CENG 466 Fundamentals of Image Processing HW2

Andaç Berkay Seval
2235521

Asrın Doğrusöz
2380301

***Abstract*—This report is prepared for the second homework of CENG 466.**

I. Introduction

This report includes methodology and rationale behind the algorithms that are implemented in the homework, difficulties and errors encountered in the design, implementation, experimentation stages and their solutions, analysis and comments on the results and the requirements of the code.

II. Requirements

This homework is done with the help of some libraires. Therefore, there are 4 requirements for the code:

- scikit-image (skimage)
- opencv (cv2)
- numpy
- scipy

The details and the purposes of these libraries will be explained in the next section.

III. Implementation

A. Fourier, Hadamard and Cosine Transformation

Fourier transformation function is implemented with the help of opencv and scipy libraries. Firstly, for working in each channel (RGB) separately, we used "cv2.split" function to seperate R, G and B dimensions of the image. Then, we used "scipy.fftpack" which is the fast fourier transform pack of the scipy library for each channel. With "fft2" function, we take the fourier transform of the 2D image. Furthermore, with the "abs" function, we take the magnitude of the fourier transform and discard the phase. With the "fftshift" function, we centered the magnitude of the fourier transform. Thus, in order to show the energy compaction precisely, we just add three channels' values R,G and B rather than merge them to obtain a RGB image. Since the idea is to display the total magnitude value, rather than obtaining a colorful image.

Discrete cosine transformation function is implemented with the help of opencv and scipy libraries. Firstly, for working in each channel (RGB) separately, we used "cv2.split" function to seperate R, G and B dimensions of the image. Then, we used "scipy.fftpack" which is the fast fourier transform pack of the scipy library for each channel. With "dctn" function, we take the discrete cosine transform of the 2D image. Therefore, we add R, G and B values to get the total cosine transform values of the image.

Hadamard transformation function is implemented with the help of opencv, numpy and scipy libraries. In order to do hadamard transformation for an image, image dimensions should be $(2^j, 2^j)$. Thus, we decided to do padding with black values for the images. Image 1 has a dimension of (814, 1600, 3). Firstly, we used "cv2.copyMakeBorder" function to do padding with black values of the image to turn the dimensions (2048, 2048) which is the closest 2^j number to 1600. Secondly, for working in each channel (RGB) separately, we used "cv2.split" function to seperate R, G and B dimensions of the image. Then, we used "hadamard" function of "scipy.linalg" library to create a (2048, 2048) hadamard matrix. Finally, for each channel, we do matrix multiplication H.f.H^T with the "numpy.matmul" function. Also, transpose of the hadamard matrix is taken by "transpose" function of the "numpy" library. Therefore, we add new R, G and B values to get the total hadamard transform values of the image. For the second image, implementation is the same. It has a dimension of (3168, 4752, 3). Thus, padding should be done until 8192, which is the closest 2^j number to 4752. However, the runtime of the function is very long, due to the (8192, 8192) hadamard matrix creation, taking its transpose and the matrix multiplications with the padded image H.f.H^T. Thus, it is not recommended to run this for image 2 together with the other functions. Because of this long runtime, a new approach is implemented too. In this new approach, instead of padding with black values, image size is changed. For example for image 2, its size is changed to (4096, 4096). However, its runtime is still long due to the (4096, 4096) hadamard matrix creation, taking its transpose and the matrix multiplications with the image H.f.H^T.

In terms of energy compaction for the 2 images, discrete cosine transformation is the best. We can see just tiny values at the top left corner of the image. Second best is the fourier transformation. The energy compaction is still really good in this one. We need to zoom image to see tiny values in the center of the image. Last one is hadamard transformation. It is still good but not as good as previous ones.

B. Ideal, Gaussian and Butterworth Low Pass Filtering

Ideal low pass filter function is implemented with the help of opencv, numpy and scipy libraries. It takes 2 argument

as image and cut-off frequency. Firstly, for working in each channel (RGB) separately, we used "cv2.split" function to seperate R, G and B dimensions of the image. Then, we used "scipy.fftpack" which is the fast fourier transform pack of the scipy library for each channel. With "fft2" function, we take the fourier transform of the 2D image. With the "fftshift" function, we centered the fourier transform. Then, we create ideal low pass filter as same dimensions of the image 3 (960, 1280) with the for loop. Keep in mind that the origin of the filter is the center of the image. If the pixel distance $|u, v|$ $\leq$ cut-off frequency, the pixel value is 1, otherwise 0. Since the filtering is done in transform domain, we multiply each channel with the filter in element wise, rather than a matrix multiplication. Then, we do inverse shift the multiplicated matrices and inverse fourier transform them with taking real values to obtain spatial domain images. Lastly, we normalize each channel and use "cv2.merge" function to obtain a RGB image with merging channels. Apart from image, filter is also returned in the function. The cut-off frequencies 10, 50 and 100 are applied.

