CENG 466 Fundamentals of Image Processing - THE2

1st Buğra Alparslan
Computer Engineering
Middle East Technical University
Ankara, Turkey
bugra.alparslan@metu.edu.tr

2nd Erce Güder
Computer Engineering
Middle East Technical University
Ankara, Turkey
guder.erce@metu.edu.tr

I. Introduction

In this paper, we are going to present our methods and algorithms towards given problems, discuss our results and comment on them.

II. Edge Detection

A. Method

In space domain we use filters like Prewitt, Sobel, etc. to get the edge map of a given image. We apply convolution operation using these filter on the images we want to get edges of. For example, let $f(x, y)$ be our image in time domain, $h(x, y)$ is the filter we apply, edge map, $g(x, y)$, can be obtained by;

$$g(x, y) = f(x, y) * h(x, y)$$

Here, for all of the pixels of the image, we slide the filter patch on the image, multiply correspondent pixel values of image and filter patch, add them together to get the value of center pixel. We know that convolution operation in time domain corresponds to multiplication operation in Fourier domain. Therefore, if we want to get the edge map of an image in Fourier domain, $G(u, v)$, we can get the Fourier transformation of the image, $F(u, v)$, and the filter, $H(u, v)$, and multiply them element-wise. In mathematical notation;

$$G(u, v) = F(u, v)H(u, v)$$

We implemented above equation to extract the edge map of given images in Fourier domain. Steps we followed are:

- First we parsed the output path given as a string, created directories if they are not present.
- Then we read the given image using OpenCV's function to read images into the memory:

```
cv2.imread(path, flag)
```

- After, we used NumPy's Fourier transform functions to carry our image to frequency domain:

```
np.fft.fft2(img)
```

- For a better visualization we carried the zero-frequency component to the center:

```
np.fft.fftshift(img)
```

- We have implemented ideal, Gaussian and Butterworth high pass filters, and applied them to the images via element-wise multiplication.
- We reversed the shift operation:

```
np.fft.ifftshift(img)
```

- We carried filtered image back to time domain using inverse Fourier transformation.

```
np.fft.ifft2(img)
```

B. Results

We implemented 3 types of high pass filter: Ideal, Gaussian and Butterworth. Below we present results of all three filters:

Fig. 1. 1.png under Ideal high pass filter

Fig. 2. 1.png under Gaussian high pass filter

Fig. 3. 1.png under Butterworth high pass filter

In ideal high pass filtered image, we see irregularities around edges, which is caused by the sharp discontinuity around radius of the filter. This effect is minimized when other 2 filters are used. We can see this effect in figures 4 through 6

Fig. 4. A portion of 1.png under Ideal high pass filter

Because of these irregularities, we did not select Ideal high pass filter. Gaussian and Butterworth high pass filters eliminate these irregularities as they present a smooth transition from 1 to 0. For this reason they are better candidates for extracting

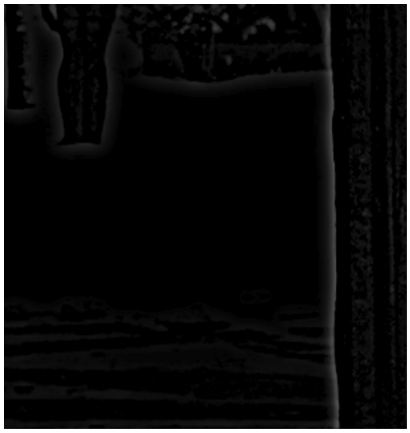

Fig. 5. Same portion of 1.png under Gaussian high pass filter

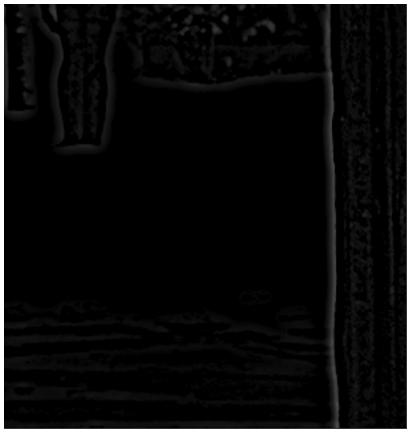

Fig. 6. Same portion of 1.png under Buttterworth high pass filter

edges. Since Butterworth high pass filters has a more flax maximum than Gaussian, outside of the edges has more smooth appearance that that of Gaussian. These factors affected our choice on Butterworth high pass filter.

