Digital Image Processing - Lab Session 1

Sofia Noemi Crobeddu
student number: 032410554
International Master of Biometrics and Intelligent Vision - Université Paris-Est Cretéil (UPEC)

Introduction

In this project it is performed an analysis on sampling, quantization and indexing color of images. Moreover, different types of extensions are introduced for compression insights.

1 Image Processing and Analysis: getting started

In this section we work on *crossroad* image. The first we analyze is the one in *dat* format. After reading the file and putting data in the array in *image_dat* variable, we cannot display the image without reshaping. This is due to the fact that through the function *fromfile*, we are reading an unidimensional array of byte, without any information about how they are organizing in the image. Below it is showed the code performed to read the file:

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

#Reading data (the image)
image_dat = np.fromfile("C:\\Users\\sofyc\\OneDrive\\Desktop\\UPEC\\Pattern recognition\\IP1\\IP1
    \\crossroad.dat", dtype=np.uint8)
```

After this starting point and continuing with the task, given the number of rows equal to 435, we can easily calculate the number of columns since

$$\text{Total pixels} = (\text{Number of rows}) \cdot (\text{Number of columns}).$$

The total number of pixels is found through the command *len(image_dat)*, i.e. the length, and in our case is equal to 252300 pixels. Hence, the number of columns is 580, and we can then reshape the image and plot it. Since we have the height and the width, we obtain a structure in gray scale. This is due to the fact that a 2D array represents an image with a unique channel, and where there are different levels of gray.
The code is shown below:

```
#Assuming the number of rows given is 435
n_rows = 435
n_cols = len(image_dat)/n_rows
n_cols = int(n_cols)

#Reshape
reshaped_image_dat = image_dat.reshape((n_rows, n_cols))

#Plot gray scale
plt.imshow(reshaped_image_dat, cmap='gray')
plt.axis("off")  #Don't show the axes
plt.show()
```

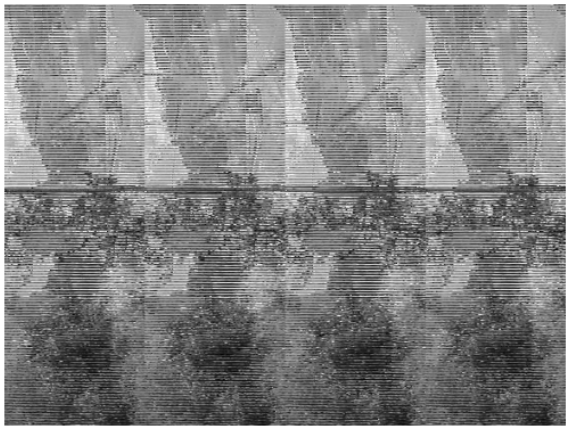

Fig. 1: *crossroad.dat* image - gray scale.

Figure 1 shows the result. As we can notice, the image doesn't show very well the content. For this reason we continue the analysis through the same image in *bmp* format. After reading directly the image data using the function *Image* from *PIL* (Pillow) library, that gives automatically a format to data matrix, we can easily display the image, i.e. Figure 2.
The code is the following one:

```
from PIL import Image

#Loading the image 'crossroad.bmp'
image_bmp = Image.open("C:\\Users\\sofyc\\OneDrive\\Desktop\\UPEC\\Pattern recognition\\IP1\\IP1\\
    crossroad.bmp")

#Plot
plt.imshow(image_bmp, cmap='gray')
plt.axis("off")  #Remove axes
plt.show()
```

Fig. 2: *crossroad.bmp* image.

Going to the next step, to analyze the first pixel of *crossroad.bmp* image, we take the value in column and row 0. To display its value we use the function *print*. The code used is shown below, and the output gives a value of 187 of gray level in the first pixel of the considered image.

```
#Converting the image into an array
image_bmp_array = np.array(image_bmp)

#Analizying the first pixel in top left in position [0,0]
```

```
first_pixel_bmp = image_bmp_array[0, 0]

#Showing the the row, the column and the value of the first pixel. We use print() for the command
    window
print(f"The first pixel in [0,0] has a value of gray level of: {first_pixel_bmp}.")
```

To find instead, the bottom right corner pixel, we need to know the number of rows and columns of the image since this pixel represents the last one. Hence, first we check if the image in *bmp* format has still the same number of total pixel (using *size* function), and shape. After this, it is possible to extract the value of the last pixel in position [Number of rows - 1, Number of columns - 1] (the indexes in an array in Python start from 0). As before, we use *print* to display the value of gray level, that is 72. The code is shown below:

```
#Checking that the information are still the same we registered in the steps before
print("The total pixels are: ", np.size(image_bmp_array))
print("Shapes: ", image_bmp_array.shape)

#Analizying the first pixel in top left in position [434,579]
last_pixel_bmp = image_bmp_array[n_rows-1, n_cols-1]
print(f"The last pixel in [434,579] has a value of gray level of: {last_pixel_bmp}.")
```

To interpret the value of gray level, we have to know that:

- the range is between 0 and 255, as for RGB scale,
- in the gray scale 0 represents the absence of light, i.e. it is entirely black,
- 255 is instead the value of maximum brightness, i.e. it is entirely white.

In our case 187 and 72 are intermediate levels, so they are shades of gray. The first pixels has more light.

Going one with the tasks, we analyze now the first row and first column of the *crossroad.bmp* image. We save them in two vectors, L1 and C1 respectively. The code is shown below:

