Digital Image Processing - Lab Session 3

Sofia Noemi Crobeddu
student number: 032410554

International Master of Biometrics and Intelligent Vision
Université Paris-Est Cretéil (UPEC)

November 26, 2024

Contents

Introduction

In this assignment we explore techniques such as stretching or equalization that modifies the histogram of an image. We also deepen the thresholding technique, in order to perform segmentation on gray scale images. Finally, we see labeling of connected components to numerate and detect objects in an image.

1 Histogram

To start this section, we calculate some relevant statistical summaries:

- **Mean**, measured as the sum of the pixels values divided by the number of total pixel, i.e.

$$\mu = \frac{1}{M \cdot N} \sum_{i=1}^{M} \sum_{j=1}^{N} I(i,j)$$

  where $I(i,j)$ is the pixel value in position (i,j), M is the number of rows and N of columns of the image;
- **Standard deviation**, measured as the spread of the values of the pixels in respect to the mean, i.e.

$$\sigma = \sqrt{\frac{1}{M \cdot N} \sum_{i=1}^{M} \sum_{j=1}^{N} (I(i,j) - \mu)^2}$$

  where μ is the mean;
- **Median**, measured as the central value of the pixels ordered based on their intensity, i.e.

$$\text{Median} = \begin{cases} V\left[\frac{n}{2}\right] & \text{if } n \text{ is odd,} \\ \frac{V\left[\frac{n}{2}\right] + V\left[\frac{n}{2}+1\right]}{2} & \text{if } n \text{ is even} \end{cases}$$

  where V is the vector of ordered pixels values and $n = M \cdot N$ is the total number of pixels in the image.

For this first step, since it's not specified which image to use, we can use *Chro* image that will be needed for the successive exercise. The code to perform this analysis it's shown below, and as you can see, we can use the functions from *Numpy* library to obtain these summaries. The results obtained are: a mean of 112.94 (this suggests a moderate brightness), a standard deviation of 32.53 and a median of 100.

```
import numpy as np
from PIL import Image

image_chro = Image.open("C:\\Users\\sofyc\\OneDrive\\Desktop\\UPEC\\Pattern recognition\\
    assignment 3 - IP\\IP3\\IP3\\chro.bmp")
#To array
image_array_chro = np.array(image_chro)

#Since mean2 and std2 are functions from Matlab, we can use here some similar functions from Numpy
    .
#Mean
mean_chro = np.mean(image_array_chro)
#Standard deviation
std_chro = np.std(image_array_chro)
#Median
median_chro = np.median(image_array_chro)
```

Going on with the task, first we visualize the histogram of the *Chro* image, that represents the intensity distribution of pixels. We analyze the histogram with 64 and 16 bins, as written in the code below. As you can see from Figure 1, the first one with more bins has a more detailed distribution, compared to the second one that aggregates more values in a single bin.

```
import matplotlib.pyplot as plt

#Plot the histogram
plt.figure(figsize=(10, 4))

#Histogram with 64 bins
plt.subplot(1, 2, 1)
plt.hist(image_array_chro.ravel(), bins=64, color='dodgerblue', alpha=0.9)
plt.title("Histogram with 64 bins.")
plt.xlabel("Pixel Intensity")
plt.ylabel("Frequency")

#Histogram with 16 bins
plt.subplot(1, 2, 2)
plt.hist(image_array_chro.ravel(), bins=16, color='coral', alpha=0.9)
plt.title("Histogram with 16 bins.")
plt.xlabel("Pixel Intensity")
plt.ylabel("Frequency")

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

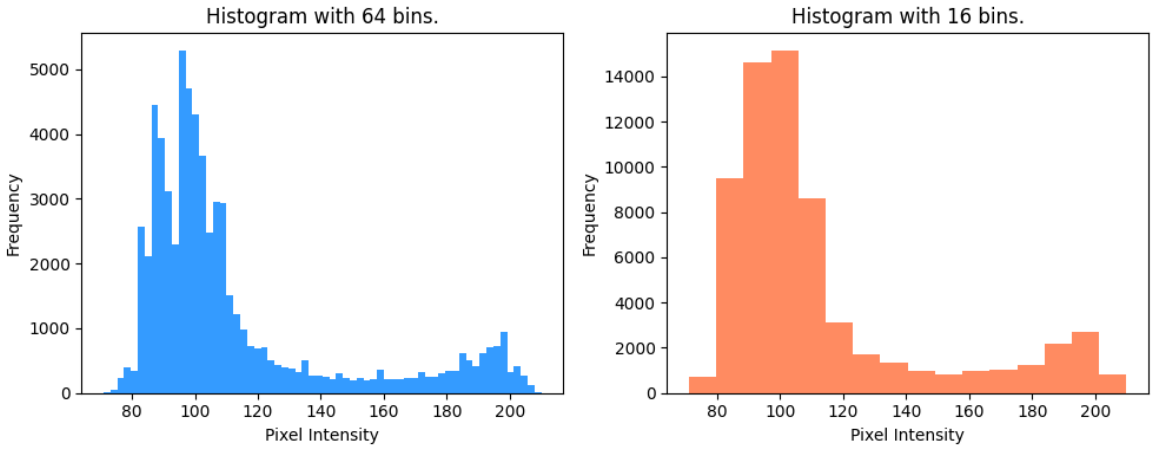

Fig. 1: Histograms of *chro* image.

We also calculate the cumulative histogram, useful also to determine the median, that is the first value right after the half of the number of total pixels. The code is shown below and the result is shown in Figure 2.

```
#Cumulative histogram
hist, bins = np.histogram(image_array_chro.ravel(), bins=256, range=(0, 256))
cumulative_hist = np.cumsum(hist)

#Plot
plt.figure(figsize=(6, 4))
#Plot of the bins
plt.bar(
    bins[:-1],
    cumulative_hist,
    width=2,  #Width of a bin
    color='gold',
    edgecolor='gray',
    alpha=0.5,
    label='Cumulative Bars'
)

#Cumulative function line
plt.plot(
    bins[:-1], cumulative_hist,
    color='red',
    linewidth=2,
    label='Cumulative Line'
)
plt.xlabel("Pixel Intensity")
plt.ylabel("Cumulative Frequency")
```

```
plt.legend()

plt.show()
```

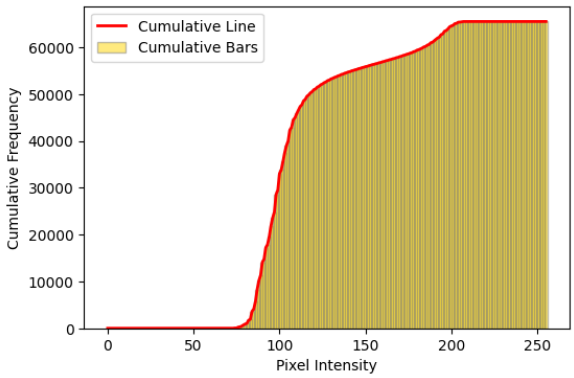

Fig. 2: Cumulative histogram of *chro* image.

```
#Obtaining the median value from the cumulative histogram
total_pixels = cumulative_hist[-1]  #Total pixels
median_value = np.where(cumulative_hist >= total_pixels / 2)[0][0]
```

If we look instead, at the histogram obtained from the rotation of the image, we can see (Figure 3) that the histogram is the same, since the intensity of the pixels is not modified.

```
#Image rotation of 90 degrees
rotated_chro = image_chro.rotate(90)
rotated_array_chro = np.array(rotated_chro)

#New histogram
plt.figure(figsize=(6, 4))
plt.hist(rotated_array_chro.ravel(), bins=64, color='forestgreen', alpha=0.9)
plt.xlabel("Pixel Intensity")
plt.ylabel("Frequency")
plt.show()
```

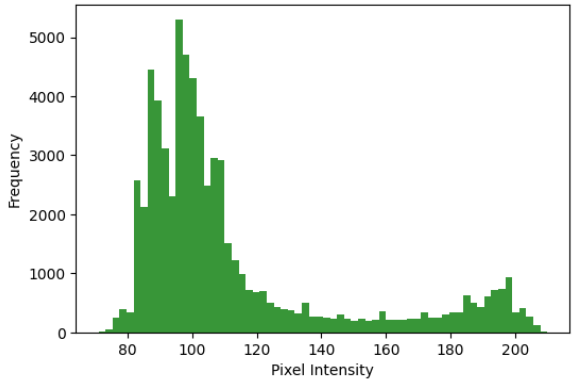

Fig. 3: Histogram of rotated *chro* image.

Since *incomplement* is a function form MatLab, in Python we can use *invert* that implements the same procedure:

$$I_{invertito}(i,j) = 255 - I(i,j)$$

where $I(i,j)$ is the intensity value of the pixel (i,j).
The result, shown in Figure 4, is the perfect opposite, horizontally speaking, of the original histogram.

```
from PIL import ImageOps

#Modifying the image with imcomplement function: it inverts the values of the pixels, transforming
     the bright pizels into dark ones and viceversa.
inverted_chro = ImageOps.invert(image_chro)
inverted_array_chro = np.array(inverted_chro)

#New histogram
plt.figure(figsize=(6, 4))
plt.hist(inverted_array_chro.ravel(), bins=64, color='limegreen', alpha=0.8)
plt.xlabel("Pixel Intensity")
plt.ylabel("Frequency")
plt.show()
```

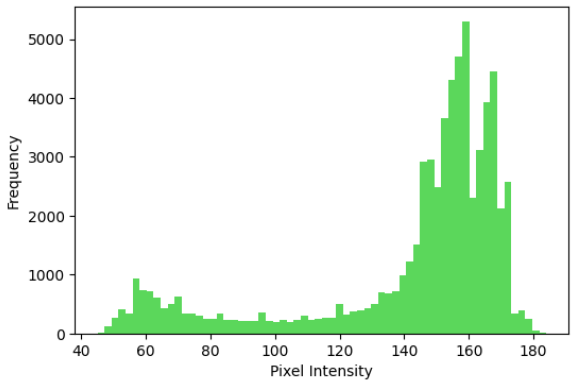

Fig. 4: Histogram of inverted *chro* image.

The next step is **histogram stretching**, a technique that increases the contrast of the image, using the entire range of the pixels (0-255 for images of 8 bits). To build the manual function of stretching, it was used the following formula:

$$I(m,n) = (I_{max} - I_{min}) * (\frac{I_{Original}(m,n) - I_{original\_min}}{I_{original\_max} - I_{original\_min}}) + I_{min}$$

where I_{min} is 0 and I_{max} is 255, $I_{original\_max}$ and $I_{original\_min}$ are the maximum and the minimum pixel value in the image. For the automatic version, it was used directly the function *rescale_intensity* from *skimage* library, that normalizes pixel intensity inside of an interval. The code to implement this is written below, and the stretched images obtained are shown in Figure 5. As we can see, the manual and the automatic method give the same result.

```
from matplotlib.colors import NoNorm
from skimage import exposure

#Function for histogram stretching
def stretching(img):
    ratio = (255-0)/(np.max(image_array_chro) - np.min(image_array_chro))
    s = (img - np.min(img)) * ratio + 0
    return s

#Manual method
stretched_chro = stretching(image_array_chro)

#Skimage method
stretched_skimage = exposure.rescale_intensity(image_array_chro, in_range='image', out_range=(0,
     255))

#Plot gray scale
plt.figure(figsize=(15, 5))
plt.subplot(1, 3, 1)
plt.imshow(image_chro, cmap='gray', norm=NoNorm())
plt.title("Chro image.")
plt.axis("off")

plt.subplot(1, 3, 2)
```

```
plt.imshow(stretched_chro, cmap='gray')
plt.title("Stretched image - manual method.")
plt.axis("off")

plt.subplot(1, 3, 3)
plt.imshow(stretched_skimage, cmap='gray')
plt.title("Stretched image - skimage method.")
plt.axis("off")

plt.show()
```

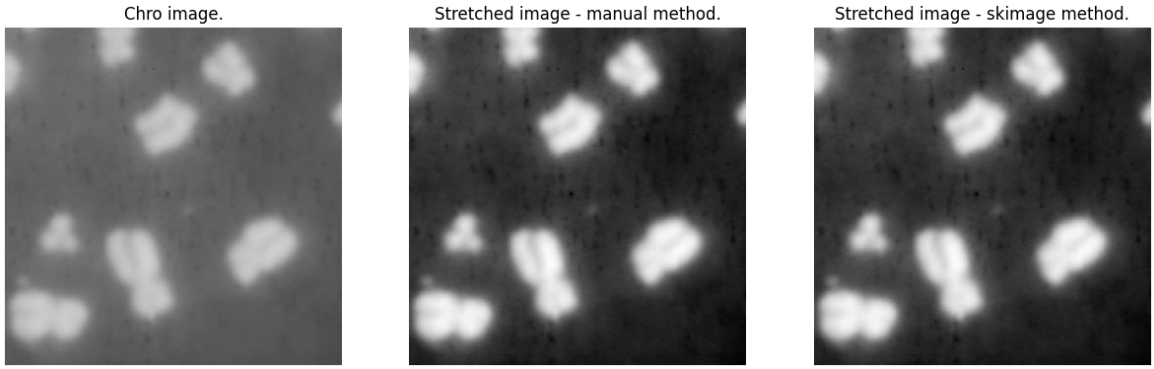

Fig. 5: Stretching on *chro* image.

After the image visualization, we compare also the histograms, as shown below. From Figure 6, the histogram of the original image has pixels in the range [71, 210], while the stretched ones in [0, 255].

