**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:

# Load the image  
image = cv2.imread('girl.jpg')  
# Display the image  
cv2.imshow('beautiful girl', image)

女人戴着帽子

描述已自动生成

**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)

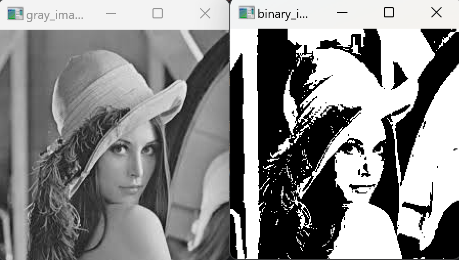
电脑合成的女孩

中度可信度描述已自动生成

**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

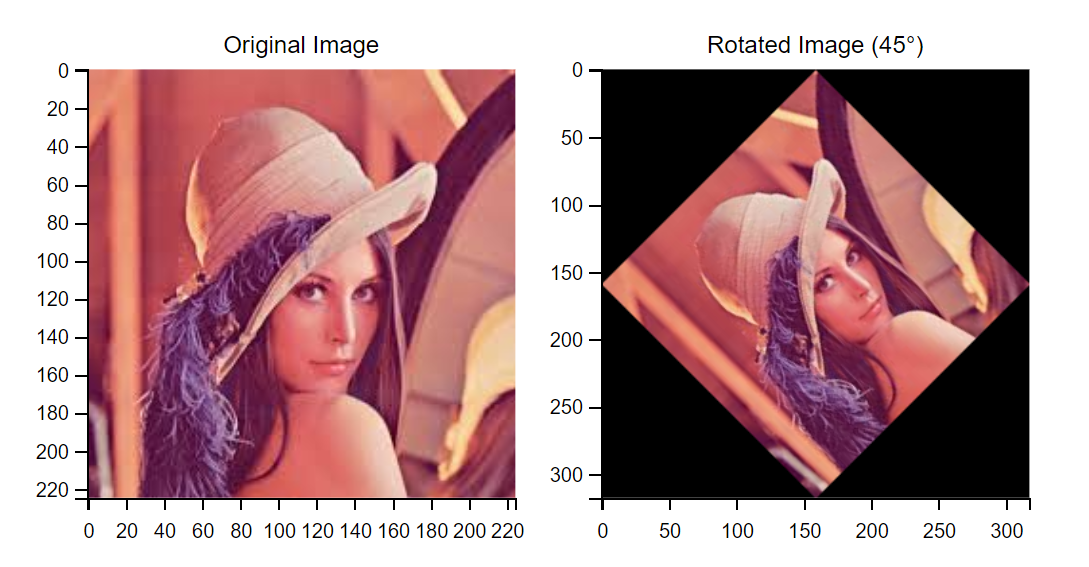
image\_rgb = cv2.cvtColor(image, cv2.COLOR\_BGR2RGB)  
# Get image dimensions  
height, width = image.shape[:2]  
# Define the translation matrix (tx = 50, ty = 30)  
tx, ty = 50, 30  
translation\_matrix = np.float32([[1, 0, tx], [0, 1, ty]])  
# Apply the translation using warpAffine  
translated\_image = 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)

**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. And then Adjust the rotation matrix to take into account the translation. After that the function cv2.warpAffine() was used for the transformation. Then the two images were displayed by the matplotlib. The code and the pictures were shown in 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)