Gaussian low pass filter function is implemented with the help of opencv, numpy and scipy libraries. It takes 2 argument as image and cut-off frequency. Firstly, for working in each channel (RGB) separately, we used "cv2.split" function to seperate R, G and B dimensions of the image. Then, we used "scipy.fftpack" which is the fast fourier transform pack of the scipy library for each channel. With "fft2" function, we take the fourier transform of the 2D image. With the "fftshift" function, we centered the fourier transform. Then, we create gaussian low pass filter as same dimensions of the image 3 (960, 1280) with the for loop. Keep in mind that the origin of the filter is the center of the image. Some implementations are tested and the most appropriate one is chosen for the pixel values which is $e^{\frac{-|u,v|^2}{2(cf)^2}}$ where cf is the cut-off frequency. Since the filtering is done in transform domain, we multiply each channel with the filter in element wise, rather than a matrix multiplication. Then, we do inverse shift the multiplicated matrices and inverse fourier transform them with taking real values to obtain spatial domain images. Lastly, we normalize each channel and use "cv2.merge" function to obtain a RGB image with merging channels. Apart from image, filter is also returned in the function. The cut-off frequencies 10, 50 and 100 are applied.

Butterworth low pass filter function is implemented with the help of opencv, numpy and scipy libraries. It takes 2 argument as image and cut-off frequency. Firstly, for working in each channel (RGB) separately, we used "cv2.split" function to seperate R, G and B dimensions of the image. Then, we used "scipy.fftpack" which is the fast fourier transform pack of the scipy library for each channel. With "fft2" function, we take the fourier transform of the 2D image. With the "fftshift" function, we centered the fourier transform. Then, we create butterworth los pass filter as same dimensions of the image 3 (960, 1280) with the for loop. Keep in mind that the origin of the filter is the center of the image. Some implementations are tested and the most appropriate one is chosen for the pixel values which is $\frac{1}{1+\frac{|u,v|^2}{cf}}$ where cf is the cut-off frequency. Since the filtering is done in transform domain, we multiply each channel with the filter in element wise, rather than a matrix multiplication. Then, we do inverse shift the multiplicated matrices and inverse fourier transform them with taking real values to obtain spatial domain images. Lastly, we normalize each channel and use "cv2.merge" function to obtain a RGB image with merging channels. Apart from image, filter is also returned in the function. The cut-off frequencies 10, 50 and 100 are applied.

For all the low pass filters, images get more clear with the increasing cut-off frequencies. All low pass filters yields a blurred image. In comparison with ideal and gaussian low pass filters, gaussian low pass filter yields a smoother output and ideal low pass filter yields some nonsmooth irregularities with the same cut-off frequency. It is probably because of the discontinuity in the ideal low pass filter. Probably because of the implementation design choice, the outputs that butterworth low pass filter yiels are more blurry and they look more smoother.

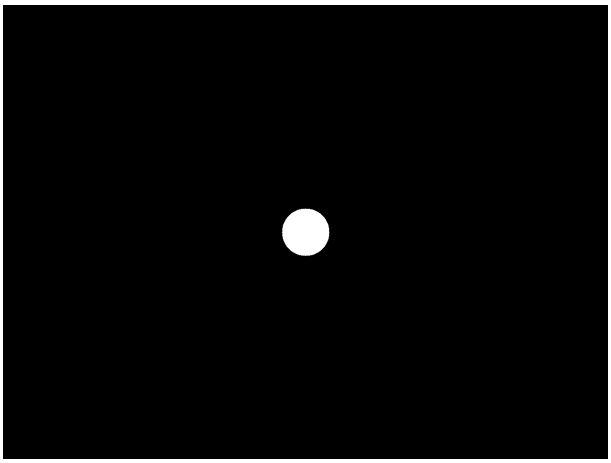

Fig. 1. Ideal Low Pass Filter with cut-off frequency 50

C. Ideal, Gaussian and Butterworth High Pass Filtering

For all the ideal, gaussian and butterworth high pass filter functions, implementations are very similar to the low pass filtering ones. The only difference is that all the pixel values are 1 - lpf where lpf is the corresponding pixel value of the low pass filter counterpart.