While determining the type and parameters of the filter, we did many tests, we tried Ideal, Gaussian and Butterwoth high pass filters with different orders and cutoff frequencies. We were able to easily choose the best filter as we had many samples.

As we discussed earlier, due to the sharp discontinuity around radius of the filter, Ideal high pass filter presented an edge map with irregularities around edges. This is an undesired behaviour as in the edge map we have to see the edges only. These irregularities disappear as we increase the radius, however this time we lose the edge information. Due to their smooth transition from 1 to 0, Gaussian and Butterworth filters do not display these irregularities.

When we compare this method with convolution in time domain, we can say that this is a more efficient method when time is considered. Convolution in time domain took nearly minutes to process an image whereas in frequency domain we can extract the edges in seconds. Another advantage is the freedom to choose the edges we want in the edge map and

eliminate the edges we don't want in the edge map. This is achieved by only adjusting the radius of the filter: bigger radius did not allow lower frequency components, so only finer edges are shown in the edge map. This allows us to be more precise when designing an edge extractor.

When we compare the result with the one we achieved at THE1 we see that in THE1 using convolution, extracted edge map is not as good as we got from THE2, using frequency domain techniques. The one extracted with convolution detected small non-edge parts as edges, whereas using high pass filters, we get more successful results in terms of getting only the edges in the edge map.

III. Noise Reduction

A. Method

In order to enhance the image in frequency domain, we went through following steps.

- First we parsed the output path given as a string, created directories if they are not present.
- Then we read the given image using OpenCV's function to read images into the memory:

```
cv2.imread(path, flag)
```

- After, we used NumPy's Fourier transform functions to carry our image to frequency domain:

```
np.fft.fft2(img)
```

- For a better visualization we carried the zero-frequency component to the center:

```
np.fft.fftshift(img)
```

- We calculated magnitude spectrum of the image to see if there is a noise pattern.

```
1+np.log(np.abs(img))
```

- We have implemented low pass filters, and applied them to the images via element-wise multiplication.
- We reversed the shift operation:

```
np.fft.ifftshift(img)
```

- We carried filtered image back to time domain using inverse Fourier transformation.

```
np.fft.ifft2(img)
```

B. Results

We started by separating the images *3.png* and *4.png* into their RGB channels. By using NumPy's Fourier transformation functions described in section A, we obtained transform domain images of all 3 channels for both of the images. Then we calculated the magnitude spectrum of all channels for both of the images to get an idea about the noise. Results are displayed in figures 7 through 12.

Looking at the magnitude spectrum of *3.png*, we see a star shaped pattern at the center in all channels. Our though was that this shape was causing the noise in the images. That's why we tried to remove that pattern from the images. In order to achieve that, we applied time domain median filtering. This actually removed the star shape as well as other useful information, actually most of the information of the image because most of the information resides in the low frequencies. As a result, this did not improve the images.

Next we tried low pass filters to remove the noise effect. For the first question we had already implemented high pass

