EE535 Digital Image Processing - Assignment 1

Honi Selahaddin (20226089)

1. Histogram Processing

In the histogram processing task, we expect to improve the image understanding by modifying the intensity distribution of the image. You can see the picture of myself in Figure 1.1 that it is really hard to recognize my face due to bad illumination conditions. Its histogram is the evidence of there is almost any brighter pixel whose intensity value greater than 150.

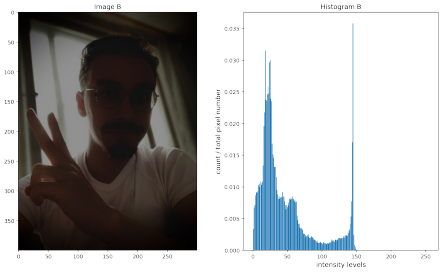


Figure 1.1. Image B and its histogram.

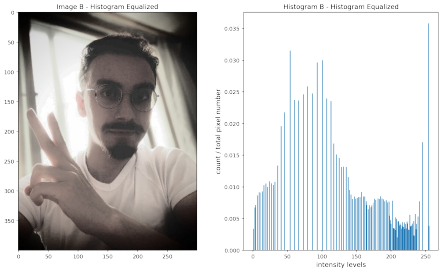


Figure 1.2. Image B and its histogram after histogram equalization is applied.

Histogram Equalization. It is possible to spread the intensity values populated in dark regions to brighter band such that histogram is equalized; in other words, the distribution approximates to uniform. A transform function based on cumulative sum of intensity occurrence probabilities is required for this purpose:

$$s_k = T(r_k) = (L-1)\sum_{j=0}^{k} p_r(r_j) \quad k = 0, 1, ..., L-1 \quad (1)$$

where, r and s denote original image and histogram-equalized image intensities, respectively. L is the number of intensity levels and since I worked on 8-bit images it is picked as 256.

I implemented histogram equalization algorithm in Python using NumPy library for matrix calculations and OpenCV for image I&O, converting&splitting color-channels. My own library, honi_he_lib, contains a HE(L=256) class whose objects initialized with L parameter. After the initialization input image must be set by setInput(img_BGR) method. This method converts given BGR image to YCbCr color-space and stores Y channel for further process. The program runs in three modes (standard histogram-equalization, local histogram-equalization and histogram specification - Modes are changed via numbers 0,1 and 2, respectively) that can be specified in apply(mode,kernel_size,target_hist) method. For the standard case, firstly, original image Y intensity map is flattened to 1d-array and its histogram is calculated; at the same time, location information for each intensity value is collected. Equation 1.1 is directly implemented here to determine the transformation function. Rounding to integer values of the transformation is resulted in new intensity values. Afterward, another loop changes original intensities to adjusted ones by collected coordinate information. Finally, getOutput() method merges new intensity map Y and original Cb, Cr channels and returns processed BGR image.

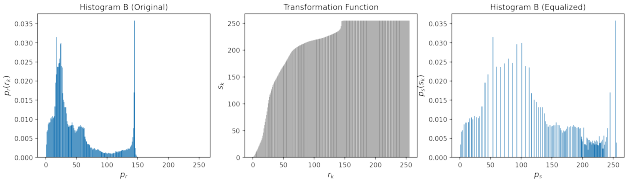


Figure 1.3. Transformation function.

Figure 1.2 shows the output image of histogram equalization if Image-B is input. The histogram is not uniformly distributed; however, it is clearly seen that intensities are spread over the band. Moreover, Figure 1.3 also includes the visualization of the transformation function which maps histogram of Figure 1.1 to histogram of Figure 1.2. Another example, Hagia Sophia, is shown in Figure 1.4.

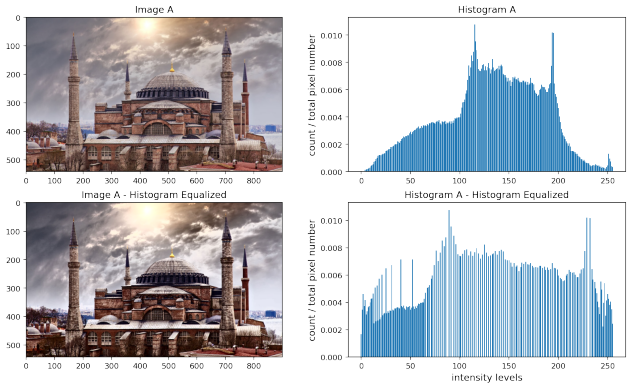


Figure 1.4. Histogram equalization on Image A.