All high pass filters works as edge detectors. As the cut-off frequency increases, they detect more edges. In ideal high pass filtering outputs, there are some undesired irregularities probably because of discontinuity of the filter. The outputs that

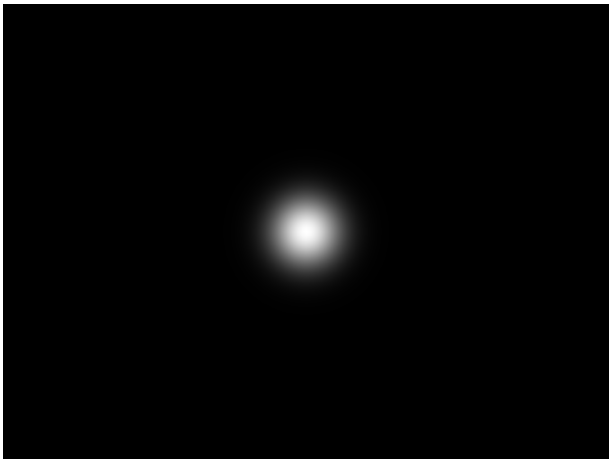

Fig. 2. Gaussian Low Pass Filter with cut-off frequency 50

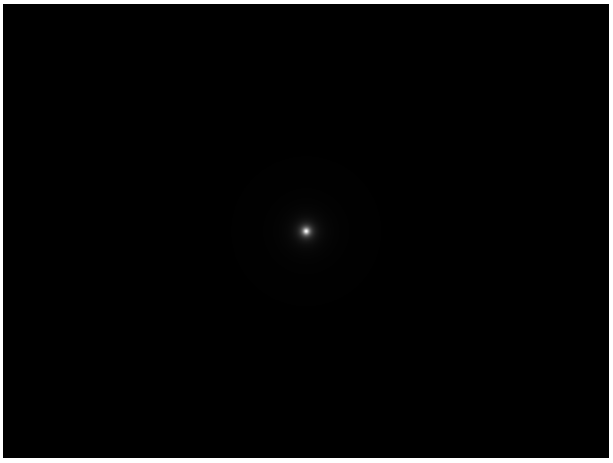

Fig. 3. Butterworth Low Pass Filter with cut-off frequency 50

are yielded by gaussian and butterworth high pass filters are more similar by the side of ideal high pass filtering outputs.

Fig. 4. Ideal High Pass Filter with cut-off frequency 50

Fig. 5. Gaussian High Pass Filter with cut-off frequency 50

Fig. 6. Butterworth High Pass Filter with cut-off frequency 50

D. Band Reject and Band Pass Filtering

Band reject filter function implementation is very similar to ideal low pass filter function. It is implemented with the help of opencv, numpy and scipy libraries. It takes 4 argument as image, cut-off frequency 1, cut-off frequency 2 and image number. Firstly, for working in each channel (RGB) separately, we used "cv2.split" function to seperate R, G and B dimensions of the image. Then, we used "scipy.fftpack" which is the fast fourier transform pack of the scipy library for each channel. With "fft2" function, we take the fourier transform of the 2D image. With the "fftshift" function, we centered the fourier transform. Then, we create band reject filter as same dimensions of the image 4 (2888, 3024) or image 5 (1536, 2048) with the for loop. Keep in mind that the origin of the filter is the center of the image. If the cut-off frequency 1 $< |u, v| <$ cut-off frequency 2, the pixel value is 1, otherwise 0. Since the filtering is done in transform domain, we multiply each channel with the filter in element wise, rather than a matrix multiplication. Then, we do inverse shift the multiplicated matrices and inverse fourier transform them with taking real values to obtain

spatial domain images. Lastly, we normalize each channel and use ”cv2.merge” function to obtain a RGB image with merging channels. Apart from image, filter is also returned in the function. The cut-off frequencies 2 and 50 are applied to image 4 which approximately remove the noise of the image and cut-off frequencies 2 and 85 are applied to image 5 which approximately remove the noise of the image.

Band pass filter function implementation is very similar to band reject filter function. The only difference is the pixel values of band pass filter are 1 - brf where brf is the corresponding pixel value in band reject filter. Since in band reject filter, selected frequency band is filtered out and in band pass filter, selected frequency band is passed to the output, the band pass filter totally disrupt the images 4 and 5 and lose their meaning.

