CENG466, Fall 2022, THE 1

1st Fırat Ağış
Department of Computer Engineering
Middle East Technical University
Ankara, Turkey
e2236867@ceng.metu.edu.tr

2nd Robin Koç
Department of Computer Engineering
Middle East Technical University
Ankara, Turkey
e2468718@ceng.metu.edu.tr

***Abstract*—We implemented affine transformation using both bilinear and bicubic interpolation as well as histogram extraction, equalization, and adaptive equalization for the first Take Home Exam of Fall 2022 semester.**

***Index Terms*—image processing, affine transformation, histogram equalization**

I. Affine Transformation

A. Rotation

Because both bilinear and bicubic interpolation require finding the closest pixels in the input image for each pixel in the output image, we created an empty output image and rotated every pixel on it counterclockwise direction, finding the position it would have fall into in the input image. To achieve this for an image with $M \times N$ dimensions rotated α degrees, we used the following formula.

$$x_t(x) = x - \frac{M}{2} \tag{1}$$

$$y_t(y) = y - \frac{N}{2} \tag{2}$$

$$rot_x(x, y) = (x_t(x)\cos\alpha - y_t(y)\sin\alpha) + \frac{M}{2} \tag{3}$$

$$rot_y(x, y) = (x_t(x)\sin\alpha + y_t(y)\cos\alpha) + \frac{N}{2} \tag{4}$$

We computed both bilinear and bicubic interpolation for $(rot_x(x, y), rot_y(x, y))$ coordinate of the input image, then assigned the result to the brightness value of (x, y) in the output image.

B. Bilinear Interpolation

We used the weighted means definition of bilinear interpolation, where (x_1, y_1), (x_1, y_2), (x_2, y_1), and (x_2, y_2) are the 4 closest pixel to a point (x, y), the brightness value at (x, y) would be calculated as,

$$w_{11} = \frac{(x_2 - x)(y_2 - y)}{(x_2 - x_1)(y_2 - y_1)} \tag{5}$$

$$w_{12} = \frac{(x_2 - x)(y - y_1)}{(x_2 - x_1)(y_2 - y_1)} \tag{6}$$

$$w_{21} = \frac{(x - x_1)(y_2 - y)}{(x_2 - x_1)(y_2 - y_1)} \tag{7}$$

$$w_{22} = \frac{(x - x_1)(y - y_1)}{(x_2 - x_1)(y_2 - y_1)} \tag{8}$$

$$f(x, y) = \sum_{i=0}^{1}\sum_{j=0}^{1} w_{ij} f(x_i, y_j) \tag{9}$$

We also added additional checks for cases where (x, y) lays directly between 2 points or directly on a point, such that $x_1 = x_2 \wedge y_1 \neq y_2$, $x_1 \neq x_2 \wedge y_1) = y_2$, or $x_1 = x_2 \wedge y_1 = y_2$. For the first two cases, we removed the argument that is equal to 0 from denominators of w_{ij} and took the weighted mean of only the two points (x, y) lies between. For the third case, we already know the value of

$$f(x, y) = f(x_1, y_1) = f(x_1, y_2) = f(x_2, y_1) = f(x_2, y_2). \tag{10}$$

Because we took bilinear interpolation for every pixel in the output image, our calculations took up to 2 minutes for the larger image "a2.png" but produced a satisfactory result for all cases.

C. Bicubic Interpolation

Bicubic interpolation is defined as

$$p(x, y) = \sum_{i=0}^{3}\sum_{j=0}^{3} a_{ij} x^i y^j, \tag{11}$$

calculated by finding the values of 16 coefficients in $[a_{ij}]$, whose values are informed by the 16 closest pixel. By only using the 4 closest pixel $(0, 0)$, $(0, 1)$, $(1, 0)$, and $(1, 1)$

and derivatives of them in horizontal, vertical, and diagonal directions, we can calculate the values of $[a_{ij}]$ as

$$A = \begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{bmatrix} \tag{12}$$