```
L1 = image_bmp_array[0,:] #First row vector
C1 = image_bmp_array[:,0] #First column vector
```

We can visualize them using the function *bar* from the library *Matplotlib*. Figures 4 and 3 show the barplots. As we can notice, regarding the plot of L1, we have higher value of gray level in the first pixels, while close to the end of the row the pixels have lower values of gray level. This is due to the fact that in the top left part of the image, as we saw in Figure 2, the brightness is higher, while in the top right part we have trees in dark gray. Same method for the analysis of C1, where we have low values of gray correspondent to the central indexes of C1, due to the fact that in the middle vertical of the image we have grass and trees.

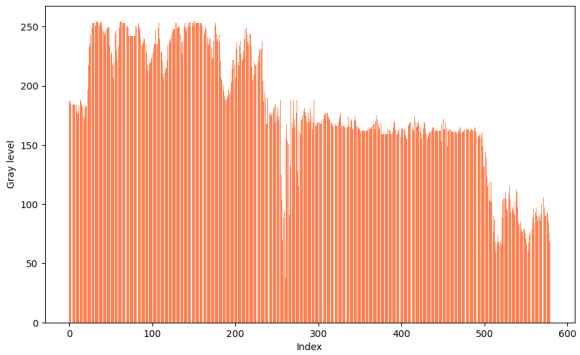

Fig. 3: Value of the first row L1 of *crossroad.bmp* image.

Fig. 4: Value of the first column C1 of *crossroad.bmp* image.

We analyze also *sonnet* image, reading it using *Image* from *PIL*, and we repeat the same process done previously to take some columns of it. In particular, we take the first column, the last one and an arbitrary column in position 200. The code is shown below, while the plots of the gray level of these columns are shown in Figure 5, 6, 7. As we can see, in all the three graphs we have decreasing values. This suggests that the top part of the image is probably more enlightened, while the bottom part is darker with values closer to 0 (i.e. black).

```
#Loading the image 'sonnet'
image_sonnet = Image.open("C:\\Users\\sofyc\\OneDrive\\Desktop\\UPEC\\Pattern recognition\\IP1\\
    IP1\\sonnet.png")

#Transforming the image into array
image_sonnet_array = np.array(image_sonnet)

#Dimensions
print(image_sonnet_array.shape)

#Taking some columns
C1_sonnet = image_sonnet_array[:,0] #First column
Clast_sonnet = image_sonnet_array[:,384-1] #Last column
C200_sonnet = image_sonnet_array[:,200-1] #Other column
```

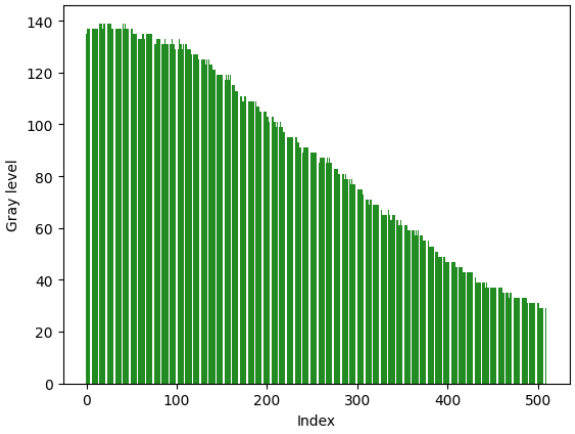

Fig. 5: Value of the first column of *sonnet* image.

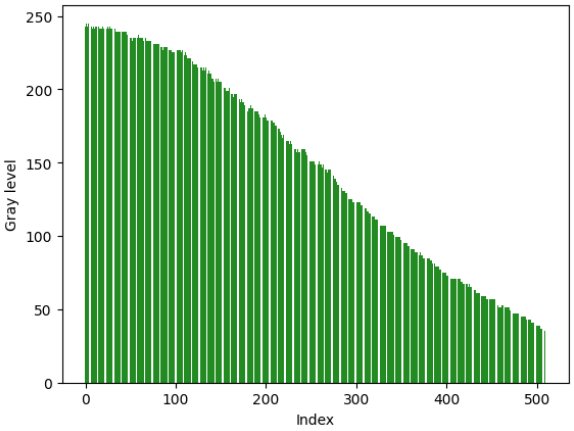

Fig. 6: Value of the last column of *sonnet* image

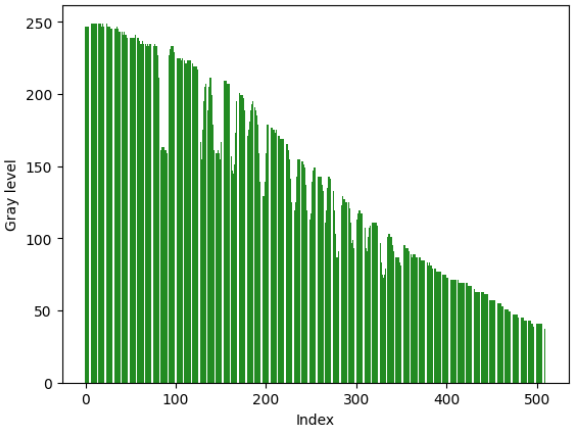

Fig. 7: Value of the 200th column of *sonnet* image.

When we display the *sonnet* image using *imshow* function, we have in fact the confirmation of the analysis that the barplots of the three columns gave us: the image in Figure 8 has an evident brightness in top part, especially in top right part.

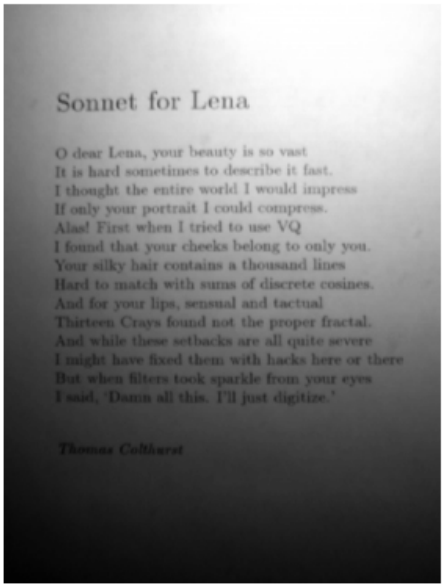

Fig. 8: *sonnet* Image.

Finally, we store *sonnet* image in PNG and JPEG format (tiff format is not necessary since it is the original format we already have). JPEG (Joint Photographic Experts Group) format is a type of compression called lossy, i.e. a part of information is lost to reduce the dimension of the file. PNG (Portable Network Graphics) is instead a lossless compression, that maintains the same information but without reducing the dimension. PNG files are usually larger than JPEG.
The code to implement them is shown below:

```
#Saving (no in png format since it is the original format of the image)
image_sonnet.save("sonnet.tif")
image_sonnet.save("sonnet.jpg")
```

2 Resolution

In this section we modify the resolution of *crossroad.bmp* image with boxes of 2, 4, 8 and 16 pixels. The first method we use is the **basic subsampling** one, where we subsample the image every n pixels, both on rows and columns. This means that we extract the image-array every 2 or 4, etc.. , pixels. As the resolution decreases (e.g. from 2x2, 4x4, ...), the details are gradually lost. This effect is more evident for n larger. The code to perform this is shown below, while the images of the output are in Figure 9. As we can see, we lose edges and details as n increases. In the last image subsampled 16x16 in fact, we can't recognize the different elements of the picture.

