CENG466 - Fundamentals of Image Processing Report 2

Bengisu Ayan
Computer Engineering
Middle East Technical University
Ankara, Turkey
e223697@metu.edu.tr

Ceren Gürsoy
Computer Engineering
Middle East Technical University
Ankara, Turkey
e2237485@ceng.metu.edu.tr

I. Introduction

In this assignment, we aimed to familiarize ourselves with fundamental frequency domain image enhancement techniques. We worked on noise reduction and edge detection using Fourier transforms and image compression using Wavelet transforms. We have used python3.5.2 along with numpy, cv2, pywt, pandas libraries.

II. Question 1 - Noise Reduction using Frequency Domain Filtering

In this part of the assignment, we are required to identify the type of noise in the images A1.png, A2.png and A3.png, investigate the present noise and remove them to recover the original image.

We planned to follow the steps listed below to reduce the noise in the images:

1) Transform the image to Fourier domain
2) Flip all the corners to the center to better analyze the Fourier transformed image. This way, the lower frequencies are at the center of the image.
3) Identify the type of noise the image contains
4) Develop a specific filter to that noise
5) Apply the filter by multiplying the image with the filter
6) Inverse flip the corners
7) Inverse Fourier transform the image to obtain the original image.

There were two steps we faced difficulties during the above process:

- Between step 2 and 3, we had trouble saving the image using cv2 $imwrite$ function. After searching on the internet, we learned that we needed to take the absolute value of the image and apply logarithm in order to properly view the magnitude component of the Fourier transformed image.

$$aX\_magnitude = 20 * np.log(np.abs(aX_{shifted})) \quad (1)$$

- In the last step, we had trouble saving the inverse Fourier transformed image since after the inverse transformation, the image contained real and imaginary parts, which caused problems while using cv2 $imwrite$, we decided to take the absolute value of inverse transformed image.

$$aX\_denoised = np.abs(aX\_denoised) \quad (2)$$

In the next three subsections, we explain the specific filters we designed for the different noises we observed in the Fourier domain.

A. A1

In A1.jpg, we observed a periodic noise with lots of spikes in the Fourier domain. Low pass filters pass only the low frequency components of the image, within a circle, around the origin. As the noise in this image were not very close to the center, we decided to use a low pass filter with a small radius. Out of ideal low pass filter, Gaussian low pass filter and Butterworth filter, we preferred to work with Butterworth filter. We eliminated ideal low pass filter to avoid non-smooth irregularities due to the effect of the discontinuity and we opted for Butterworth filter for the extra smoothing effect it provides compared to the Gaussian filter.

While designing the filter, we used the following formula from the book [1]:

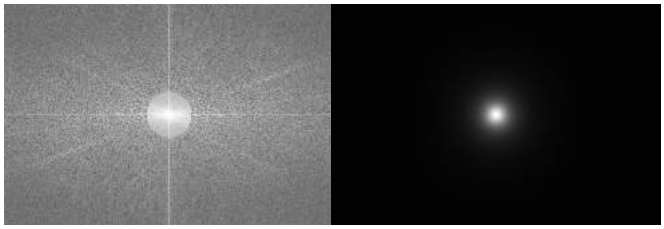
$$H(u,v) = \frac{1}{1 + [D(u,v)/D_0]^{2n}} \quad (3)$$

where $D(u,v)$ is given by:

$$D(u,v) = [(u - P/2)^2 + (v - Q/2)^2]^{1/2} \quad (4)$$

where P and Q are the sizes of the image. The n in the formula is the filter order and D_0 (or r) is the cutoff frequency. While experimenting with the filter design, as we increased the filter order, it got harder to discern the tulips from the background, thus we settled on 0.48 as n value, which is pretty small. The more we increased r, the less frequency values passed the filter, making the image blurrier. We settled on 480 for r value after a lot of experiments. We had to trade off the clarity of the image in order to reduce the noise in the image. However, we tried to make it less blurry while keeping the general shape of the tulips.

We found working with A1 really hard as we did not manage to completely eliminate the noise from the image without making it blurry. We tried to reserve informative



(a) A1 in Fourier domain before (b) Butterworth filter after

Fig. 1: A1 in Fourier domain and the Butterworth filter we designed

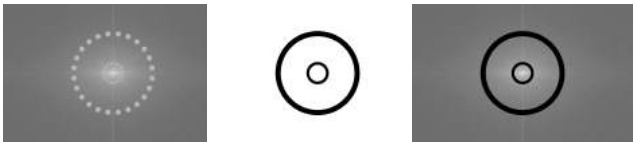
structures like edges and boundaries as much as possible. However, in our case a complete recovery was not obtained.