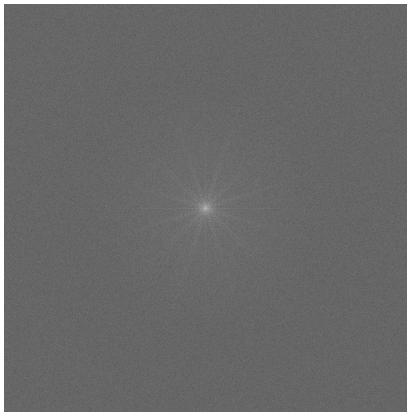

Fig. 7. Magnitude spectrum of R channel of 3.png

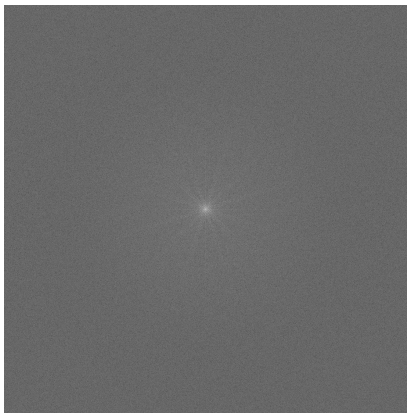

Fig. 8. Magnitude spectrum of G channel of 3.png

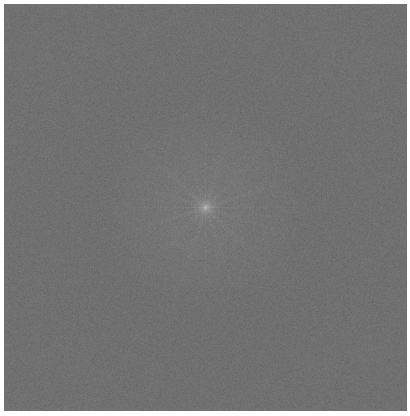

Fig. 9. Magnitude spectrum of B channel of 3.png

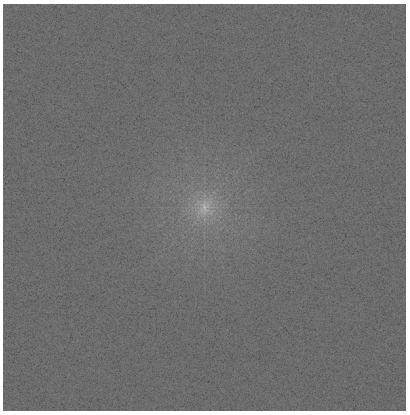

Fig. 10. Magnitude spectrum of R channel of 4.png

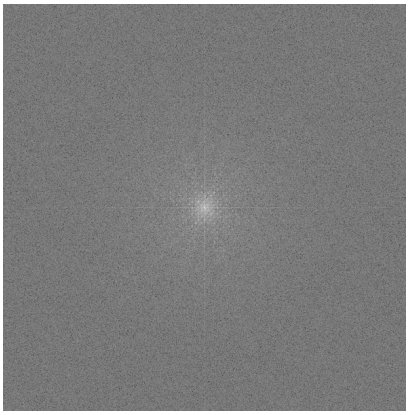

Fig. 11. Magnitude spectrum of G channel of 4.png

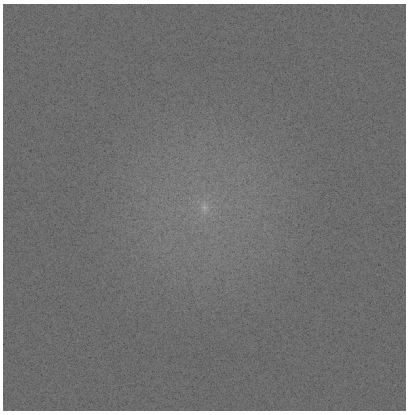

Fig. 12. Magnitude spectrum of B channel of 4.png

filters of three types: Ideal high pass filter, Gaussian high pass filter and Butterworth high pass filter. Let $H_{HP}(u, v)$ denote a high pass filter, we can get the low pass filter of that type by applying following equation:

$$H_{LP}(u, v) = 1 - H_{HP}(u, v)$$

In our implementation we used above formula, by this way we did not have to implement a separate filter for smoothing images.

In order to smooth our images, we used Ideal, Gaussian and Butterworth low pass filters. We applied these filters for R, G and B channels separately, so we tried these filters with different radius lengths for different channels.

Upon our experiments, we concluded that ideal low pass filters were poorly enhancing the images. Figures 13 and 14 display examples ideal low pass filter.

Fig. 13. 3.png under ideal low pass filter

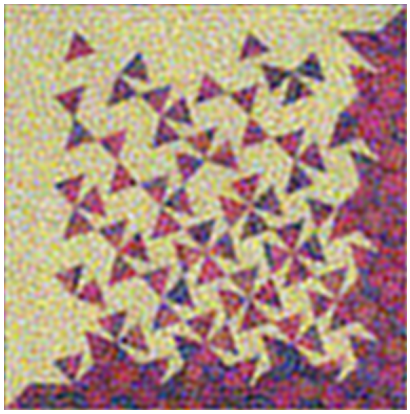

Fig. 14. 4.png under ideal low pass filter

Ideal low pass filters are not only insufficient enhancing the images, they are also introducing a new kind of noise caused by sharp transition from 1 to 0 in the filter.

