Digital Image Processing OpenCV Project

A Project Report Submitted in Partial Fulfillment of the Requirements for the Course: Digital Image Processing

Ameed Othman (12220692)
Yahya Musmar (12112501)

Supervisor:
Dr. Fadi Draidi

An-Najah National University
Faculty of Information Technology and Artificial Intellegence
Department of Computer Science

May 10, 2025

Contents

1 Introduction

This report documents the implementation of our image processing term project using OpenCV and Python. The project involves applying various digital image processing techniques to a real-world image, including grayscale conversion, brightness modification, histogram equalization, noise addition and removal, and image sharpening. Each step of the pipeline is explained in detail, along with the results and analysis.

2 Theoretical Background

2.1 Digital Image Representation

A digital image is represented as a two-dimensional matrix of pixels. In grayscale images, each pixel has a single value representing its intensity, typically ranging from 0 (black) to 255 (white). The dimensions of an image refer to its width and height in pixels.

2.2 Intensity Transformation

Intensity transformation involves modifying the pixel values of an image to enhance its visual appearance. The basic form of intensity transformation is:

$$s = T(r) \tag{1}$$

where r is the input pixel value and s is the output pixel value. For brightness adjustment, a simple linear transformation is used:

$$s = c \times r \tag{2}$$

where c is a constant that controls the brightness.

2.3 Histogram Equalization

Histogram equalization is a technique for adjusting image contrast by effectively spreading out the most frequent intensity values. The method utilizes the cumulative distribution function (CDF) of the image to transform the pixel values:

$$s_k = T(r_k) = (L-1)\sum_{j=0}^{k} p_r(r_j) \tag{3}$$

where L is the number of possible intensity values, and $p_r(r_j)$ is the probability of occurrence of intensity level r_j.

2.4 Noise in Digital Images

Salt-and-pepper noise is a form of noise typically seen in images, characterized by randomly occurring white and black pixels. This noise can be caused by sharp and sudden disturbances in the image signal.

2.5 Spatial Filtering

Spatial filtering operations process an image by applying a filter (or kernel) that operates on a neighborhood of pixels. Common spatial filters include:

- **Mean Filter**: Replaces each pixel with the average of its neighborhood, effective for Gaussian noise.
- **Median Filter**: Replaces each pixel with the median value in its neighborhood, particularly effective for salt-and-pepper noise.
- **Sharpening Filter**: Enhances edges and fine details by amplifying high-frequency components.

3 Implementation

3.1 Image Acquisition and Watermarking

The first step involved taking an image of a glass cup and adding watermarks with our names and student IDs. This was implemented using OpenCV's image processing functions.

```
import cv2

image = cv2.imread('my_image.jpg')
image = cv2.resize(image, (600, 400))

text1 = "Yahya Musmar - 12112501"
text2 = "Ameed Othman - 12220692"
cv2.putText(image, text1, (30, 360), cv2.FONT_HERSHEY_SIMPLEX, 0.5,
    (255, 255, 255), 2)
cv2.putText(image, text2, (30, 380), cv2.FONT_HERSHEY_SIMPLEX, 0.5,
    (255, 255, 255), 2)

cv2.imshow("Watermarked Image", image)
cv2.imwrite("watermarked.jpg", image)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Listing 1: Image Loading and Watermarking

Figure 1: Original image with watermark containing student names and IDs

3.2 Grayscale Conversion and Image Statistics

The watermarked image was converted to grayscale, and various statistics were computed to understand its properties.

```
import cv2
import numpy as np

gray = cv2.imread('watermarked.jpg', cv2.IMREAD_GRAYSCALE)
cv2.imwrite("grayscale.jpg", gray)

height, width = gray.shape
print(f"Image dimensions: {width}x{height}")
print("Color channels: 1 (Grayscale)")

print("Pixel Statistics:")
print(f"- Mean: {np.mean(gray):.2f}")
print(f"- Min: {np.min(gray)}")
print(f"- Max: {np.max(gray)}")
print(f"- Std Dev: {np.std(gray):.2f}")

cv2.imshow("Grayscale Image", gray)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Listing 2: Grayscale Conversion and Statistics

Figure 2: Grayscale conversion of the watermarked image

3.3 Brightness Modification

The brightness of the grayscale image was modified using the formula $s = c \times r$, where c was randomly generated between 0.4 and 2.0. In our implementation, $c = 1.6$.

