Artificial Intelligence and Computer Vision Laboratory

Report from laboratory 3

Ewa Kobiela

Aim of the laboratory

The aim of this laboratory was to generate a set of image transformation. This contained image quantization, stretching and equalizing an image's histogram, image thresholding, discrete Fourier transform and inverse Fourier transform. This report contains detailed explanation of created code. Scripts which were written to perform laboratory's tasks are attached to this file.

Task 1

In Task 1 an image histogram stretching for quantized greyscale low and high frequency images was performed.

```
#Load images
img = cv2.imread(r'lena.png',0)
cv2.imshow('High frequency image',img)
imgHighFreq = img.copy()
img=img*0
img2 = cv2.imread(r'lowfreq.png',0)
#cv2.imshow('Low frequency image',img2)
imgLowFreq = img2.copy()
img2=img2*0
```

To begin with, images were read from the directory in greyscale and copied to new variables to avoid mistakes and errors in further code. The base variables were cleared. Only high frequency picture was displayed as low frequency image was not used in this example.

```
#Define qualtization level and perform quantization
lvl_of_intensity = 32     #32, 64 or 128
quant = quantize(imgHighFreq, lvl_of_intensity)
#OR
#quant = quantize(imgLowFreq, lvl_of_intensity)
cv2.imshow('Quantized image',quant)

#Stretch histogram for imgs (Task 1)
str = stretch(quant)
cv2.imshow("Stretched histogram", str)
```

Secondly, quantized image was prepared using function quantize() which is described below in this report. As the function takes two arguments, base image and level of quantization, a place for choosing the level was prepared. Values 32, 64 and 128 were recommended in task description. once again, operations were performed only on high frequency image, but place for inserting low frequency image is prepared in code. The quantized image was then displayed.

Image stretching was obtained in function stretch() which took quantized image as an argument. Results were displayed in a separate window.

```
def quantize(img, lvl):
    image = np.float32(img)
    step = 256//lvl
    list = np.zeros(lvl, dtype=int)
    for s in range(0,lvl):
        list[s]=s*step
    rows, cols = img.shape[:2]
    for row in range(0, cols):
        for col in range(0, cols):
            val = img[row, col]
            new_val = list.flat[np.abs(list - val).argmin()]
            img[row,col]=new_val
    return img
```

In quantize() function image was converted into an array of float32 type to allow further calculations. Firstly, step of quantization was calculated using predefined value of desired levels and in the first loop a vector containing all step values was created. In the second loop script gathered value of each pixel of an image and found nearest level in the step list. Then it replaced pixel value with nearest value from the list.

```
def stretch(img):
    rows, cols = img.shape[:2]
    max_val = np.max(img)
    min_val = np.min(img)
    for row in range(0, cols):
        for col in range(0, cols):
            val = img[row, col]
            new_val = ((val-min_val)/(max_val-min_val))*255
            img[row, col] = new_val
    return img
```

In stretch() function histogram of a given image was stretched. Operations were based on algorithm created with the use of equation given in lecture:

$$newValue = \frac{currentValue - minValue}{maxValue - minValue} * 255$$

.