```
#Histograms in the three cases
plt.figure(figsize=(20, 5))

#Original image
plt.subplot(1, 3, 1)
plt.hist(image_array_chro.ravel(), bins=64, color='gold', alpha=0.8)
plt.title("Histogram chro image - original.")

#Stretched image - manual method
plt.subplot(1, 3, 2)
plt.hist(stretched_chro.ravel(), bins=64, color='orange', alpha=0.8)
plt.title("Histogram stretched image - manual method.")

#Stretched image - skimage method
plt.subplot(1, 3, 3)
plt.hist(stretched_skimage.ravel(), bins=64, color='orange', alpha=0.8)
plt.title("Histogram stretched image - skimage method.")

plt.show()
```

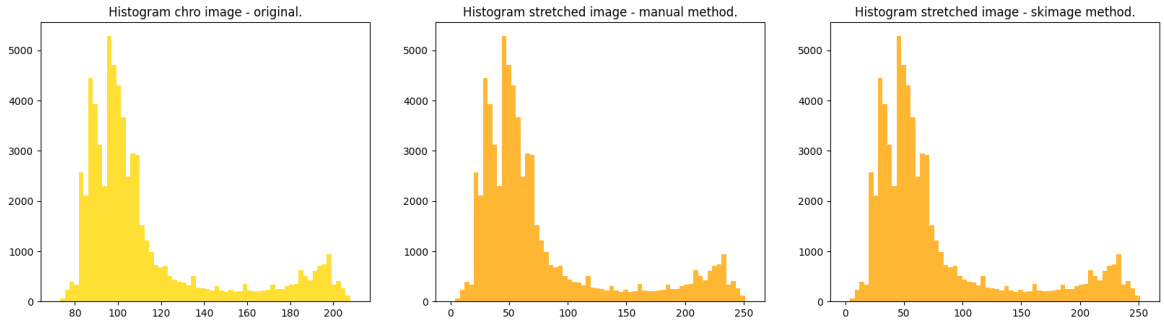

Fig. 6: Histogram of stretched *chro* image.

In the last step of this section we analyze the **Histogram equalization**. It is a technique that improves the contrast quality of an image to make the pixel intensity distribution be uniform. The aim it is improve the quality without losing details and information.
The first group of images where we analyze this technique, regards various modified versions of *Elaine* image. The code to load the image is shown below, and they are represented in Figure 7. In this case, to visualize them, we use *NoNorm()* to avoid the normalization before plotting.

```
#Loading the images
image_elaine = Image.open("C:\\Users\\sofyc\OneDrive\\Desktop\\UPEC\\Pattern recognition\\
    assignment 3 - IP\\IP3\\IP3\\elaine.jpg")
image_elaineContrasted = Image.open("C:\\Users\\sofyc\\OneDrive\\Desktop\\UPEC\\Pattern
    recognition\\assignment 3 - IP\\IP3\\IP3\\elaineContrasted.jpg")
image_elaineDark = Image.open("C:\\Users\\sofyc\OneDrive\\Desktop\\UPEC\\Pattern recognition\\
    assignment 3 - IP\\IP3\\IP3\\elaineDark.jpg")
image_elaineLight = Image.open("C:\\Users\\sofyc\\OneDrive\\Desktop\\UPEC\Pattern recognition\\
    assignment 3 - IP\\IP3\\IP3\\elaineLight.jpg")
image_elaineLowContrasted = Image.open("C:\\Users\\sofyc\OneDrive\\Desktop\\UPEC\\Pattern
    recognition\\assignment 3 - IP\\IP3\\IP3\\elaineLowContrasted.jpg")

#Plots
plt.figure(figsize=(20, 6))
plt.subplot(1, 5, 1)
plt.imshow(image_elaine, cmap='gray', norm=NoNorm())
plt.title("Elaine image.")
plt.axis("off")

plt.subplot(1,5, 2)
plt.imshow(image_elaineContrasted, cmap='gray', norm=NoNorm())
plt.title("ElaineContrasted image.")
plt.axis("off")

plt.subplot(1, 5, 3)
plt.imshow(image_elaineDark, cmap='gray', norm=NoNorm())
plt.title("ElaineDark image.")
plt.axis("off")

plt.subplot(1, 5, 4)
plt.imshow(image_elaineLight, cmap='gray', norm=NoNorm())
plt.title("ElaineLight image.")
plt.axis("off")

plt.subplot(1, 5, 5)
plt.imshow(image_elaineLowContrasted, cmap='gray', norm=NoNorm())
plt.title("ElaineLowContrasted image.")
plt.axis("off")
```

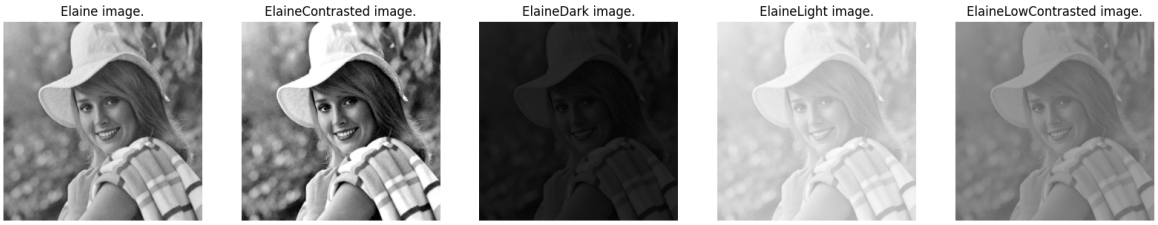

Fig. 7: *Elaine* images (different versions).

Through the function *equalize_hist*, we can apply this technique and the resulting images are in Figure 8. As we can notice, all the equalized images are very similar even if the starting versions are different, since the technique creates uniformity.

```
#To array
image_array_elaine = np.array(image_elaine)
image_array_Contrasted = np.array(image_elaineContrasted)
image_array_Dark = np.array(image_elaineDark)
image_array_Light = np.array(image_elaineLight)
image_array_LowContrasted = np.array(image_elaineLowContrasted)

#Applying the function
equalized_elaine = exposure.equalize_hist(image_array_elaine)
equalized_elaineContrasted = exposure.equalize_hist(image_array_Contrasted)
```

```
equalized_elaineDark = exposure.equalize_hist(image_array_Dark)
equalized_elaineLight = exposure.equalize_hist(image_array_Light)
equalized_elaineLowContrasted = exposure.equalize_hist(image_array_LowContrasted)

#Plots
plt.figure(figsize=(20, 6))
plt.subplot(1, 5, 1)
plt.imshow(equalized_elaine, cmap='gray', norm=NoNorm())
plt.title("Eq. Elaine image.")
plt.axis("off")

plt.subplot(1,5, 2)
plt.imshow(equalized_elaineContrasted, cmap='gray', norm=NoNorm())
plt.title("Eq. ElaineContrasted image.")
plt.axis("off")

plt.subplot(1, 5, 3)
plt.imshow(equalized_elaineDark, cmap='gray', norm=NoNorm())
plt.title("Eq. ElaineDark image.")
plt.axis("off")

plt.subplot(1, 5, 4)
plt.imshow(equalized_elaineLight, cmap='gray', norm=NoNorm())
plt.title("Eq. ElaineLight image.")
plt.axis("off")

plt.subplot(1, 5, 5)
plt.imshow(equalized_elaineLowContrasted, cmap='gray', norm=NoNorm())
plt.title("Eq. ElaineLowContrasted image.")
plt.axis("off")
```

Fig. 8: Equalized *Elaine* images.

For a final check, we also compare the means and the standard deviations before and after the application of equalization. As we can see from Table 1, the starting images had very different means (for example *ElaineDark* of 22.5 while *ElaineLight* of 214.6), but after the application they all reach a mean around 0.5.

```
import pandas as pd

#Check differences on the means
mean_el = np.mean(image_elaine)
mean_Contr = np.mean(image_elaineContrasted)
mean_Dark = np.mean(image_elaineDark)
mean_Light = np.mean(image_elaineLight)
mean_LowContr = np.mean(image_elaineLowContrasted)

mean_eq_el = np.mean(equalized_elaine)
mean_eq_Contr = np.mean(equalized_elaineContrasted)
mean_eq_Dark = np.mean(equalized_elaineDark)
mean_eq_Light = np.mean(equalized_elaineLight)
mean_eq_LowContr = np.mean(equalized_elaineLowContrasted)

#Take a look
mean_res = {
    'Image': ['Elaine', 'ElaineContrasted', 'ElaineDark', 'ElaineLight', 'ElaineLowContrasted'],
    'Mean image': [mean_el, mean_Contr, mean_Dark, mean_Light, mean_LowContr],
    'Mean Equalized Image': [mean_eq_el, mean_eq_Contr, mean_eq_Dark, mean_eq_Light,
     mean_eq_LowContr]
}

mean_res_df = pd.DataFrame(mean_res)
mean_res_df
```

Image	Mean Image	Mean Equalized Image
Elaine	136.354843	0.503001
ElaineContrasted	139.455868	0.504514
ElaineDark	22.487926	0.517779
ElaineLight	214.560787	0.508761
ElaineLowContrasted	130.402126	0.509144

Table 1: Mean comparison between *Elaine* images and equalized versions.

Below, you can also find the same analysis of before for the standard deviation. The results are shown in Table 2 and, as we can see, the standard deviation is around 0.29 for all the images.

```
#Check differences on the standard deviation
std_el = np.std(image_elaine)
std_Contr = np.std(image_elaineContrasted)
std_Dark = np.std(image_elaineDark)
std_Light = np.std(image_elaineLight)
std_LowContr = np.std(image_elaineLowContrasted)

std_eq_el = np.std(equalized_elaine)
std_eq_Contr = np.std(equalized_elaineContrasted)
std_eq_Dark = np.std(equalized_elaineDark)
std_eq_Light = np.std(equalized_elaineLight)
std_eq_LowContr = np.std(equalized_elaineLowContrasted)

#Take a look
std_res = {
    'Image': ['Elaine', 'ElaineContrasted', 'ElaineDark', 'ElaineLight', 'ElaineLowContrasted'],
    'Std. image': [std_el, std_Contr, std_Dark, std_Light, std_LowContr],
    'Std. Equalized Image': [std_eq_el, std_eq_Contr, std_eq_Dark, std_eq_Light, std_eq_LowContr]
}

std_res_df = pd.DataFrame(std_res)
std_res_df
```

Image	Std. image	Std. Equalized Image
Elaine	45.921128	0.288613
ElaineContrasted	65.183496	0.291405
ElaineDark	7.757789	0.288155
ElaineLight	15.856192	0.288465
ElaineLowContrasted	15.068619	0.288470

Table 2: Standard deviation comparison between *Elaine* images and equalized versions.

We apply the same technique on the images *Neck* and *Covering*. The effect is the same as before: it reduced the contrast in the darker areas, and increased it in the parts brighter. The results are shown in Figure 9 and 10.