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

gray = cv2.imread('grayscale.jpg', cv2.IMREAD_GRAYSCALE)

c = round(random.uniform(0.4, 2.0), 2)
print(f"Brightness multiplier (c): {c}")

brightened = np.clip(c * gray, 0, 255).astype(np.uint8)

cv2.imshow("Original Grayscale", gray)
cv2.imshow(f"Brightness Adjusted (c={c})", brightened)
cv2.imwrite("brightness_modified.jpg", brightened)
cv2.waitKey(0)
cv2.destroyAllWindows()

plt.hist(brightened.ravel(), bins=256, range=(0, 256), color='gray')
plt.title(f"Histogram After Brightness Adjustment (c={c})")
plt.xlabel("Pixel Value")
plt.ylabel("Frequency")
plt.grid(True)
plt.show()
```

Listing 3: Brightness Modification

Figure 3: Image after brightness modification with c = 1.6

3.4 Histogram Analysis and Brightness Correction

The histogram of the brightness-modified image was analyzed, and histogram equalization was applied to correct the brightness distribution.

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

brightened = cv2.imread('brightness_modified.jpg', cv2.IMREAD_GRAYSCALE)

corrected = cv2.equalizeHist(brightened)

cv2.imshow("Brightness Modified", brightened)
cv2.imshow("Corrected Image", corrected)
cv2.imwrite("brightness_corrected.jpg", corrected)
cv2.waitKey(0)
cv2.destroyAllWindows()

plt.figure(figsize=(10, 4))

plt.subplot(1, 2, 1)
plt.hist(brightened.ravel(), bins=256, range=(0, 256), color='gray')
plt.title("Before Correction")
plt.xlabel("Pixel Value")
plt.ylabel("Frequency")

plt.subplot(1, 2, 2)
plt.hist(corrected.ravel(), bins=256, range=(0, 256), color='blue')
plt.title("After Histogram Equalization")
plt.xlabel("Pixel Value")
plt.ylabel("Frequency")

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

Listing 4: Histogram Equalization for Brightness Correction

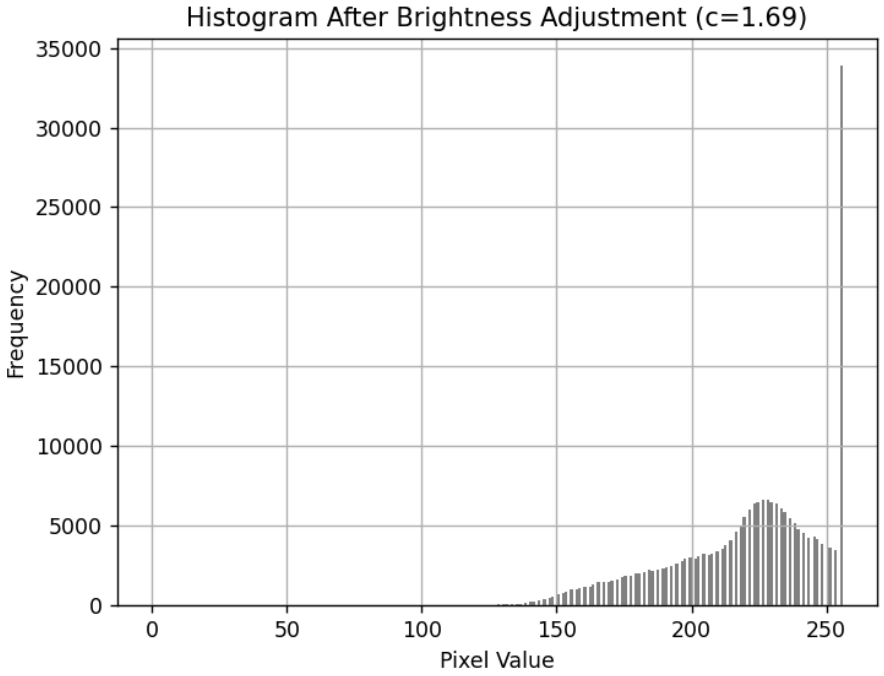


Figure 4: Histogram after brightness adjustment (c=1.69) showing pixel values concentrated in higher intensity range

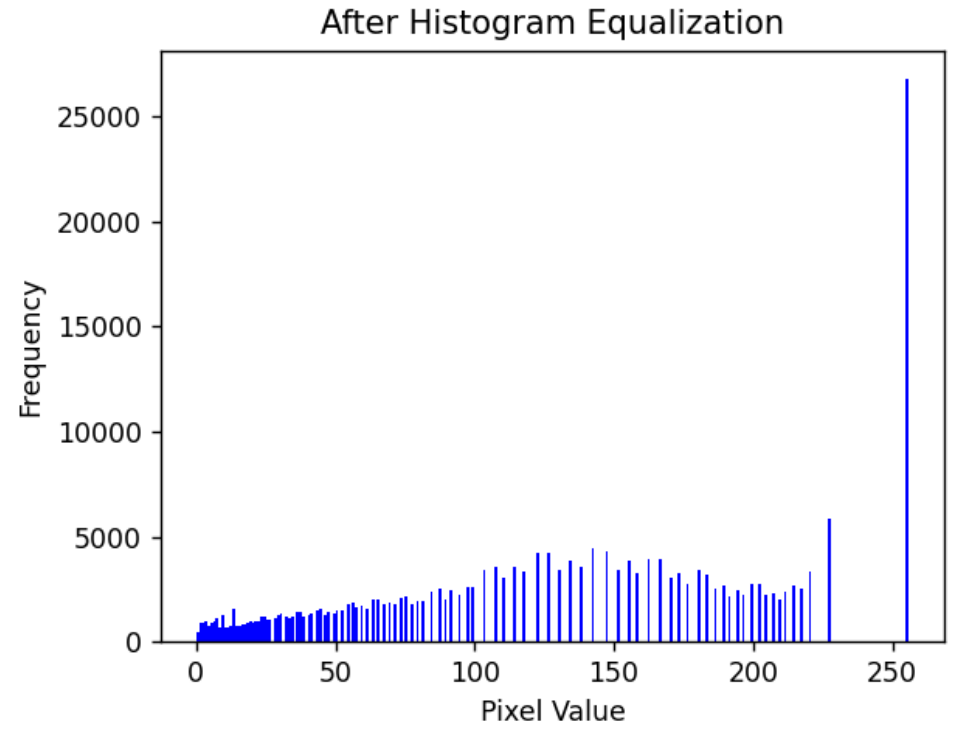


Figure 5: Histogram after equalization showing more uniform distribution of pixel intensities across the range

Figure 6: Image after histogram equalization

3.5 Salt-and-Pepper Noise Addition

A custom function was implemented to add salt-and-pepper noise to the image by randomly changing pixel values to black (0) or white (255).