(a) original A1 before (b) denoised A1 after

Fig. 2: A1 before and after noise reduction

B. A2

In A2.jpg, we clearly observed the noise which consisted of 2 nested circular shapes. We decided to apply band reject filter to A2 to cover those circles.



(a) A2 in Fourier domain (b) Band reject filter (c) Filtered image in Fourier domain

Fig. 3: Filter applied to A2 in Fourier Domain

While implementing the filter, we used the idea of distance in order to create circular shapes. The following equation is used to create a circular shape with a distance d from the center of the image and width w.

$$d + q \geq \sqrt{[(u - P/2)^2 + (v - Q/2)^2]^2} \geq d \qquad (5)$$

with P and Q indicating the size of the image. We found the exact bandwidth and distance of the circular shapes from the center of the image by experimenting with different numbers.

As can be seen from Figure 4, the small noise in the corners of the image are totally reduced. This was the image we believe we are the most successful as we managed to elevate the noise completely without loss in substantial information such as edges and boundaries.

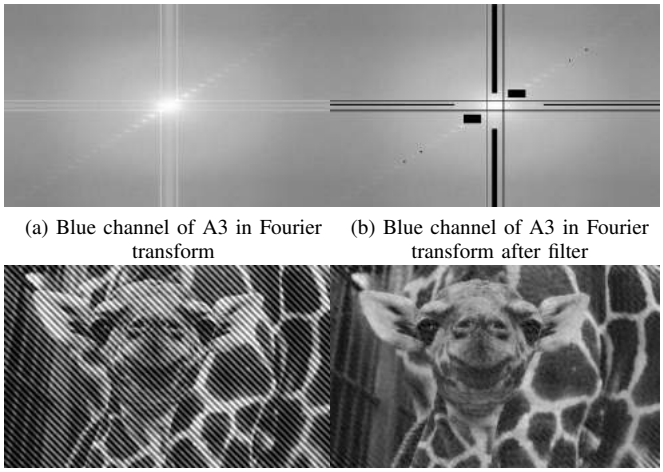
(a) original A2 before (b) denoised A2 after

Fig. 4: A2 before and after noise reduction

C. A3

A3.png is the only RGB image among the images we needed to apply noise reduction. We decided to separate the image to different channels and apply the general algorithm mentioned above to each channel, thus designing specific filters for each channel just like they are separate images.

After separating the images into 3 different channels and applying Fourier transforms, we realized that while there were slight changes between the width of the lines, the most prominent noise was cause by thin lines across the images in all three of them. For this problem, we decided to design a different kind of band reject filter compared to A2. We concealed the white lines with this band reject filter. After covering these lines, we had reduced most of the noise (crossed lines) on the giraffe's face. However, we decided to experiment a bit more with this type of noise reduction and tried to cover the bright spots in different parts of the image that appeared like noise similar to the ones in our lectures notes. However, the later procedure did not result in a dramatic change on our noise reduction but we decided to keep them. While finding the pixels corresponding to the lines and sports, we used this website [2].

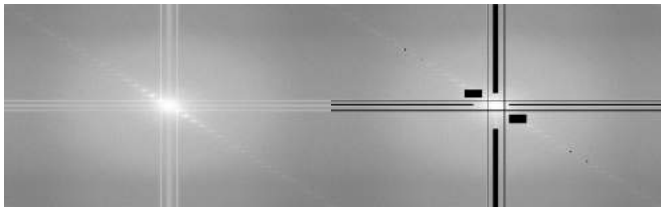


(a) Blue channel of A3 in Fourier transform (b) Blue channel of A3 in Fourier transform after filter

(c) Blue channel of A3 (d) Blue channel of A3 after filter

Fig. 5: A3 blue channel before and after filter applied

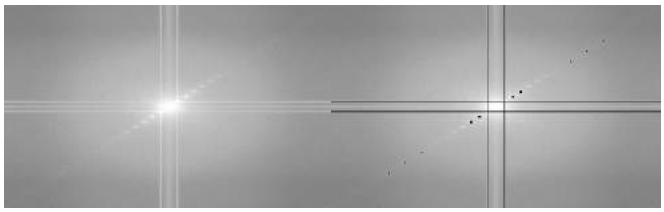
We then merged the inverse Fourier transformed images we denoised from each channel back to an RGB channel to obtain the denoised version of the original image. While we still have some noise near the corner of the image, we believe we mostly