**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:

图片包含 图形用户界面

描述已自动生成

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.**

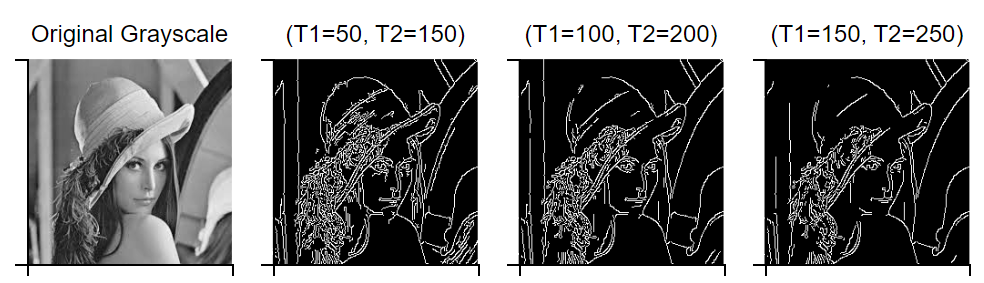
A Gaussian filter is a low-pass filter that reduces noise while preserving edges better than a mean filter. The degree of smoothing is controlled by σ (standard deviation), which determines the spread of the Gaussian kernel. In the issue, we use the kernel(9,9) for the Gaussian filter. And 1, 3, 5, 10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



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



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 produces a balanced edge map, detecting most significant edges while reducing noise.

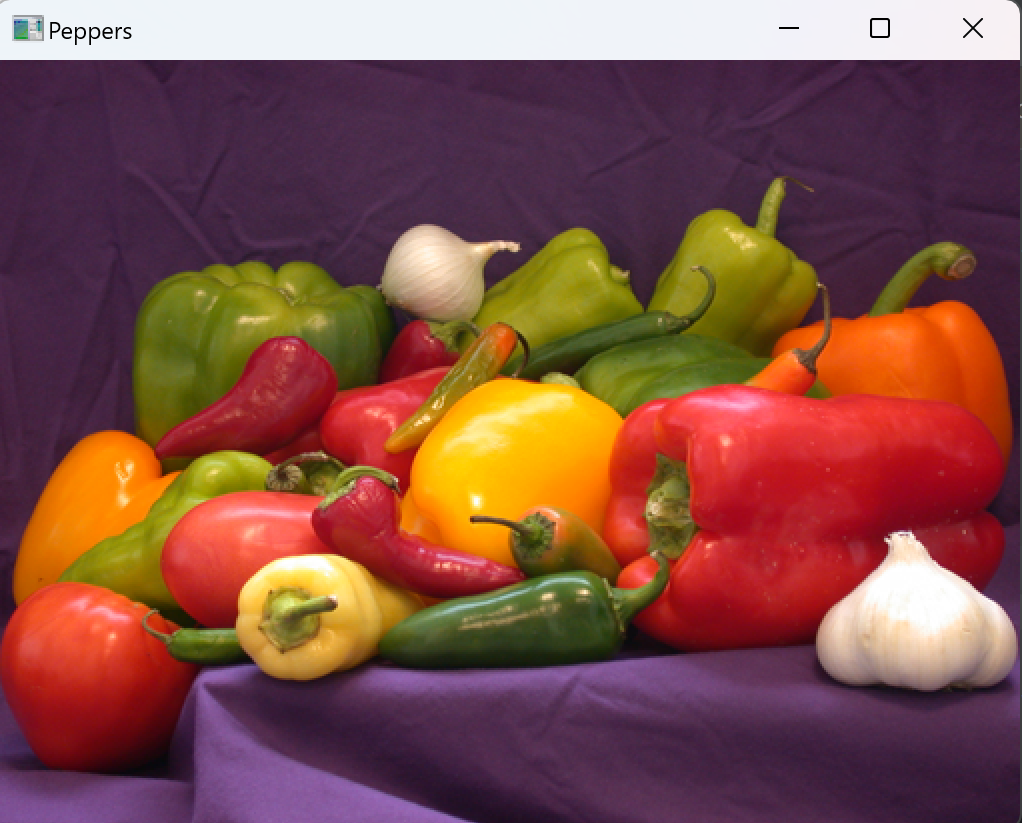
High Thresholds (150,250) can detects only strong edges, losing fine details but effectively removing noise.