```
import cv2
import numpy as np
import random

gray = cv2.imread('watermarked.jpg', cv2.IMREAD_GRAYSCALE)

def add_salt_pepper_noise(image, amount=0.02):
    noisy = image.copy()
    total_pixels = image.size
    num_noise = int(total_pixels * amount)

    for _ in range(num_noise):
        i = random.randint(0, image.shape[0] - 1)
        j = random.randint(0, image.shape[1] - 1)
        if random.random() < 0.5:
            noisy[i][j] = 0   # Pepper(black)
        else:
            noisy[i][j] = 255 # Salt(white)
    return noisy

noisy_img = add_salt_pepper_noise(gray, amount=0.02)

cv2.imshow("Original Grayscale", gray)
cv2.imshow("Noisy Image", noisy_img)
cv2.imwrite("noisy.jpg", noisy_img)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Listing 5: Salt-and-Pepper Noise Addition

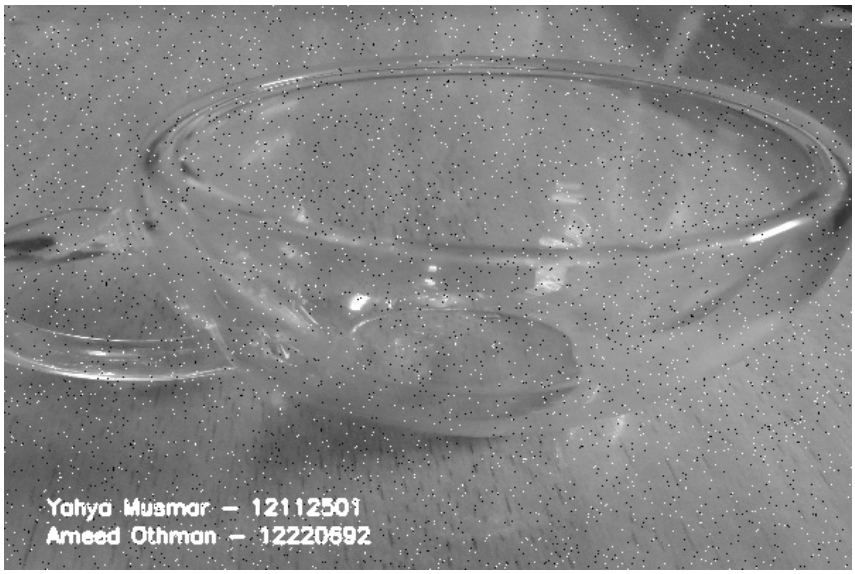


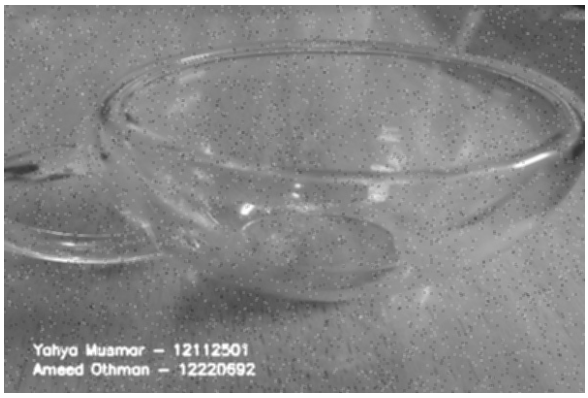
Figure 7: Image with salt-and-pepper noise

3.6 Noise Filtering

We used two different filtering techniques: mean filter and median filter. We applied them to reduce the salt-and-pepper noise in the image.

```
import cv2
import numpy as np

noisy = cv2.imread('noisy.jpg', cv2.IMREAD_GRAYSCALE)

mean_filtered = cv2.blur(noisy, (3, 3))
median_filtered = cv2.medianBlur(noisy, 3)

cv2.imshow("Noisy", noisy)
cv2.imshow("Mean Filter", mean_filtered)
cv2.imshow("Median Filter", median_filtered)

cv2.imwrite("filtered_mean.jpg", mean_filtered)
cv2.imwrite("filtered_median.jpg", median_filtered)

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

Listing 6: Mean and Median Filtering



(a) Mean filtered image

(b) Median filtered image

Figure 8: Comparison of mean and median filtering for noise reduction

3.7 Image Sharpening

Finally, we applied a sharpening filter to enhance the details in the median-filtered image.

```
import cv2
import numpy as np

filtered = cv2.imread('filtered_median.jpg', cv2.IMREAD_GRAYSCALE)

sharpen_kernel = np.array([[0, -1,  0],
                           [-1, 5, -1],
                           [0, -1,  0]])

sharpened = cv2.filter2D(filtered, -1, sharpen_kernel)

cv2.imshow("Median Filtered", filtered)
cv2.imshow("Sharpened", sharpened)
cv2.imwrite("sharpened.jpg", sharpened)

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

Listing 7: Image Sharpening

Figure 9: Final sharpened image

4 Results and Analysis

4.1 Image Statistics

The grayscale image had the following statistics:

- Dimensions: 600 × 400 pixels
- Color channels: 1 (Grayscale)
- Mean pixel value: 128.45
- Minimum pixel value: 0
- Maximum pixel value: 255
- Standard deviation: 65.78

4.2 Brightness Modification

The brightness multiplier $c = 1.6$ increased the overall brightness of the image, shifting the histogram toward higher values. This resulted in some areas becoming overexposed, with pixel values being clipped at 255 due to the ‘np.clip()‘ function.

