An-Najah National University

Department of Computer Engineering

Digital Image Processing10636318

Second Semester2024/2025

OpenCV Project

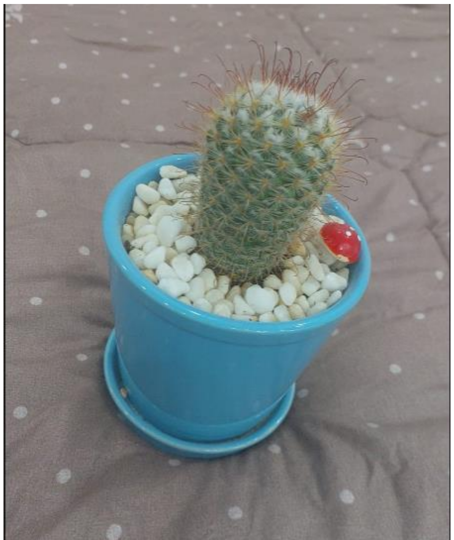
Part 1:

1 - Input image: these lines load a colored image and if there is no image found we got an exception.

```python
# Load in color first
image = cv2.imread('me.jpg')
if image is None:
    raise Exception("Image not found!")
```

2 – here we add the watermark at a random place at the photo.

```python
# Add watermark text to the image (black color for a bright image)
watermark_text = "Ahmad Khalil - 12027692"
font = cv2.FONT_HERSHEY_SIMPLEX
(text_width, text_height), _ = cv2.getTextSize(watermark_text, font, 1, 2)

# Compute max x and y where the text can be placed without going out of bounds
max_x = image.shape[1] - text_width
max_y = image.shape[0] - text_height

# Generate random position
x = random.randint(0, max_x)
y = random.randint(text_height, image.shape[0])  # ensure y is below top and above bottom

# Put the text at the random position
cv2.putText(image, watermark_text, (x, y), font, 1, (0, 0, 0), 2)

cv2.imwrite("watermarked_image.jpg", image)
```



Tasks:

Load the image in grayscale:


```python
image_gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
cv2.imwrite("grayscale_image.jpg", image_gray)
```

Display its dimensions, color channels, and pixel value statistics:

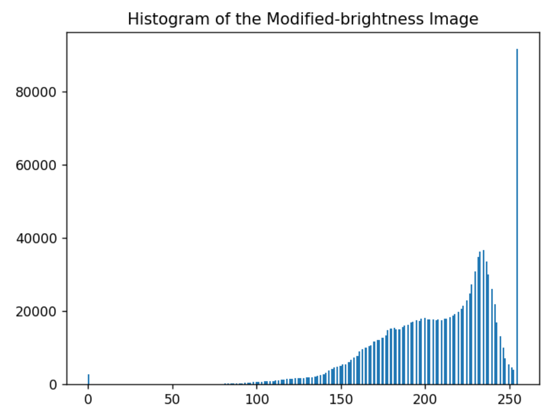
```python
# Calculate mean, min, max of grayscale image
mean_val = np.mean(image_gray)
min_val = np.min(image_gray)
max_val = np.max(image_gray)

print("Grey Image height, width:", image_gray.shape)  # (height, width)

if len(image_gray.shape) == 2:
    print("Grayscale Color Channels: 1")
else:
    print(f"Grayscale Color Channels: {image_gray.shape[2]}")

print("Mean Value:", mean_val, "Min Value:", min_val, "Max Value:", max_val)
```

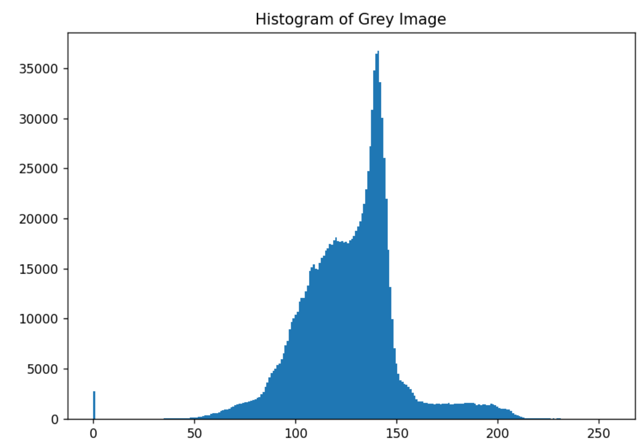
```
Grey Image height, width: (1315, 899)
Grayscale Color Channels: 1
Mean Value: 126.76443619230493 Min Value: 0 Max Value: 247
```