(a) Red channel of A3 in Fourier transform (b) Red channel of A3 in Fourier transform after filter

(c) Red channel of A3 (d) Red channel of A3 after filter

Fig. 6: A3 red channel before and after filter applied

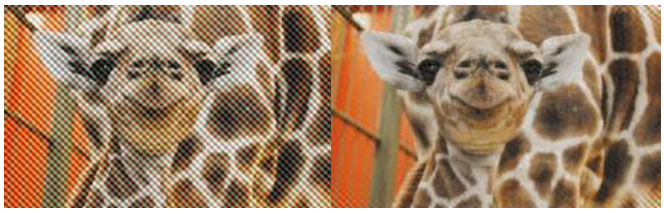


(a) Green channel of A3 in Fourier transform (b) Green channel of A3 in Fourier transform after filter

(c) Green channel of A3 (d) Green channel of A3 after filter

Fig. 7: A3 green channel before and after filter applied

managed to successfully denoise this image. This image took a lot longer than the last 2 images simply because it was actually 3 images to process and finding the pixels of the noise was a bit time consuming.



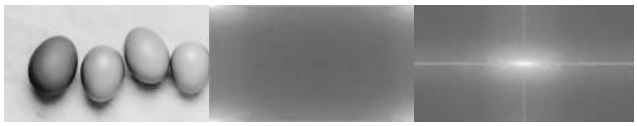
(a) original A3 before (b) denoised A3 after

Fig. 8: A1 before and after noise reduction

III. QUESTION 2 - EDGE DETECTION

In this section, we found edge maps of images in frequency domain using Fourier Transform. Our algorithm consists of these steps:

1) We transformed our images into Fourier domain using $fft2$ function of numpy. $fft2$ takes an image and applies 2D discrete Fourier transform.
2) Since light colors are placed at four corners due to properties of Fourier transform, we shifted Fourier image and brought them to the center using $fftshift$ function of numpy. This process is shown in Figure 9.
3) We applied a filter on images in frequency domain. Since we wanted to find edges of images and edges are high frequency components of an image, we passed only high frequency components. Therefore, we used Butterworth high pass filter.
4) This filter was applied by multiplying it by the shifted image. Since this multiplication must be element wise, we used $multiply$ function of numpy.
5) We applied inverse Fourier transform to filtered image in order to change its domain as space domain. Therefore, firstly we brought zero frequency components back to the corners using $ifftshift$ function of numpy. Then, we applied inverse Fourier transform using $ifft2$ function of numpy.
6) Since image consists of complex numbers due to Fourier transform, we took its absolute value using abs function of numpy in order to save it.



(a) (b) (c)

Fig. 9: a) Grayscale B3 b) Magnitude spectrum of B3 c) Magnitude spectrum centered of B3

A. Butterworth High Pass Filter

Before applying Butterworth high pass filter, we tried ideal high pass filter with different cutoff frequencies. However, ideal high pass filter caused irregularities in images as shown in Figure 10. In (a), some parts that are not belong to edges can be seen. Therefore, we chose Butterworth high pass filter. We wrote this filter as a function according to formula from the book [3]:

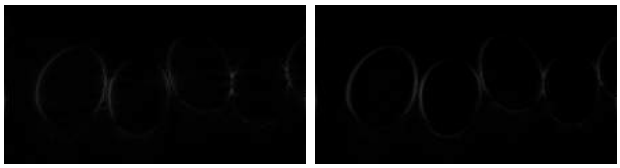
$$H(u,v) = \frac{1}{1 + [D_0/D(u,v)]^{2n}} \tag{6}$$

where $D(u,v)$ is given by:

$$D(u,v) = [(u - P/2)^2 + (v - Q/2)^2]^{1/2} \tag{7}$$

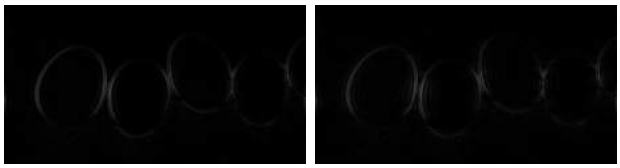
where P and Q are the sizes of the image.