$$B = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -3 & 3 & -2 & -1 \\ 2 & -2 & 1 & 1 \end{bmatrix} \tag{13}$$

$$F = \begin{bmatrix} f(0,0) & f(0,1) & f_y(0,0) & f_y(0,1) \\ f(1,0) & f(1,1) & f_y(1,0) & f_y(1,1) \\ f_x(0,0) & f_x(0,1) & f_{xy}(0,0) & f_{xy}(0,1) \\ f_x(1,0) & f_x(1,1) & f_{xy}(1,0) & f_{xy}(1,1) \end{bmatrix} \tag{14}$$

$$A = BFB^{-1}. \tag{15}$$

As most of the available libraries in our disposal does not calculate the values of A for each given image, instead assigning them predetermined values (such as opencv), we decided to implement it by ourselves. To achieve this, we used a 3 step process.

- *Precomputation* We started by calculated the values of $f_x(x,y)$, $f_y(x,y)$, and $f_{xy}(x,y)$ for every pixel of the original input image where

$$f_x(x,y) = \frac{f(x+1,y) - f(x-1,y)}{2} \tag{16}$$

$$f_y(x,y) = \frac{f(x,y+1) - f(x,y-1)}{2} \tag{17}$$

$$f_{xy}(x,y) = \frac{f_x(x,y) + f_y(x,y)}{2} \tag{18}$$

  using 0 padding.
- *Interpolation* We calculated the brightness value of every pixel of the output image using Equations **??** and **??**.
- *Normalization* We normalized the brightness values of the output image to $[0, 255]$, then rounded them to nearest integer value.

Because this order of operations requiring multiple matrix multiplications for each pixel in the output image, even with optimizations we implemented, this method took upto 10 minutes for the larger "a2.png" image. But if we disregard execution time, we obtained high quality images with no visible flaws, using mathematically accurate bicubic interpolation.

II. Histogram Equalization

A. Histogram Extraction

For histogram extraction we needed the information of distribution of intensity values. For this we basically created an array that will hold the information of how many pixels image has from each intensity value. Our array had 256 zero elements that we are going to increase with each intensity value that corresponds to that elements index. Our function returned an *numpy* array with information of how many pixels the image has from each intensity value. In the homework text the image given was written as gray scale image but the code

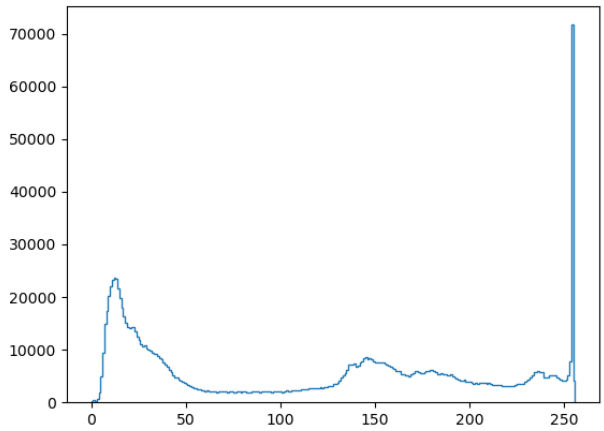

Fig. 1. Original histogram plot of the image b1.png

template did not take the b1.png as gray scale. TA's did change the image with grayscale one but instead we rgb parameter to false to get the corresponding gray scale image.

After having the intensity values and their distribution we basically fed this information to *stairs* function from *matplotlib* library. This function takes two lists and plots a continuous histogram graph.