```
#Resolutions
res_boxes = [2,4,8,16]

plt.figure(figsize=(20, 10))
for i, n in enumerate(res_boxes):
    #Subsampling the image every n pixels
    subsampled_image = image_bmp_array[::n, ::n]

    #Plot of the subsampled image
    plt.subplot(1, len(res_boxes), i + 1)
    plt.imshow(subsampled_image, cmap='gray')
    plt.title(f'Subsampling {n}x{n}')
    plt.axis('off') #Without axes

plt.show()
```

Fig. 9: Subsampling method on *crossroad* image.

The second method used is instead the **resize** one. We use *resize* function from library *PIL*, that offers different option di interpolation. The image is first resized based on the scale factor n both on height and width in order to have a comparison with the previous method's output, and after this we apply the **NEAREST interpolation** method. It is basically a method that select the closer pixel to the new position during the resizing. It was chosen this one, for its simplicity and its low computational cost. The code performed is shown below, and the output is in Figure 10.

```
plt.figure(figsize=(20, 10))
for i, n in enumerate(res_boxes):
    #Ridimensioning the image with resize function from PIL
    resized_image = image_bmp.resize(
        (image_bmp.width // n, image_bmp.height // n), Image.Resampling.NEAREST
    )

    #Plot resized image
    plt.subplot(1, len(res_boxes), i + 1)
    plt.imshow(resized_image, cmap='gray')
    plt.title(f'Resize {n}x{n}')
    plt.axis('off')

plt.show()
```

Fig. 10: Resize method on *crossroad* image.

Comparing the two methods, we can see that the images performed are quite similar, but *resize* usually offers more uniform results, since it has a controlled interpolation.

Now we re-do the same process with *test pattern* and *patterns* images.
Below there is the code for *subsampling* method in the case of *test pattern* image. Figure 11 shows the result.

```
#Loading the image 'test pattern'
image_test_pattern = Image.open("C:\\Users\\sofyc\\OneDrive\\Desktop\\UPEC\\Pattern recognition\\
    IP1\\IP1\\test pattern.jpg")
image_test_array = np.array(image_test_pattern)

# i) Subsampling method
plt.figure(figsize=(20, 10))
for i, n in enumerate(res_boxes):
    #Subsampling the image every n pixels
    subsampled_image = image_test_array[::n, ::n]

```

```
    #Plot of the subsampled image
    plt.subplot(1, len(res_boxes), i + 1)
    plt.imshow(subsampled_image, cmap='gray')
    plt.title(f'Subsampling {n}x{n}')
    plt.axis('off') #Without axes

plt.show()
```

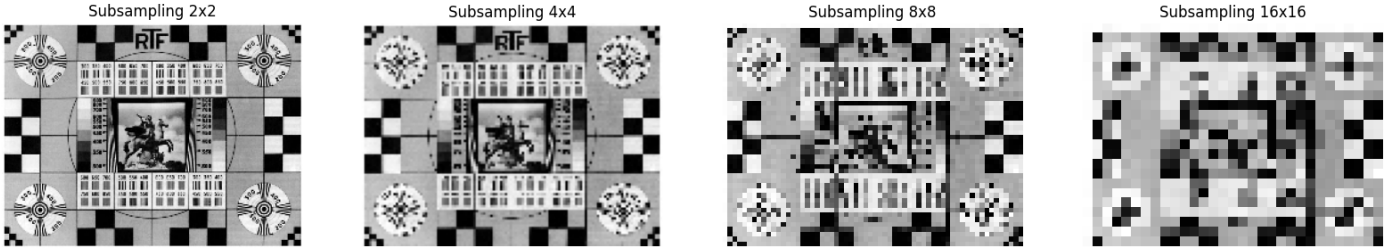

Fig. 11: Subsampling method on *test pattern* image.

Here's the code for *resize* method in the case of *test pattern* image, and figure 12 shows the result.

```
# ii) Resizing method
plt.figure(figsize=(20, 10))
for i, n in enumerate(res_boxes):
    #Ridimensioning the image with resize function from PIL
    resized_image = image_test_pattern.resize(
        (image_test_pattern.width // n, image_test_pattern.height // n), Image.Resampling.NEAREST
    )

    #Plot resized image
    plt.subplot(1, len(res_boxes), i + 1)
    plt.imshow(resized_image, cmap='gray')
    plt.title(f'Resize {n}x{n}')
    plt.axis('off')

plt.show()
```

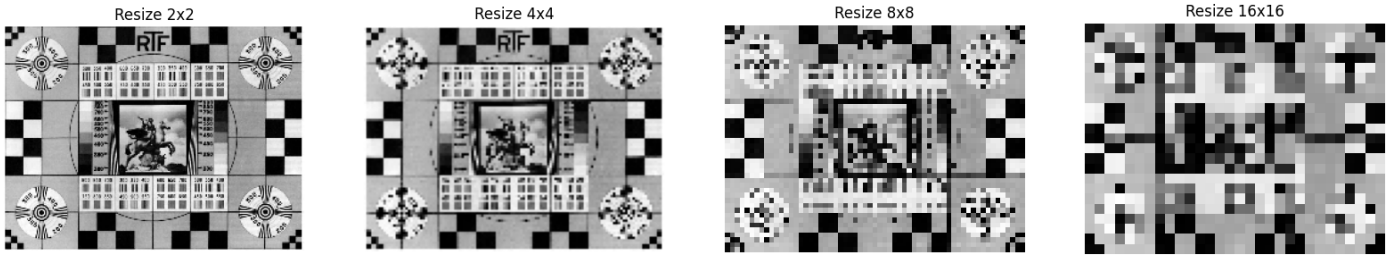

Fig. 12: Resize method on *test pattern* image.

Below there is the code for *subsampling* method for *patterns* image. In this case we had two channels and the channel 1 shows completely black images. This means that it contains just 0 values for all the pixels, and it is like an empty channel. Hence we select the channel 0 (with brightened images). Figure 13 shows the result.