```
#Load image
image_neck = Image.open("C:\\Users\\sofyc\OneDrive\\Desktop\\UPEC\\Pattern recognition\\assignment
    3 - IP\\IP3\\IP3\\neck.bmp")

#Plot
plt.figure(figsize=(10, 6))
plt.subplot(1, 2, 1)
plt.imshow(image_neck, cmap='gray', norm=NoNorm())
plt.title("Neck image.")
plt.axis("off")

#To array
image_array_neck = np.array(image_neck)
#Equalized image
equalized_neck = exposure.equalize_hist(image_array_neck)

#Plot
plt.subplot(1, 2, 2)
plt.imshow(equalized_neck, cmap='gray', norm=NoNorm())
plt.title("Eq. Neck image.")
plt.axis("off")
```

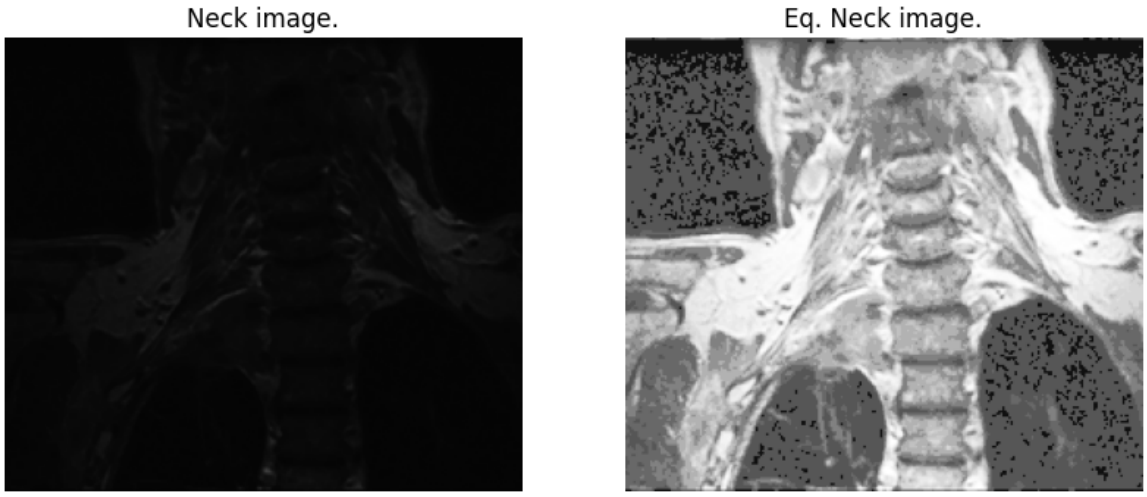

Fig. 9: *Neck* image and equalized version.

```
#Load image
image_covering = Image.open("C:\\Users\\sofyc\OneDrive\\Desktop\\UPEC\\Pattern recognition\\
    assignment 3 - IP\\IP3\\IP3\\covering.bmp")

#Plot
plt.figure(figsize=(10, 6))
plt.subplot(1, 2, 1)
plt.imshow(image_covering, cmap='gray', norm=NoNorm())
plt.title("Covering image.")
plt.axis("off")

#To array
image_array_covering = np.array(image_covering)
#Equalized image
equalized_covering = exposure.equalize_hist(image_array_covering)

#Plot
plt.subplot(1, 2, 2)
plt.imshow(equalized_covering, cmap='gray', norm=NoNorm())
plt.title("Eq. Covering image.")
plt.axis("off")
```

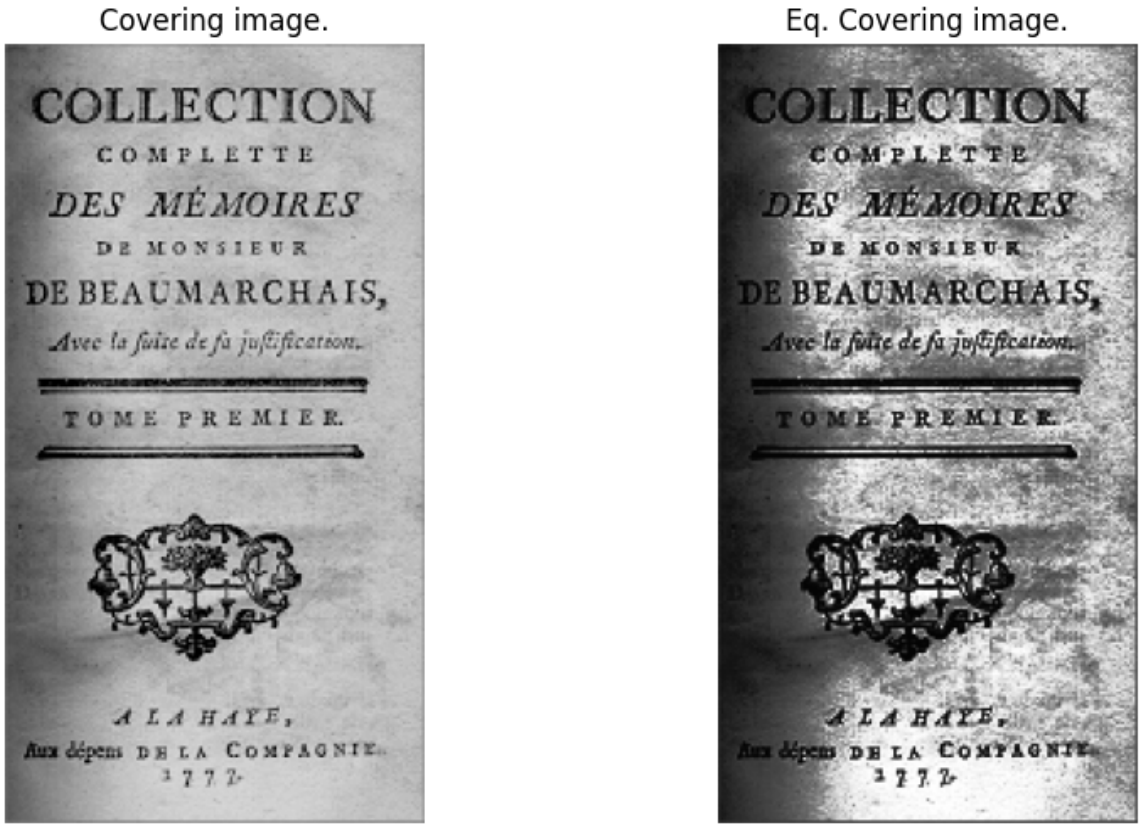

Fig. 10: *Covering* image and equalized version.

The last application regards instead colored images, *Rocks* and *Umbrella*. in both cases, we divided the RGB channels, and we applied separately the equalization. After this, we combined again them in order to form the final equalized image. The visual effect is very evident for *Rocks* image since it was quite dark, and it is shown in Figures 11 and 12.

```
#Loading image
image_rocks = Image.open("C:\\Users\\sofyc\\OneDrive\\Desktop\\UPEC\\Pattern recognition\\
    assignment 3 - IP\\IP3\\IP3\\rocks.jpg")

#To array
image_array_rocks = np.array(image_rocks)

#Split of the RGB channels
R = image_array_rocks[:, :, 0]
G = image_array_rocks[:, :, 1]
B = image_array_rocks[:, :, 2]

#Equalization for each channel
R_eq_rocks = exposure.equalize_hist(R)
G_eq_rocks = exposure.equalize_hist(G)
B_eq_rocks = exposure.equalize_hist(B)

#Combining the equalized RGB channels
#We basically convert the values from [0,1] to [0,255]
equalized_rocks = np.stack([
    (R_eq_rocks * 255).astype(np.uint8),
    (G_eq_rocks * 255).astype(np.uint8),
    (B_eq_rocks * 255).astype(np.uint8)
], axis=-1)

#Plot
plt.figure(figsize=(10, 6))
plt.subplot(1, 2, 1)
plt.imshow(image_rocks)
plt.title("Rocks image.")
plt.axis("off")
```

```
31
32 #Plot
33 plt.subplot(1, 2, 2)
34 plt.imshow(equalized_rocks)
35 plt.title("Eq. Rocks image.")
36 plt.axis("off")
```

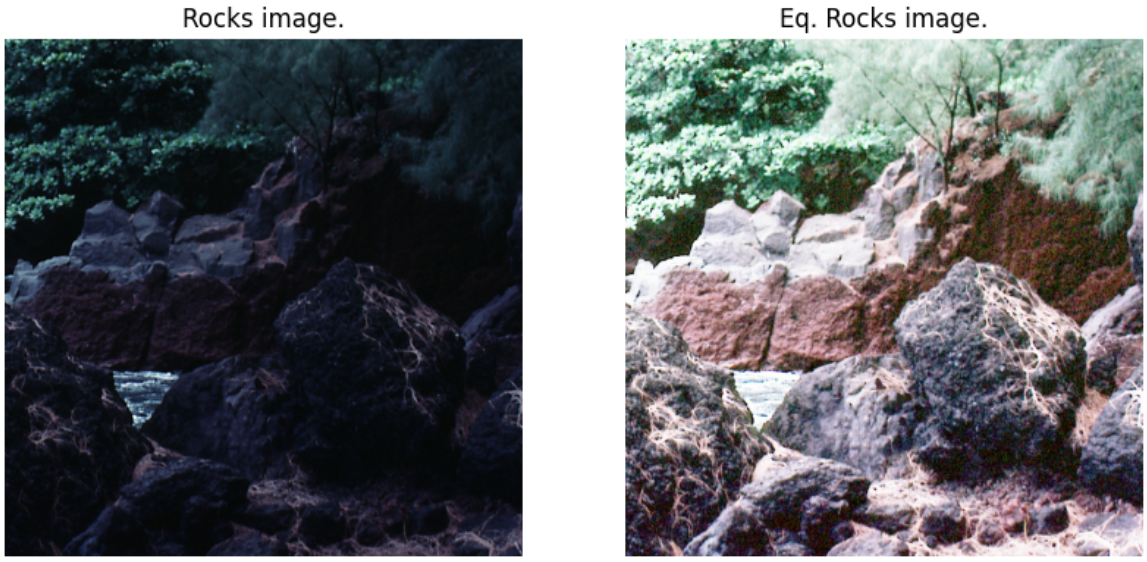

Fig. 11: *Rocks* image and equalized version.

```
1  #Loading image
2  image_umbrella = Image.open("C:\\Users\\sofyc\\OneDrive\\Desktop\\UPEC\\Pattern recognition\\
       assignment 1 - IP\\IP1\\IP1\\umbrella.jpg")
3
4  #To array
5  image_array_umbrella = np.array(image_umbrella)
6
7  #Split of the RGB channels
8  R = image_array_umbrella[:, :, 0]
9  G = image_array_umbrella[:, :, 1]
10 B = image_array_umbrella[:, :, 2]
11
12 #Equalization for each channel
13 R_eq_umbrella = exposure.equalize_hist(R)
14 G_eq_umbrella = exposure.equalize_hist(G)
15 B_eq_umbrella = exposure.equalize_hist(B)
16
17 #Combining the equalized RGB channels
18 #We basically convert the values from [0,1] to [0,255]
19 equalized_umbrella = np.stack([
20     (R_eq_umbrella * 255).astype(np.uint8),
21     (G_eq_umbrella * 255).astype(np.uint8),
22     (B_eq_umbrella * 255).astype(np.uint8)
23 ], axis=-1)
24
25 #Plot
26 plt.figure(figsize=(10, 6))
27 plt.subplot(1, 2, 1)
28 plt.imshow(image_umbrella)
29 plt.title("Umbrella image.")
30 plt.axis("off")
31
32 #Plot
33 plt.subplot(1, 2, 2)
34 plt.imshow(equalized_umbrella)
35 plt.title("Eq. Umbrella image.")
36 plt.axis("off")
```

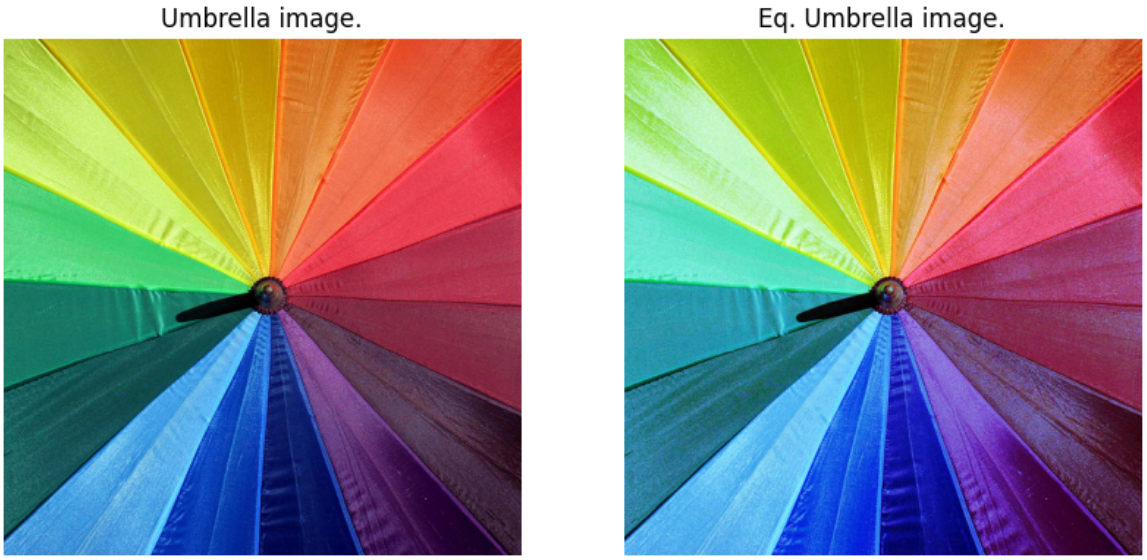

Fig. 12: *Umbrella* image and equalized version.

2 Thresholding

In this section we analyze some **thresholding** techniques, i.e. techniques for segmentation.
In this first step, we consider the images *Gdr* and *Objects*, applying the manual version of this technique. Thresholding helps to separate objects from the background in binary images. For both images, we normalize them into the scale [0,1], and after that, we perform the binarization using a specified threshold: the pixels that exceed it are assigned a value of 1, while for pixels lower than the threshold, the value assigned is 0. As the task suggested, for both images we test various thresholds. For *Gdr* image, as we can see from Figures 13 and 14, the best results are obtained through a threshold of 0.6. The code implemented are shown below, and as you can see, we also used the default value of 0.5 of *im2bw* function in Matlab (if no specified).

