Rosie Manoli
Course: Digital Image Processing and Applications
Assignment #2: Edge Detection, Compression, and Feature Extraction

Introduction: For code design I used the online Octave environment and the book "Ψηφιακή επεξεργασία & ανάλυση εικόνας" by Papamarkos Nikolaos H.. In the repository you will find all my scripts, providing the code for solving the following questions.

Exercise 1 - Edge Detection: Consider the grayscale images bridge.bmp, girlface.bmp, and lighthouse.bmp.

Questions:
A. Compute the edges of the image lighthouse.bmp using the following first derivative-based methods: Sobel, Roberts, Prewitt, and Kirsch. For the Sobel, Roberts, and Prewitt methods, use Otsu's global thresholding method, while for the Kirsch method, determine the optimal threshold experimentally. Compare the methods in terms of their performance.
B. Compute the edges of the image lighthouse.bmp using second derivative-based methods (LoG filter). Experimentally determine the optimal values for the variance and threshold parameters in the implementation of the LoG operator. Present indicative results and provide corresponding commentary.
C. Compute the edges of the image lighthouse.bmp using the Canny method (considering the "default" parameterization provided by Octave). Comment on the edge detection result.
D. Compute the edges for the images bridge.bmp and girlface.bmp using the methods applied in steps (A)-(C) and compare their performance across all three images. Provide only the optimal edge mask for each method.

Answers:
A. The design and implementation of the code for all methods are provided in the repository. Initially, for the Sobel method, we observe that it detects the main edges of the image smoothly. This is evident in areas such as the rooftops, the fence, the lighthouse, and the street pole. Next, applying the Roberts method, we see that the detected edges are fewer, thinner, and perhaps not as clear as those detected by the Sobel method. Then, using the Prewitt method, the edges are detected quite well, with results similar to the Sobel method, though slightly thinner and fewer in number. Finally, the Kirsch method was applied, and various threshold values were tested to determine the optimal one. It was observed that as the threshold (T) decreases, the number of detected edges increases; however, this does not always yield satisfactory results. Among the different values tested, the image with T=35 provides a fairly good approximation.

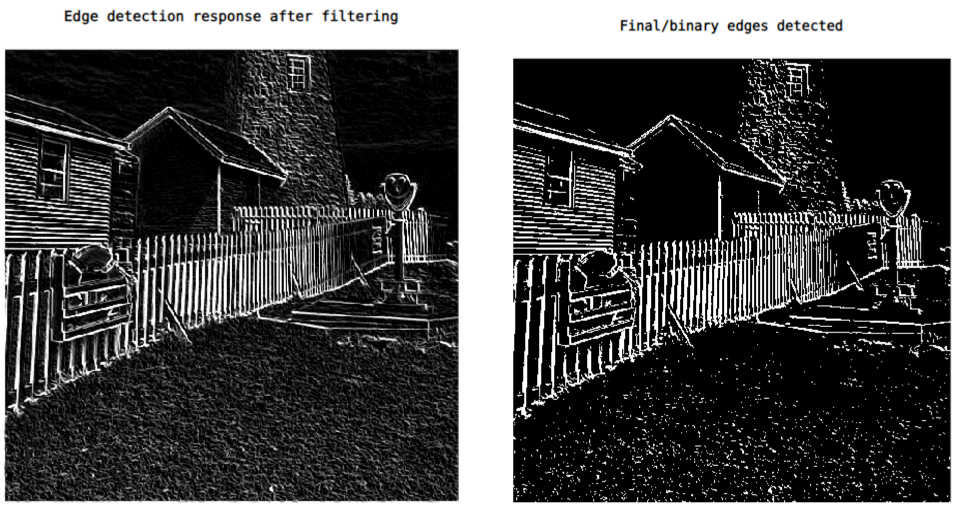

Edge detection response after filtering
Final/binary edges detected

Figure 1: The image after filtering with the Sobel method and the final edge image

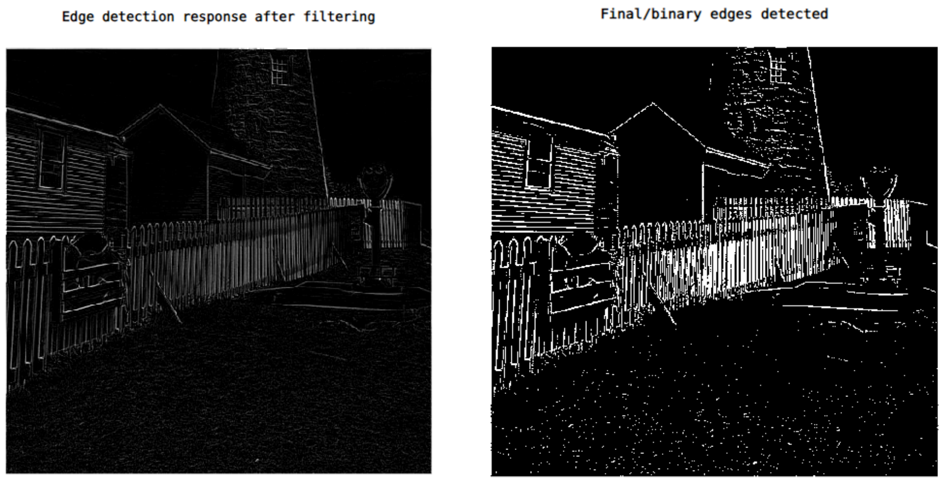

Edge detection response after filtering
Final/binary edges detected

Figure 2: The image after filtering with the Roberts method and the final edge image

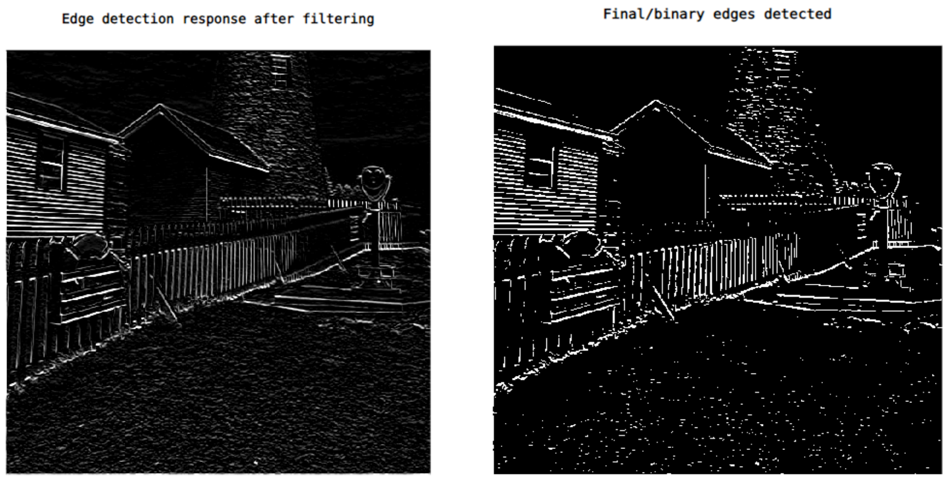

Edge detection response after filtering
Final/binary edges detected

Figure 3: The image after filtering with the Prewitt method and the final edge image

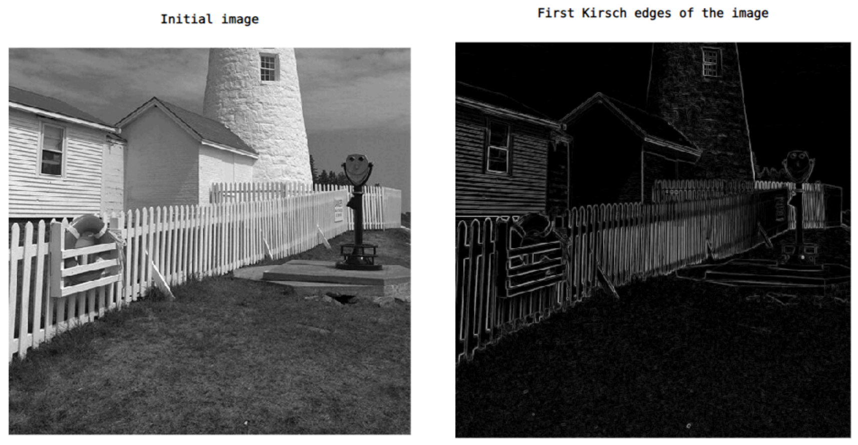


Figure 4: The original image and the original image after filtering with the Kirsch method

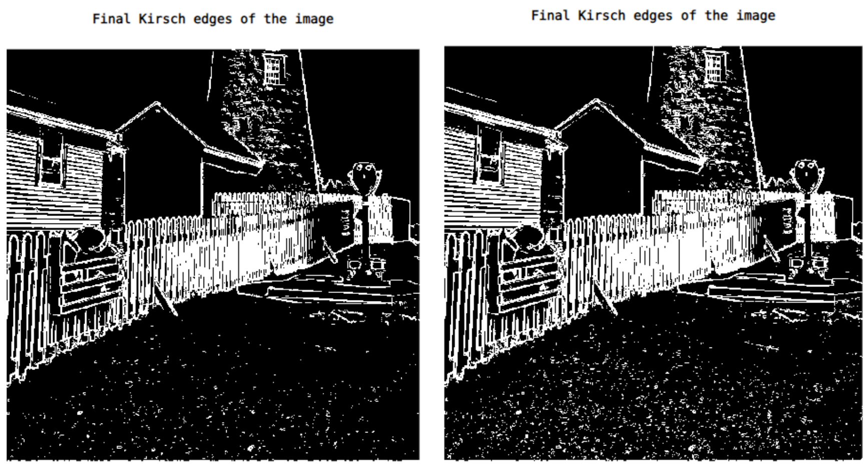

Final Kirsch edges of the image
Final Kirsch edges of the image

Figure 5: The final edge image with a threshold value of T=30 and T=25, respectively

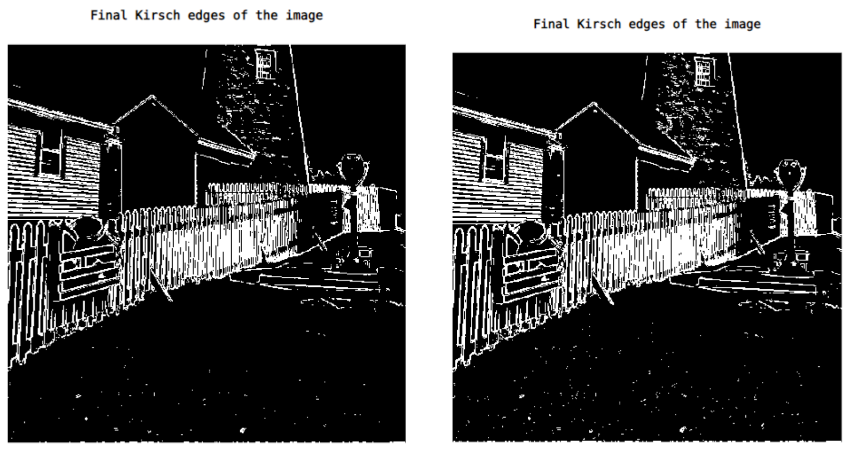

Final Kirsch edges of the image
Final Kirsch edges of the image

Figure 6: The final edge image with a threshold value of T=40 and T=35, respectively

B. For the LoG filter, the code is provided in the repository. Applying the filter to the image requires testing various values for both variance and threshold to determine the optimal parameters. The first test used a variance of 1.4 and a threshold value of T=4, which produced well-detected edges for the image. Further experimentation showed that increasing σ while simultaneously decreasing T resulted in an excessive number of edges and noise. For a lower threshold (Figure 8), the image appeared cleaner but still did not achieve the desired result. By slightly decreasing the variance and slightly increasing the threshold, a good approximation of the image was obtained.