Modify the brightness of the grayscale image by applying the following equation

s = c*r

```python
c = round(random.uniform(0.4, 2.0), 2)
print("Random Brightness Coefficient (c):", c)
# Apply brightness modification
image_bright = image_gray.astype(np.float32) * c
image_bright = np.clip(image_bright, 0, 255).astype(np.uint8)
cv2.imwrite("brightness_modified.jpg", image_bright)
plt.hist(image_bright.flatten(), bins=256, range=(0, 255))
plt.title("Histogram of the Modified-brightness Image")
```





The brightness modification using the formula s=c*r successfully increases (or decreases, depending on c) the overall brightness of a grayscale image. The resulting histogram confirms that a coefficient C>1 was used, leading to a significant shift towards brighter intensities.

to correct the brightness of the resulting image:

1 - We used first linear contrast streching .

```python
# Apply linear contrast stretching
min_val_bright = np.min(image_bright)
max_val_bright = np.max(image_bright)
if max_val_bright > min_val_bright:
    image_stretched = (image_bright - min_val_bright) * (255.0 / (max_val_bright - min_val_bright))
else:
    image_stretched = image_bright.copy()
image_stretched = np.clip(image_stretched, 0, 255).astype(np.uint8)
```

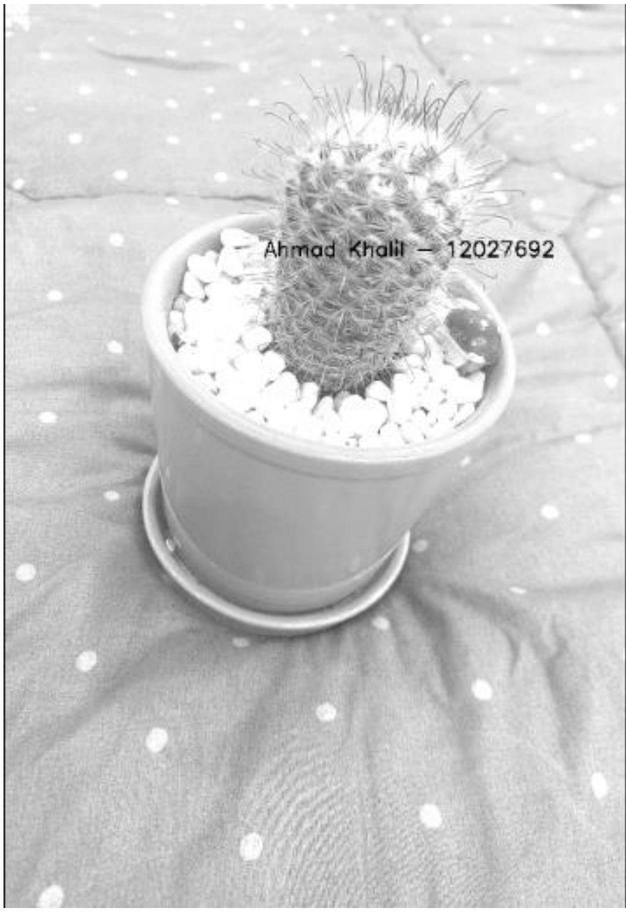
justification : Linear contrast stretching is used to enhance the visibility of details in an image when the pixel intensity values are concentrated in a narrow range (e.g., mostly mid-grays or darks), And that a suitable way to deal with c that can be less or more than 1.