```
#Loading image
image_gdr = Image.open("C:\\Users\\sofyc\OneDrive\\Desktop\\UPEC\\Pattern recognition\\assignment 3 - IP\\IP3\\IP3\\gdr.bmp").convert("L") #converting to gray scale

#To array with also the normalization to the scale [0,1]
image_array_gdr = np.array(image_gdr)/255.0

#Thresholding function: the binarization is made putting 1 to pixels > threshold, and 0 to pixels < threshold.
def manual_threshold(img, threshold):
    binary_img = (img > threshold).astype(np.uint8)
    return binary_img

#In the function im2bw of MatLab, the usually default value is 0.5. In this case, we will see the outputs for different values.
#Values for threshold
thresholds = [0.3, 0.35, 0.4, 0.5, 0.6, 0.65, 0.7]

#Plot
#Original image
plt.figure(figsize=(15, 8))
plt.subplot(2, 4, 1)
plt.imshow(image_gdr, cmap='gray', norm=NoNorm())
plt.title("Gdr Image.")
plt.axis("off")
#Thresholded images
for i, thr in enumerate(thresholds):
    binary_img = manual_threshold(image_array_gdr, thr)
    plt.subplot(2, 4, i + 2)
    plt.imshow(binary_img, cmap='gray')
    plt.title(f"Threshold = {thr}.")
    plt.axis("off")

```

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

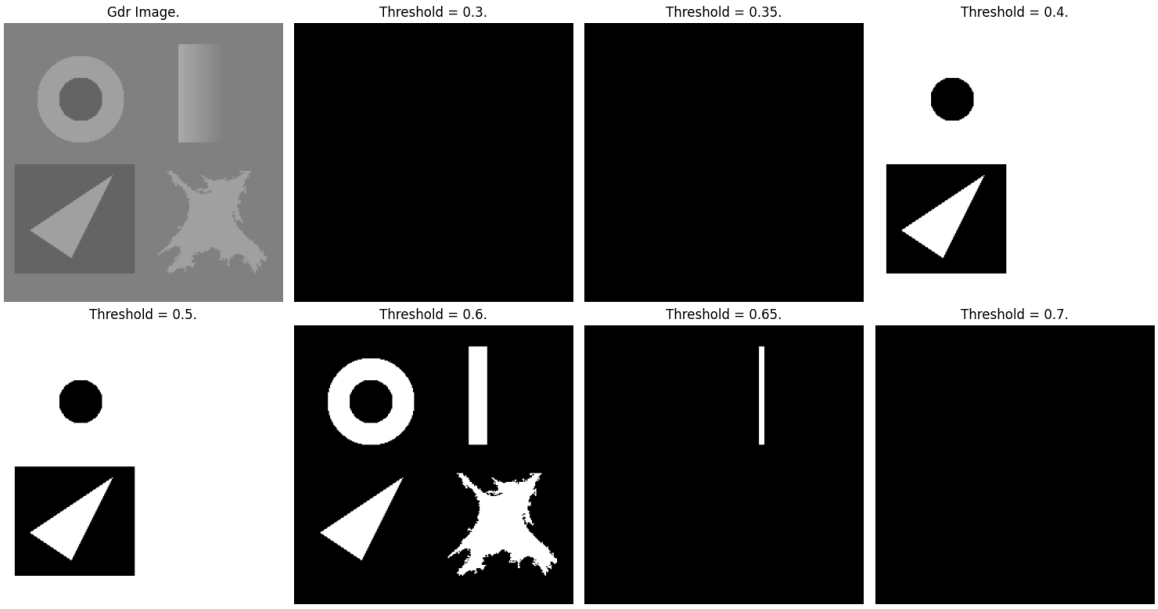

Fig. 13: Manual thresholding on *Gdr* image.

```
#Loading image
image_objects = Image.open("C:\\Users\\sofyc\OneDrive\\Desktop\\UPEC\\Pattern recognition\\
    assignment 3 - IP\\IP3\\IP3\\objects.bmp").convert("L") #converting to gray scale

#To array with also the normalization to the scale [0,1]
image_array_objects = np.array(image_objects) / 255.0

#New values for threshold
thresholds2 = [0.1, 0.3, 0.5, 0.55, 0.575, 0.6, 0.8]

#Plot
#Original image
plt.figure(figsize=(15, 8))
plt.subplot(2, 4, 1)
plt.imshow(image_objects, cmap='gray', norm=NoNorm())
plt.title("Object Image.")
plt.axis("off")
#Thresholded images
for i, thr in enumerate(thresholds2):
    binary_img = manual_threshold(image_array_objects, thr)
    plt.subplot(2, 4, i + 2)
    plt.imshow(binary_img, cmap='gray')
    plt.title(f"Threshold = {thr}.")
    plt.axis("off")

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

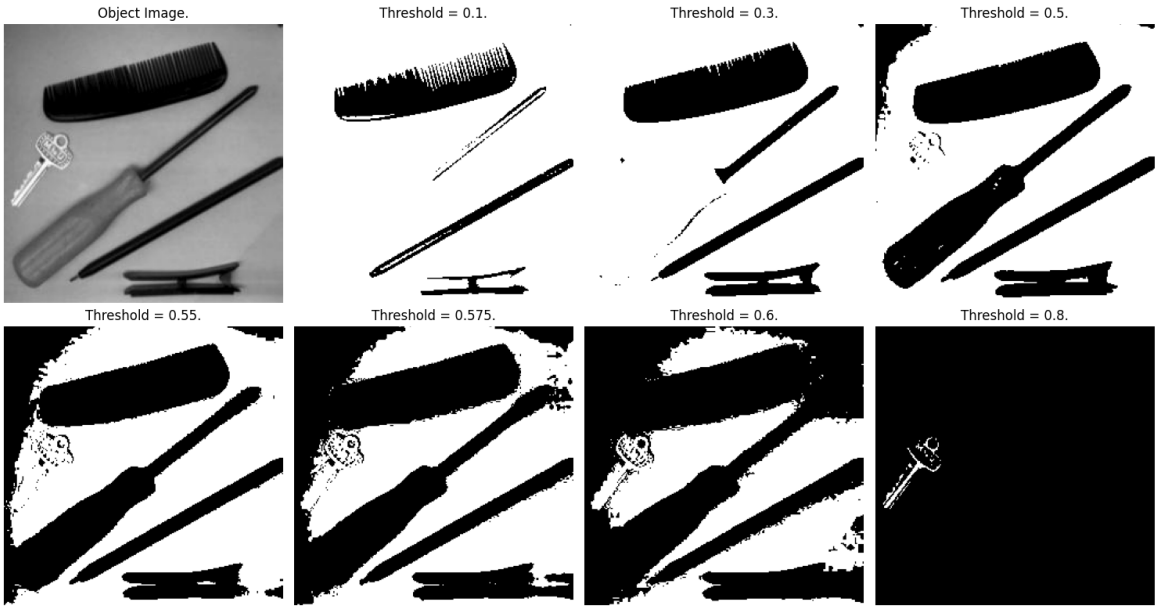

Fig. 14: Manual thresholding on *Objects* image.