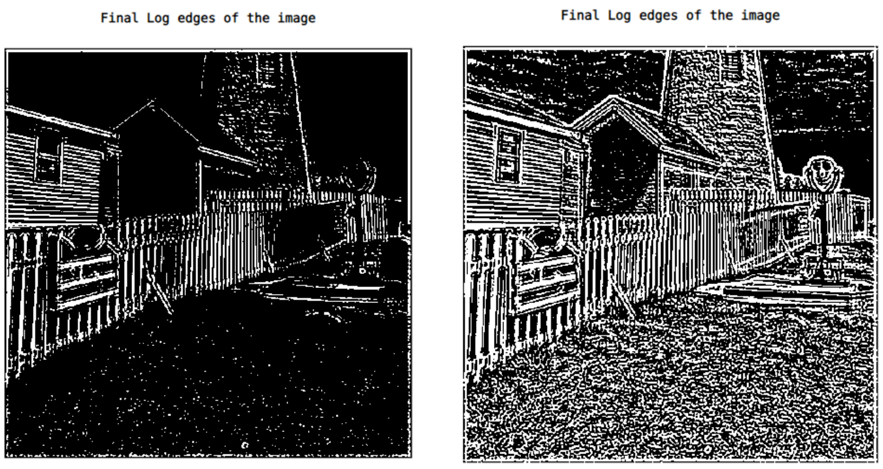


Figure 7: The image with the final edges for σ=1.4 and threshold value T=4 and for σ=2 and T=1, respectively.

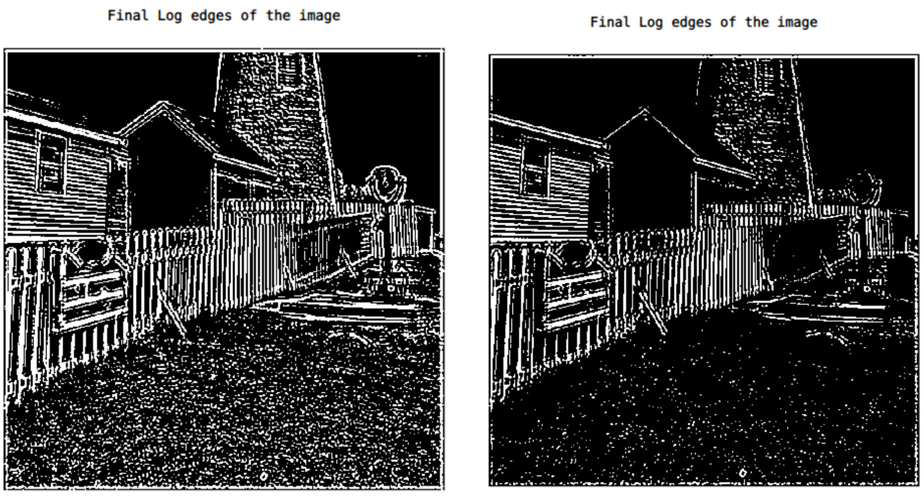


Figure 8: The image with the final edges for σ=2 and threshold value T=2 and for σ=1.5 and T=3, respectively.

C. For the Canny method, the code is provided in the repository. The Canny method has detected the edges of the image very well. The edges are continuous and clear, and the essential information of the image has been identified. Compared to the other methods, this is perhaps the cleanest edge detection.

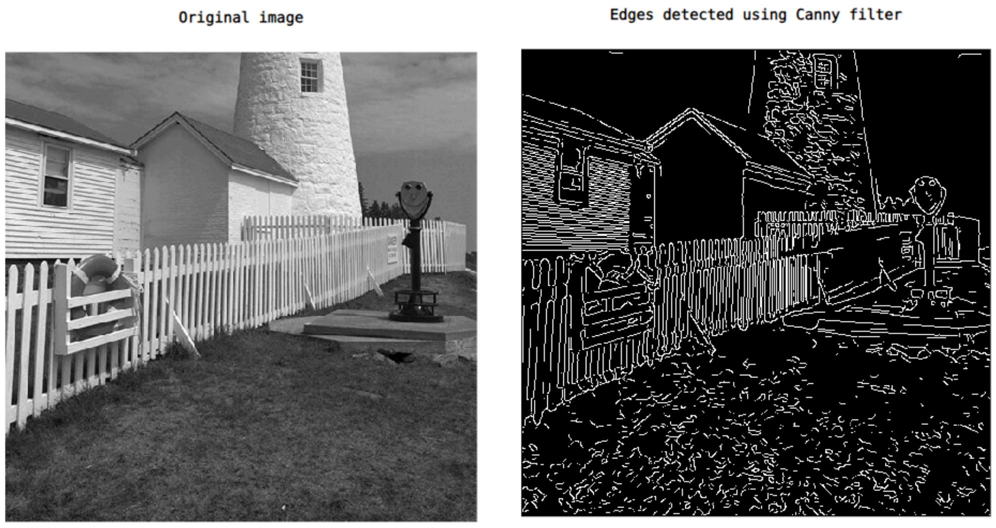


Figure 9: The image before and after edge detection using the Canny method.

D. For the image "bridge.jpg":

As with the previous image, for the Sobel method, we observe that it has detected the main edges of the image smoothly. Then, applying the Roberts method, we see that the detected edges are fewer and thinner, and although it captures fine details, there is also noise, especially in the area of the leaves. Next, using the Prewitt method, edges are detected quite well, and the results are similar to the Sobel method, though possibly slightly thinner and fewer in number. Then, the Kirsch method was used with an optimal threshold of T=35. Finally, the LoG filter was applied with σ=1.4 and T=4 as the best values, along with the Canny method, which also produced clear and continuous edges in this case.

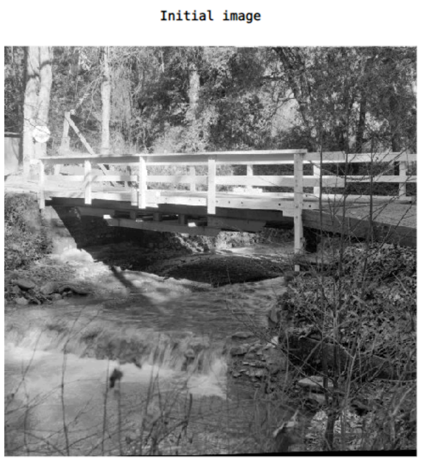


Figure 10: The original image "bridge.jpg"

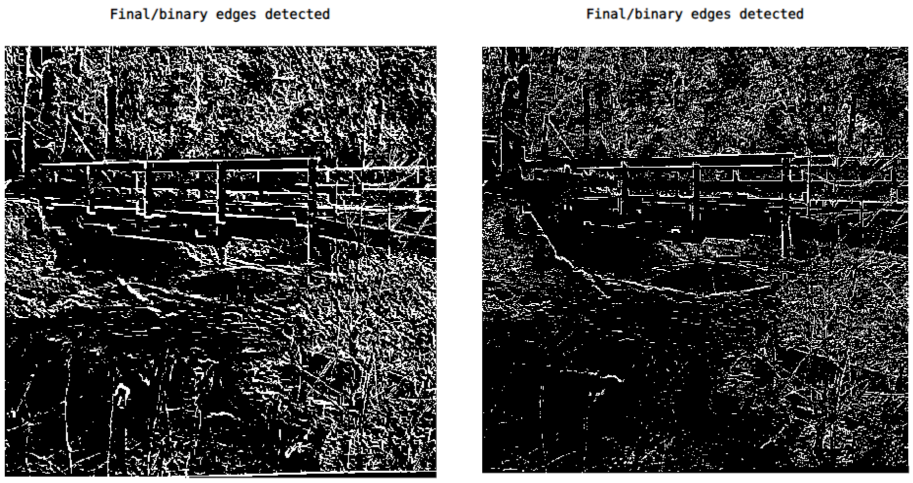


Figure 11: The image after edge detection using the Sobel and Roberts methods

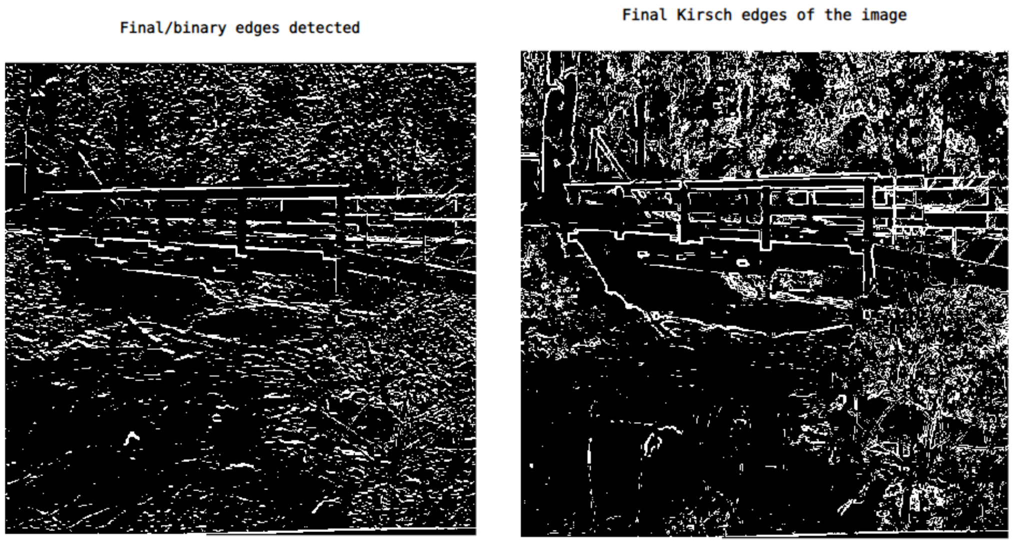


Figure 12: The image after edge detection using the Prewitt and Kirsch methods with T=35