**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 file pepper.png is showed in the following figure:



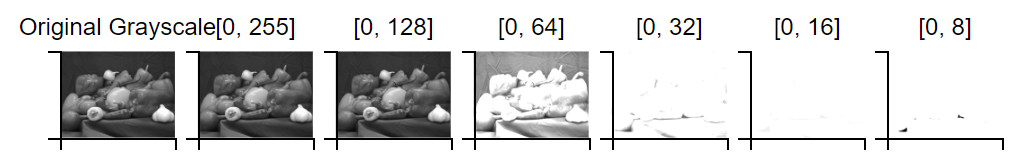
**2. Convert the color image to grayscale and display the grayscale image in its full  
intensity range ([0,255]).**

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 function imshow(gray\_image, cmap="gray", vmin=0, vmax=255) from matplotlib was used to display grayscale image in full intensity range [0,255]. The two pictures of original image and gray(with full intensity range) are show in the following figure:



**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 reduce 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. We reduce the intensity range using np.round(gray\_image / 255 \* N) \* (255 // N). This scales the grayscale values down to [0, N] and then scales them back to [0,255]. Then we used the function imshow(rescaled\_image, cmap='gray', vmin=0, vmax=N) in matplotlib to display the images. The results were showed in the following figure:



From the above pictures, we can , 

**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.

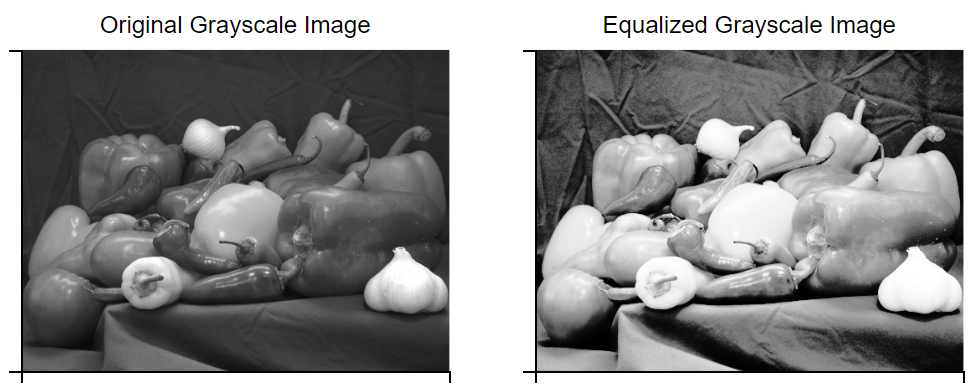
**Report: Identifying Noticeable Distortions in the Image with Reduced Intensity Levels.**

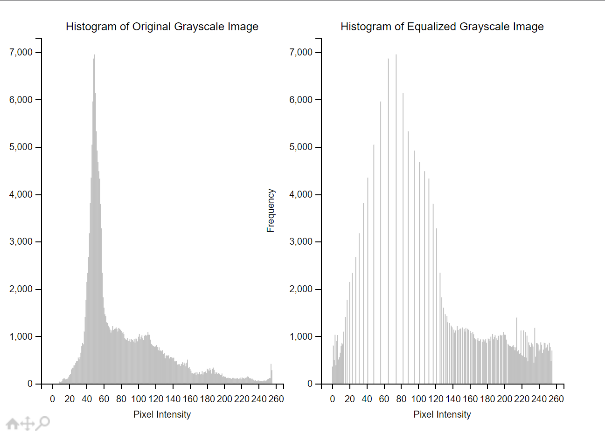
Let’s analysis the above figure. For N=255 and N=128, the image appears almost identical to the original and there is no noticeable distortions. For N=64, there is slight loss of fine details, but the image is still smooth. No strong visual artifacts were observed. For N=32, Posterization begins, and smooth gradients start breaking into visible intensity bands and edges remain relatively sharp. For N=16 and N=8, severe distortions appear, the image looks highly quantized, and gradients turn into blocky patches.

 So in conclusion, noticeable distortions begin at N=32 with visible intensity banding. Strong distortions appear at N=16, 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 image(grayscale image and equalized grayscale image) were displayed with matplotlib. 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.





Let’s analysis 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.