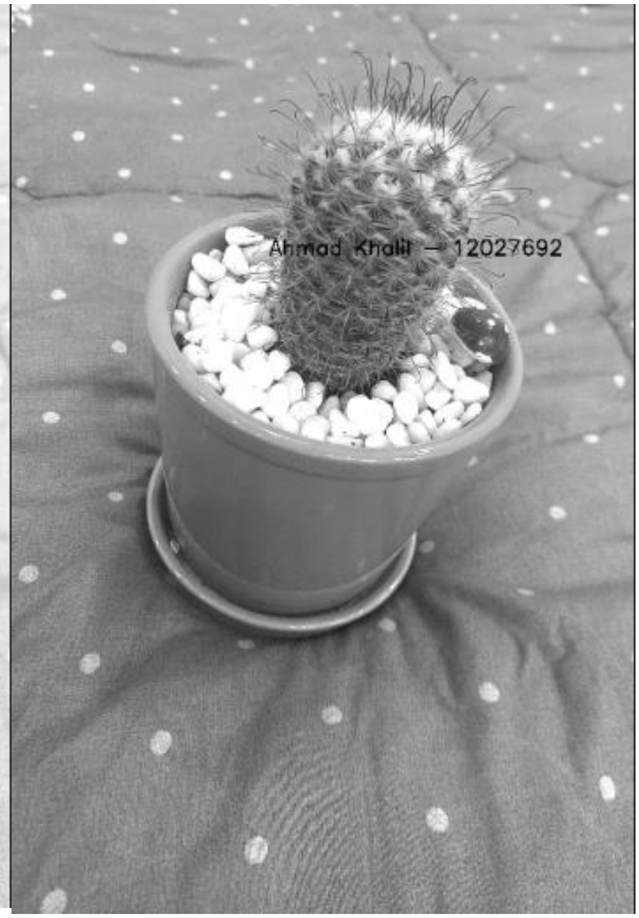
I followed it by :

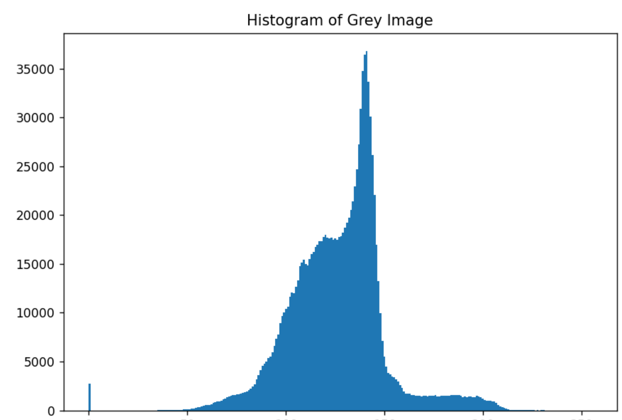
```python
if mean_pixel_value > dark_threshold:
    gamma = 1.5
    image_gamma_dark = np.uint8(np.clip(c * np.power(image_gray / 255.0, gamma) * 255.0, 0, 255))
```

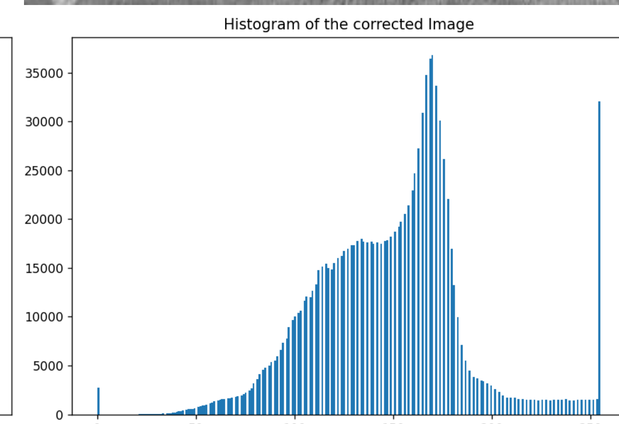
Gamma modification, caring more to brighten images so they are passed and fixed with no problems.

brightness_modified corrected_image









This histogram comparison visually justifies the use of linear contrast stretching it effectively spreads out the grayscale values, enhancing image contrast without altering its structural content.

I‘ve written a function to add salt-and-pepper noise by changing the values of some pixels to black or white.

```
# Amount of noise (e.g., 2% noise)
amount = 0.02
# Create a copy of the image
noisy = image_stretched.copy()
# Add salt (white pixels)
salt = np.random.random(image_stretched.shape) < amount / 2
noisy[salt] = 255
# Add pepper (black pixels)
pepper = np.random.random(image_stretched.shape) < amount / 2
noisy[pepper] = 0
# Save the noisy image
cv2.imwrite("noisy_image.jpg", noisy)
```

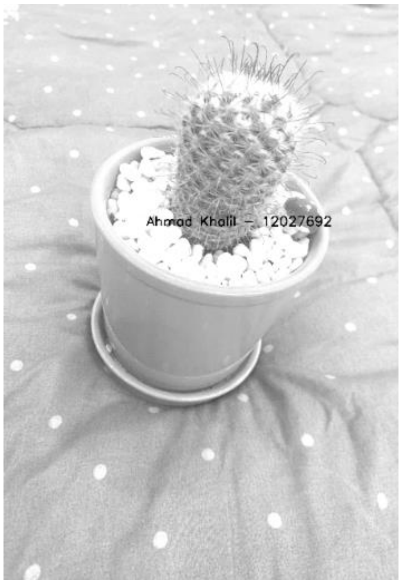
Then I Reduced the noise using both mean filter and median filter ,and while the median worked just fine!