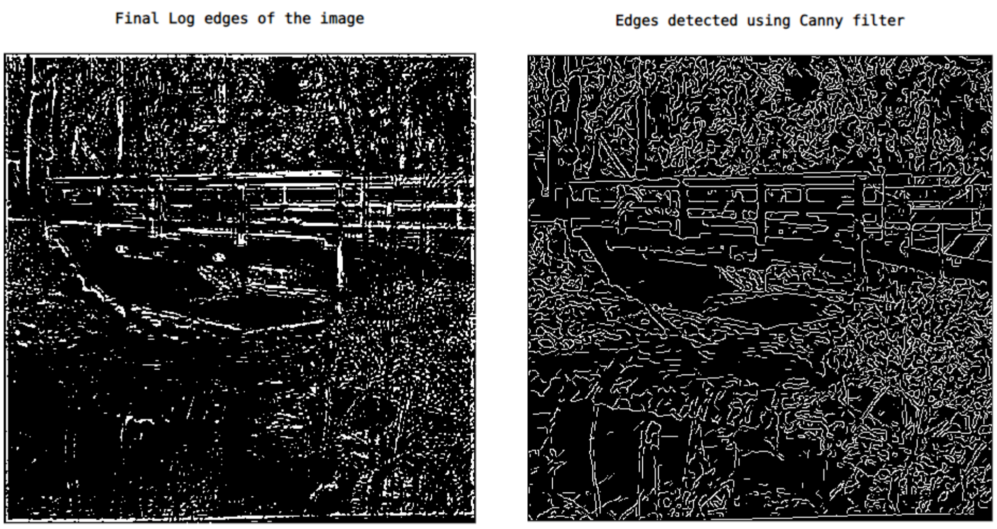

Final Log edges of the image
Edges detected using Canny filter

Figure 13: The image after edge detection using the LoG method for σ=1.4 and T=4 and Canny

For the image "girlface.jpg":
As with the previous images, for the Sobel method, we observe that it has detected the main edges of the image in a smooth and clear manner. Then, applying the Roberts method, we see that in this image as well, the detected edges are significantly fewer and thinner. Next, using the Prewitt method, edges are detected quite well, and the results are similar to the Sobel method. Then, the Kirsch method was used with an optimal threshold of T=25, which is lower than in the other images. Finally, the LoG filter was applied with σ=1.5 and T=2 as the best values, along with the Canny method, which also produced clear and continuous edges in this case.

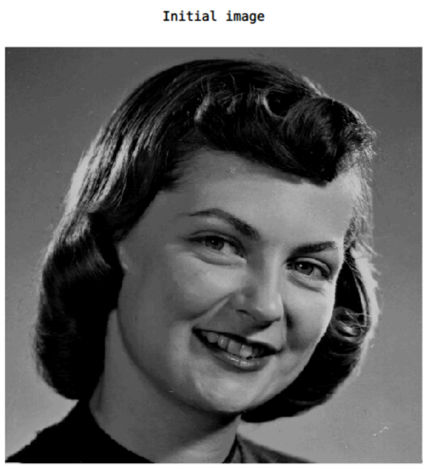


Figure 14: The original image "girlface.jpg"

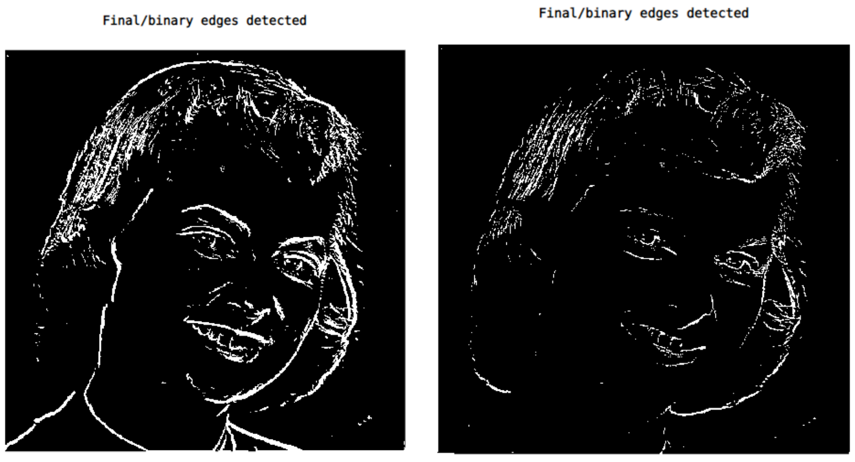
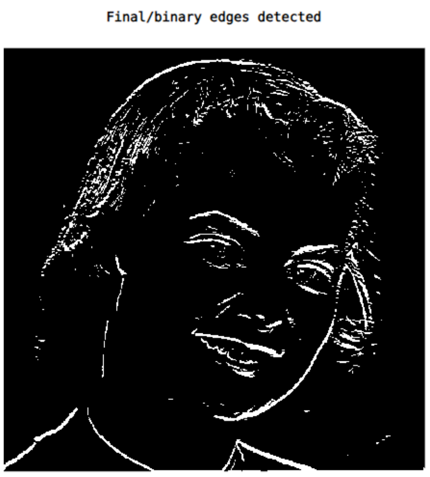


Figure 15: The image after edge detection using the Sobel and Roberts methods



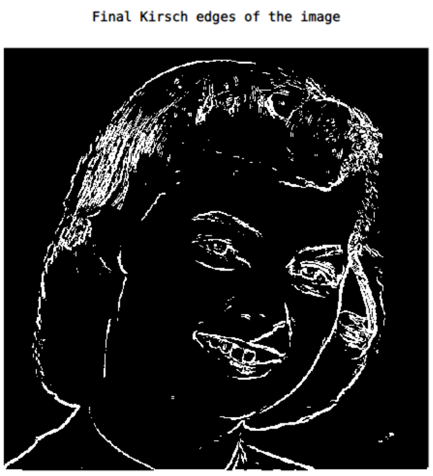
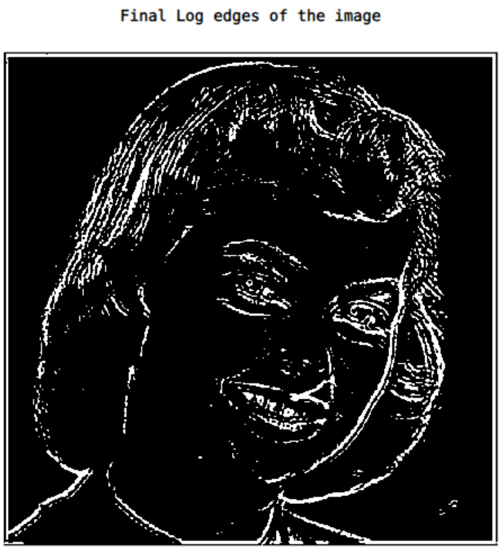


Figure 16: The image after edge detection using the Prewitt and Kirsch methods with T=25



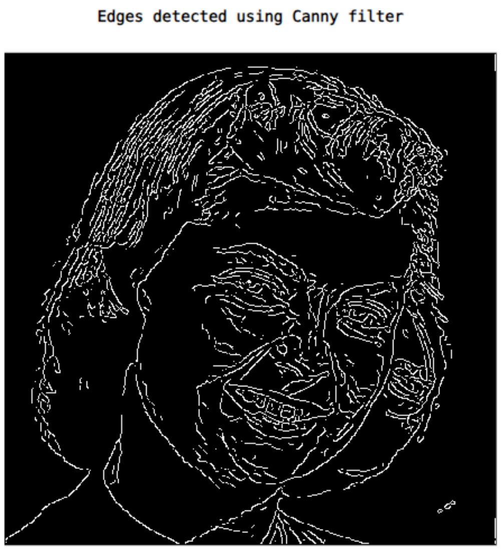


Figure 17: The image after edge detection using the LoG method for σ=1.5 and T=2 and Canny

Exercise 2 - Huffman Coding: Consider the grayscale images "bridge.bmp", "girlface.bmp", and "lighthouse.bmp".

Questions:

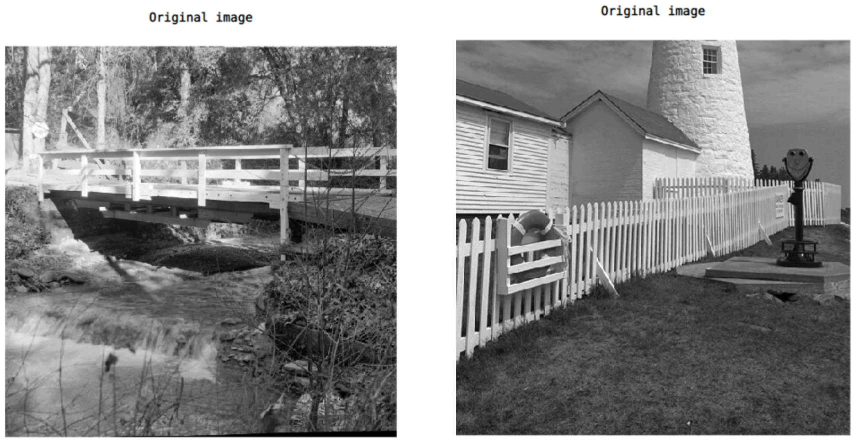
A. Implement and apply the Huffman coding method (lossless) for the images "bridge.bmp," "girlface.bmp," and "lighthouse.bmp." Do not use pre-existing functions or implementations of the Huffman method.

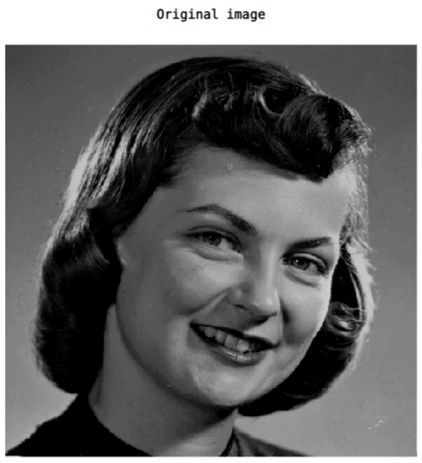
B. For each image, provide: 1) The resulting codewords (mapping of brightness values to binary codewords), 2) The average codeword length, 3) The entropy value, and 4) The achieved compression ratio.