In this next step of the second section, after implementing the technique manually, we implement it using the automatic **Otsu's method**. The basic idea is to find the optimal threshold to separate the foreground (objects) from the background in an image. This algorithm, maximizes the variance between classes, such that the separation is made based on their intensities.
We apply this method to a group of ten images: *Gdr*, *Objects*, *Bacteria*, *Cell*, *Chro*, *Gear-Wheel*, *Fibers*, *Fibers2*, *I10*, *I12*. For each image we perform the binarization through Otsu's method, and we also compare the histogram from the original image with the one from the binarized image: as we can see from Figures 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, the thresholding successfully separated the objects from the background, with the correspondent histograms showing a clear separation between the two intensity regions (they have value just on 0 and 1).

```
#Loading the images
image_bacteria = Image.open("C:\\Users\\sofyc\OneDrive\\Desktop\\UPEC\\Pattern recognition\\
    assignment 3 - IP\\IP3\\IP3\\bacteria.jpg").convert("L") #converting to gray scale
image_cell = Image.open("C:\\Users\\sofyc\OneDrive\\Desktop\\UPEC\\Pattern recognition\\assignment
     3 - IP\\IP3\\IP3\\cell.bmp").convert("L") #converting to gray scale
image_gear_wheel = Image.open("C:\\Users\\sofyc\OneDrive\\Desktop\\UPEC\\Pattern recognition\\
    assignment 3 - IP\\IP3\\IP3\\gear-wheel.png").convert("L") #converting to gray scale
image_fibers = Image.open("C:\\Users\\sofyc\OneDrive\\Desktop\\UPEC\\Pattern recognition\\
    assignment 3 - IP\\IP3\\IP3\\fibers.jpg").convert("L") #converting to gray scale
image_fibers2 = Image.open("C:\\Users\\sofyc\OneDrive\\Desktop\\UPEC\\Pattern recognition\\
    assignment 3 - IP\\IP3\\IP3\\fibers2.jpg").convert("L") #converting to gray scale
image_I10 = Image.open("C:\\Users\\sofyc\OneDrive\\Desktop\\UPEC\\Pattern recognition\\assignment
    3 - IP\\IP3\\IP3\\I10.bmp").convert("L") #converting to gray scale
image_I12 = Image.open("C:\\Users\\sofyc\OneDrive\\Desktop\\UPEC\\Pattern recognition\\assignment
    3 - IP\\IP3\\IP3\\I12.bmp").convert("L") #converting to gray scale
image_chro = Image.open("C:\\Users\\sofyc\OneDrive\\Desktop\\UPEC\\Pattern recognition\\assignment
     3 - IP\\IP3\\IP3\\chro.bmp").convert("L") #converting to gray scale

#To array with also the normalization to the scale [0,1]
image_array_bacteria = np.array(image_bacteria) / 255.0
image_array_cell = np.array(image_cell) / 255.0
image_array_gear_wheel = np.array(image_gear_wheel) / 255.0
image_array_fibers = np.array(image_fibers) / 255.0
image_array_fibers2 = np.array(image_fibers2) / 255.0
image_array_I10 = np.array(image_I10) / 255.0
image_array_I12 = np.array(image_I12) / 255.0
image_array_chro = np.array(image_chro) / 255.0
```

```
from skimage.filters import threshold_otsu

#Function for Otsu thresholding
def otsu_method(img):
```

```
    otsu_threshold = threshold_otsu(img)
    #Binarization
    binary_image = (img > otsu_threshold).astype(np.uint8)
    return binary_image, otsu_threshold

#List with all the images
img_list = [image_array_gdr, image_array_objects, image_array_bacteria, image_array_cell,
    image_array_chro, image_array_gear_wheel, image_array_fibers, image_array_fibers2,
    image_array_I10, image_array_I12]
#Names
image_names = ["Gdr", "Objects", "Bacteria", "Cell", "Chro", "Gear-Wheel", "Fibers", "Fibers2", "
    I10", "I12"]

for i, (img, name) in enumerate(zip(img_list, image_names)):
    #Original image
    plt.figure(figsize=(18, 40))
    plt.subplot(10, 4, i * 4 + 1)
    plt.imshow(img, cmap='gray', norm=NoNorm())
    plt.title(f"{name} image.")
    plt.axis("off")

    #Histogram of the original image
    plt.subplot(10, 4, i * 4 + 2)
    plt.hist(img.ravel(), bins=64, color='seagreen', label="Histogram")
    plt.title(f"Histogram of {name} image.")
    plt.xlabel("Pixel Intensity")
    plt.ylabel("Frequency")
    plt.legend()

    #Thresholded image
    binary_img, otsu_threshold = otsu_method(img)
    #Plot
    plt.subplot(10, 4, i * 4 + 3)
    plt.imshow(binary_img, cmap='gray')
    plt.title(f"Otsu Thresholding = {otsu_threshold:.2f}.")
    plt.axis("off")

    #Histogram of the thresholded image
    plt.subplot(10, 4, i * 4 + 4)
    plt.hist(binary_img.ravel(), bins=64, color='salmon', label="Binary Histogram")
    plt.title(f"Histogram of Otsu-binarized {name} image.")
    plt.xlabel("Pixel Intensity")
    plt.ylabel("Frequency")
    plt.legend()

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

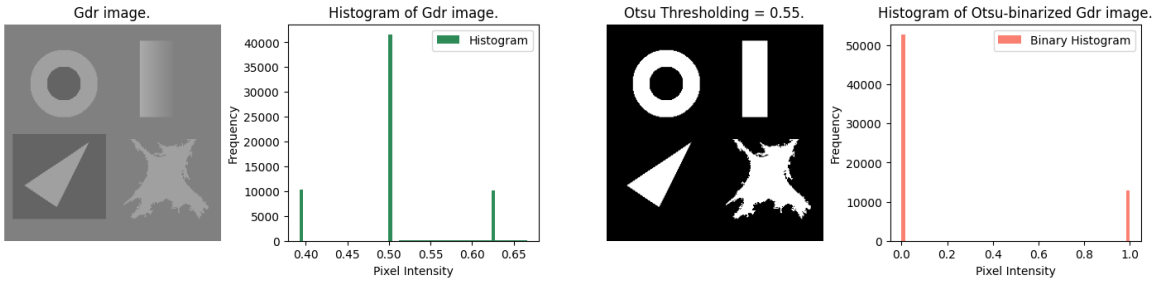

Fig. 15: Otsu thresholding on *Gdr* image.

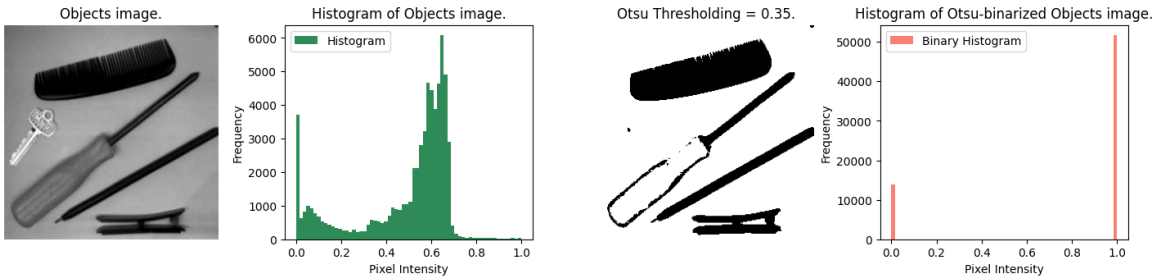

Fig. 16: Otsu thresholding on *Objects* image.

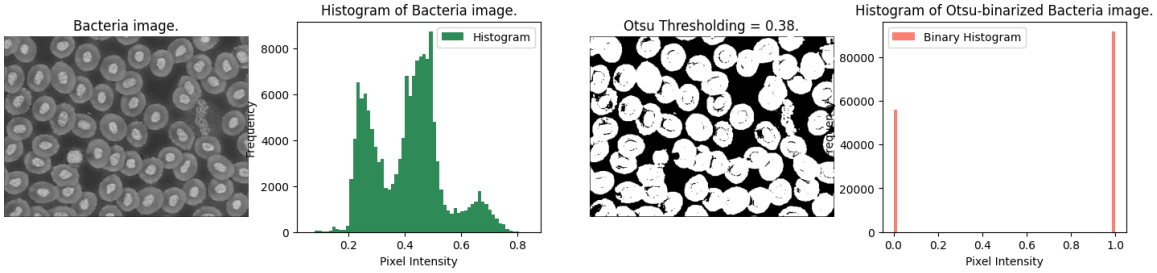

Fig. 17: Otsu thresholding on *Bacteria* image.

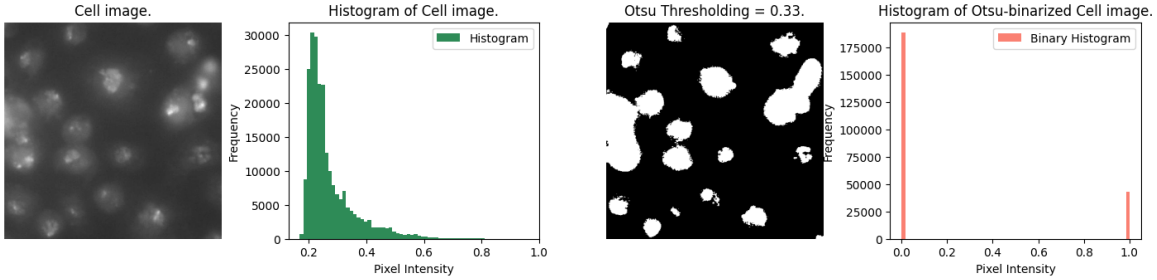

Fig. 18: Otsu thresholding on *Cell* image.

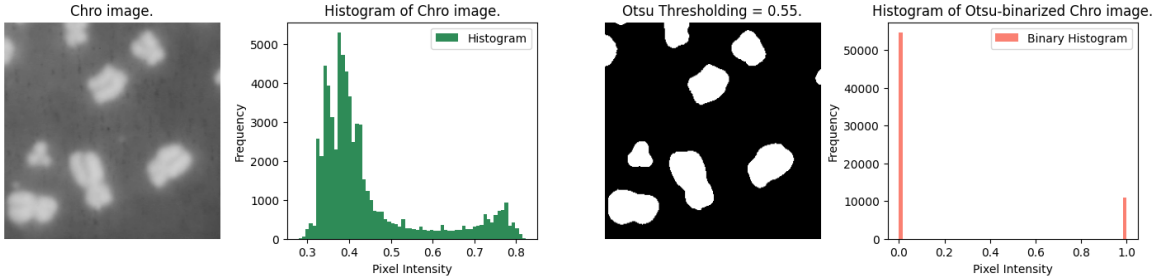

Fig. 19: Otsu thresholding on *Chro* image.

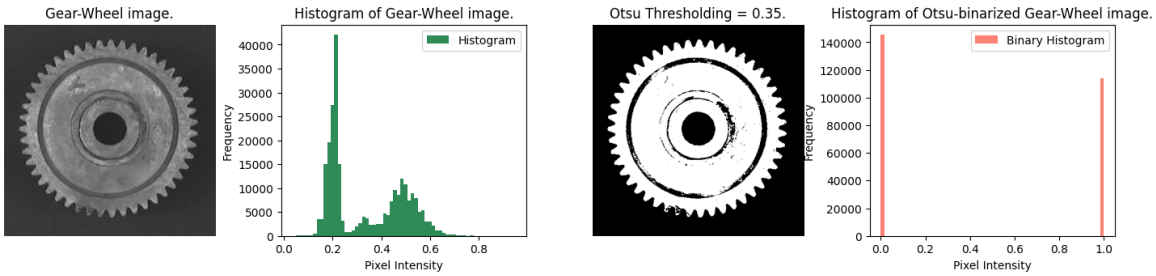

Fig. 20: Otsu thresholding on *Gear-Wheel* image.

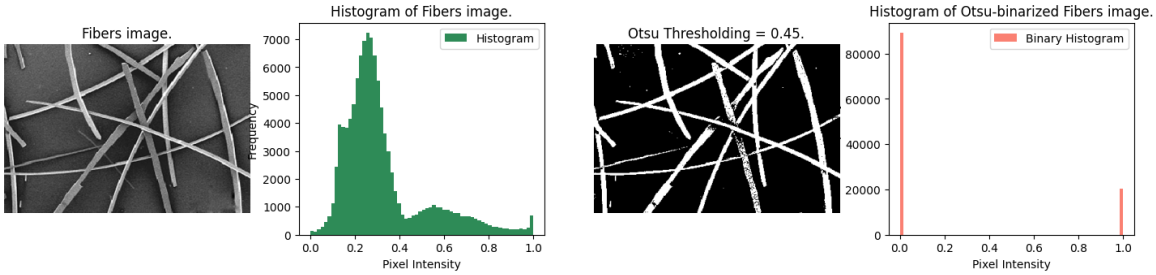

Fig. 21: Otsu thresholding on *Fibers* image.

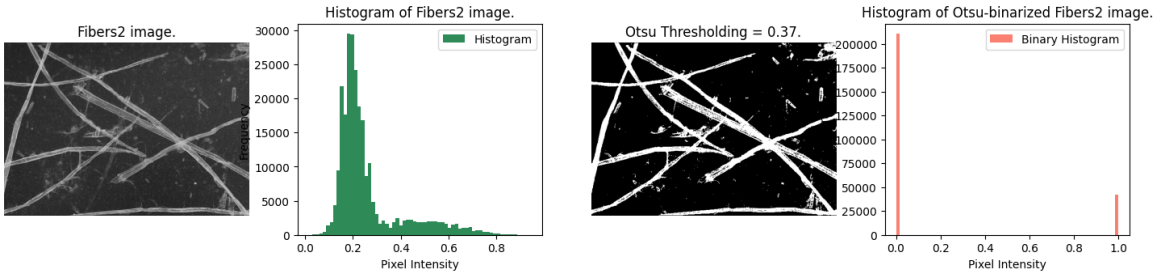

Fig. 22: Otsu thresholding on *Fibers2* image.

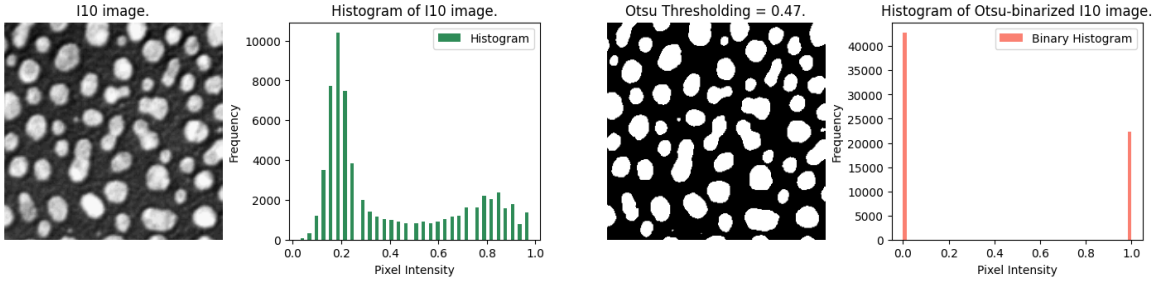

Fig. 23: Otsu thresholding on *I10* image.

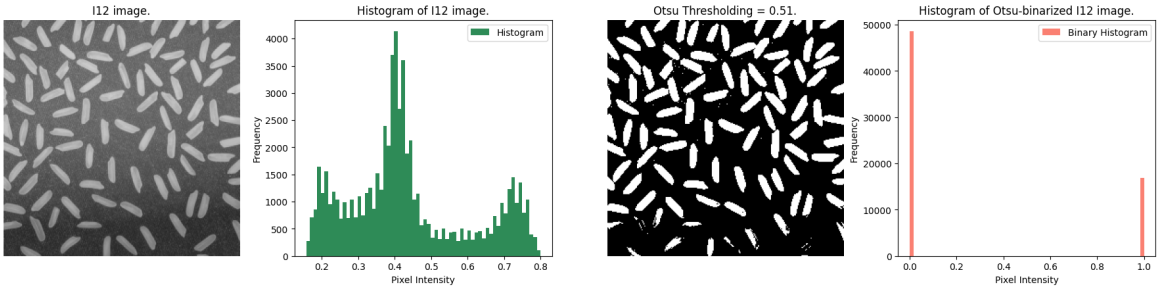

Fig. 24: Otsu thresholding on *I12* image.

In this step instead, we evaluate the effect of histogram stretching and equalization on Otsu's method, i.e. on segmentation. We start comparing it on stretched images, with the function created in the first section. Since stretching increases the contrast of the image, it potentially improves the separability of foreground and background for Otsu's algorithm. The final consideration, as we can see from Figures 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, is actually that visually the difference between segmentation on the original image and segmentation on stretched one, is not so evident, besides in *Gdr*, since it didn't have a strong contrast. In images like *Cell*, the contrast was already very evident, so there is not a big difference (the Otsu threshold is almost the same). However, in both cases the segmentation is good.