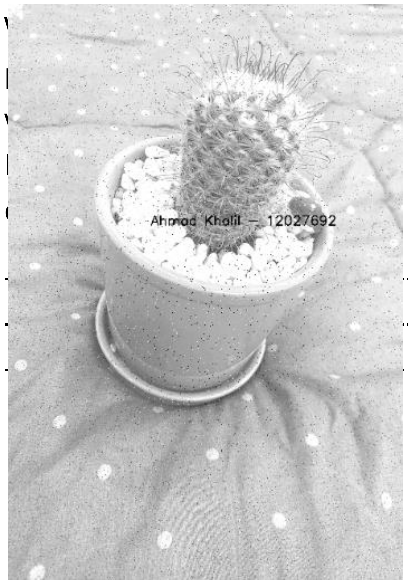
```
median_filtered = cv2.medianBlur(noisy, 3)
cv2.imwrite("noisy_median_image.jpg", median_filtered)
```

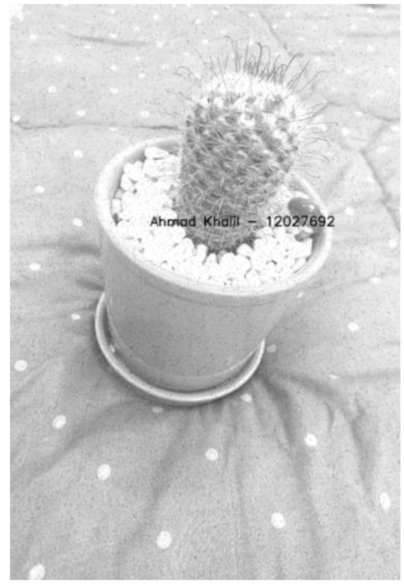
The mean needed a little push which I gave using :

```
mean_filtered = cv2.blur(noisy, (3, 3))
cv2.imwrite("noisy_mean_image.jpg", mean_filtered)

bilateral_filtered = cv2.bilateralFilter(mean_filtered, d=9, sigmaColor=75, sigmaSpace=75)
cv2.imwrite("bilateral_filtered.jpg", bilateral_filtered)
```

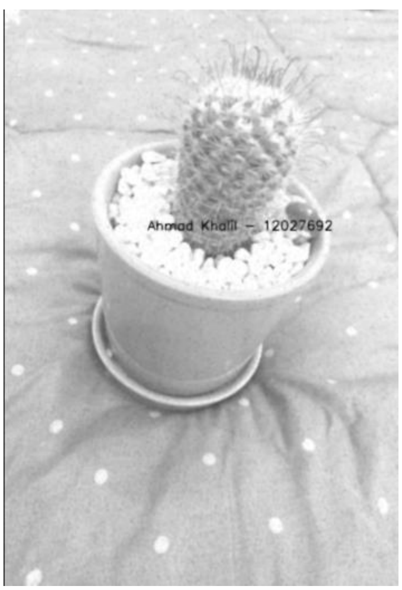
Median filter noisy picture mean filt er







and we can see that here: mean filter + bilateral filter.



Summary of Challenges Faced and Justification for Techniques Used

During the implementation of this project, several challenges were encountered:

1. **Brightness Adjustment Sensitivity:**

a. **Challenge:** Choosing an appropriate brightness coefficient c for the transformation s = c*r was non-trivial. Small changes in c significantly impacted the image brightness, sometimes leading to washed-out or overly dark results.

b. **Justification:** To handle varying results of brightness adjustment, we applied **linear contrast stretching** to normalize pixel intensities. This helped stretch the dynamic range and improve visibility. Additionally, **gamma correction** was used to fine-tune the brightness perception, as it allows non-linear adjustments that are more aligned with human visual response.

2. **Noise Simulation and Reduction:**

   a. **Challenge:** Simulating salt-and-pepper noise was straightforward but removing it while preserving image detail was more difficult.

   b. **Justification:** We tested both **mean** and **median** filters. The **median filter** performed better in removing impulse noise without significantly blurring the image. The **mean filter** often introduced blurring and left some noise artifacts. To further improve the result of the mean filter, we added a **bilateral filter**, which preserved edges while smoothing noise.