C. Compare and evaluate the algorithm's performance (as calculated in step B) across the different images.

Answers:

A. The code implementation will be provided in the repository. Initially, I created a function that calculates the frequency of each pixel value in an image. Since there are 256 possible pixel values (0-255), the freq is an array with a length of 256. The value of each pixel increases the corresponding position in the freq array. Then, I created a function that constructs the Huffman tree based on the pixel value frequencies, from left to right. After that, the function huffman_encode_image encodes each pixel of the image using the Huffman dictionary, which is essentially an array of cells where each cell contains the binary sequence representing the pixel value. Finally, the functions are used, and the result is obtained.

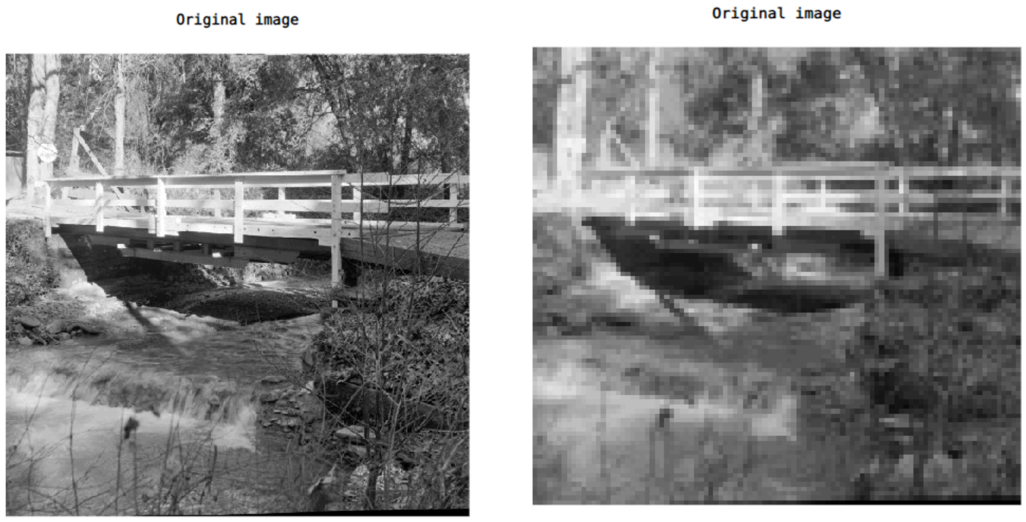




```
Images have been encoded using Huffman encoding.
```

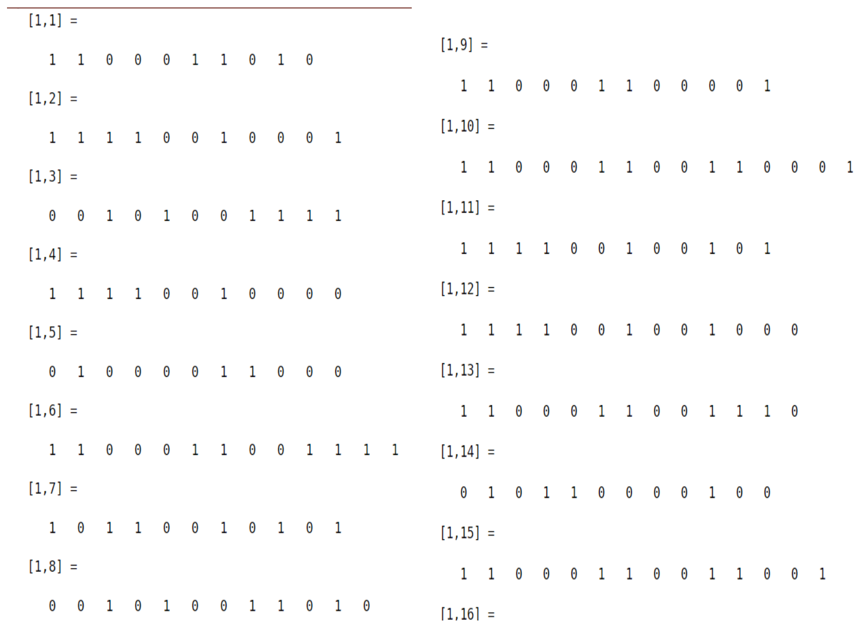
Figure 18: The images before Huffman coding

B. For the image "bridge.jpg":



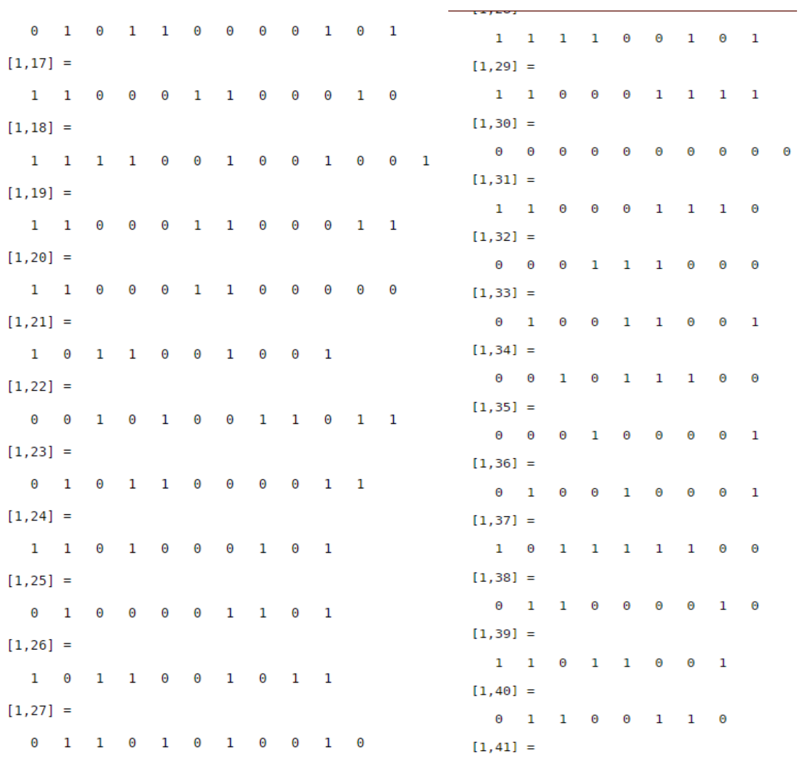
```
ans = Entropy original image =  7.569263.
ans = Average word length Huffman for original image =  7.595600.
ans = Compression ratio using Huffman =  0.949450.
```

Figure 19: The entropy, average codeword length, and compression ratio for the image "bridge.jpg"



```
[1,1] =

   1   1   0   0   0   1   1   0   1   0

[1,2] =

   1   1   1   1   0   0   1   0   0   0   1

[1,3] =

   0   0   1   0   1   0   0   1   1   1   1

[1,4] =

   1   1   1   1   0   0   1   0   0   0   0

[1,5] =

   0   1   0   0   0   0   1   1   0   0   0

[1,6] =

   1   1   0   0   0   1   1   0   0   1   1   1   1

[1,7] =

   1   0   1   1   0   0   1   0   1   0   1

[1,8] =

   0   0   1   0   1   0   0   1   1   0   1   0
```

```
[1,9] =

   1   1   0   0   0   1   1   0   0   0   0   1

[1,10] =

   1   1   0   0   0   1   1   0   0   1   1   0   0   0   1

[1,11] =

   1   1   1   1   0   0   1   0   0   1   0   1

[1,12] =

   1   1   1   1   0   0   1   0   0   1   0   0   0

[1,13] =

   1   1   0   0   0   1   1   0   0   1   1   1   0

[1,14] =

   0   1   0   1   1   0   0   0   0   1   0   0

[1,15] =

   1   1   0   0   0   1   1   0   0   1   1   0   0   1

[1,16] =
```



```
   0   1   0   1   1   0   0   0   0   1   0   1
[1,17] =
   1   1   0   0   0   1   1   0   0   0   1   0
[1,18] =
   1   1   1   1   0   0   1   0   0   1   0   0   1
[1,19] =
   1   1   0   0   0   1   1   0   0   0   1   1
[1,20] =
   1   1   0   0   0   1   1   0   0   0   0   0
[1,21] =
   1   0   1   1   0   0   1   0   0   1
[1,22] =
   0   0   1   0   1   0   0   1   1   0   1   1
[1,23] =
   0   1   0   1   1   0   0   0   0   1   1
[1,24] =
   1   1   0   1   0   0   0   1   0   1
[1,25] =
   0   1   0   0   0   0   1   1   0   1
[1,26] =
   1   0   1   1   0   0   1   0   1   1
[1,27] =
   0   1   1   0   1   0   1   0   0   1   0
```

```
   1   1   1   1   0   0   1   0   1
[1,29] =
   1   1   0   0   0   1   1   1   1
[1,30] =
   0   0   0   0   0   0   0   0   0   0
[1,31] =
   1   1   0   0   0   1   1   1   0
[1,32] =
   0   0   0   1   1   1   0   0   0
[1,33] =
   0   1   0   0   1   1   0   0   1
[1,34] =
   0   0   1   0   1   1   1   0   0
[1,35] =
   0   0   0   1   0   0   0   0   1
[1,36] =
   0   1   0   0   1   0   0   0   1
[1,37] =
   1   0   1   1   1   1   1   0   0
[1,38] =
   0   1   1   0   0   0   0   1   0
[1,39] =
   1   1   0   1   1   0   0   1
[1,40] =
   0   1   1   0   0   1   1   0
[1,41] =
   0   1   1   0   0   1   0   1
[1,42] =
   1   1   1   1   0   0   1   1
[1,43] =
   1   1   0   1   1   0   0   0
[1,44] =
   1   0   1   0   1   0   0   0
[1,45] =
   1   1   1   1   0   1   1   0
[1,46] =
   1   1   1   1   1   0   1   0
[1,47] =
   1   1   1   0   0   0   0   0
[1,48] =
   0   0   1   1   0   1   1   0
[1,49] =
   1   0   1   0   1   0   1   0
[1,50] =
   0   1   0   1   1   0   0   1
[1,51] =
   0   0   0   1   1   0   0   0   0
[1,52] =
   0   0   1   0   1   1   1   1
[1,53] =
   1   1   1   1   1   1   0
[1,54] =
```