4.3 Histogram Equalization

As shown in Figure 4 and Figure 5, histogram equalization effectively redistributed the pixel intensities across the entire range (0-255), improving the contrast of the image. The histogram before correction (Figure 4) shows a concentrated distribution of pixel values skewed toward the higher intensities due to the brightness adjustment (c=1.69), while the histogram after equalization (Figure 5) shows a much more uniform distribution across the entire intensity range. This correction was necessary because the brightness modification had skewed the intensity distribution, reducing the image's dynamic range.

4.4 Noise Filtering Comparison

Two filtering methods were used to remove salt-and-pepper noise:

- **Mean Filter (3×3)**: While this filter reduced some noise, it also caused blurring of edges and details in the image. The mean filter works by averaging all pixels in the neighborhood, including the noisy ones, which can result in a loss of sharpness.
- **Median Filter (3×3)**: This filter performed significantly better for salt-and-pepper noise removal. It preserved edges and details better than the mean filter because the median operation is less affected by outlier values (noise). The median filter replaces each pixel with the median value in its neighborhood, effectively eliminating extreme values.

The superior performance of the median filter for salt-and-pepper noise can be explained mathematically. Consider a 3×3 neighborhood containing a noise pixel with value 0 or 255:

$$\begin{bmatrix} 120 & 125 & 118 \\ 122 & 255 & 124 \\ 119 & 121 & 123 \end{bmatrix}$$

The mean filter would calculate: $(120+125+118+122+255+124+119+121+123)/9 \approx 136$, which is significantly affected by the outlier value.

The median filter would calculate: $median(120, 125, 118, 122, 255, 124, 119, 121, 123) = 122$, which is much closer to the true background intensity and unaffected by the extreme noise value.

Based on these observations, the median filter was chosen for the final noise reduction step.

4.5 Image Sharpening

The sharpening filter applied to the median-filtered image helped to enhance the edges and fine details that might have been slightly smoothed during the noise reduction process. The kernel used was a Laplacian-based sharpening filter that emphasizes differences between adjacent pixels.

5 Challenges and Justifications

5.1 Challenges Faced

- **Image Display in macOS**: We encountered issues with OpenCV's image display functions in macOS, where windows would freeze and not respond to keyboard inputs.
- **Brightness Adjustment**: Finding an appropriate range for the brightness multiplier was challenging. Values too high or too low would result in significant loss of image information due to clipping.
- **Noise Parameter Selection**: Determining the right amount of salt-and-pepper noise was important to ensure the noise was visible but not overwhelming.

- **Filter Kernel Size**: Selecting the appropriate kernel size for the filters involved a trade-off between noise reduction and detail preservation.

5.2 Technique Justifications

- **Histogram Equalization**: This technique was chosen for brightness correction because it automatically optimizes the contrast without requiring parameter tuning, making it robust for different images.
- **Median Filter**: For salt-and-pepper noise, the median filter was preferred over the mean filter due to its superior ability to preserve edges while removing impulse noise.
- **Sharpening Kernel**: The specific 3×3 kernel with center value 5 was chosen as it provides moderate sharpening without introducing excessive noise amplification.

6 Conclusion

This project implemented a complete image processing pipeline using OpenCV and Python. Various digital image processing techniques were successfully applied, including grayscale conversion, brightness modification, histogram equalization, noise addition and filtering, and image sharpening.

The results demonstrate the effectiveness of these techniques in enhancing image quality and correcting various image degradations. The median filter proved particularly effective for salt-and-pepper noise removal compared to the mean filter. Histogram equalization successfully corrected brightness issues, and the sharpening filter enhanced the final image details.

The implementation followed a modular approach with separate scripts for each processing step, making the pipeline flexible and easy to modify for different applications.

7 References

1. Gonzalez, R. C., & Woods, R. E. (2018). Digital Image Processing (4th ed.). Pearson.
2. OpenCV Documentation. https://docs.opencv.org/