Task 2

In the second task, image reading and quantization were done exactly the same as it the first task. Thus, the explanation will not be repeated. However, function stretch() was replaced with function equalize() which was required in this task.

```
#Equalize historgam for imgs (Task 2)
#for base images:
high_eq = equalize(imgHighFreq)
cv2.imshow("Original image with equalized histogram", high_eq)
#OR
#low_eq = equalize(imgHighFreq)
#cv2.imshow("Original image with equalized histogram", low_eq)
#for quantized image:
equ = equalize(quant)
cv2.imshow("Quantized image with equalized histogram", equ)

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

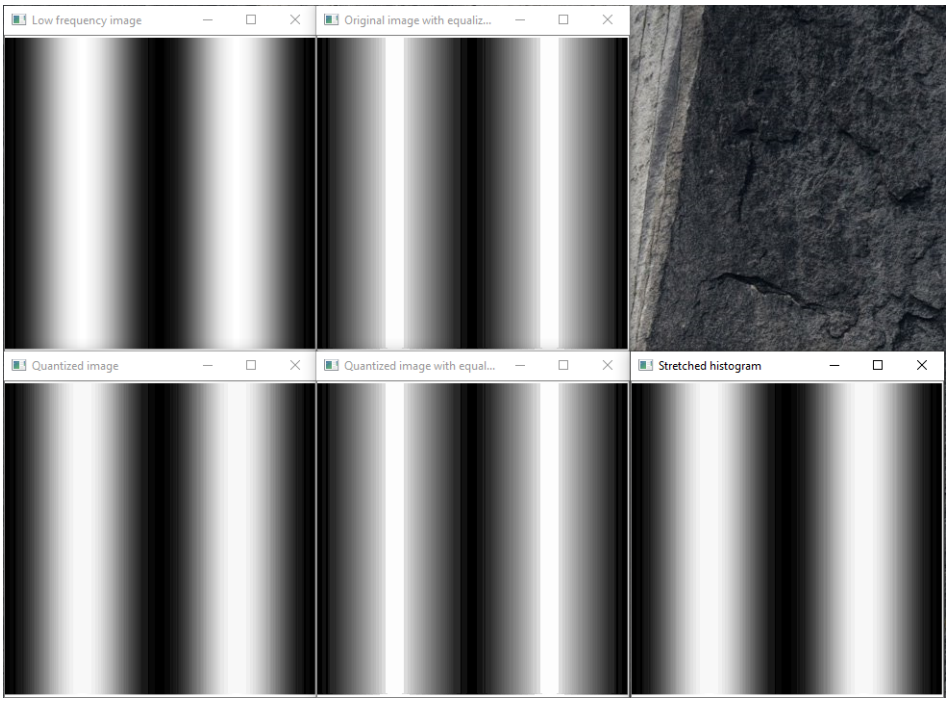
This code also gives result only for high frequency image, but can be easily reversed to be used with low frequency image.

```
def equalize(img):
    equ=cv2.equalizeHist(img)
    return equ
```

Function equalize() uses function equalizeHist() from OpenCV library and returns image with equalized histogram.

Results of tasks 1 and 2

The image above presents difference between different stages of high frequency image transformation. Quantization was performed with levels of intensity equal 32, which is the lowest value prepared for this exercise. However, the difference between quantized and original image is very small and not so visible. When using 64 or even 128 levels, the difference is even more subtle. It shows that a high frequency image can be stored in quantized form without a great loss of information. When histogram of a quantized image is stretched, it looks even more qualitative then the original one. Images with equalized histogram, both of original and quantized image, contains more amount of pixels with values close to white or black.



Like in previous example, the difference between original and quantized images of low frequency is hardly visible. Thus, image with stretched histogram also looks similar. Greater difference occurs when histogram of original and quantized image is equalized: lines of pure white colour are visibly shrinked and transition between black and white colour is less smooth.

Task 3

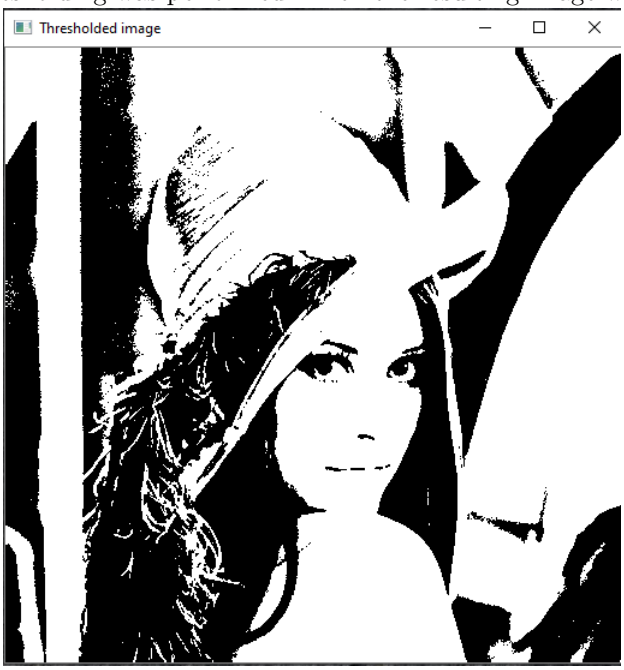
In the third task image tresholding and finding negative value of an image was performed.

```
#Load image in grayscale
img = cv2.imread(r'lena.png', 0)
cv2.imshow('Original image',img)
img1 = img.copy()
img1=np.uint8(img1)
img=img*0
```

Firstly, and image was read from a directory in greyscale and converted into type uint8. Once again, a copy of an image was created in a new variable and the previous one was cleared in order to avoid any mistakes in code.