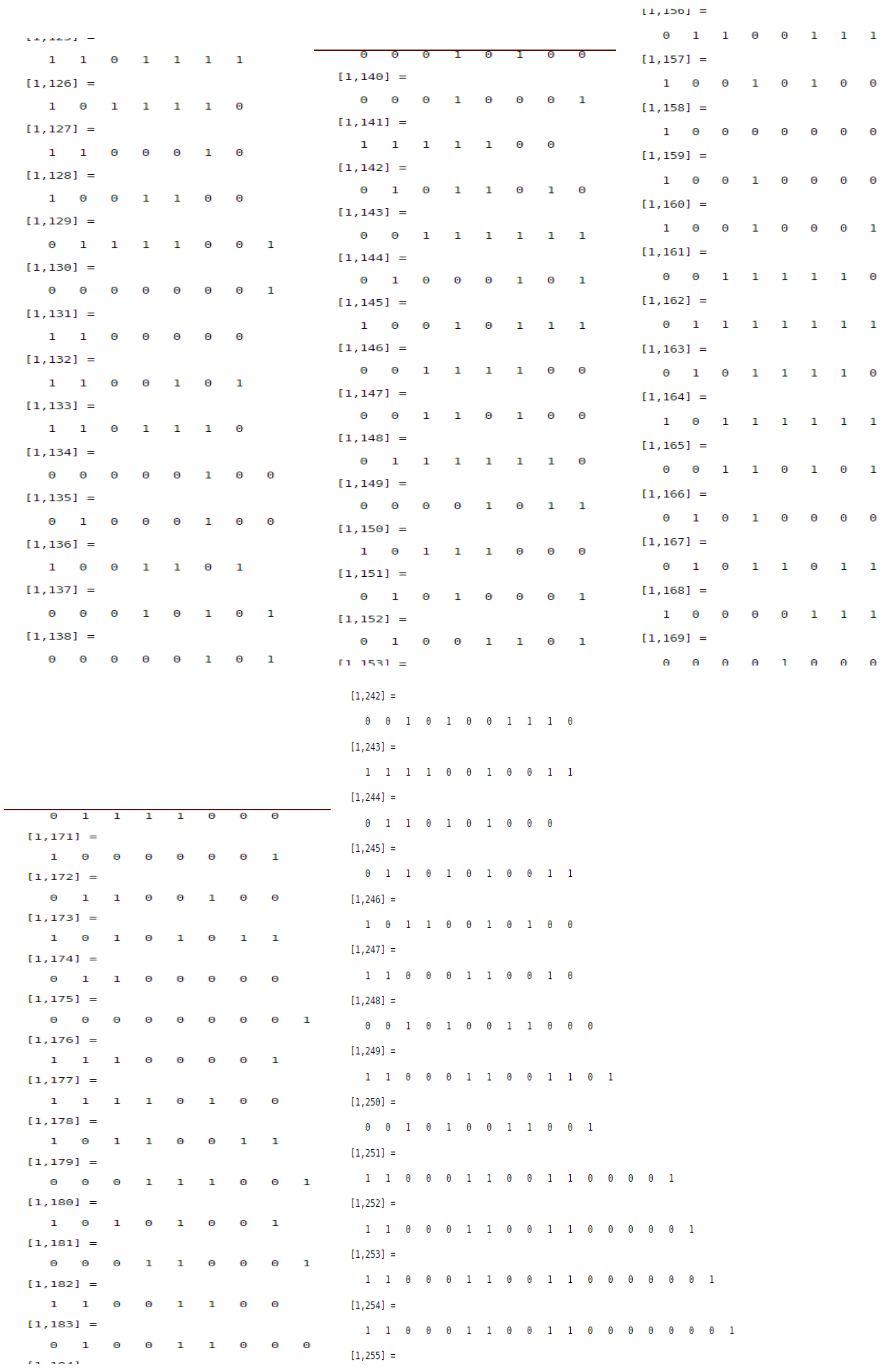
```
   1   1   1   1   1   1   1
[1,55] =
   1   1   1   0   0   1   1
[1,56] =
   1   0   0   1   0   1   1   0
[1,57] =
   0   0   0   0   0   1   1   1
[1,58] =
   0   0   0   0   1   0   1   0
[1,59] =
   0   0   0   0   0   1   1   0
[1,60] =
   0   0   0   1   1   1   0   1
[1,61] =
   1   1   0   0   1   1   1
[1,62] =
   0   0   1   0   1   0   0   0
[1,63] =
   1   1   0   0   0   0   1
[1,64] =
   1   1   1   0   0   0   1
[1,65] =
   1   0   1   0   0   0   1
[1,66] =
   1   1   1   0   0   1   0
[1,67] =
   1   1   1   0   1   0   0
```

```
[1,69] =
   0   1   0   0   1   1   1
[1,70] =
   0   1   0   1   0   0   1
[1,71] =
   1   0   1   1   0   1   1
[1,72] =
   1   0   1   0   1   1   0
[1,73] =
   0   1   0   0   1   0   1
[1,74] =
   1   0   0   0   1   1   1
[1,75] =
   0   0   1   0   0   0   1
[1,76] =
   0   0   1   0   0   1   0
[1,77] =
   0   1   0   1   1   1   0
[1,78] =
   0   0   0   0   1   1   0
[1,79] =
   0   0   1   1   1   0   1
[1,80] =
   0   0   0   0   1   1   1
[1,81] =
   0   0   0   1   0   1   1
[1,82] =
   0   1   0   0   0   1   1
```

```
   0   0   1   1   0   0   1
[1,84] =
   0   0   0   0   0   0   1
[1,85] =
   0   0   0   1   1   0   1
[1,86] =
   0   0   0   1   0   0   1
[1,87] =
   0   0   1   0   0   1   1
[1,88] =
   0   0   0   1   1   1   1
[1,89] =
   0   0   1   1   0   0   0
[1,90] =
   1   1   1   0   1   1
[1,91] =
   0   0   1   0   1   1   0
[1,92] =
   0   1   0   0   0   0   0
[1,93] =
   1   0   1   0   0   0   0
[1,94] =
   0   1   0   1   0   1   1
[1,95] =
   0   1   1   1   0   0   1
[1,96] =
   0   0   1   1   1   0   0
```

```
[1,97] =
   0   1   0   1   0   1   0
[1,98] =
   0   0   1   0   0   0   0
[1,99] =
   0   1   1   1   0   1   0
[1,100] =
   1   1   0   1   0   0   1
[1,101] =
   1   0   0   0   1   1   0
[1,102] =
   1   0   1   1   0   0   0
[1,103] =
   1   1   0   0   1   0   0
[1,104] =
   1   0   0   0   0   0   1
[1,105] =
   1   0   1   0   0   1   1
[1,106] =
   1   0   0   0   1   0   0
[1,107] =
   0   1   1   0   1   1   0
[1,108] =
   0   1   1   0   1   1   1
[1,109] =
   1   0   1   0   1   1   1
[1,110] =
```

```
[1,111] =
   0   1   1   1   0   0   0
[1,112] =
   0   1   1   0   1   0   0
[1,113] =
   0   1   1   1   0   1   1
[1,114] =
   1   1   0   1   0   1   1
[1,115] =
   1   0   0   1   1   1   0
[1,116] =
   1   0   1   0   0   1   0
[1,117] =
   1   0   1   1   0   1   0
[1,118] =
   1   0   0   1   1   1   1
[1,119] =
   1   1   1   1   0   0   0
[1,120] =
   1   0   1   1   1   0   1
[1,121] =
   1   1   0   1   0   1   0
[1,122] =
   1   0   0   0   0   1   0
[1,123] =
   1   1   0   1   1   0   1
[1,124] =
```



```
   1   1   0   1   1   1   1
[1,126] =
   1   0   1   1   1   1   0
[1,127] =
   1   1   0   0   0   1   0
[1,128] =
   1   0   0   1   1   0   0
[1,129] =
   0   1   1   1   1   0   0   1
[1,130] =
   0   0   0   0   0   0   0   1
[1,131] =
   1   1   0   0   0   0   0
[1,132] =
   1   1   0   0   1   0   1
[1,133] =
   1   1   0   1   1   1   0
[1,134] =
   0   0   0   0   0   1   0   0
[1,135] =
   0   1   0   0   0   1   0   0
[1,136] =
   1   0   0   1   1   0   1
[1,137] =
   0   0   0   1   0   1   0   1
[1,138] =
   0   0   0   0   0   1   0   1
```

```
   0   0   0   1   0   1   0   0
[1,140] =
   0   0   0   1   0   0   0   1
[1,141] =
   1   1   1   1   1   0   0
[1,142] =
   0   1   0   1   1   0   1   0
[1,143] =
   0   0   1   1   1   1   1   1
[1,144] =
   0   1   0   0   0   1   0   1
[1,145] =
   1   0   0   1   0   1   1   1
[1,146] =
   0   0   1   1   1   1   0   0
[1,147] =
   0   0   1   1   0   1   0   0
[1,148] =
   0   1   1   1   1   1   1   0
[1,149] =
   0   0   0   0   1   0   1   1
[1,150] =
   1   0   1   1   1   0   0   0
[1,151] =
   0   1   0   1   0   0   0   1
[1,152] =
   0   1   0   0   1   1   0   1
[1,153] =
```