Local Histogram Equalization. This algorithm totally covers the standard equalization. Only difference comes from, it is applied on local windows on the image. In practice, this time HE() object runs in second mode by passing such arguments to apply(mode=2,kernel_size=7) method. Standard histogram-equalization is performed for each 7×7 window and equalized intensities in local windows are updated in the output image, recursively.

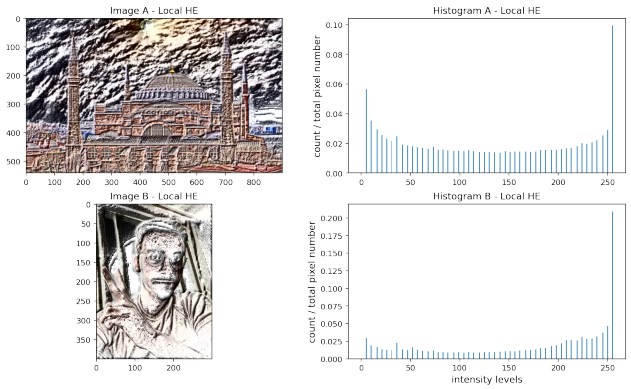


Figure 1.5. Local Histogram Equalization on both Images A and B.

Local-HE works better if there are details in local areas of the image. It is more robust to background compared to standard-HE.

Histogram Specification. We have freedom of not only equalizing the histogram to uniform distribution but also any target histogram. My HE() implementation runs in third mode to realize the histogram specification if target histogram is specified in apply(mode=2,target_hist=any). Again, it is similar to standard procedure; but, an inverse operation is included. Intensity values s in the histogram-equalized image are matched to the closest arg of z correspond where z is defined as:

$$z = G^{-1}(s) = G^{-1}[T(r)]. \tag{2}$$

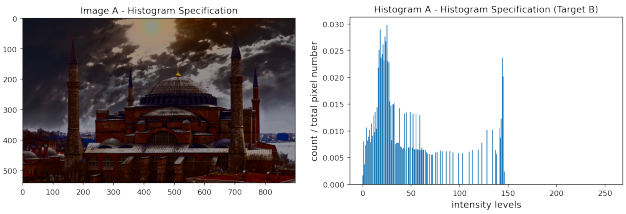


Figure 1.6. Histogram specification.

Figure 1.6 shows an example of histogram specification (matching) on Image-A. Histogram of original Image-B is chosen as target, see Figure 1.1. Since target histogram has lower intensities more frequently, result image also became darker.

Histogram specification needs a bit human-control to reach desirable images.

2. Bilateral Filtering

In the filtering tasks, main goal is smoothing the image while preserving the edges. The most famous smoothing tool is Gaussian Blur, obviously. Simply, it generates a square kernel weighted by Gaussian distribution based on the distance to the center point of the kernel; then, slides this kernel on the image.

Again, I implemented my own filtering library, honi_filter_lib. Main class ImgFilter() has same I&O methods like HE() class. Gaussian smoothing is effortless for both coding and running. applyGauss(sigma) method gets one argument sigma and then calculates the kernel size based on a thumb rule:

$$kernel\_size = 2 \times ceil(3 \times sigma) + 1. \tag{3}$$

Later, Gaussian kernel values are computed in a loop walking through the neighborhood. Obtained kernel is slided by the help of OpenCV's filter2D() function. It is too easy; yet, it does not provide a sufficient performance to preserve the edges, see Figures 2.2-2.3 parts (a) and (b). Because, considering only spatial distance ignores the actual content of the image: edges.

Bilateral Filter adds one more term, K_{range} to the function which also takes into account of intensity similarity:

$$\hat{I}(x) = \frac{1}{C(x)} \sum_{y \in N(x)} K_d(||y - x||) K_r(|I(y) - I(x)|) I(y) \tag{4}$$

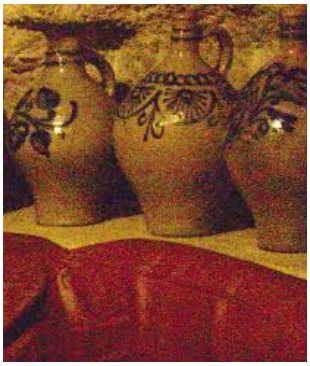
where x is the center pixel location, N(x) is neighborhood around x, y is the neighbor pixel in the neighborhood, I(x) and I(y) are the intensities belongs to center and neighbor pixels, respectively.