```
#Selecting some valuefor tresholding:
tr_val = 120
#Binary thresholding and showing results:
ret, img_thre = cv2.threshold(img1, tr_val, 255, cv2.THRESH_BINARY)
cv2.imshow('Thresholded image', img_thre)
```

Then, thresholding of the image was performed. A tresholding value was set as a default to 120, but can be easily replaced with another proper one. In thresholding itself a function threshold() from OpenCV library was used. The function returns two arguments, one being a threshlded image, and takes four arguments: base image, thresholding value, maximum value of pixel allowed and method of thresholding, in this example set to THRESH BINARY as binary thresholding was performed. Then the resulting image was displayed.



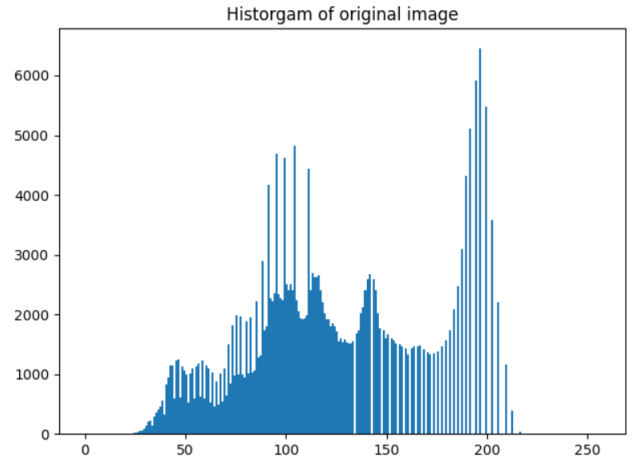

```
#Calculating and showing negative image:
img_neg = negative(img1)
cv2.imshow('Negative image', img_neg)
def negative(img):
    rows, cols = img.shape[:2]
    for row in range(0,rows):
        for col in range(0,cols):
            img[row, col] = 255 - img[row, col]
    return img
```

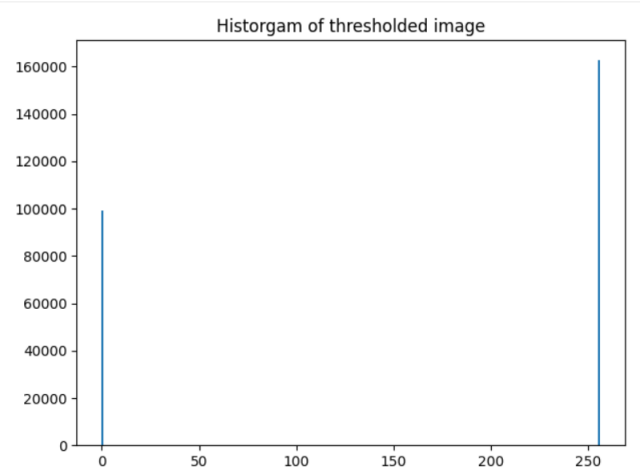
In order to obtain negative image, the function negative() was created and the mentioned above base image was passed into it. In the function a value of each pixel is found and is subtracted from 255, which is the maximum allowed

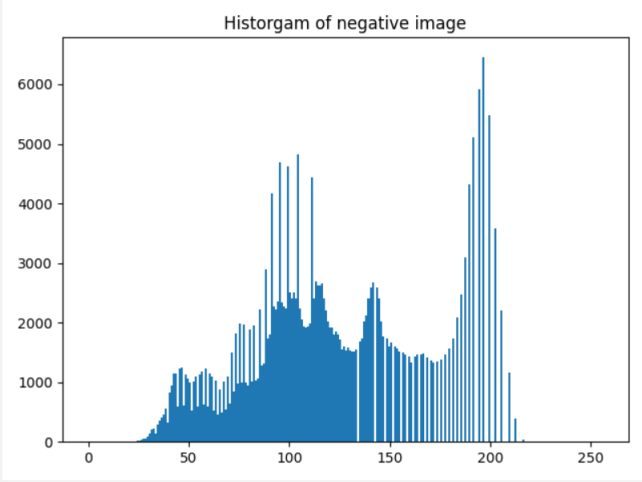
value of pixel. This way, a negative image was created.


```
#Plotting image historgams
plt.hist(img1.ravel(),256,[0,256]); plt.title('Historgam of original image'); plt.show()
plt.hist(img_thre.ravel(),256,[0,256]); plt.title('Historgam of thresholded image'); plt.show()
plt.hist(img_neg.ravel(),256,[0,256]); plt.title('Historgam of negative image'); plt.show()
```

Last step was to plot histograms of each step of image transformation.







As it is visible on images above, histograms of original and negative image are similar. However, a histogram of thresholded image differs from them. This occurs, because in thresholding a value of a pixel can take only one of two values, 0 or 255, which is visible on the graph in two bars.

Task 4

In the third task Fourier transform and inverse Fourier transform was performed.