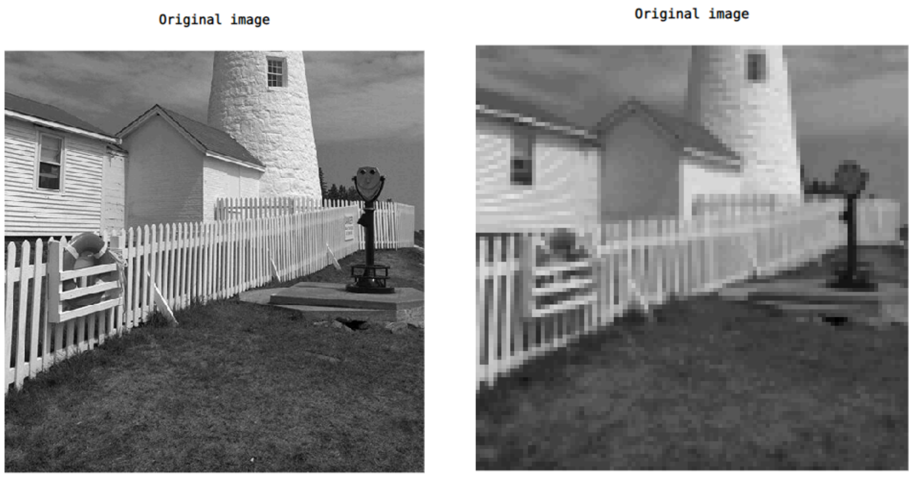
```
[1,156] =
   0   1   1   0   0   1   1   1
[1,157] =
   1   0   0   1   0   1   0   0
[1,158] =
   1   0   0   0   0   0   0   0
[1,159] =
   1   0   0   1   0   0   0   0
[1,160] =
   1   0   0   1   0   0   0   1
[1,161] =
   0   0   1   1   1   1   1   0
[1,162] =
   0   1   1   1   1   1   1   1
[1,163] =
   0   1   0   1   1   1   1   0
[1,164] =
   1   0   1   1   1   1   1   1
[1,165] =
   0   0   1   1   0   1   0   1
[1,166] =
   0   1   0   1   0   0   0   0
[1,167] =
   0   1   0   1   1   0   1   1
[1,168] =
   1   0   0   0   0   1   1   1
[1,169] =
   0   0   0   0   1   0   0   0
```

```
   0   1   1   1   1   0   0   0
[1,171] =
   1   0   0   0   0   0   0   1
[1,172] =
   0   1   1   0   0   1   0   0
[1,173] =
   1   0   1   0   1   0   1   1
[1,174] =
   0   1   1   0   0   0   0   0
[1,175] =
   0   0   0   0   0   0   0   0   1
[1,176] =
   1   1   1   0   0   0   0   1
[1,177] =
   1   1   1   1   0   1   0   0
[1,178] =
   1   0   1   1   0   0   1   1
[1,179] =
   0   0   0   1   1   1   0   0   1
[1,180] =
   1   0   1   0   1   0   0   1
[1,181] =
   0   0   0   1   1   0   0   0   1
[1,182] =
   1   1   0   0   1   1   0   0
[1,183] =
   0   1   0   0   1   1   0   0   0
```

```
[1,242] =
  0  0  1  0  1  0  0  1  1  1  0
[1,243] =
  1  1  1  1  0  0  1  0  0  1  1
[1,244] =
  0  1  1  0  1  0  1  0  0  0
[1,245] =
  0  1  1  0  1  0  1  0  0  1  1
[1,246] =
  1  0  1  1  0  0  1  0  1  0  0
[1,247] =
  1  1  0  0  0  1  1  0  0  1  0
[1,248] =
  0  0  1  0  1  0  0  1  1  0  0  0
[1,249] =
  1  1  0  0  0  1  1  0  0  1  1  0  1
[1,250] =
  0  0  1  0  1  0  0  1  1  0  0  1
[1,251] =
  1  1  0  0  0  1  1  0  0  1  1  0  0  0  0  1
[1,252] =
  1  1  0  0  0  1  1  0  0  1  1  0  0  0  0  0  1
[1,253] =
  1  1  0  0  0  1  1  0  0  1  1  0  0  0  0  0  0  1
[1,254] =
  1  1  0  0  0  1  1  0  0  1  1  0  0  0  0  0  0  0  1
[1,255] =
```

Figure 20: The resulting codewords

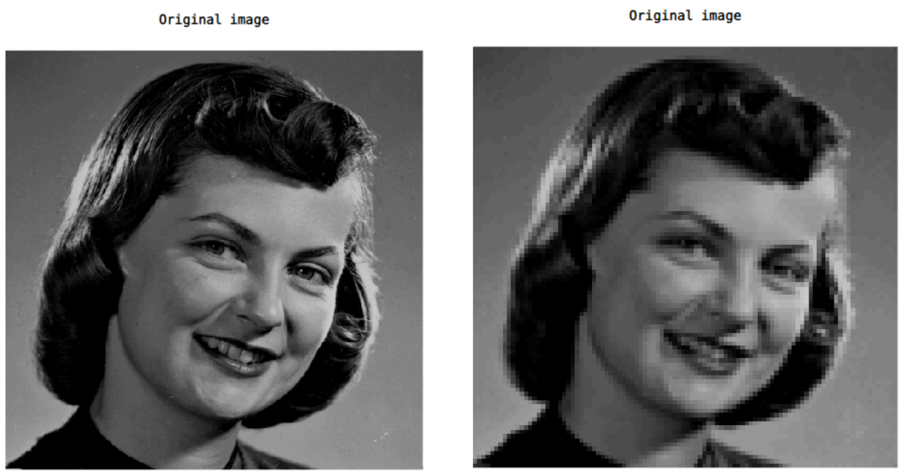
For the image “lighthouse.jpg”:



```
ans = Entropy original image =  7.302893.
ans = Average word length Huffman for original image =  7.331900.
ans = Compression ratio using Huffman =  0.916487.
```

Figure 21: The entropy, average codeword length, and compression ratio for the image “lighthouse.jpg”

For the image “girlface.jpg”:

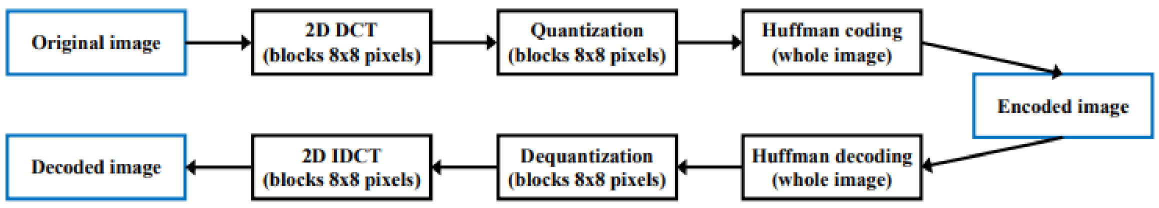


```
ans = Entropy original image =  7.248799.
ans = Average word length Huffman for original image =  7.274500.
ans = Compression ratio using Huffman =  0.909313.
```

Figure 22: The entropy, average codeword length, and compression ratio for the image “girlface.jpg”

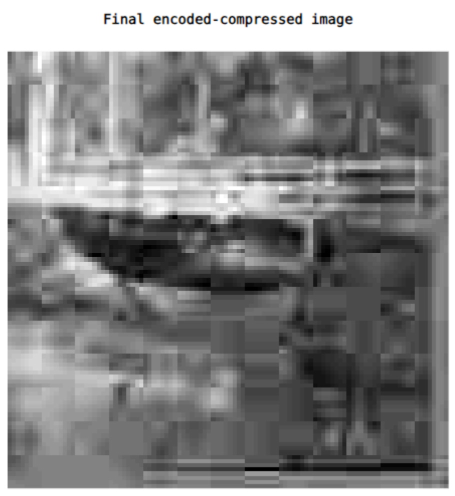
C. For the algorithm's performance, we can say the following: Initially, for all three images, the average Huffman code length is very close to the entropy, meaning the encoding is efficient. Among the images, the “bridge.jpg” image has the best Huffman encoding, as it achieves the highest compression ratio and its average code length is very close to the entropy. Next is the “lighthouse.jpg” image, and finally, the “girlface.jpg” image seems to have the lowest compression ratio, although the difference is not very large.

Exercise 3 - Image Compression according to the JPEG standard: Consider the grayscale images “bridge.bmp,” “girlface.bmp,” and “lighthouse.bmp.” Also, consider the following variation of the JPEG standard for image compression (in this work, only grayscale images will be considered).

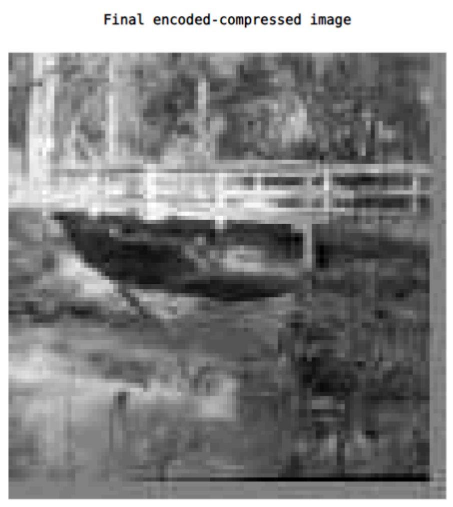

Original image
2D DCT (blocks 8x8 pixels)
Quantization (blocks 8x8 pixels)
Huffman coding (whole image)
Encoded image
Huffman decoding (whole image)
Dequantization (blocks 8x8 pixels)
2D IDCT (blocks 8x8 pixels)
Decoded image

Questions:
A. Implement the image encoder and decoder according to the JPEG standard (as shown in the figure above). Consider two alternative choices for the quantization table (specifically the Q10 and Q50 tables).
B. Encode the images “bridge.bmp,” “girlface.bmp,” and “lighthouse.bmp” (considering both options for the quantization table). Calculate the average codeword length (Huffman algorithm) and the compression ratio achieved for each image in each case.
C. For the images “bridge.bmp,” “girlface.bmp,” and “lighthouse.bmp,” print their decoded versions (considering both options for the quantization table sequentially) and determine the amount of noise introduced (metric Peak Signal-to-Noise Ratio (PSNR)) in each case.

Answers:
A - B - C. For the encoding and decoding, the code was written based on the previous exercise. The images for the two alternative quantization tables are shown below. Regarding the average codeword length and compression ratio, unfortunately, I encountered an error because of the issue with image sizes changing continuously during compression. For the Peak Signal-to-Noise Ratio (PSNR) metric, we observe that the greater the compression, the more noise is introduced, leading to a decrease in image quality. This is why the PSNR values are lower for higher compression levels.

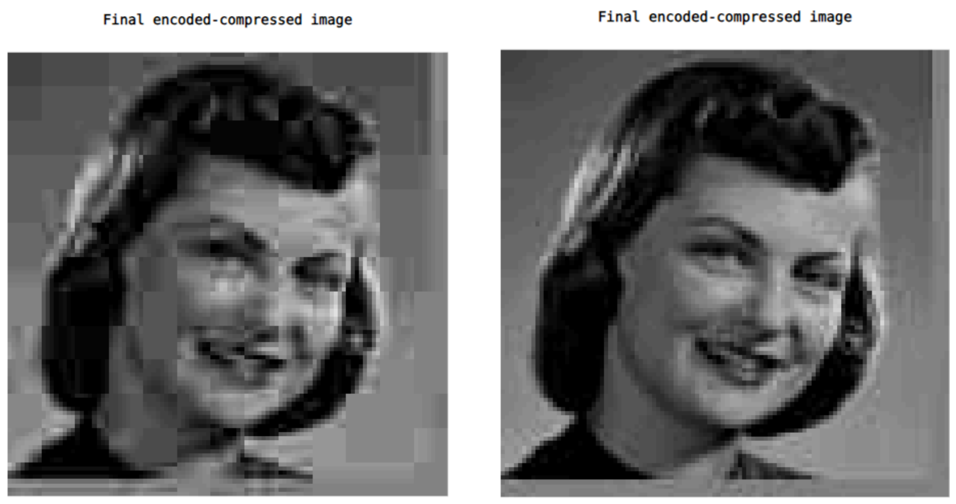


```
PSNR
26.080
```



```
PSNR
30.004
```