Then we tried better results by arranging parameters. For example, when we increased order value (n) of the filter, ringing around edges occurred more for the same cutoff frequencies (D_0) as shown in Figure 11.



(a) (b)

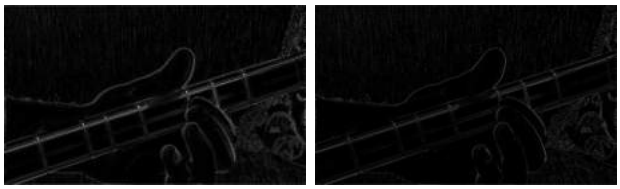
Fig. 10: a) Ideal high pass filter on B3 b) Butterworth high pass filter on B3



(a) (b)

Fig. 11: a) BHPF with $n = 2$ on B3 b) BHPF with $n = 8$ on B3

Moreover, we observed that when D_0 was increased, boundaries were less distorted. For example, in Figure 12, first image was created with $D_0 = 30$ and second image was created with $D_0 = 80$. In the first image, irregularities around fingers in front occur more and boundaries are not as clean as the second image.



(a) (b)

Fig. 12: a) BHPF with $D_0 = 30$ for B1 b) BHPF with $D_0 = 80$ for B1

All these results showed us, we should choose D_0 as high value, whereas we should choose n as small value. However, we should decide these parameters according to images. For example, for B3, when we applied $D_0 = 80$ and $n = 2$, we obtained the result shown in Figure 13. As can be seen from the figure, edges are almost invisible because after a point, some parts of the edges that needed to be detected were also filtered out.

Therefore, after application of this filter with different cutoff frequencies and n values, we obtained best result from $D_0 = 80$ and $n = 2$ for B1 and B2 and $D_0 = 50$ and $n = 2$ for B3. These results are shown in Figure 14, Figure 15 and Figure 16.

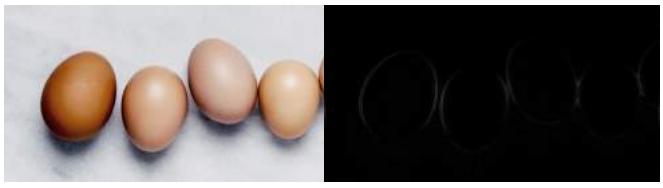
Fig. 13: BHPF with $D_0 = 80$ and $n = 2$ for B3

(a) (b)

Fig. 14: a) Original B1 b) Edge detection with BHPF with $D_0 = 80$ and $n = 2$ for B1

(a) (b)

Fig. 15: a) Original B2 b) Edge detection with BHPF with $D_0 = 80$ and $n = 2$ for B2



(a) (b)

Fig. 16: a) Original B3 b) Edge detection with BHPF with $D_0 = 50$ and $n = 2$ for B3

IV. QUESTION 3 - IMAGE COMPRESSION WITH WAVELET DECOMPOSITION

In this part of the assignment, we developed an image compression algorithm based on Wavelet transforms. From the two methods of compression algorithms, we chose to implement a lossy method as lossy methods tend to have a higher compression rate. We tried to design the algorithm such that the information losses caused by the algorithm are generally tolerable by the end user.

A. Algorithm

In the compress method, we planned to Wavelet transform the image, quantize the image, encode it if possible and then save it. In the decompress method, we planned to decode and then apply inverse Wavelet transform it.

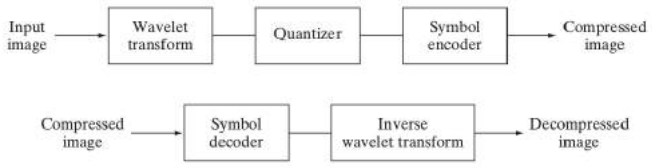


Fig. 17: Compression steps

1) Compress Method:

We first applied 2d Wavelet transform using the $wavedec2$ method from $pywt$ library of python to the given images. We decided to apply a level 4 Wavelet transform as [4] suggests that Wavelet decomposition level is an important factor in image compression. After experimenting with 1, 2, 4 and 8 decomposition levels, we decided on level 4 and we believe it achieved a good balance between reconstruction error and compression ratio. We then had to choose the Wavelet we wanted to apply Wavelet transform with. As [5] suggests, the most widely used expansion functions for wavelet-based compression are the Daubechies wavelets and biorthogonal wavelets. We decided on $bior4.4$ as JPEG2000 also uses that one. We then kept the coefficients as arrays using $pywt.coeffs\_to\_array$ function since it makes it easier to work with the coefficients resulting from a 4 level Wavelet decomposition.