```
#Loading the image 'patterns'
image_patterns = Image.open("C:\\Users\\sofyc\\OneDrive\\Desktop\\UPEC\\Pattern recognition\\IP1\\
    IP1\\patterns.png")
image_patterns_array = np.array(image_patterns)

print(image_patterns_array.shape)

#We select just the first channel to obtain an image in gray scale
image_patterns_array = image_patterns_array[:, :, 0]

```

```
# i) Subsampling method
plt.figure(figsize=(20, 10))
for i, n in enumerate(res_boxes):
    #Subsampling the image every n pixels
    subsampled_image = image_patterns_array[::n, ::n]

    #Plot of the subsampled image
    plt.subplot(1, len(res_boxes), i + 1)
    plt.imshow(subsampled_image, cmap='gray')
    plt.title(f'Subsampling {n}x{n}')
    plt.axis('off') #Without axes

plt.show()
```

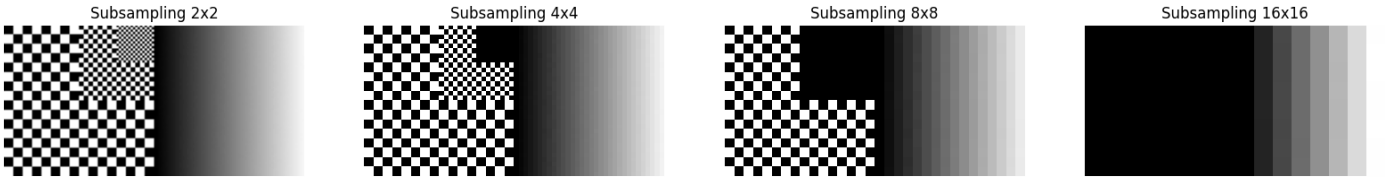

Fig. 13: Subsampling method on *patterns* image.

Finally, the code for *resize* method in the case of *patterns* image, and figure 14 shows the result.

```
# ii) Resizing method
plt.figure(figsize=(20, 10))
for i, n in enumerate(res_boxes):
    #Ridimensioning the image with resize function from PIL
    resized_image = image_patterns.resize(
        (image_patterns.width // n, image_patterns.height // n), Image.Resampling.NEAREST
    )

    #Plot resized image
    plt.subplot(1, len(res_boxes), i + 1)
    plt.imshow(resized_image, cmap='gray')
    plt.title(f'Resize {n}x{n}')
    plt.axis('off')

plt.show()
```

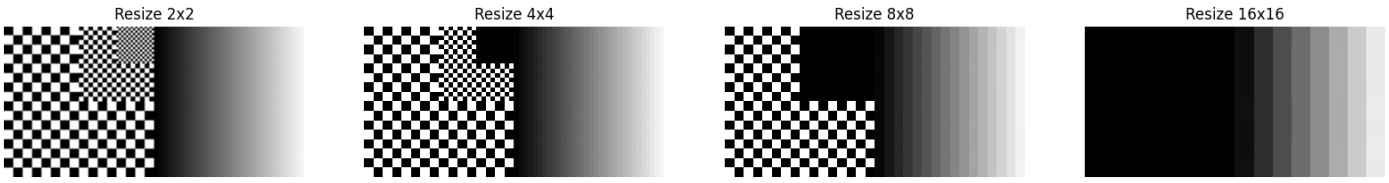

Fig. 14: Resize method on *patterns* image.

3 Quantization

In this section we apply the **quantization**. The objective is to reduce the intensity of the gray level of images *crossroad.bmp*, *test pattern* and *patterns*. In all the cases we have 256 levels of gray (from 0 to 255 included), and we apply the quantization in order to have 128, 64, 32, 16, 8, 4 and 2 levels. To perform this analysis, we create a function called *quantize_fun* that takes the image and a certain number of gray levels as inputs, and gives the quantized image as output. First, it calculated the step of quantization through a floor division (i.e. the result is an integer) between 256 and the level required. After this, the image is divided (again floor division) by the step in order to have pixels values between 0 and this specific level required, and finally these new quantized levels are multiplied by the step to bring them in the original scale 0-255. The output, for each gray level wanted in the case of *crossroad.bmp* image, is shown in Figure 15, while the code is shown below. As we can notice,

for 128 and 64 levels the image maintains a good quality, similar to the original one. Since this image has not a good original appearance, also the 32 and 16 levels don't show evident differences, while 8 and 4 levels have more white pixels and the color transitions are more clear. In the last level 2, we have a very different effect from the original one, and we don't have real intermediate levels of gray, just black and white, so the shades are lost.