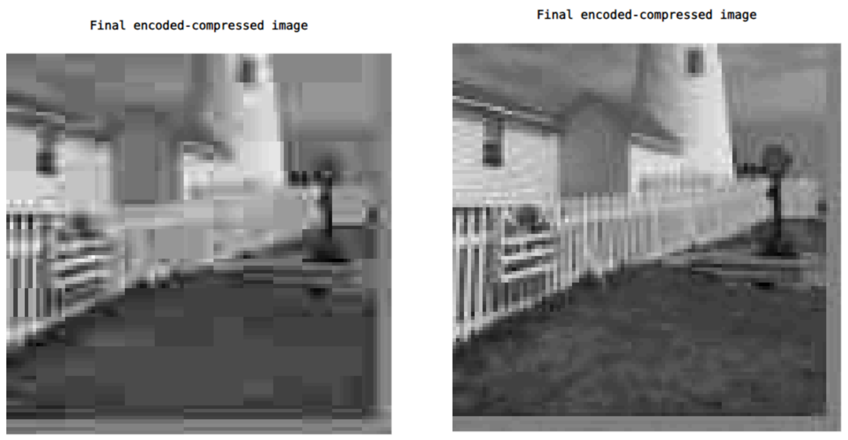
Figure 23: The image “bridge.jpg” after compression with Q10 and Q50 and the corresponding PSNR value


Final encoded-compressed image
Final encoded-compressed image

```
PSNR
30.702
```

```
PSNR
35.859
```

Figure 24: The image "girlface.jpg" after compression with Q10 and Q50 and the corresponding PSNR value

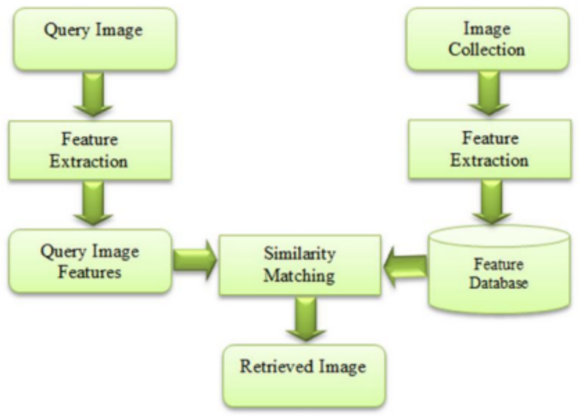

Final encoded-compressed image
Final encoded-compressed image

```
PSNR
27.946
```

```
PSNR
32.910
```

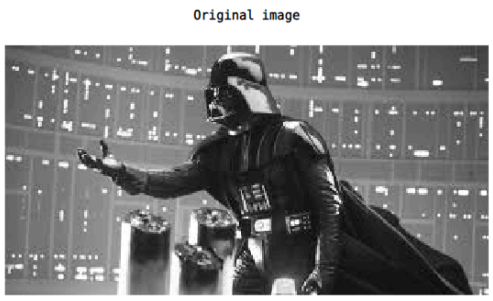
Figure 25: The image "lighthouse.jpg" after compression with Q10 and Q50 and the corresponding PSNR value

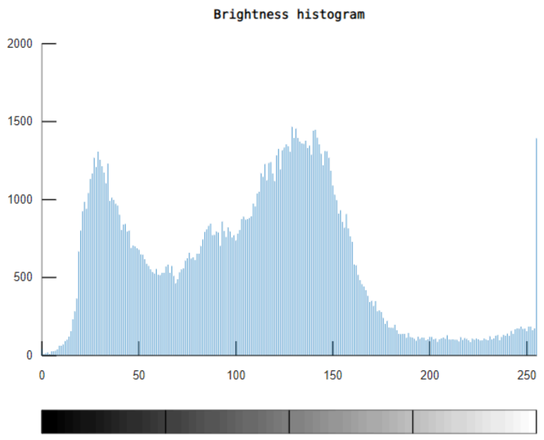
Exercise 4 - Feature Extraction for Image Retrieval: Consider the general architecture of a Content-Based Image Retrieval (CBIR) system, as shown in the figure below. Download the public image dataset 'Villains - Image Classification,' which contains photos of comic characters belonging to the following five categories: a) Darth Vader, b) Green Goblin, c) Joker, d) Thanos, and e) Venom.



Questions:
A. For each image in the dataset (after converting it to grayscale), extract the following features: A1) Normalized 256-bin brightness histogram, A2) Normalized 256-bin Local Binary Pattern (LBP) feature histogram.
B. Implement the following two similarity metrics between two feature vectors f1(i) and f2(i) (smaller values of the metric indicate more "similar" feature vectors): B1) L1 = Σi|f1(i)-f2(i)|, B2) L2 = sqrt(Σi|f1(i) - f2(i)|^2).

Answers:
A. For the normalized brightness histogram and the LBP feature histogram, as we can see from the images below, in the first case, the light and dark regions of the image are identified, while in the LBP histogram, the textures, meaning the variations and patterns in the image, are identified.





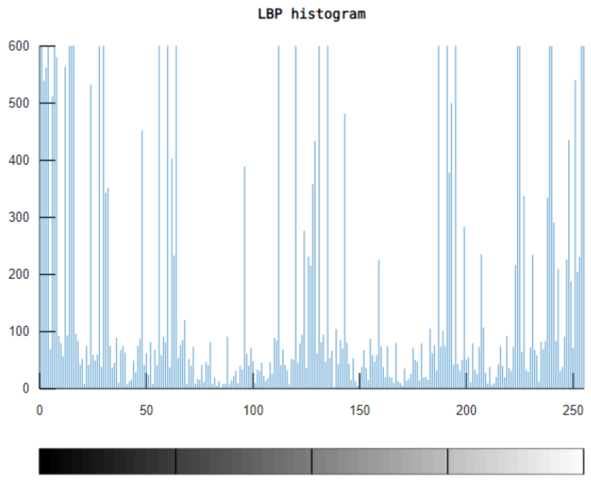


Figure 26: The image with Vader in grayscale and the histogram for the brightness and texture values, respectively

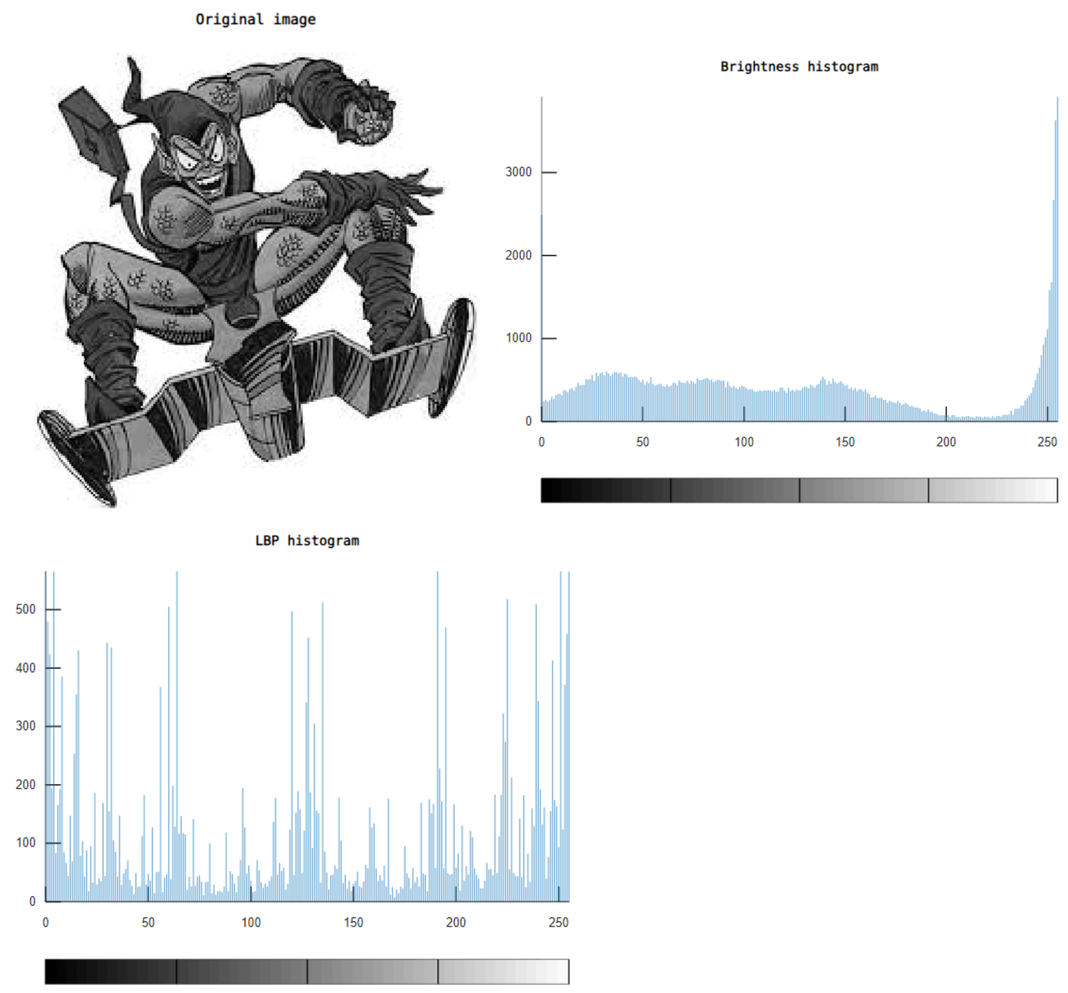


Figure 27: The image with the Green Goblin in grayscale and the histogram for the brightness and texture values, respectively