We separated the quantization step to 2 different parts.

First, we applied thresholding to the coefficients that we obtained from the Wavelet transformation. We decided to keep the coefficient that are bigger than a certain threshold and set the other smaller coefficients to 0, as they do not carry very important information and there is not a big difference between these small coefficients and 0. To find this threshold value, we decided to keep a certain percentage of the values, thus we sorted the coefficients and found the last value we will allow past the treshold. In order to obtain the coefficients bigger than the threshold in the unsorted coefficient array, we created an array consisting of 1 and 0s. 1s signify the values bigger than the threshold and 0s signify the values below the threshold in the original unsorted coefficient array. We then multiplied this array with the coefficient array and set the smaller values to 0. After experimentation, we decided to keep the first %10 of the largest coefficient, as we decrease these percentages, we get higher compression rates but we then have to compromise the quality of the decompressed images. In fact C1 caused the bottleneck value for this value in our algorithm. When we kept the first %5 of the values, there was not much visible differences in C2 and C3, however C1 got blurry in certain parts. We aimed to get rid of the psychovisual redundancy with this technique as the differences very small Wavelet coefficients make are not very discernable to human eyes. This was also a lossy compression since we cannot recover the values we set to 0. During this procedure, [6] influenced us a lot.

Secondly, we decided to keep the coefficient values as integers rather than floats in our algorithm. After a few observations with the coefficients, we realized that we do not need to keep the integer values as 32 bit integers and 16 bits are enough with level 4 Wavelet decompositions of these images. Thus we decided to quantize the floats to the floor values of 16 bit integers, which significantly increased our compression rate. This is also one of the steps that make our compression algorithm lossy, there is no way we can recover these float values from integer values we obtained. Furthermore, we reduce coding redundancy in this part of the algorithm as we get rid of 16 extra bits required to keep integers when 16 bit integers suffice to store our coefficient values.

We then decided to save the output of the compression as npz files. We needed to keep both the slices of the arrays since this is a 4 level Wavelet decomposition based image compression, thus we first saved the quantized coefficients, then the slices that indicate where each coefficient begins and ends. Furhermore, as npz files are already compressed versions of themselves, it helped us save space. Previously we had tried saving the result as png, however the recovery was not preferable as we had large values in our quantized coefficients and png only allows values between 0 and 255. Moreover, the size of the numpy arrays we obtained were even bigger than the original png file which resulted in us deciding on compressed numpy arrays.

2) Decompress Method:

We loaded the npz file we saved in the compress method and extracted the quantized coefficients and the slices. We then converted the coefficient array back to the coefficient format using $pywt.array\_to\_coeffs(coeffs\_array, slices,' wavedec2')$. Then we inverse Wavelet transformed the image using $pywt.waverec2(coeffs, wavelet =' bior4.4')$. This process was fairly simple compared to compress method.

As can be seen from Figure 18, Figure 19 and Figure 20, there is not much difference visible to human eyes between the original and decompressed images, even though compressed files are smaller.

(a) original C1 before (b) decompressed C1 after

Fig. 18: C1 before and after compression

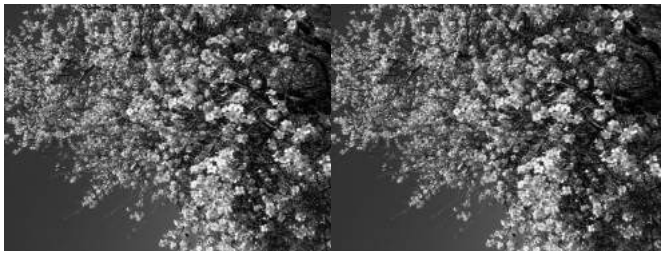
(a) original C2 before (b) decompressed C2 after

Fig. 19: C2 before and after compression

B. MSE and Compression Ratio Comparisons

We compared our results with JPEG standards according to same images compressed with JPEG in different quality values as shown in Figure 21 and Figure 22.