```
#Loading image in grayscale
img = cv2.imread(r'lena.png', 0)
img1 = img.copy()
img1=np.uint8(img1)
img=img*0
#Showing original image
plt.subplot(221),plt.imshow(img1, cmap = 'gray')
plt.title('Original Image'), plt.xticks([]), plt.yticks([])
```

As previously, an image was read from directory and copied to a new variable and a case variable was cleared. The base image was displayed using the subplot().

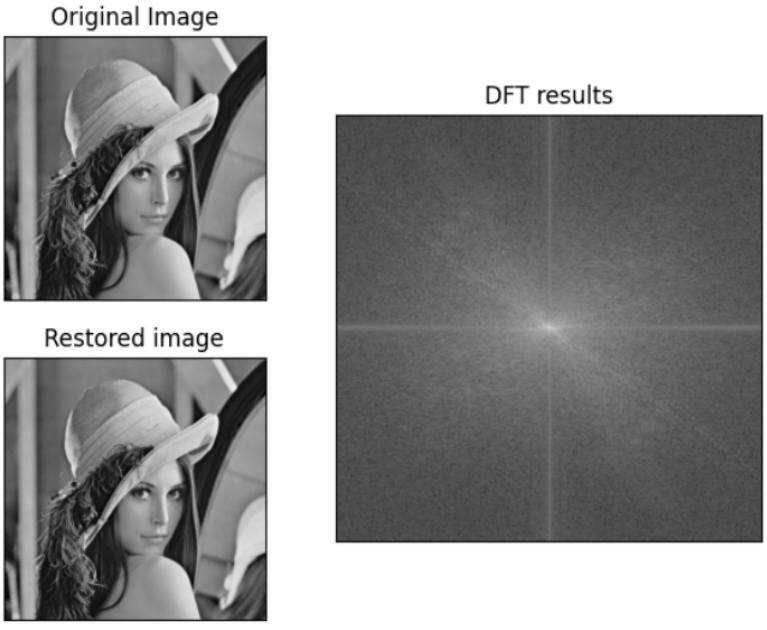
```
#Computing DFT
dft = cv2.dft(np.float32(img1),flags = cv2.DFT_COMPLEX_OUTPUT)
dft_shift = np.fft.fftshift(dft)
magnitude_spectrum = 20*np.log(cv2.magnitude(dft_shift[:,:,0],dft_shift[:,:,1]))
#Showing DFT results
plt.subplot(122),plt.imshow(magnitude_spectrum, cmap = 'gray')
plt.title('DFT results'), plt.xticks([]), plt.yticks([])
```

In order to perform DFT, the image was converted to float32. To perform DFT itself, a ready function dft() was used from OpenCV library. The function takes four arguments, which is base image, output image, flags and non-zero-rows. In this case only the first three arguments were passed with flags set to DFT COMPLEX OUTPUT in order to ensure that the output matrix is of the same size as the input. Next step was to move DC component of the resolt from the top left corner to the center which was performed in the following two line. Lastly, the magnitude spectrum was presented in the form of image.

```
#Calculating image dimensions
row = img1.shape[0]
col = img1.shape[1]
mid_row = row // 2
mid_col = col // 2
#Creating mask (centre pixel = 0, rest = 1)
mask = np.ones((row, col, 2), np.uint8)
mask[mid_row:mid_row, mid_col:mid_col] = 0
#Applying mask to DFT image
fshift = dft_shift * mask
f_ishift = np.fft.ifftshift(fshift)
#Inverse DFT transformation
res = cv2.idft(f_ishift)
res = cv2.magnitude(res[:, :, 0], res[:, :, 1])
#Showing inverse transformation results
plt.subplot(223),plt.imshow(res, cmap = 'gray')
plt.title('Restored image'), plt.xticks([]), plt.yticks([])

plt.show()
```

In order to perform an inverse transform, a mask was created. It consists of ones in every element but the middle one, which is set to zero. Then, the mask was multiplied by the shift calculated in the previous step. The inverse transform was obtained using idft() function from the OpenCV library and the result was displayed.



Personally, I cannot see any significant differences between the original and restored images. The restored one is little blurred in comparison to the original one, but not much difference was noticed.