

EE 417 COMPUTER VISION LAB#1 POST-LAB REPORT

GOKTUG KORKULU

27026

Brief description of the aim of the lab: In this lab, the images that have degraded quality for some reasons like *sudden scene changes*, *motion blur*, *excessive or inefficient lightning* will be enhanced and made more suitable for further processing.

Initial requirements: All the images that will be processed on should be converted to *grayscale* in the very beginning of all the processing.

1) Point Image Processing

In this step, there will be 3 sub-steps all of which will be implemented to improve noisy images by transforming individual pixel values of the images.

Linear Transformation is one of the point image processing methods which represented by;

$$g(u) = b * (u + a)$$

1.1) Linear Scaling:

Description:

Linear Scaling is an example of a *Linear Point Transformation* in which pixel values of images are linearly scaled between u_{min} and u_{max} , which are the minimum and maximum pixel values of a given image, using the function $g(u)$ where;

$$a = -u_{min}$$

$$b = \frac{G_{max}}{(u_{max} - u_{min})}$$

$$G_{max} = 255$$

Implementation:

“Now write a function which takes an image as input and returns the “linearly scaled version” of it. u_{min} and u_{max} of the returned image should be 0 and 255, respectively. Your function’s name should be “lab1linscale.m”. Plot histograms of the original and the transformed images using Matlab’s built-in “imhist” function.”

```

1 function I_new = lab1linscale(img)
2
3 [row,col,ch] = size(img);
4
5 if(ch == 3) %if the image id rgb
6     img = rgb2gray(img);
7 end
8 img = double(img);
9
10 umin = min(img(:)); %converts matrix into a single row then returns the minimum pixel value. Alternatively can use min(min(img));
11 umax = max(img(:)); %converts matrix into a single row then returns the maximum pixel value. Alternatively can use max(max(img));
12
13 a = -umin;
14 Gmax = 255.0;
15 b = Gmax / (umax - umin);
16 I_new = b * (img-a);
17
18 %Display
19 figure;
20 subplot(2,2,1), imshow(uint8(img)), title('Original Image');
21 subplot(2,2,3), imshow(uint8(I_new)), title('Linear Scaled Image');
22 subplot(2,2,2), imhist(uint8(img)), title('Original Image Histogram');
23 subplot(2,2,4), imhist(uint8(I_new)), title('Linear Scaled Image Histogram');
24 end

```

Handwritten annotations in the image: 1 points to the function output `I_new`; 2 points to the input parameter `img`; 3 points to the `size` function call; 4 points to the `if` statement; 5 points to the `umin` and `umax` calculations; 6 points to the scaling formula; 7 points to the display section.

Figure 1 : Linear Scaling of an arbitrary image with `lab1linscale.m` function.

The function called “`lab1linscale`” above takes an image parameter as `img` (indicated by 2) and returns linearly scaled version of it as `I_new` (indicated by 1).

The part that is indicated by 3 is to gather row, column and channel amount of the input image.

The part that is indicated by 4 is to transform the input image grayscale version if the input image is rgb scale. Note that, since we have no assumption beforehand about the input image, (e.g., whether it is rgb or grayscale) the `ch` value we gathered on part3 which corresponds to the channel amount indicates if it is rgb (`ch = 3`) or grayscale (`ch = 1`). So we iterate if condition according to `ch` value. And also in line 8, the pixel values are changed from integer to double values so that the mathematical operations are not going to cause loss of decimal values.

Part 5 is to obtain the minimum and the maximum pixel value of the entire image and to keep those values on variables called `umin` and `umax` respectively.

In part 6, we apply the formula that is mentioned on previous page in order to obtain new image which is the linearly scaled version of the input image according to `umax` and `umin` values of the input image.

Finally, part 7 is to display all the values and changes that was made in the code. Note that the original version of the input image is not displayed on the final figure. It can be seen below as a separate image.