```
#Function to quntize the image
def quantize_fun(image, level):
    step = 256 // level
    image_quantized = (image // step) * step
    return(image_quantized)

#Gray levels
gray_levels = [128, 64, 32, 16, 8, 4, 2]

#a) Image crossroad dat
#array format
image_dat_array = np.array(reshaped_image_dat)

plt.figure(figsize=(25, 10))
for i, levels in enumerate(gray_levels):
    quantized_image = quantize_fun(image_dat_array, levels)
    plt.subplot(2, 4, i + 1)
    plt.imshow(quantized_image, cmap='gray')
    plt.title(f'{levels} gray level')
    plt.axis('off')

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

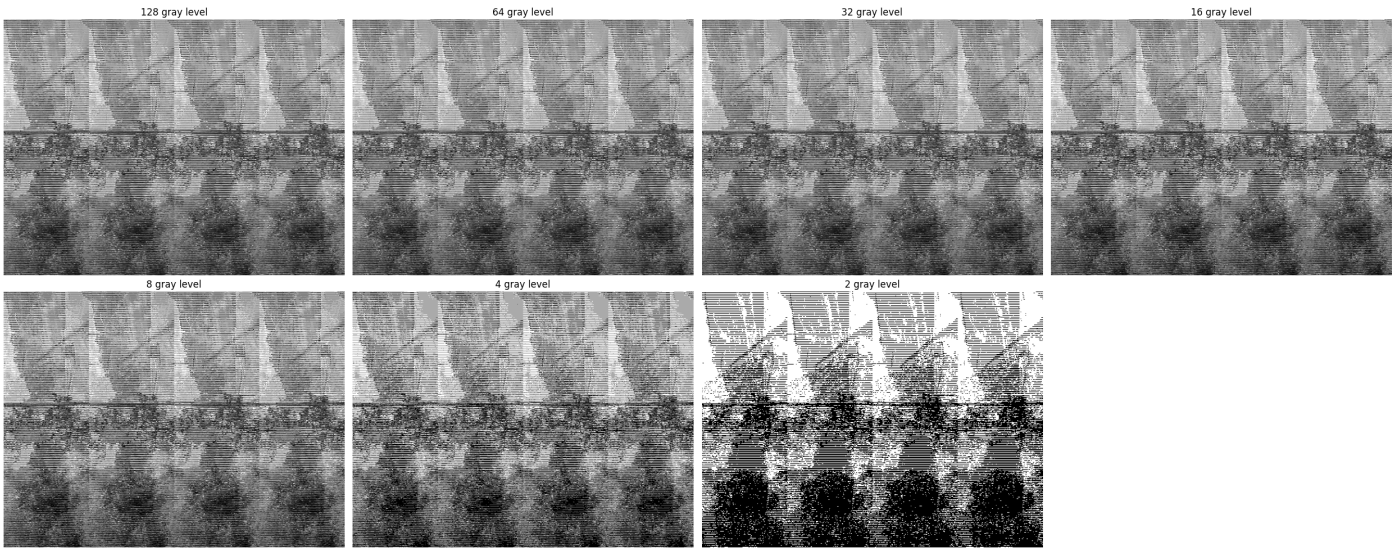

Fig. 15: Quantization of *crossroad.dat* image.

The same process of quantization is applied to *test pattern* image. The code in shown below and the output is in Figure 16. We have a similar output to *crossroad.bmp* image, and in this case the difference between the original one and the last one (2 levels), is really evident.

```
#b) Image test pattern
plt.figure(figsize=(25, 10))
for i, levels in enumerate(gray_levels):
    quantized_image = quantize_fun(image_test_array, levels)
    plt.subplot(2, 4, i + 1)
    plt.imshow(quantized_image, cmap='gray')
    plt.title(f'{levels} gray level')
    plt.axis('off')

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

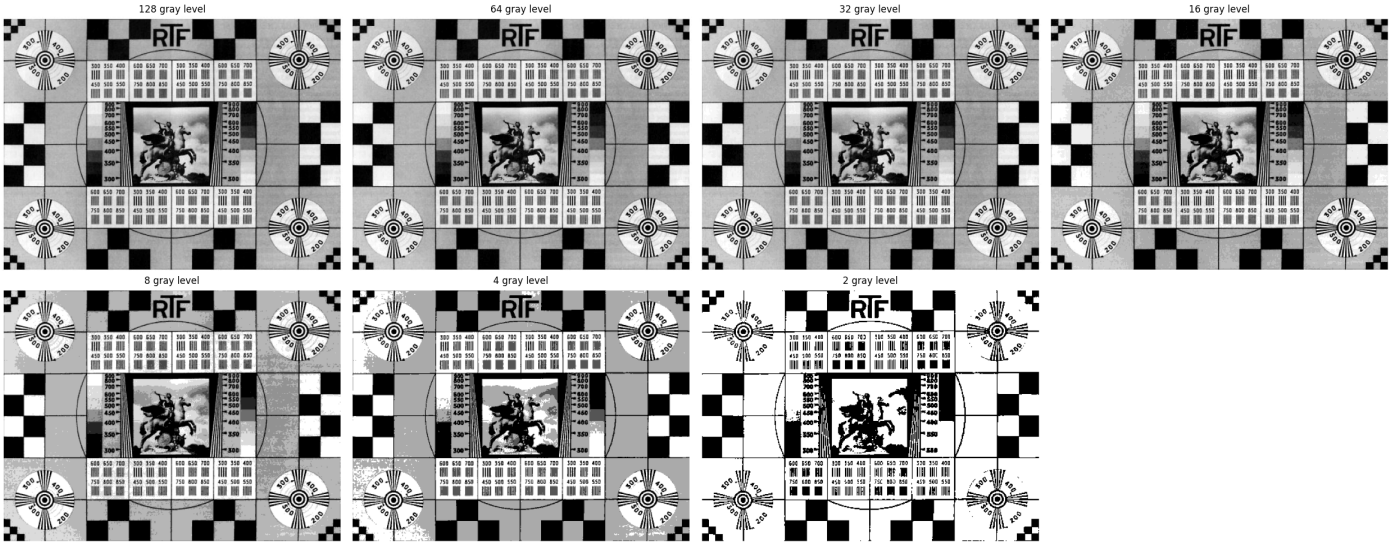

Fig. 16: Quantization of *test pattern* image.

Finally, we apply the quantization to *patterns* image. The code in shown below and the output is in Figure 17. Since in this case half of the original image has evident shades between black and white, we notice the effect of quantization already from the usage of 16 levels.