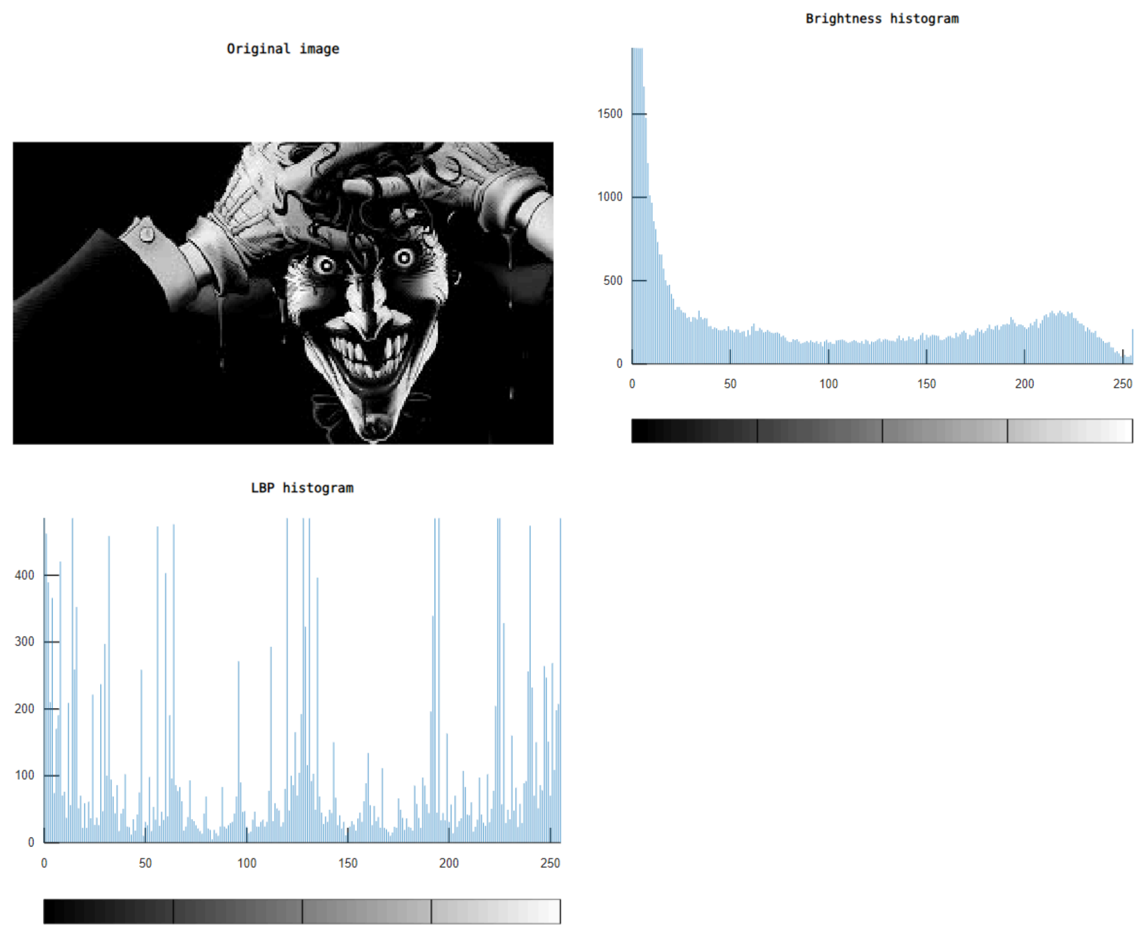


Figure 28: The image with the Joker in grayscale and the histogram for the brightness and texture values, respectively

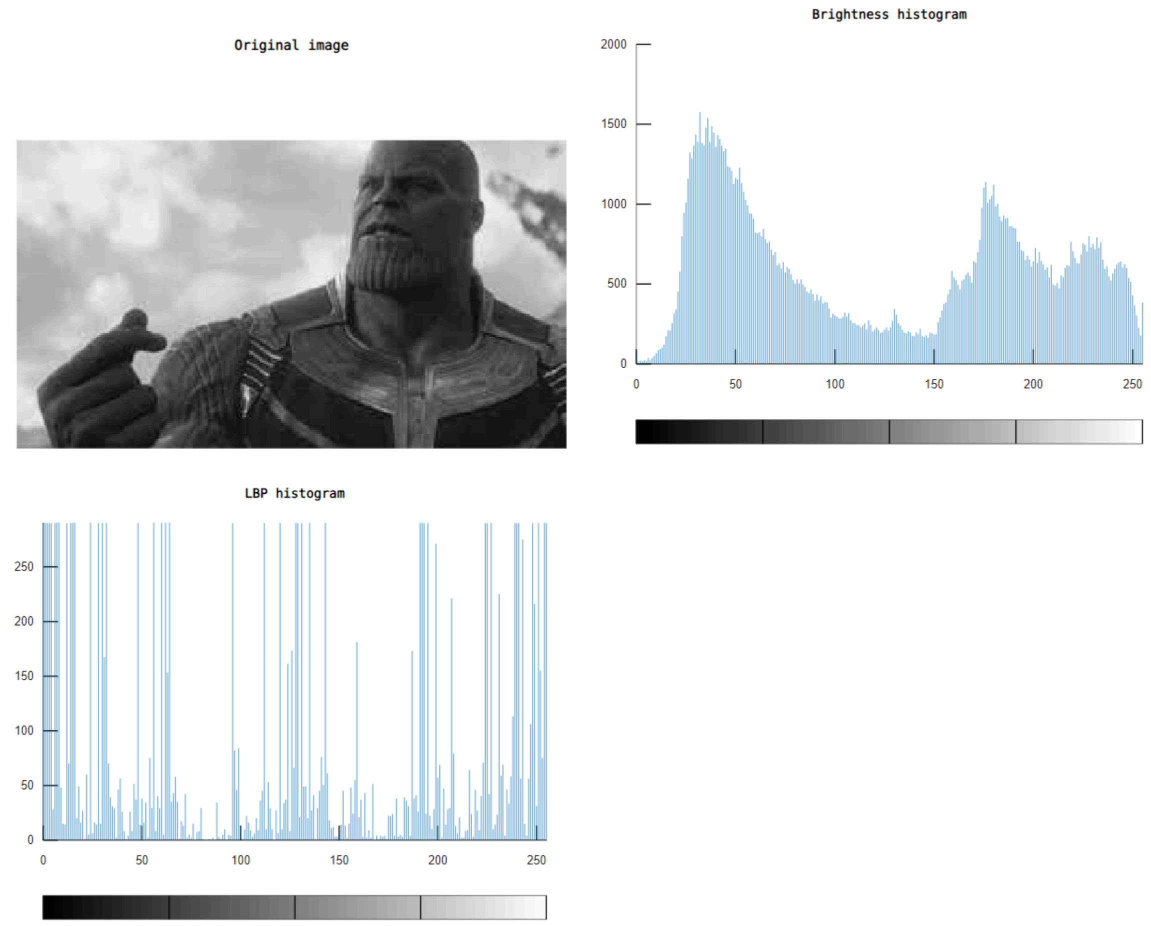
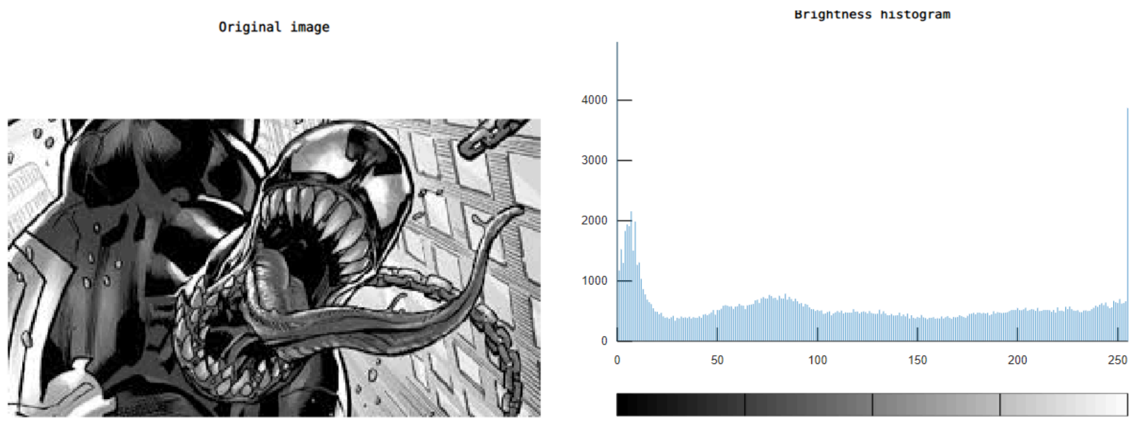


Figure 29: The image with Thanos in grayscale and the histogram for the brightness and texture values, respectively



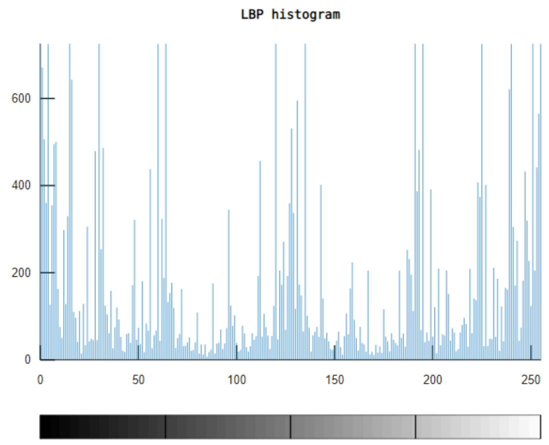
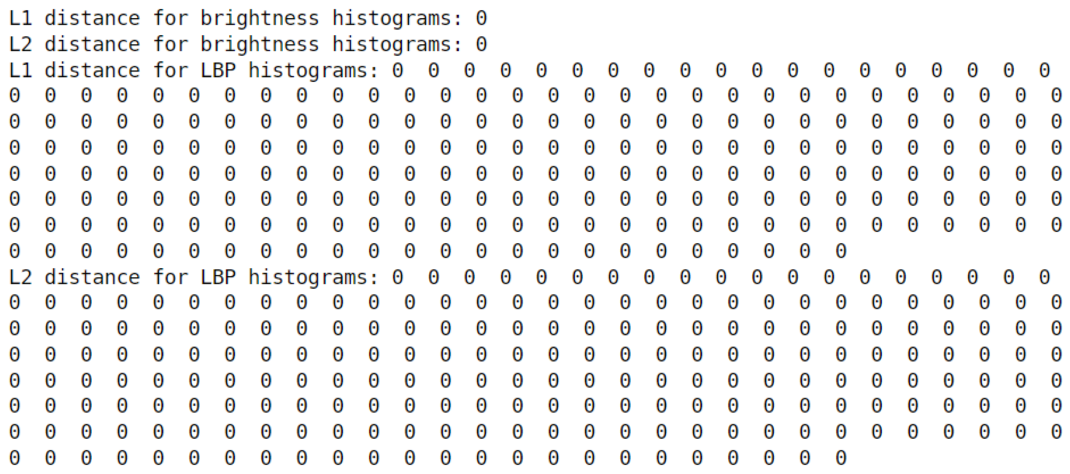


Figure 30: The image with Venom in grayscale and the histogram for the brightness and texture values, respectively

B. For the two similarity metrics between two feature vectors f1(i) and f2(i),
L1 = Σi|f1(i)-f2(i)| and L2 = sqrt(Σi|f1(i) - f2(i)|^2), I tried to write the code, but the vectors came out with zeros. However, the code will be provided in the repository.



```
L1 distance for brightness histograms: 0
L2 distance for brightness histograms: 0
L1 distance for LBP histograms: 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
L2 distance for LBP histograms: 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
```