In my implementation, class of ImgFilter()'s applyBF(sigma_space,sigma_range) method is used to run the

algorithm. Similarly, it determines the kernel size (neighborhood) by Equation 3 and calculates the kernel for each pixel location with a simple loop-inside-loop. Since, kernel is updated every iteration I could not use a built-in sliding (?) window function and this resulted in slow run-time. In the main loop, first it checks current window (neighborhood) is in the image boundaries; if it satisfies, inside-loop starts to evaluate kernel. At the end, updated center pixel intensity stored in the output intensity map.

The method applyBF(...,img_guidance) method gets another argument, img_guidance, None by default. If an image is specified then neighbor pixel intensities I(y) are picked from that guidance image. This approach is called Joint Bilateral Filtering.

Figure 2.1 shows the original input images. Figures 2.2 - 2.3 are the outputs of Gaussian Blur, Bilateral Filter and Joint Bilateral Filter. Each process is repeated for the spatial variation sigma equals to 1 and 3. Bilateral and Joint Bilateral Filters have another parameter range variation sigma. This parameter is chosen as 80 after an experimental study. Of course, there must be better configurations; whereas, it is not that hard to sense the differences. For both images, Gaussian Blur disrupts the image content by eliminating the edges, especially when spatial sigma getting larger. Bilateral Filter outputs a balanced result, see part (d) in Figures 2.2 - 2.3, it protects the boundaries while performing smoothing. Finally, flashed corresponds to input images are used in JBF to obtain clean image with no-flash atmosphere.



(a) Image 1

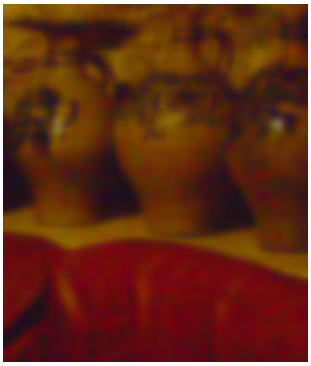
(b) Image 2

Figure 2.1. Original images.

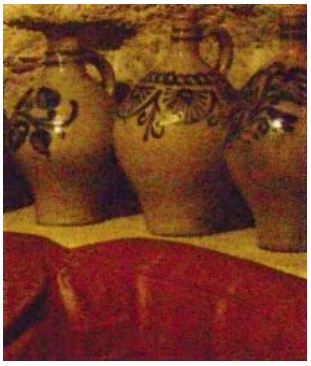
3. Rolling Guidance Filtering

This type of filters are used to remove the small object/details in the image and it adopts an iterative approach by using Joint Bilateral Filtering. Algorithm consists of two phases: first, it applies Gaussian Blur to obtain first guidance image; second, fixed input image and last guidance image is used in a JBF process to generate next guidance image and it repeats itself until iteration limit is exceeded.

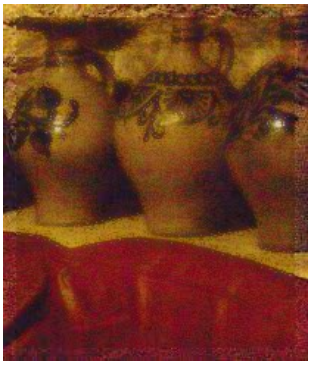
(a) Gaussian ($s = 1$)



(b) Gaussian ($s = 3$)



(c) BF ($s = 3)(s_r = 80$)



(d) BF ($s = 3)(s_r = 80$)

(e) JBF ($s = 3)(s_r = 80$)

(f) JBF ($s = 3)(s_r = 80$)

Figure 2.2. Applying Gaussian Blur, Bilateral Filter and Joint Bilateral Filter on image 1.

Implementation of this algorithm requires to calling JBF from an outer-loop. Therefore, my function applyRollingGuidance() gets iteration size argument and continuously calls the applyBF(img_guidance) function by replacing the guidance image with the new one for each step.

Figure 3.1 shows a visual example for Rolling Guidance Filtering. Original image has mosaic details and we want to get rid of them, somehow. (b) Step 2 and (c) Step 5 images demonstrate how iterating over the guidance images gain the filtering performance. It is possible to reach much better results by increasing the iteration limit.

(a) Gaussian $(s = 1)$

(b) Gaussian $(s = 3)$

(c) BF $(s = 3)(s_r = 80)$

(d) BF $(s = 3)(s_r = 80)$

(e) JBF $(s = 3)(s_r = 80)$

(f) JBF $(s = 3)(s_r = 80)$

Figure 2.3. Applying Gaussian Blur, Bilateral Filter and Joint Bilateral Filter on image 2.

(a) Original

(b) Step 2

(c) Step 5

Figure 3.1. Rolling Guidance Filtering.