As can be seen from the tables, when the quality factor (represented as JPEG-quality in the tables) is high, image is less compressed and closer to its original version, therefore, mean square error (mse) values and compression ratios become low. For example, as can be seen from Figure 23 information loss in C1 compressed with JPEG1 is higher than JPEG80, and the mse and compression ratios are high in JPEG1.
For MSE values, we measured original images and their compressed versions with our algorithm, and similarly with



(a) original C3 before (b) decompressed C3 after

Fig. 20: C3 before and after compression

	Our Algorithm	JPEG1	JPEG10	JPEG50	JPEG80
C1	41.08574930826823	72.63200505574544	51.757397969563804	35.43237559000651	5.800960540771484
C2	1.1178322916666668	81.26527083333333	20.3663468 75	5.343703645833333	0.5889135416666667
C3	36.664334297180176	87.2390292485555	56.71092160542806	32.287237803141274	20.95677661895752

Fig. 21: Mean Square Error Values of the Images

	Our Algorithm	JPEG1	JPEG10	JPEG50	JPEG80
C1	2.536281844138023	31.65517955801105	11.883720930232558	3.9885399535332966	2.7871879845550454
C2	1.8646078501980554	24.230757711889787	16.37451932807124	6.7169498033436765	4.6515364935176935
C3	3.3941919620771897	37.63102917580798	14.600238845411 12	5.432621330768012	3.1500633869895633

Fig. 22: Compression Ratios of the Images

(a) (b) (c)

Fig. 23: a) Original C1 b) C1 compressed by JPEG1 c) C1 compressed by JPEG80

JPEG according to the formula:

$$MSE = \frac{1}{MN}\sum_{y=1}^{M}\sum_{x=1}^{N}[I(x,y) - R(x,y)]^2 \qquad (8)$$

where original image is I whose size is $N * M$ and R is the decompressed image.
MSE value denotes the average difference of the pixels all over the image. A higher value of MSE designates a greater difference amid the original image and processed image. This means that it is better the lower the MSE value is for the compression algorithm.

(a) (b) (c)

Fig. 24: a) C1 compressed by JPEG1 b) C1 compressed by our algorithm c) C1 compressed by JPEG80

(a) (b) (c)

Fig. 25: a) C2 compressed by JPEG1 b) C2 compressed by our algorithm c) C2 compressed by JPEG80

As a result, we obtained best results according to MSE values with C2 and its result is closer to JPEG80 as you can see in Figure 25. It is nearly impossible to see the differences between our compression and JPEG80. However, the resolution is pretty bad for JPEG1. Other two images

(a) (b) (c)

Fig. 26: a) C3 compressed by JPEG1 b) C3 compressed by our algorithm c) C3 compressed by JPEG80

are also good, because their result is closer to JPEG50. This means, information loss on images compressed with our algorithm is low and these compressed images are close to their original versions.

Secondly, we measured compression ratios by dividing size of original images to size of our compressed files and similarly, size of images compressed with JPEG. According to Figure 22, our compression ratios are low and our results are similar to JPEG80. Low compression ratios show us that our algorithm does not compress images into very small sizes compared to JPEG1, for example. Our compression for C2 is worse compared to C1 and C3, and also for JPEG for 80 quality. However, our compression ratio for C1 and C3 are generally between JPEG 50 and 80 qualities.

As a result, While our MSE values are similar to JPEG50 and JPEG80, they generally have better compression rates for similar MSE values. We could have improved the compression rate by keeping less of the biggest coefficients as explained in the algorithm section of the assignment, however we would then need to give up the quality of the images. We tried to find a middle ground between quality and compression ratio.

REFERENCES

[1] R. C. Gonzalez, Digital Image Processing, 3rd ed., Prentice-Hall, Inc., 2006, pp.269–273.

[2] https://yangcha.github.io/iview/iview.html

[3] R. C. Gonzalez, Digital Image Processing, 3rd ed., Prentice-Hall, Inc., 2006, pp.269–284.

[4] R. C. Gonzalez, Digital Image Processing, 3rd ed., Prentice-Hall, Inc., 2006, pp.606.

[5] R. C. Gonzalez, Digital Image Processing, 3rd ed., Prentice-Hall, Inc., 2006, pp.604-605.

[6] https://www.youtube.com/watch?v=eJLF9HeZA8I&ab_channel=SteveBrunton