E. Contrast Streching

For improving the contrast of the images 6 and 7, adaptive histogram equalization is applied to them with the function ”equalize_adapthist” from ”skimage.exposure” library. Thanks to that, contrast of the images is clearly improved. For image 6, it has now more realistic colors rather than a shiny photograph. Facial features also become visible in the image. It has more darker and powerful color values which puts forward details. For image 7, since it is originally relatively a dark image, it has now more live colors. Brightness of the image is increased to show details. It looks more realistic now.

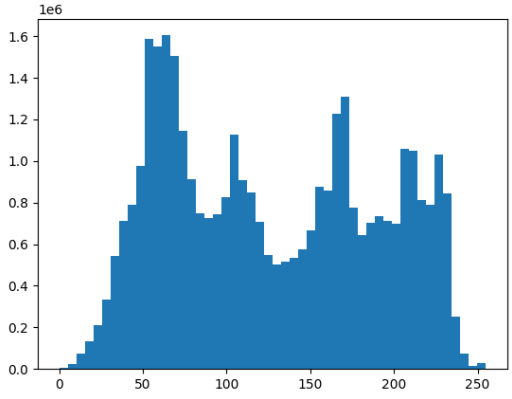

Fig. 7. Original Histogram of Image 6

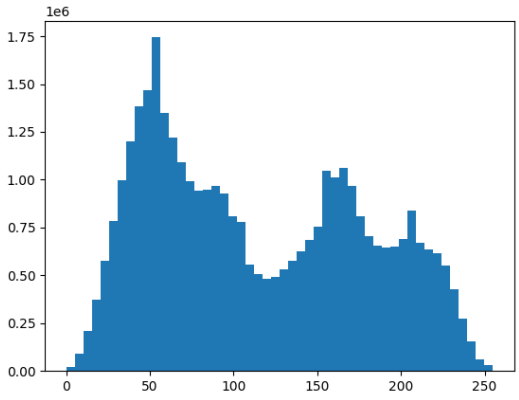

Fig. 8. Adaptive Equalized Histogram of Image 6

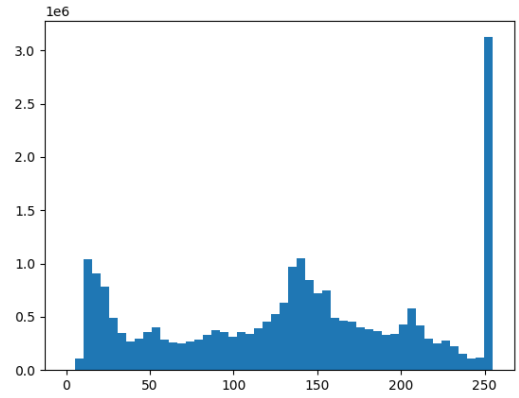

Fig. 9. Original Histogram of Image 7

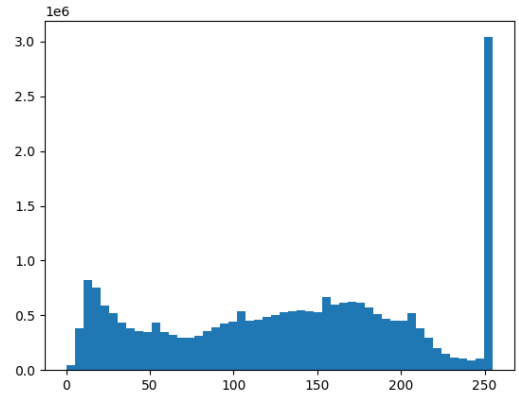

Fig. 10. Adaptive Equalized Histogram of Image 7

REFERENCES

[1] Stéfan van der Walt, Johannes L. Schönberger, Juan Nunez-Iglesias, François Boulogne, Joshua D. Warner, Neil Yager, Emmanuelle Gouillart, Tony Yu, and the scikit-image contributors. scikit-image: Image processing in Python. PeerJ 2:e453 (2014) https://doi.org/10.7717/peerj.453.

[2] J. D. Hunter, ”Matplotlib: A 2D Graphics Environment”, Computing in Science & Engineering, vol. 9, no. 3, pp. 90-95, 2007.

[3] Harris, Charles R. and Millman, K. Jarrod. ”Array programming with NumPy”. Nature. 2020. doi: 10.1038/s41586-020-2649-2

[4] Bradski, G. Dr. Dobb’s Journal of Software Tools. ”The OpenCV Library”. 2000.

[5] Virtanen, Pauli and Gommers, Ralf and Oliphant, Travis E. ”SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python” Nature Methods. 2020. https://rdcu.be/b08Wh. doi: 10.1038/s41592-019-0686-2