)]
filtered_images = [cv2.blur(image_rgb, ksize) for ksize in
kernel_sizes]

# Display original and filtered images
fig, axes = plt.subplots(1, 4, figsize=(30, 10))

# Show original image
axes[0].imshow(image_rgb)
axes[0].set_title("Original Image")
axes[0].axis("off")
axes[0].set_xticks([])
axes[0].set_yticks([])
```

```
# Show filtered images
for i, (ksize, img) in enumerate(zip(kernel_sizes, filtered_images)):
    axes[i + 1].imshow(img)
    axes[i + 1].set_title(f"Mean Filter {ksize}")
    axes[i + 1].axis("off")
    axes[i + 1].set_xticks([])
    axes[i + 1].set_yticks([])


# Display all images
plt.show(block=False)
# waiting for the use to press the enter key to close the plot window
input("close the plot window if in pycharm and Press Enter to continue
... ")
# close the plot window
plt.close()
```

From the above result, we can find larger kernels result in more blurring, reducing noise but also removing details. Meanwhile, small kernels preserve more sharpness while applying minimal smoothing.

2. Apply a Gaussian filter to the input image. Experiment with different standard deviations (σ) and describe how changing σ influences the result.

A Gaussian filter is a low-pass filter that reduces noise while preserving edges better than a mean filter. The degree of smoothing is controlled by σ (standard deviation), which determines the spread of the Gaussian kernel. In the issue, we use the kernel(9,9) for the Gaussian filter. The function cv2.GaussianBlur() was used to smooth the image. And 1, 3, 5, 10 were chosen as σ. The code and the result was show in the following figure:

```
    # Load the image
    image = cv2.imread("girl.jpg")

    # Convert from BGR to RGB for correct color display in Matplotlib
    image_rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

    # Apply Gaussian blur with different standard deviations (σ)
    sigma_values = [1, 3, 5, 10]
    blurred_images = [cv2.GaussianBlur(image_rgb, (9, 9), sigma) for
sigma in sigma_values]

    # Display original and blurred images
    fig, axes = plt.subplots(1, 5, figsize=(30, 10))

    # Show original image
    axes[0].imshow(image_rgb)
    axes[0].set_title("Original Image")
    axes[0].axis("off")
    axes[0].set_xticks([])
    axes[0].set_yticks([])

    # Show blurred images for different sigma values
    for i, (sigma, img) in enumerate(zip(sigma_values, blurred_images)):
        axes[i + 1].imshow(img)
        axes[i + 1].set_title(f" σ={sigma}")
```

```
        axes[i + 1].axis("off")
        axes[i + 1].set_xticks([])
        axes[i + 1].set_yticks([])

    # Display all images
    plt.show(block=False)
    # waiting for the use to press the enter key to close the plot
window
    input("close the plot window if in pycharm and Press Enter to
continue ... ")
    # close the plot window
    plt.close()
```

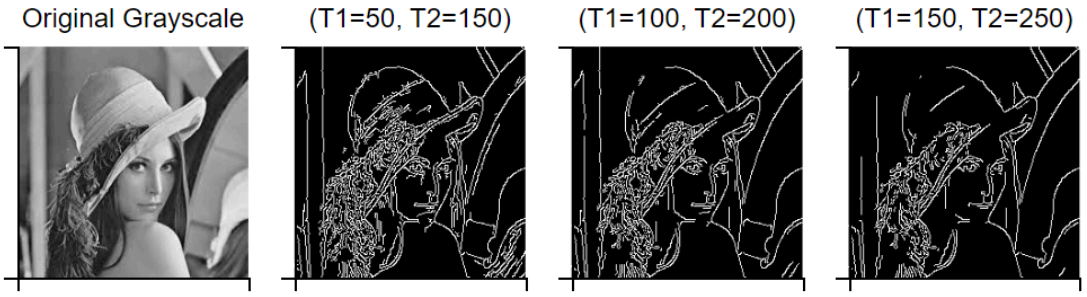
From the above figure, we can find that similar as the mean filter, Smaller σ can retain more details but removes less noise, while Larger σ can increase blurring but may remove important fine details. Compared with the mean filter, the Gaussian filter works better at preserving edges because weights decrease smoothly, and is more effective for Gaussian noise.

3. Apply a Canny edge detector to the grayscale image. Display the result and analyze the impact of different threshold values on the detected edges.

Canny Edge Detection is a multi-step process used to detect edges in an image. It relies on gradient-based edge detection with two key thresholds that determine which edges are preserved. For detection, the gray picture should be used. The function cv2.Canny() was used for the detection in the code, and three parameters were passed to the function, the first one is the gray image data; the other two is the thresholds. The first threshold (threshold1) is pixels with a gradient magnitude below this value are suppressed (not considered edges). The second threshold(threshold2) is that pixels with a gradient magnitude above this value are definitely considered edges. Strong edges (above threshold2) are preserved. Weak edges (between threshold1 and threshold2) are kept only if they are connected to strong edges.