```
for i, (img, name) in enumerate(zip(img_list, image_names)):
    #Original image
    plt.figure(figsize=(18, 40))
    plt.subplot(10, 4, i * 4 + 1)
    plt.imshow(img, cmap='gray', norm=NoNorm())
    plt.title(f"{name} image.")
    plt.axis("off")

    #Thresholded image on original one
    binary_img, otsu_threshold = otsu_method(img)
    #Plot
    plt.subplot(10, 4, i * 4 + 2)
    plt.imshow(binary_img, cmap='gray')
    plt.title(f"Otsu on original image = {otsu_threshold:.2f}.")
    plt.axis("off")

    #Stretched image
    stret_img = stretching(img)
    #Plot
    plt.subplot(10, 4, i * 4 + 3)
    plt.imshow(stret_img, cmap='gray')
    plt.title(f"Stretched {name} image.")
    plt.axis("off")

    #Thresholded image on stretched one
    stret_array_img = np.array(stret_img)/255.0
    binary_img2, otsu_threshold2 = otsu_method(stret_array_img)
    #Plot
    plt.subplot(10, 4, i * 4 + 4)
    plt.imshow(binary_img2, cmap='gray')
    plt.title(f"Otsu on stretched image = {otsu_threshold2:.2f}.")
    plt.axis("off")

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

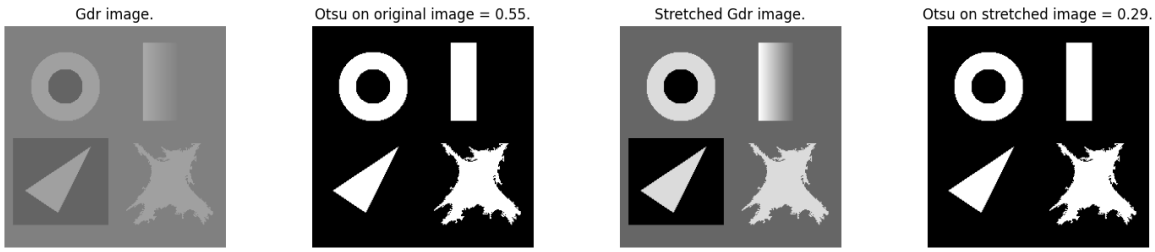

Fig. 25: Otsu thresholding on *stretched Gdr* image.

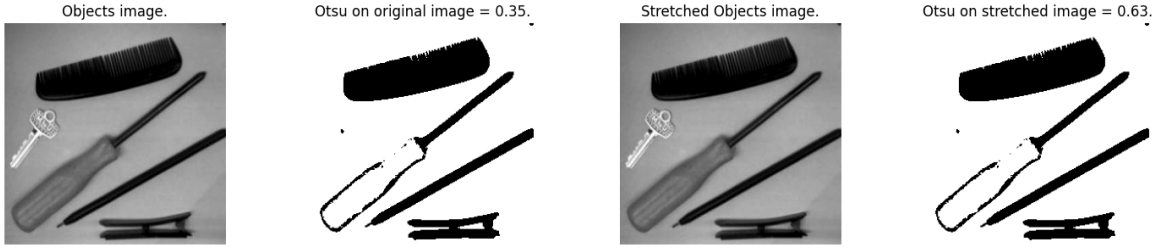

Fig. 26: Otsu thresholding on *stretched Objects* image.

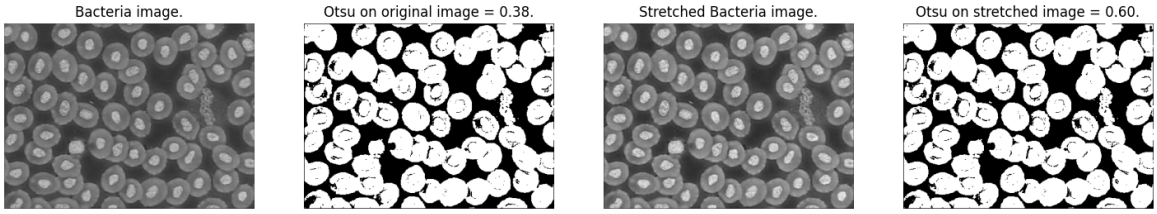

Fig. 27: Otsu thresholding on *stretched Bacteria* image.

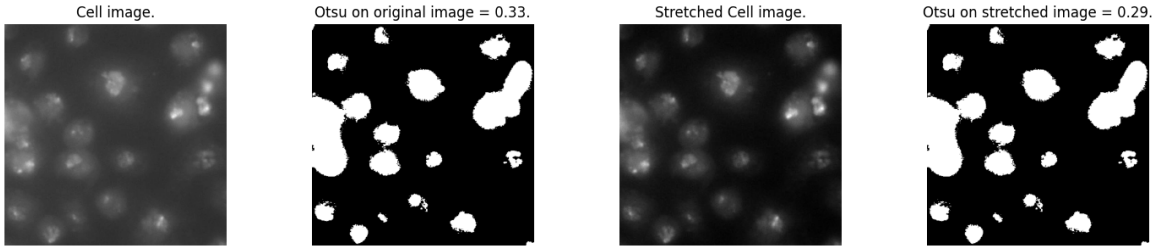

Fig. 28: Otsu thresholding on *stretched Cell* image.

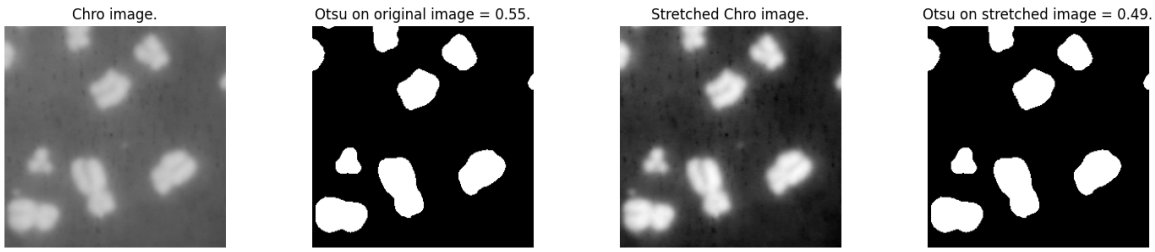

Fig. 29: Otsu thresholding on *stretched Chro* image.

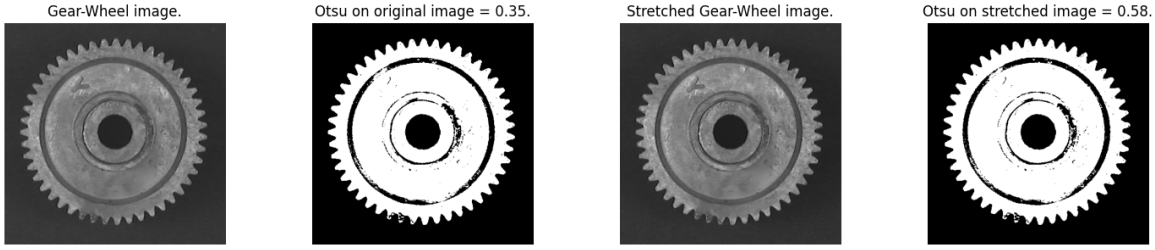

Fig. 30: Otsu thresholding on *stretched Gear-Wheel* image.

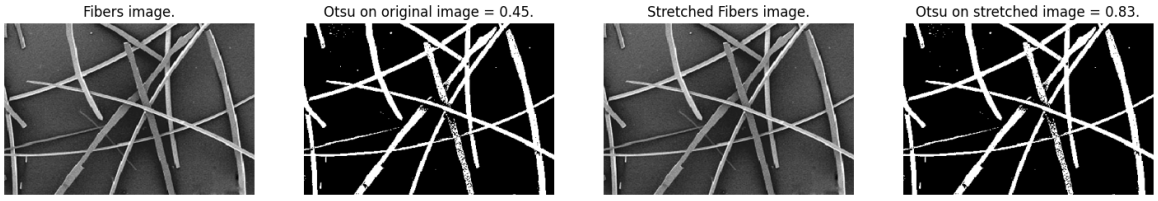

Fig. 31: Otsu thresholding on *stretched Fibers* image.

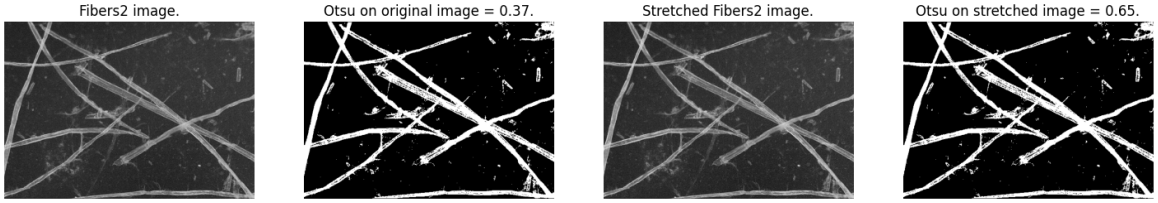

Fig. 32: Otsu thresholding on *stretched Fibers2* image.

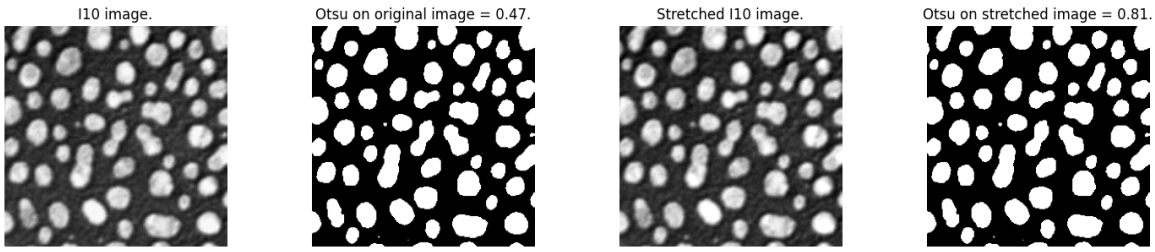

Fig. 33: Otsu thresholding on *stretched I10* image.

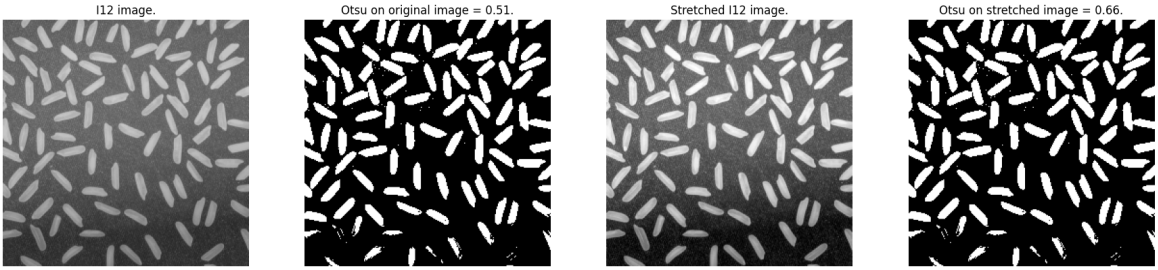

Fig. 34: Otsu thresholding on *stretched I12* image.