B. Histogram Equalization

After having the information intensity value distributions, rest was easy. We first computed the cumulative histogram of the image by adding the sum of previous intensity values from the original histogram to each element. Cumulative histogram is different than the original histogram in an important aspect, it gives us a non-decreasing function so we can calculate the equalized intensity values from it. We do this by calculating a coefficient from the intensity range and image size with the bottom formula and multiplying it with corresponding values from the cumulative histogram for each value in the original histogram.

$$s_k = round[\frac{L-1}{N \cdot M} \cdot h_c \cdot (r_k)] \tag{19}$$

Where,

$$h_c \cdot (r_k) = \sum h(r_i) \tag{20}$$

C. Adaptive Histogram Equalization

Adaptive histogram equalization differs from the histogram equalization by the context they are taking account. Histogram equalization takes the whole image and compute the distributions from all pixels but adaptive histogram equalization works more locally, takes some part of the image into account and only affects that part of the image. This gives us more detail piece wise but to make the final image more visually appealing we need to apply more filtering techniques or change

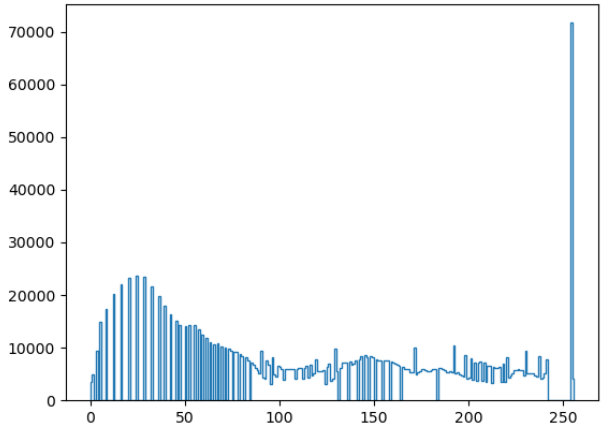

Fig. 2. Equalized histogram plot of the image b1.png

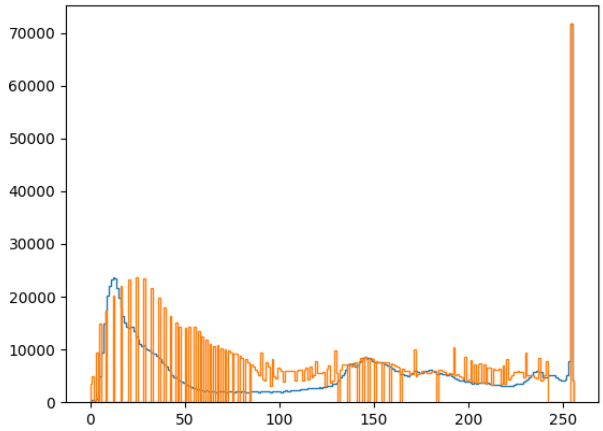

Fig. 3. Comparison of equalized histogram and original histogram on the same plot b1.png

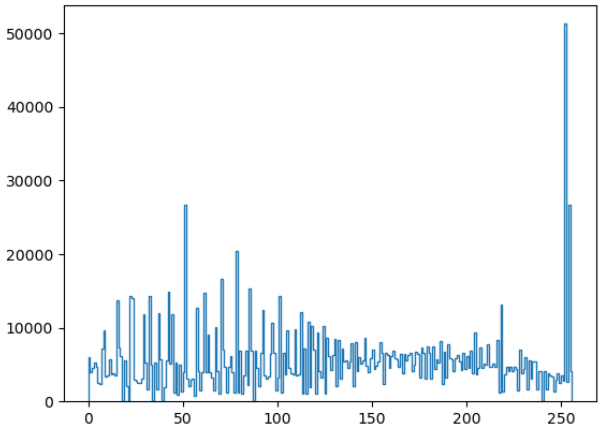

Fig. 4. Adaptively equalized histogram plot of the image b1.png

the equalization technique fundamentally. Our image has visible difference lines between the parts that our equalization technique uses as local areas.

III. Dependencies

We used following libraries for the described reasons.

- *os:* Handling non-existant input or output paths.
- *PIL:* Reading images and converting them arrays.
- *numpy:* Executing array and matrix operations.
- *math:* Performing trigonometric operations.
- *mathplotlib:* Creating histograms as graphics and writing arrays as image files.