```
    # Load the image
    image = cv2.imread("girl.jpg")

    # Convert to grayscale
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

    # Apply Canny edge detection with different threshold values
    thresholds = [(50, 150), (100, 200), (150, 250)]
    edges_list = [cv2.Canny(gray, t1, t2) for t1, t2 in thresholds]

    # Display original and edge-detected images
    fig, axes = plt.subplots(1, 4, figsize=(30, 10))

    # Show original grayscale image
    axes[0].imshow(gray, cmap="gray")
    axes[0].set_title("Original Grayscale")
    axes[0].axis("off")
    axes[0].set_xticks([])
    axes[0].set_yticks([])
```

```
  # Show Canny edge-detected images with different thresholds
  for i, (t, edges) in enumerate(zip(thresholds, edges_list)):
     axes[i + 1].imshow(edges, cmap="gray")
     axes[i + 1].set_title(f"(T1={t[0]}, T2={t[1]})")
     axes[i + 1].axis("off")
     axes[i + 1].set_xticks([])
     axes[i + 1].set_yticks([])

  # Display all images
  plt.show(block=False)
  # waiting for the use to press the enter key to close the plot window
  input("close the plot window if in pycharm and Press Enter to
continue ... ")
  # close the plot window
  plt.close()
```



From the above figure, we can find the threshold impact:

Low Thresholds (50,100) can detect many edges, but also introduces false edges (noise).

Medium Thresholds (100,200) can produce a balanced edge map, detecting most significant edges while reducing noise.

High Thresholds (150,250) can detects only strong edges, losing fine details but effectively removing noise.

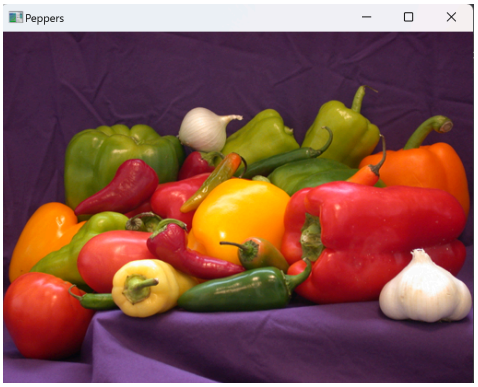
Part B: Image Analysis Using Peppers.png

Q4: Intensity Range Adjustment and Histogram Equalization

1. Load and display the color image 'peppers.png' (build-in in MATLAB or download it from online source for Python users) and examine and report the size of the image (width, height, and number of channels)

This issue is simple, and similar to the Q1 issue1, we can use the cv2.imread() and cv2.imshow() function to read and show the image. And width, height, and number of channels can be obtained from the property of image by the shape of image data. The width, height and the number of channels are 512, 384 and 3 respectively. The code and the file pepper.png is showed in the following:

```
# Load the image (make sure 'peppers.png' is in the same directory or
provide the correct path)
image = cv2.imread("peppers.png")
# display the image
cv2.imshow("Peppers", image)
cv2.waitKey(0)
cv2.destroyAllWindows()
# Get image dimensions: height, width, number of channels
height, width, channels = image.shape
# Report the size
print(f"Width: {width}, Height: {height}, Number of channels:
{channels}")
```



2. Convert the color image to grayscale and display the grayscale image in its full intensity range ([0,255]).

In this issue, the function cv2.imread("peppers.png") was used for loading the color image; the function cv2.cvtColor(image, cv2.COLOR_BGR2GRAY) was used to convert the image to grayscale. The pictures were displayed by the matplotlib library. The function imshow(gray_image, cmap="gray", vmin=0, vmax=255) from matplotlib was used to display the grayscale image in full intensity range [0,255]. The code and the two pictures of original image and gray(with full intensity range) are show in the following figure:

```
# Load the color image
image = cv2.imread("peppers.png")
# Convert BGR to RGB for correct display
image_rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

# Convert to grayscale
gray_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

# Display original and grayscale images
fig, axes = plt.subplots(1, 2, figsize=(15, 8))

# Original color image
axes[0].imshow(image_rgb)
axes[0].set_title("Original Color Image")
axes[0].axis("off")
axes[0].set_xticks([])
axes[0].set_yticks([])

# Grayscale image
axes[1].imshow(gray_image, cmap="gray", vmin=0, vmax=255)  # Ensure
full intensity range
axes[1].set_title("Grayscale Image (0-255)")
axes[1].axis("off")
axes[1].set_xticks([])
axes[1].set_yticks([])