As a final analysis of this step, we also analyze the effect of equalization on Otsu's method. As we can see from Figures 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, the segmentation is not good. Sometimes we also lose objects, as for example in the case of *Gdr* image. This application may help when the original image has concentrated intensity values in specific areas (for example in *Gear-Wheel* or *Fibers* images), but it may not improve the segmentation if the image has already a balanced contrast.

```
for i, (img, name) in enumerate(zip(img_list, image_names)):
    #Original image
    plt.figure(figsize=(18, 40))
    plt.subplot(10, 4, i * 4 + 1)
    plt.imshow(img, cmap='gray', norm=NoNorm())
    plt.title(f"{name} image.")
    plt.axis("off")

    #Equalized image
    equalized_img = exposure.equalize_hist(img)
    #Plots
    plt.subplot(10, 4, i * 4 + 2)
    plt.imshow(equalized_img, cmap='gray', norm=NoNorm())
    plt.title(f"Eq. {name} image.")
    plt.axis("off")

    #Thresholded image on the equalized one
    binary_img_eq, otsu_threshold_eq = otsu_method(equalized_img)
    #Plot
    plt.subplot(10, 4, i * 4 + 3)
    plt.imshow(binary_img_eq, cmap='gray')
    plt.title(f"Otsu on equalized image = {otsu_threshold_eq:.2f}.")
    plt.axis("off")

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

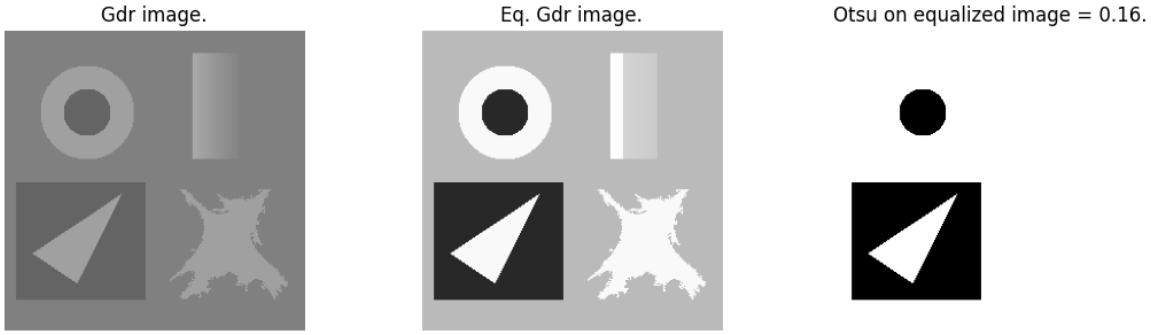

Fig. 35: Otsu thresholding on *equalized Gdr* image.

Fig. 36: Otsu thresholding on *equalized Objects* image.

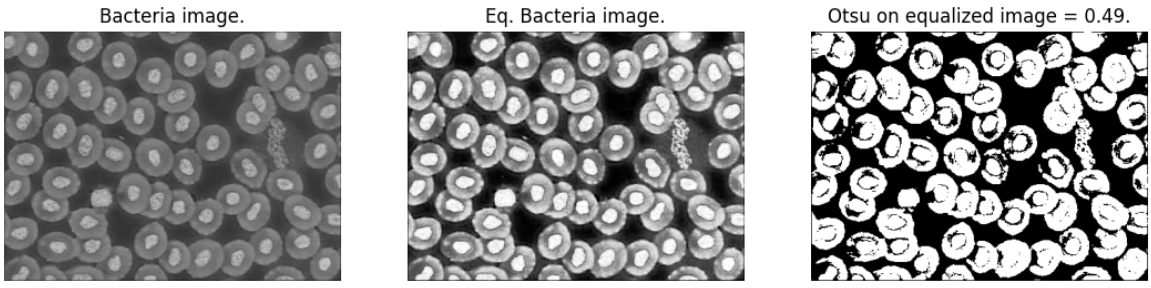

Fig. 37: Otsu thresholding on *equalized Bacteria* image.

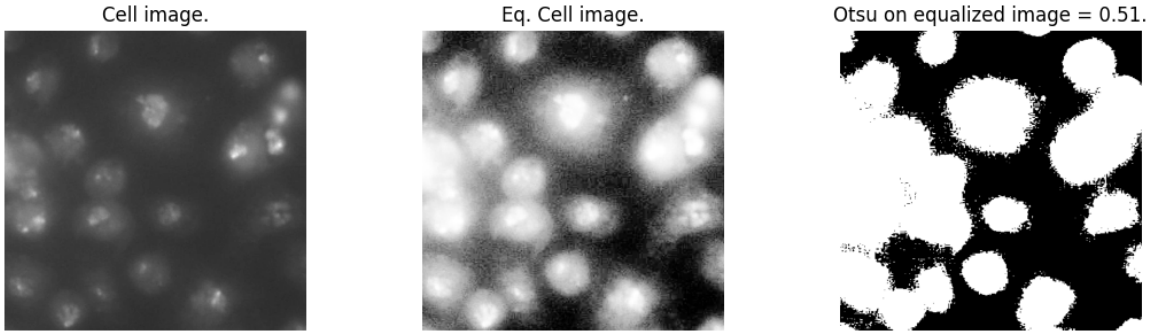

Fig. 38: Otsu thresholding on *equalized Cell* image.

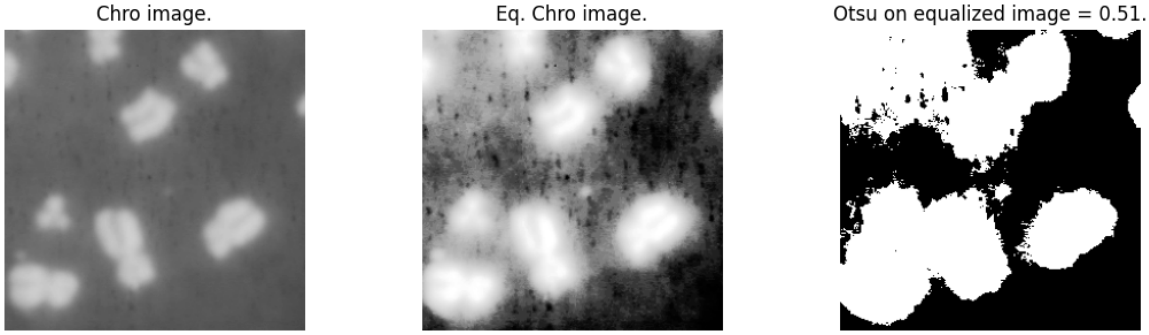

Fig. 39: Otsu thresholding on *equalized Chro* image.

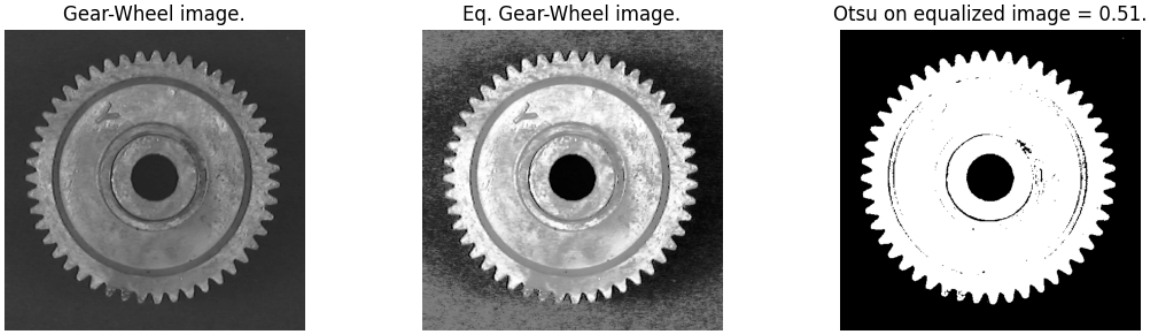

Fig. 40: Otsu thresholding on *equalized Gear-Wheel* image.

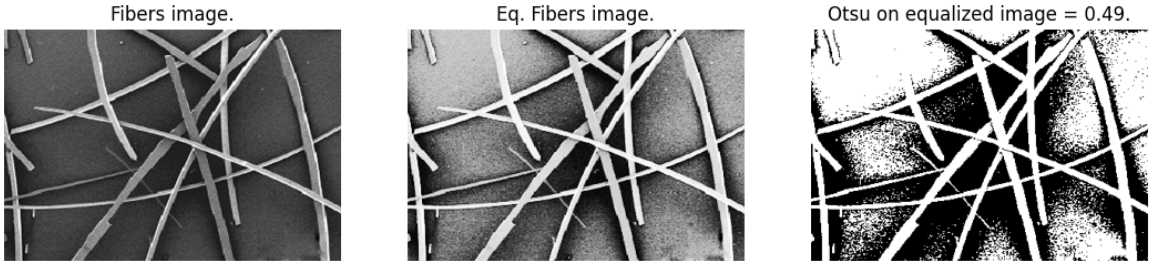

Fig. 41: Otsu thresholding on *equalized Fibers* image.

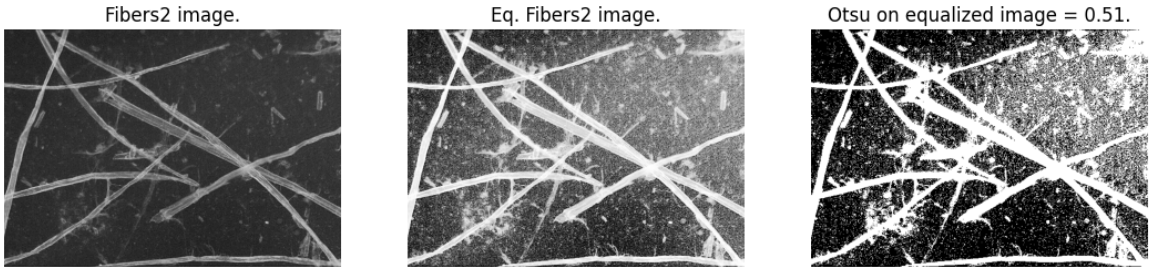

Fig. 42: Otsu thresholding on *equalized Fibers2* image.

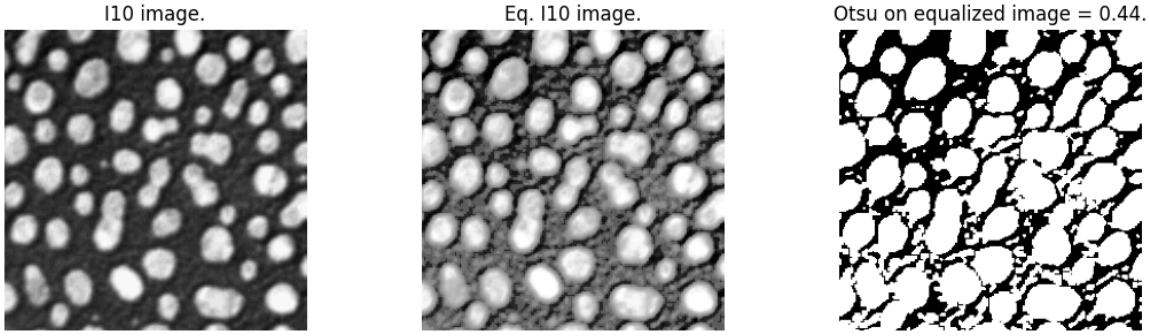

Fig. 43: Otsu thresholding on *equalized I10* image.

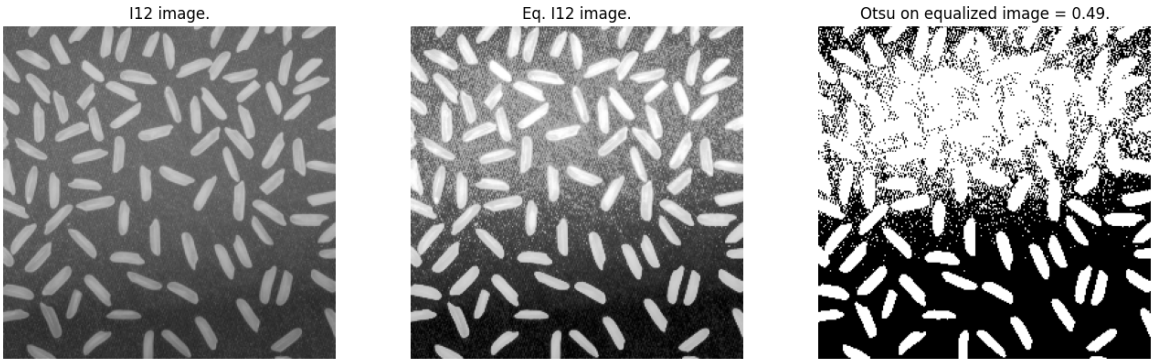

Fig. 44: Otsu thresholding on *equalized I12* image.

For the last step of this section, we apply the segmentation on a colored image, *Pills2*: we segment two green pills. To do this, we need to isolate the green color using the HSV color model. After uploading the image, we convert its RGB array to HSV (Hue, Saturation, Value) color space using the function *rgb2hsv*. After, we isolate hue and saturation into two different vectors, where hue channel provides the color information. From Figure 48 in the Appendix, we can see that green color usually range between 80 and 180 on a total of 360 in the hue spectrum. For this reason, we take the range [0.22, 0.5] for the bounds of hue. The minimum level of saturation is instead set at 0.1 looking at the best result achieved, after some trials. Saturation channel helps filter out colors that are not saturated (like white or gray). Hence, the mask is created using these three limits, and it is after screen to remove eventual noise with the function *remove_small_objects*. Finally, the binary mask is created and here the green pills are represented by white pixels (i.e. 255), while everything else is black (i.e. 0). The code to implement this is shown below, and the result of the segmentation is in Figure 45.