```
#c) Image patterns
plt.figure(figsize=(25, 10))
for i, levels in enumerate(gray_levels):
    quantized_image = quantize_fun(image_patterns_array, levels)
    plt.subplot(2, 4, i + 1)
    plt.imshow(quantized_image, cmap='gray')
    plt.title(f'{levels} gray level')
    plt.axis('off')

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

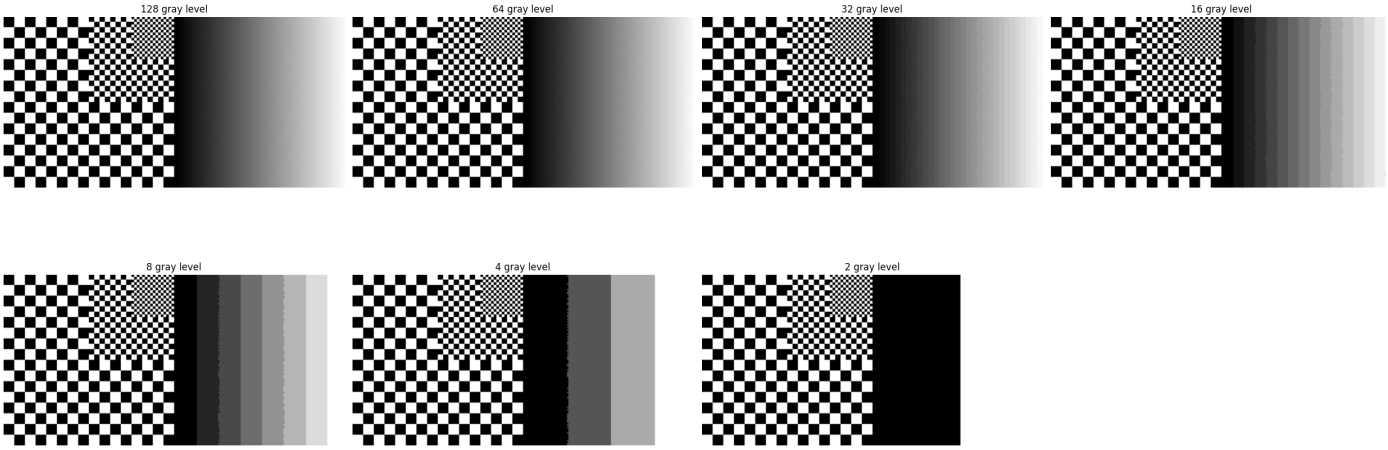

Fig. 17: Quantization of *patterns* image.

In general we know that

$$\text{Number of levels} = 2^{\text{Number of bits}}.$$

Hence, after the previous considerations, we can say that in all the cases we need at least 5 bits to have a good quality image, that correspond to 32 levels of gray. This can change from place to place of the image: more detailed zones can require more bits, i.e. higher gray levels.

4 Indexed color

In this section we analyze the indexing of the color. We first analyze it through the single channel *chro* image. The code for its acquisition is shown below and the original image is in Figure 18.

```
#Loading the image 'chro'
image_chro = Image.open("C:\\Users\\sofyc\\OneDrive\\Desktop\\UPEC\\Pattern recognition\\IP1\\IP1
    \\chro.tif")
image_chro_array = np.array(image_chro)

print(image_chro_array.shape)

#Plot without colormap
plt.imshow(image_chro_array, cmap='gray')
plt.axis('off')  #Without axes
plt.show()
```

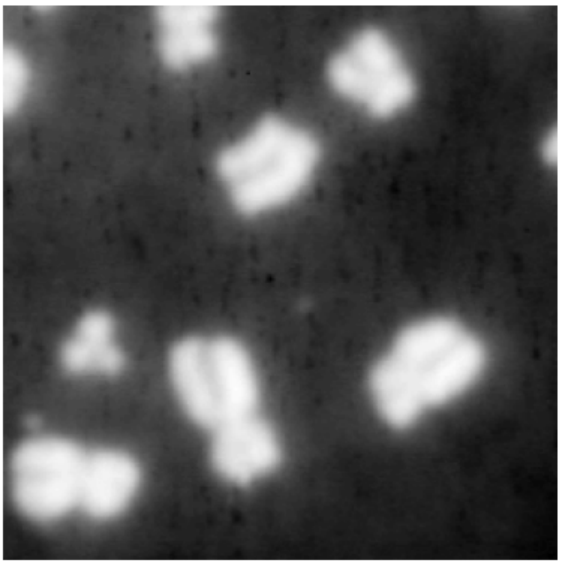

Fig. 18: *chro* image - single channel.

After displaying it without any map, we repeat this passage using *jet*, *hsv*, *hot* and *tab10* color maps (*lines* is just in Matlab, so we used the most similar in Python, *tab10*). The results are shown in Figure 19.
Some important considerations to are:

- Jet is a map with colors between blue and red, including yellow and green. It is useful to highlight gradual differences, and it is usually used in geospatial data, temperatures or applications where it is important to underline peaks.
- Hsv is a map based on Hue, Saturation and Value, where colors alternate in a cycling way. It is usually used in periodic data, such as phase signals.
- Hot is a map that varies between black and red, including orange, yellow and white. It is usually used in data with increasing intensity levels like heat map, where it is important to enfasise high values.
- Tab10 is a map with different colors used to distinguish between different categories/classes. Hence, it is usually used for categorical data, for example in bar or line plots.
- Lines in Matlab is usually used in line graphs with different variables to recognize.

```
#'lines' is not in matplot.lib. It is a colormap from Matlab. The most similar one is 'tab10'.
colormaps_list = ['jet', 'hsv', 'hot', 'tab10']

#Plot of the image with different colormaps
plt.figure(figsize=(20, 10))
```

```
6
7  for i, map in enumerate(colormaps_list):
8      plt.subplot(1, len(colormaps_list), i + 1)
9      plt.imshow(image_chro_array, cmap=map)
10     plt.title(f'Colormap: {map}')
11     plt.axis('off')  #Without axes
12
13 plt.tight_layout()
14 plt.show()
```