plt.tight_layout()
# Show the images
plt.show(block=False)
# waiting for the use to press the enter key to close the plot window
input("close the plot window if in pycharm and Press Enter to continue
... ")
# close the plot window
plt.close()
```

3. Reduce the intensity range of the grayscale image to a lower range ([0,N]) for values of N ranging from 255 to 8. Display the resulting images.

I took the grayscale version of the peppers.png image and reduced its intensity range from [0,255] to a lower range [0, N], where N varies from 255 to 8. This simulates reducing the number of gray levels, effectively reducing image contrast and introducing a posterization effect. Scales pixel values using:

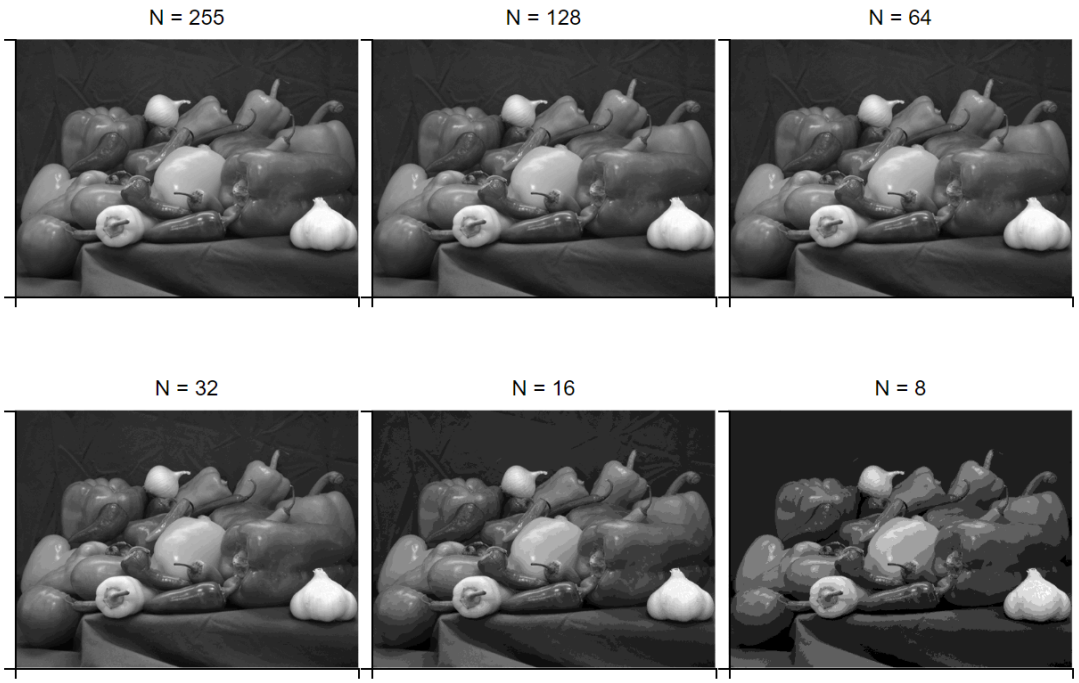
$$I' = \frac{I \times N}{255}$$

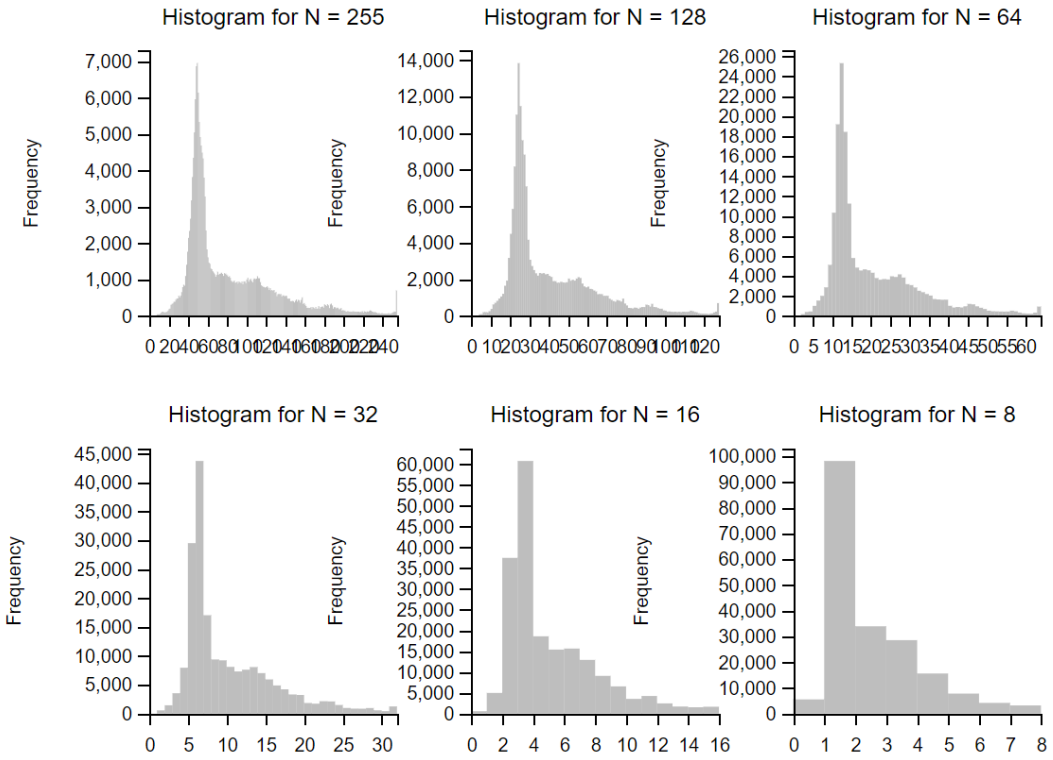
when I is the original pixel intensity

We reduce the intensity range using numpy package scaled_image = (image.astype(np.float32) * N / 255).astype(np.uint8). This scales the grayscale values down to [0, N]. Then we used the function imshow(rescaled_image, cmap='gray', vmin=0, vmax=N) in matplotlib to display the images. To compare the results, the histogram plot was displayed too. The code and the results were displayed in the following:

```
    # Load the grayscale image
    image_path = "peppers.png"
    image = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)