```
from skimage.color import rgb2hsv
from skimage.morphology import remove_small_objects

#Load image
image_pills2 = Image.open("C:\\Users\\sofyc\OneDrive\\Desktop\\UPEC\\Pattern recognition\\
    assignment 3 - IP\\IP3\\IP3\\pills2.bmp")

#To array
image_array_pills2 = np.array(image_pills2)

#Conversion RGB to HSV
#- Hue: [0,1]. As hue increases from 0 to 1, the color transitions from red to orange, yellow,
    green, cyan, blue, magenta, and finally back to red.
#- Saturation: [0,1]. 0 = neutral shade, 1 = maximum saturation.
#- Value: maximum value among the red, green, and blue components of a specific color.
hsv_pills2 = rgb2hsv(image_array_pills2)
#Extraction of Hue and Saturation channels
hue = hsv_pills2[:, :, 0]  #Hue channel
saturation = hsv_pills2[:, :, 1]  #Saturation channel
#Thresholds for green in HSV space:
#The Hue range has 360 degrees. The color green goes more or less between 80 to 180 degrees. Hence
    , 80/360 = 0.22, while 180/360 = 0.5.
lower_hue = 0.22  #Lower bound for green hue
upper_hue = 0.5  #Upper bound for green hue
minimum_saturation = 0.1  #Minimum saturation to exclude white --> 0.1 after some trials.
#Mask for green points in the image
green_mask = (hue >= lower_hue) & (hue <= upper_hue) & (saturation >= minimum_saturation)
#Removing eventual noise (if there is any)
final_mask = remove_small_objects(green_mask, min_size=200)  #min_size indicates that each object
    wuth an area lower than 50 pixels is removed.
#Converting the mask to binary one (black = 0 and white = 1)
binary_mask = final_mask.astype(np.uint8) * 255

#Plot
plt.figure(figsize=(10, 6))
plt.subplot(1, 2, 1)
plt.imshow(image_pills2, cmap='gray', norm=NoNorm())
plt.title("Pills2 image.")
plt.axis("off")

#Segmentation plot
plt.subplot(1, 2, 2)
plt.imshow(binary_mask, cmap='gray')
plt.title("Segmentation for green pills.")
plt.axis("off")
plt.tight_layout()
plt.show()
```

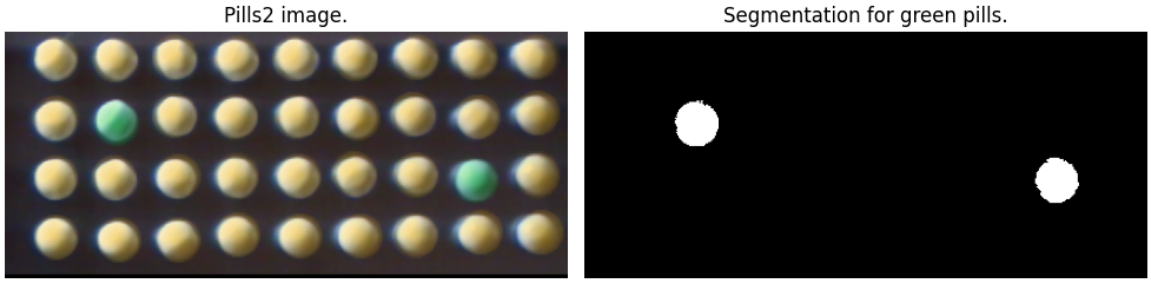

Fig. 45: Segmentation in *Pills2* image.

As a final information extraction, we calculate the surface area of green pixels by summing up the values of the final mask (i.e. white pixels). The output is of 1806 pixels. If we would like to have the surface in square millimeters, we would need the area of a pixel (information we don't have).

```
#Calculation of the area in pixels
green_pixels_surface = np.sum(final_mask)
```

3 Labeling

In this section, we perform **labeling** using the **connected component analysis**, in order to automatically count the number of distinct objects present in a image. We apply this on *Chro* image. After converting it to a binary image using Otsu's method, we label the connected components in it using *label* function from *skimage* library. Just for precision, it is important to say that connected components are connected groups of pixels that have some properties in common, for example the color or the intensity. There are two types of connectivities:

- **4-connectivity**, where a pixel is connected with its neighbors that lie horizontally or vertically, i.e. it has 4 possible neighbors (up, down, left, right). This connectivity is indicated by *connectivity=1* in *label* function.
- **8-connectivity**, where a pixel is connected with all its neighbors, including the diagonal ones, i.e. it has 8 possible neighbors (4 horizontal or vertical and 4 diagonal). This connectivity is indicated by *connectivity=2* in *label* function and it is the one we use.

Label function assigns a label, i.e. an integer, to every connected region of white pixels in the binary image. Moreover, this function also returns the number of distinct objects that in our case is 10. Finally, on the RGB labeled image where the labels are mapped to distinct colors is applied a black background, using the function *label2rgb*. The resulting image is shown in Figure 46.

```
from skimage.measure import label

#Converting inot binary image through Otsu method
binary_img_chro, otsu_thr_chro = otsu_method(image_array_chro)

#Labeling the connected components
labeled_image, num_objects = label(binary_img_chro, connectivity=2, return_num=True)

print(f"Number of objects found: {num_objects}")

#Plot
plt.figure(figsize=(10, 6))
#Binary mask - black background
plt.subplot(1, 3, 1)
plt.imshow(binary_img_chro, cmap='gray', norm=NoNorm())
plt.title("Binary mask - background.")
plt.axis("off")

#Labeled image with Jet map
plt.subplot(1, 3, 2)
plt.imshow(labeled_image, cmap='jet')
plt.title("Labeled image with the map Jet.")
plt.axis("off")

```

```
#Labeled image with label2rgb (black background)
labeled_image_rgb = label2rgb(labeled_image, bg_label=0, bg_color=(0, 0, 0), colors=None)
plt.subplot(1, 3, 3)
plt.imshow(labeled_image_rgb)
plt.title("RGB labeled image.")
plt.axis("off")

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

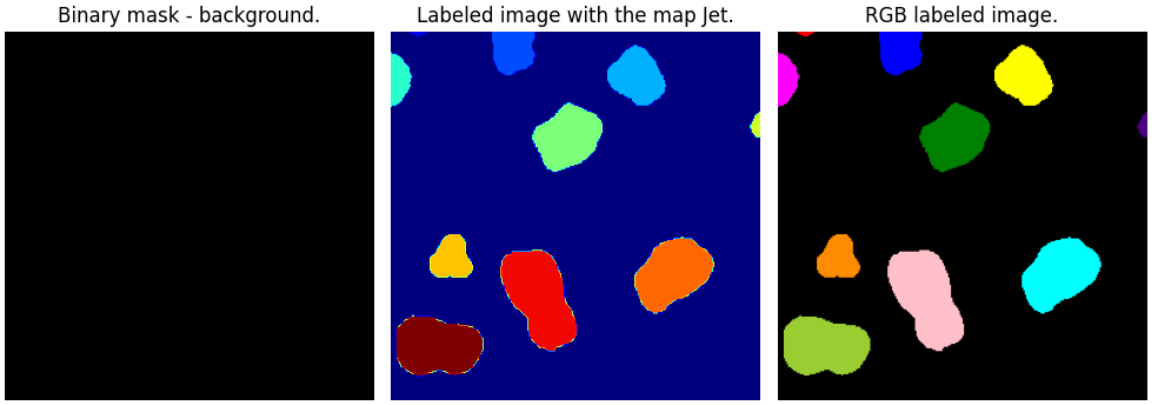

Fig. 46: Labeling on *Chro* image.

In this last part of the section, we try to label the difference between *Original* and *Original2* images. The binarization is still made using Otsu's method, and the difference is perform through the logical XOR operator. This operator returns an image where each pixel is 1 if the pixel in the images is different, and 0 if not. Finally, the function *label* catches the connective components in the image of differences. In this case we have 8 differences labeled, and the result obtained is shown in Figure 47.

```
from skimage.color import label2rgb

#Load image
image_original = Image.open("C:\\Users\\sofyc\OneDrive\\Desktop\\UPEC\\Pattern recognition\\
    assignment 3 - IP\\IP3\\IP3\\original.bmp")
image_original2 = Image.open("C:\\Users\\sofyc\OneDrive\\Desktop\\UPEC\\Pattern recognition\\
    assignment 3 - IP\\IP3\\IP3\\original2.bmp")

#To array
image_array_original = np.array(image_original)
image_array_original2 = np.array(image_original2)

#Converting inot binary image through Otsu method
binary_img_original, otsu_thr_original = otsu_method(image_array_original)
binary_img_original2, otsu_thr_original2 = otsu_method(image_array_original2)

#We calculate the difference with XOR operation
#We can't use np.diff since it calculates differences between consecutive elements in a specified
    axis. It is not adapt for entire images.
difference_img = np.logical_xor(binary_img_original, binary_img_original2)

#Labelling the difference
labeled_diff, num_diff = label(difference_img, connectivity=2, return_num=True)

#Plot
plt.figure(figsize=(15, 10))
plt.subplot(1, 3, 1)
plt.imshow(image_original, cmap='gray', norm=NoNorm())
plt.title("Original image.")
plt.axis("off")

plt.subplot(1, 3, 2)
plt.imshow(image_original2, cmap='gray', norm=NoNorm())
plt.title("Original2 image.")
plt.axis("off")

#Plot of difference between labelling
```

```
#To RGB
labeled_diff_rgb = label2rgb(labeled_diff, bg_label=0, bg_color=(0, 0, 0), colors=None)
plt.subplot(1, 3, 3)
plt.imshow(labeled_diff_rgb)
plt.title(f"Number of labeled difference = {num_diff}.")
plt.axis("off")

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

Fig. 47: Labeling on the difference between *Original* and *Original2* images.

Appendix

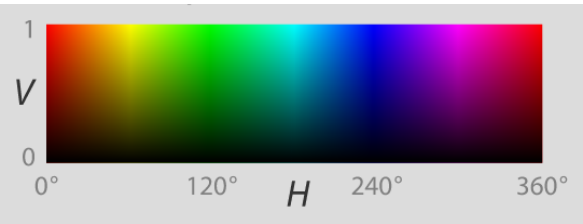

Fig. 48: Hue system colors.

References

[1] Majidzadeh, F.: Digital Image Processing and Pattern Recognition, (2023)

[2] YouTube: Video on Color Detection (2024). https://www.youtube.com/watch?v=qe4fpdNzfxU

[3] MathWorks: rgb2hsv - MATLAB Documentation (2024). https://fr.mathworks.com/help/matlab/ref/rgb2hsv.html

[4] Overflow, S.: MATLAB - How to Detect Green Color on Image (2016). https://stackoverflow.com/questions/37684903/matlab-how-to-detect-green-color-on-image