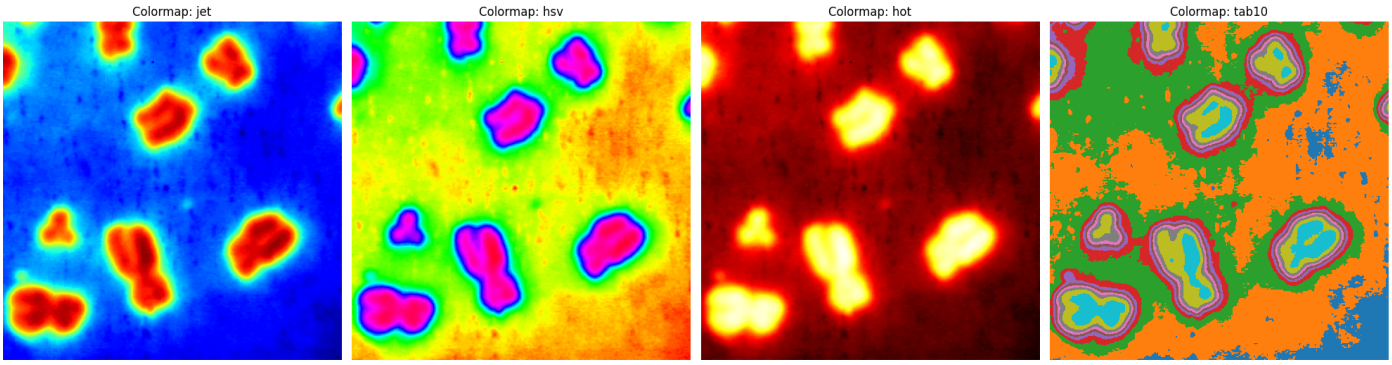

Fig. 19: Different color maps for *chro* image.

Now, we repeat the process with the image *glucose*. The image acquired is a matrix in gray scale. To convert it in RGB scale, we use the original palette of the image to create a personalized color map. Using it, I converted the gray scale of the image into a RGB one, mapping the intensity values of the colors correspondent to the palette. The image is shown in Figure 20. The code performed is shown below:

```
1  #Loading the image 'glucose'
2  image_glucose = Image.open("C:\\Users\\sofyc\\OneDrive\\Desktop\\UPEC\\Pattern recognition\\IP1\\
       IP1\\glucose.tif")
3  image_glucose_array = np.array(image_glucose)
4
5  print(image_glucose_array.shape)
6
7  #In Python we don't have the function 'ind2rgb', but have 'mcolors' from matplotlib.colors.
8  #First, we take the colormap
9  palette = image_glucose.getpalette()
10
11 #Then, we convert the palette to RGB
12 if palette:
13     #Converting the colormap in a list of tuples RGB
14     RGB_palette = [tuple(palette[i:i+3]) for i in range(0, len(palette), 3)]
15     #Creating a personalized colormap, though the normalization of RGB values in [0, 1]
16     cmap = mcolors.ListedColormap(np.array(RGB_palette) / 255.0)
17 else:
18     print("Colormap Error.")
19
20 #Plot
21 image_RGB = cmap(image_glucose_array / 255.0)[:, :, :3]  #We normalize the image
22
23 plt.imshow(image_RGB)
24 plt.axis('off') #Without axes
25 plt.show()
```

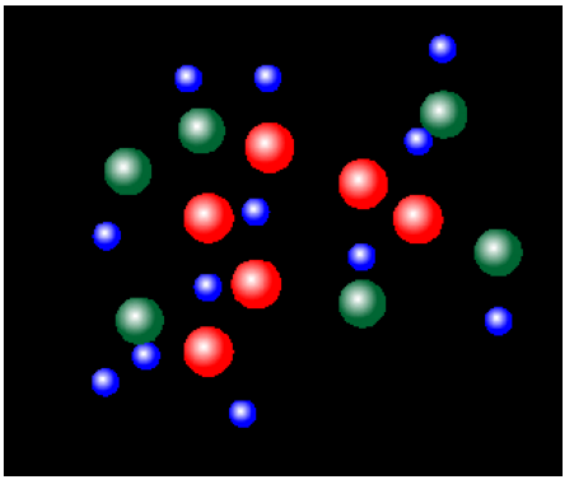

Fig. 20: *glucose* image with personalized color map.

After this, I saved the image into *JPEG* (lossy compression) and *TIFF* format (lossless compression), in order to compare the size of the files. To see the difference in terms of size we use the function *getsize* from *os* library. The JPEG image has 7466 bytes, while the TIFF image has 242870 bytes. As expected, in JPEG format we lose information and reduce the dimension. The code is the following:

```
import os

#Converting the array numpy in a object Image from PIL
image_RGB_PIL = Image.fromarray((image_RGB * 255).astype(np.uint8))

#Saving in TIFF format - without compression, and JPEG format - with compression
image_RGB_PIL.save("glucose no compression.tif")
image_RGB_PIL.save("glucose compression.jpg")

print(f"JPEG size: {os.path.getsize('glucose compression.jpg')} bytes")
print(f"TIFF size: {os.path.getsize('glucose no compression.tif')} bytes")
```

Now, we work on the three channels *spectrum* image. We want to transform it into a single channel. First, we load the RGB image and we display it (see code below). The Figure obtained is 22 and it represents the original image.

```
#Loading the image 'spectrum'
image_spectrum= Image.open("C:\\Users\\sofyc\\OneDrive\\Desktop\\UPEC\\Pattern recognition\\IP1\\
    IP1\\spectrum.png")
image_spectrum_array = np.array(image_spectrum)

#Plot of the original RGB image
plt.imshow(image_spectrum_array)
plt.title("Original image Spectrum.")
plt.axis("off") #Without axes
plt.show()
```

Fig. 21: Original image *spectrum*.