    # Define different intensity ranges
    N_values = [255, 128, 64, 32, 16, 8]

    # Create subplots
    fig, axes = plt.subplots(2, 3, figsize=(12, 8))
    axes = axes.ravel()

    # Process and display images
    for i, N in enumerate(N_values):
        scaled_image = (image.astype(np.float32) * N /
255).astype(np.uint8)
        axes[i].imshow(scaled_image, cmap='gray', vmin=0, vmax=N)
        axes[i].set_title(f"N = {N}")
        axes[i].axis('off')
        axes[i].set_xticks([])
        axes[i].set_yticks([])

    plt.tight_layout()
    # Show the images
    plt.show(block=False)
    # waiting for the use to press the enter key to close the plot
window
    input("close the plot window if in pycharm and Press Enter to
continue ... ")
    # close the plot window
    plt.close()

    # Create subplots
```

```
    fig, axes = plt.subplots(2, 3, figsize=(20, 15),squeeze=False)
    fig.subplots_adjust(hspace=0.5, wspace=0.3)
    axes = axes.ravel()

    # Process and display images
    for i, N in enumerate(N_values):
        scaled_image = (image.astype(np.float32) * N /
255).astype(np.uint8)
        #  plot the histogram
        axes[i].hist(scaled_image.ravel(), bins=N, range=[0, N],
color='gray', alpha=0.7)
        axes[i].set_xlim([0, N])
        axes[i].set_title(f"Histogram for N = {N}", pad=10)
        axes[i].set_ylabel("Frequency",labelpad=10)
    # Show the images
    plt.show(block=False)
    # waiting for the use to press the enter key to close the plot
window
    input("close the plot window if in pycharm and Press Enter to
continue ... ")
    # close the plot window
    plt.close()
```





Report: Identifying Noticeable Distortions in the Image with Reduced Intensity Levels.

From the above pictures, we can find the effect on the image quality. For N = 255, the image remains unchanged since all intensity levels are preserved; for N = 128, 64, 32, etc, image details start to reduce because fewer intensity values are used; For N = 8, the image becomes highly posterized, losing smooth gradients. Meanwhile from the histogram behavior, for N = 255, the histogram is widely spread across the range [0, 255]; for lower N values, the histogram compresses into fewer bars, meaning pixel intensities are forced into a smaller set of values, leading to a quantization effect. for N = 8,the histogram shows only 8 spikes, representing only 8 available intensity levels.

So in conclusion, noticeable distortions begin at N=16 with visible intensity banding. Strong distortions appear at N=8 making the image look unnatural. The lower the value of N, the more the image loses fine details and smooth gradients, making it appear posterized.

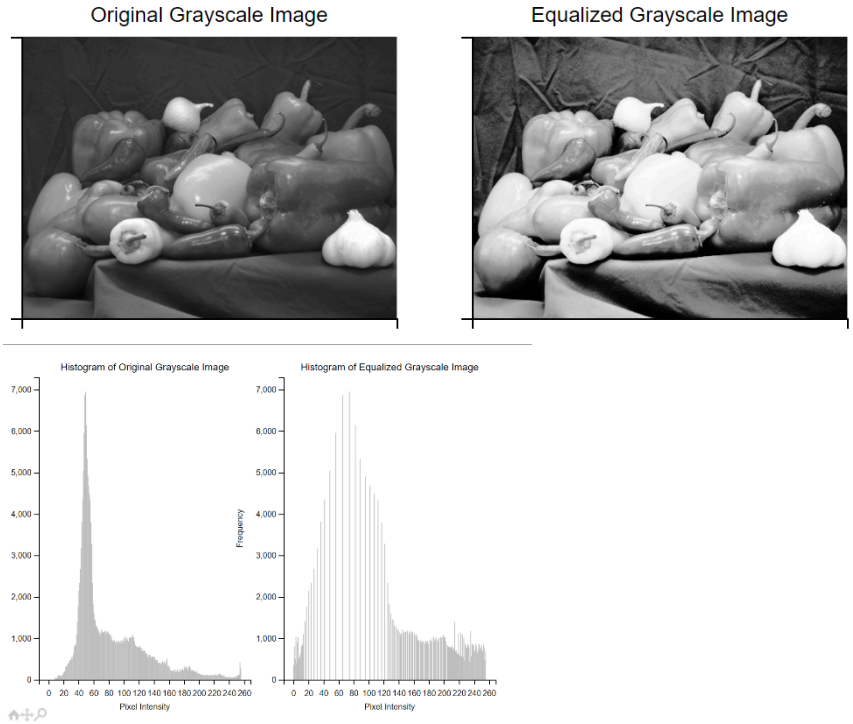
2. Apply histogram equalization to the grayscale image. Display the result and compare it to the original grayscale image. Discuss the differences in brightness and contrast.

Histogram equalization was applied to a grayscale image to enhance contrast. We will then compare the original and equalized images to observe changes in brightness and contrast. First the color image was converted to grayscale. Then cv2.equalizeHist (gray_image) was used to apply histogram equalization to the gray image. After that, the two images(grayscale image and equalized grayscale image) were displayed with matplotlib. The code and the result was displayed in the following figure. The histogram

plot of original grayscale and equalized grayscale images was displayed in the following figure too.