After deciding not to use ideal low pass filters, we were left with Gaussian and Butterworth low pass filters. We did several experiments, and we agreed on using Butterworth filter with radius 100 for R and B channels, radius 50 for G channel, and order of 2 for all of the channels in *3.png*. For *4.png* we used Gaussian filter with radius of 40 for all of the channels. Again, this result is found experimentally. Figures 15 and 16 display enhanced images.

Using low pass filters for image enhancement is very time effective when compared with time domain smoothing filters. Filters can be applied and results are achieved very fast. Time domain smoothing takes minutes to enhance an image but

frequency domain low pass filters give the result in seconds, so this saves huge time constructing images.

This gives another advantage: Since filters produce results really fast, someone who lacks the domain information about image can just try different radius (and order) values for the filter, and choose the one that best improves the image. It's even better for someone one has the domain information about the image, he/she can precisely decide on which frequencies to keep and which frequencies to eliminate.

When we compare this approach to the one in THE1, we see that smoothing in time domain depends of both the filter size and the entries of the filter whereas in frequency domain, it is enough to decide on radius of the filter, i.e., frequencies we pass and reject. So constructing filters in frequency domain is more straightforward.

C. Critic of our own work

In our work, we could not completely eliminate the noise, rather we could only enhance the images by simply cutting out the high frequency components, which we thought contain **the noise together with the actual signal**. Transform domain techniques certainly allow us to do more precise enhancement than the one we performed. To put it in technical terms, one can only filter out the coefficients which s/he believes that belong to noise and keep the signal corresponding to the image. This can be achieved with more detailed filters and of course, trial-error process which takes significant time, the one we lack the most.

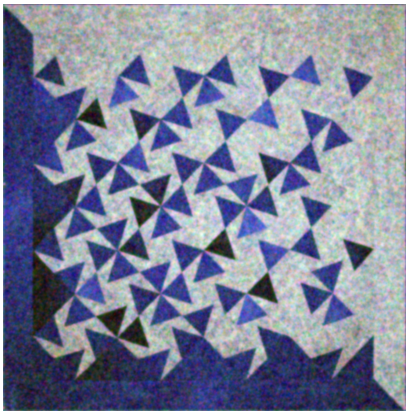

Fig. 15. Enhanced 3.png

IV. JPEG Image Compression

A. Write

In order to compress images, we followed some of the JPEG compression options, including separation to 8×8 blocks, DCT, quantization, and Huffman coding. To ease the implementation, and not to deal with parsing-related problems, we chose `JSON` format for our output file.

- First, we read the image using `cv2.imread()` which returns us the channels of the image in BGR order.

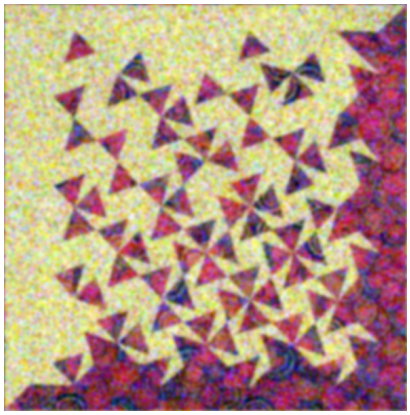

Fig. 16. Enhanced 4.png

- Then we simply convert to the $YCbCr$ color space using `cv2.cvtColor()`. In this color space, the luminance constituent (Y) is neatly separated from the colour constituents (Cb, Cr). The reason behind choosing that particular color space is that it is more coherent with our human visual system. Why? Because, us, humans have much more rods than we have cones in our eyes, resulting in our perception being dominated by the luminance/shapes rather than colours. Hence, we can tolerate much of the information loss that is **purely** on the colour. To put it another way, we can, indeed should, easily throw out some of the information on the colour channels Cb and Cr.