After this, we create a personalized color map using the file *map.txt* that contains one channel colors. We create a function *rgb_to_index* that searches the closest color in the map for each pixel, since the suggested function **??** mentioned in the task, is just for Matlab. This function is applied to

each pixel, and the value obtained are saved in an array. The result is a single channel image where every pixel is represented by its index in the color map. The indexing is useful to reduce the data size. The image obtained is shown in Figure 22, while the code performed is the one below.

```
#Reading the colormap
colormap = np.loadtxt("C:\\Users\\sofyc\\OneDrive\\Desktop\\UPEC\\Pattern recognition\\IP1\\IP1\\map.txt")
#'colormap' is a numpy array of lists of RGB values.

#Step 1: Create a personalized/custom colormap, through ListedColormap which takes an array of RGB colors.
cmap = mcolors.ListedColormap(colormap)

#Step 2: Check to see if the image has four channels (RGBA). In this case we remove the alpha channel.
if image_spectrum_array.shape[2] == 4:
    image_spectrum_array = image_spectrum_array[..., :3]  #We maintain just the first three channels (RGB).

#Step 3: Reduce the RGB image to a single channel.
#We index the image based on the custom colormap by finding the closest color for each pixel.
#To do this, we create a function to find the index of the closest color in the colormap.
def rgb_to_index(rgb_image_pixel, colormap):
    #We calculate the Euclidean distance between the pixel of the image and all colors in the colormap.
    diff = np.linalg.norm(colormap - rgb_image_pixel, axis=1)
    return np.argmin(diff)  #Return the index of the closest color.

#Step 4: Apply the indexing function to the image.
#We create an empty array to store the indexed image.
indexed_image_spectrum = np.zeros((image_spectrum_array.shape[0], image_spectrum_array.shape[1]), dtype=int)

#Iterating over each pixel of the image to find its index in the colormap.
for i in range(image_spectrum_array.shape[0]):
    for j in range(image_spectrum_array.shape[1]):
        #Getting the index of the closest color in the colormap for the current pixel.
        indexed_image_spectrum[i, j] = rgb_to_index(image_spectrum_array[i, j, :3], colormap)

#Plot of the imndexed image
plt.imshow(indexed_image_spectrum)
plt.title("Indexed Image from Spectrum.")
plt.axis("off") #Without axes
plt.show()
```

Fig. 22: *spectrum* image in one channel.

As last step, we repeat the same process on the image *umbrella*. The original image is shown in Figure 23, in RGB format.

```
#Loading the image 'umbrella'
image_umbrella= Image.open("C:\\Users\\sofyc\\OneDrive\\Desktop\\UPEC\\Pattern recognition\\IP1\\IP1\\umbrella.jpg")
image_umbrella_array = np.array(image_umbrella)

#Plot of the original RGB image
plt.imshow(image_umbrella_array)
plt.title("Original image Umbrella.")
plt.axis("off") #Without axes
plt.show()
```

Fig. 23: Original image *umbrella*.

Now, we apply again the same indixing function created before, and we use the same color map values from the text file. Again, each pixel of the image is associated to a closer value in the color map, transforming the RGB image into a single channel one. The image obtained is shown in Figure 24.

```
#Step 4: Apply the indexing function to the image.
#We create an empty array to store the indexed image.
indexed_image_umbrella = np.zeros((image_umbrella_array.shape[0], image_umbrella_array.shape[1]),
    dtype=int)

#Iterating over each pixel of the image to find its index in the colormap.
for i in range(image_umbrella_array.shape[0]):
    for j in range(image_umbrella_array.shape[1]):
        #Getting the index of the closest color in the colormap for the current pixel.
        indexed_image_umbrella[i, j] = rgb_to_index(image_umbrella_array[i, j,:], colormap)

#Plot of the imndexed image
plt.imshow(indexed_image_umbrella)
#plt.title("Umbrella image in one channel.")
plt.axis("off") #Without axes
plt.show()
```

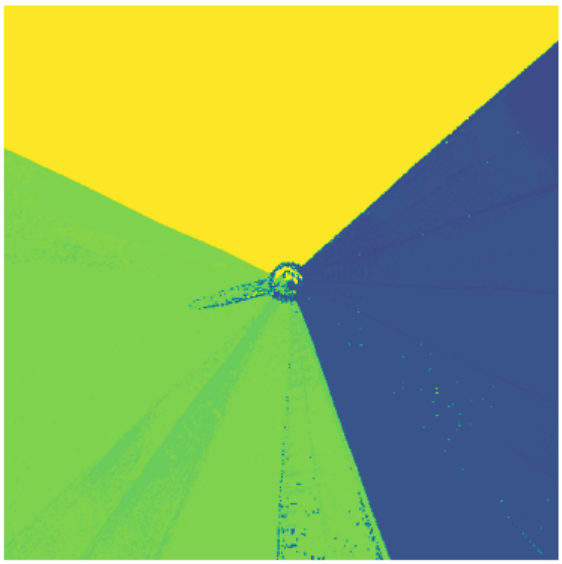

Fig. 24: *umbrella* image in one channel.

As a final consideration, we can say that both *spectrum* and *umbrella* image are full of different colors and shades, and this indexing technique is useful in these cases to avoid color redundancy, and where a visual representation based on a predefined color map is still desired.

References

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

[2] Kathamali, W.: Sampling & Quantization in Digital Image Processing. https://wimarshikathamali1995.medium.com/sampling-quantization-in-digital-image-processing-8c4490357039

[3] Community, S.O.: Convert RGB Image to Index Image. https://stackoverflow.com/questions/42750910/convert-rgb-image-to-index-image

[4] Community, S.O.: Equivalent to MATLAB's Ind2rgb in Python. https://stackoverflow.com/questions/72063045/equivalent-to-matlabs-ind2rgb-in-python

[5] Community, S.O.: What Algorithms Does Rgb2ind in MATLAB Use? https://stackoverflow.com/questions/18817997/what-algorithms-does-rgb2ind-in-matlab-use

[6] MathWorks: Rgb2ind Documentation. (2024). https://fr.mathworks.com/help/matlab/ref/rgb2ind.html