```
    # Load the image and convert to grayscale
    image = cv2.imread("peppers.png")
    gray_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

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

    # Create a figure with two subplots to compare original and
equalized images
    fig, axes = plt.subplots(1, 2, figsize=(15, 5))

    # Display the original grayscale image
    axes[0].imshow(gray_image, cmap='gray', vmin=0, vmax=255)
    axes[0].set_title("Original Grayscale Image")
    axes[0].axis('off')
    axes[0].set_xticks([])
    axes[0].set_yticks([])

    # Display the equalized grayscale image
    axes[1].imshow(equalized_image, cmap='gray', vmin=0, vmax=255)
    axes[1].set_title("Equalized Grayscale Image")
    axes[1].axis('off')
    axes[1].set_xticks([])
    axes[1].set_yticks([])



    # Show the images
    plt.show(block=False)
    # waiting for the use to press the enter key to close the plot
window
    input("close the plot window if in pycharm and Press Enter to
continue ... ")
    # close the plot window
    plt.close()


    # Optionally, print intensity range to compare
    print(f"Original Grayscale image intensity range:
[{gray_image.min()}, {gray_image.max()}]")
    print(f"Equalized Grayscale image intensity range:
[{equalized_image.min()}, {equalized_image.max()}]")

    fig, axes = plt.subplots(1, 2, figsize=(12, 5))

    # Histogram of original grayscale image
    axes[0].hist(image.ravel(), bins=256, range=[0, 256], color='gray',
alpha=0.7)
    axes[0].set_title("Histogram of Original Grayscale Image")
    axes[0].set_xlabel("Pixel Intensity")
    axes[0].set_ylabel("Frequency")

    # Histogram of equalized grayscale image
    axes[1].hist(equalized_image.ravel(), bins=256, range=[0, 256],
color='gray', alpha=0.7)
    axes[1].set_title("Histogram of Equalized Grayscale Image")
    axes[1].set_xlabel("Pixel Intensity")
    axes[1].set_ylabel("Frequency")
```

```
    # Show the images
    plt.show(block=False)
    # waiting for the use to press the enter key to close the plot
window
    input("close the plot window if in pycharm and Press Enter to
continue ... ")
    # close the plot window
    plt.close()
```



Let's analyze the above pictures. For the original grayscale image, some areas appear dark, while others are bright, leading to low contrast. And the histogram is not evenly spread; pixel values may cluster in a specific range. While for the equalized grayscale image, the contrast is enhanced, making the image appear clearer. And the contrast is enhanced, making the image appear clearer. In conclusion, histogram equalization enhances contrast by redistributing intensity values, it works well for low-contrast images, making details more visible.