- Following, we split the image into 8×8 blocks and apply type-II Discrete Cosine Transform on each patch. The motivation to apply DCT is that the transformed image consists mostly of (almost) zero components that we can easily compress.

  `scipy.fftpack.dct()`

- Upon transforming 8×8 patches, we perform element-wise integer division by some quantization table(s)[1](which is also recorded at the output file). This is a **lossy** operation, but the loss is tolerable **even for the Y channel**. The motivation is to create further zero entries on the patch for the next step.

- Then, we simply follow the entries of the patch not row by row but in a zigzag manner starting from the top-left ending in bottom-right. The motive is that by following a zigzag path, the aforementioned zero entries get consequently placed.

- Upon receiving the elements in the zigzag order, we simply replace them with their computed Huffman codes. These Huffman codes are computed and saved per-channel rather than per-patch to avoid further load

[1]https://www.sciencedirect.com/topics/engineering/quantization-table

on the output file.

- *(This step is later omitted)*[2] After that, we simply do run-length-encoding. Which mostly works by compressing the consequently placed same number.

- The resulting encoded patch is stored for the corresponding patch, and channel in the output file.

B. Read

In order to read and show our compressed image, we take the reverse of the steps taken when compressing (writing). Here they go:

- The input image (indeed a `JSON` file) is read by using `json.load()`

- The shape of the image and the quantization tables used when compressing are also read.

- Then, for each patch, the huffman encoded string is decoded a list of length 64.

- After decoding, the list elements are simply placed to a new patch in the order they were put in - zigzag order.

- Following that, the patch is element-wise multiplied, according to the channel it belongs, with the quantization table extracted before.

- The result of the multiplication is fed to the IDCT transform and put the the corresponding patch. Here we used

  `scipy.fftpack.idct()`

- After applying the above steps for all the patches, the resulting image is in $YCbCr$ colour space. Thus, again, using `cv2.cvtColor()` we revert it back to the RGB space.

- Finally, using `matplotlib.pyplot.imshow()`, we show the re-constructed image to the user.

C. Critic of our own work

Ultimately, the resulting `JSON` file is a text file. That means it stores each character in a single byte. This deteriorates our compression ratio because for instance the Huffman coded string `11000101` takes 8 bytes in this case, where it could only take one if it was stored as pure binary.

This is actually the reason why we omitted the run-length-encoding step. Our RLE function returns a string that contains – delimited `(char, repetition)` pairs. This delimiter was essential for our implementation to be able to parse, but it was also making the Huffman coded string larger in length. Thus, omitting that step actually improved our compression ratio.

As the very last words, the zigzag ordering, after omitting the run-length-encoding, became useless. Because it doesn't matter in which order you do store the characters, as long you store all of them. Thus, one should also omit it to reduce run-time of our algorithm.

[2]As it is requires some delimiter character to separate (char, repetition) pairs it took more space.

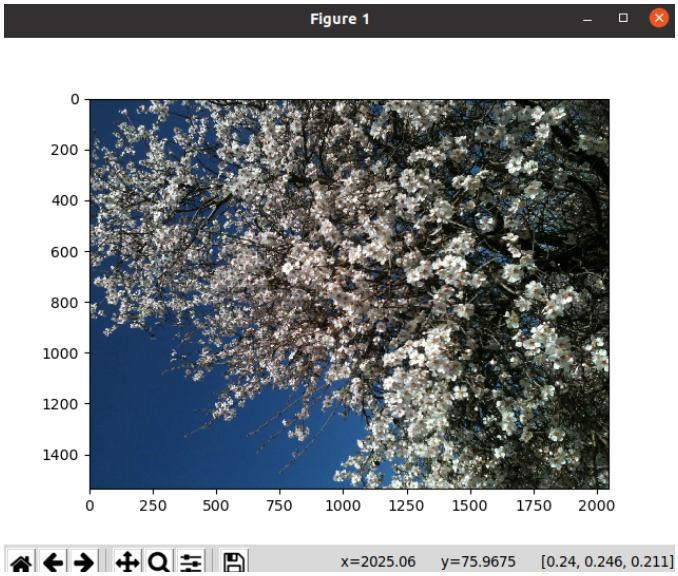

Fig. 17. Resulting image after compression