Result of the Implementation:



Figure 2 : The original version of the input image. 'girl.jpg'

```
clc;  
clear;  
close all;  
  
image = imread('girl.jpg');  
  
%% Linear scaling  
lab1linscale(image);
```

Figure 3 : The code in order to call the lab1linscale.m



Figure 4 : The result of the lab1linscale.m function

Inferences on the outcome are these;

- As it is seen, the linear scaled image is brighter than the original image since the pixel values are scaled linearly towards the value 255 as it can be seen from the difference between the original image histogram and linearly scaled image histogram.

1.2) Conditional Scaling:

Description:

Conditional Scaling is another example of the *Linear Point Transformation* in which an input image is mapped into new image such that new image has the same mean and variance as a reference image.

$$a = \mu_{I_{ref}} * \frac{\sigma_I}{\sigma_{I_{ref}}} - \mu_I$$

$$b = \frac{\sigma_{I_{ref}}}{\sigma_I}$$

Where $\sigma_{I_{ref}}$, σ_I are standard deviations of the reference and input image, respectively; and $\mu_{I_{ref}}$, μ_I are mean values of the reference and input image, respectively.

Implementation:

“Now write a function which takes “two images”, current image I and reference image I_{ref} , as inputs and returns I_{new} , the “conditionally scaled version” of the current image. Map the current image into the returned image such that the resultant image has the same mean and standard deviation as the reference image. The current image is given in Figure 1 and the reference image is shown in Figure 2. Plot current, reference and output images, and compute mean and standard deviation for each. Your function’s name should be “lab1condscale.m”. “

```

1 function I_new = lab1condscale(img, img_ref)
2
3 [row,col,ch] = size(img);
4 if(ch == 3) %if the image is rgb
5     img = rgb2gray(img);
6 end
7 img = double(img);
8
9 [row2,col2,ch2] = size(img_ref);
10 if(ch2 == 3) %if the image is rgb
11     img_ref = rgb2gray(img_ref);
12 end
13 img_ref = double(img_ref);
14
15 meanOfReference = mean2(img_ref);
16 stdOfReference = std2(img_ref);
17 meanOfImage = mean2(img);
18 stdOfImage = std2(img);
19
20 a = meanOfReference * (stdOfImage/stdOfReference) - meanOfImage;
21 b = stdOfReference / stdOfImage;
22
23 I_new = b * (img + a);
24
25 %Displays
26 figure;
27
28 subplot(1,3,1), imshow(uint8(img)), title('Original Image'), xlabel(['Mean: ', num2str(meanOfImage)], ['Std: ', num2str(stdOfImage)]);
29 subplot(1,3,2), imshow(uint8(img_ref)), title('Reference Image'), xlabel(['Mean: ', num2str(meanOfReference)], ['Std: ', num2str(stdOfReference)]);
30 subplot(1,3,3), imshow(uint8(I_new)), title('Output Image'), xlabel(['Mean: ', num2str(mean2(I_new))], ['Std: ', num2str(std2(I_new))]);
31
32 end
  
```

Handwritten annotations in the image:

- 1: Points to the function definition line.
- 2: Points to the input arguments `img` and `img_ref`.
- 3: Points to the conversion of `img` to grayscale and double.
- 4: Points to the conversion of `img_ref` to grayscale and double.
- 5: Points to the calculation of mean and standard deviation for both images.
- 6: Points to the final calculation of `I_new` and the display of the three images.

Figure 5 : Conditional Scaling of an arbitrary image with lab1condscale.m function.

The function is called `lab1condscale` and it takes image to be processed on (`img`, indicated by 2) and a reference image (`img_ref`, indicated by 2) whose standard deviation and mean will be used to map a new image (`I_new`, indicated by 1).

Part 3 is about checking whether both the input and reference images are grayscaled previously or not and transforms them if they're not initially.

Part 4 is to change the pixel values from integer to double values so that the mathematical operations are not going to cause loss of decimal values.

In the part 5, initially the mean and variance of input and reference images are calculated. Then those values are used in the equation stated above. Therefore, at the end new image is mapped.

Part 6 is for displaying the result of the entire function. Note that the original version of the input image is not displayed on the final figure. It can be seen below as a separate image.

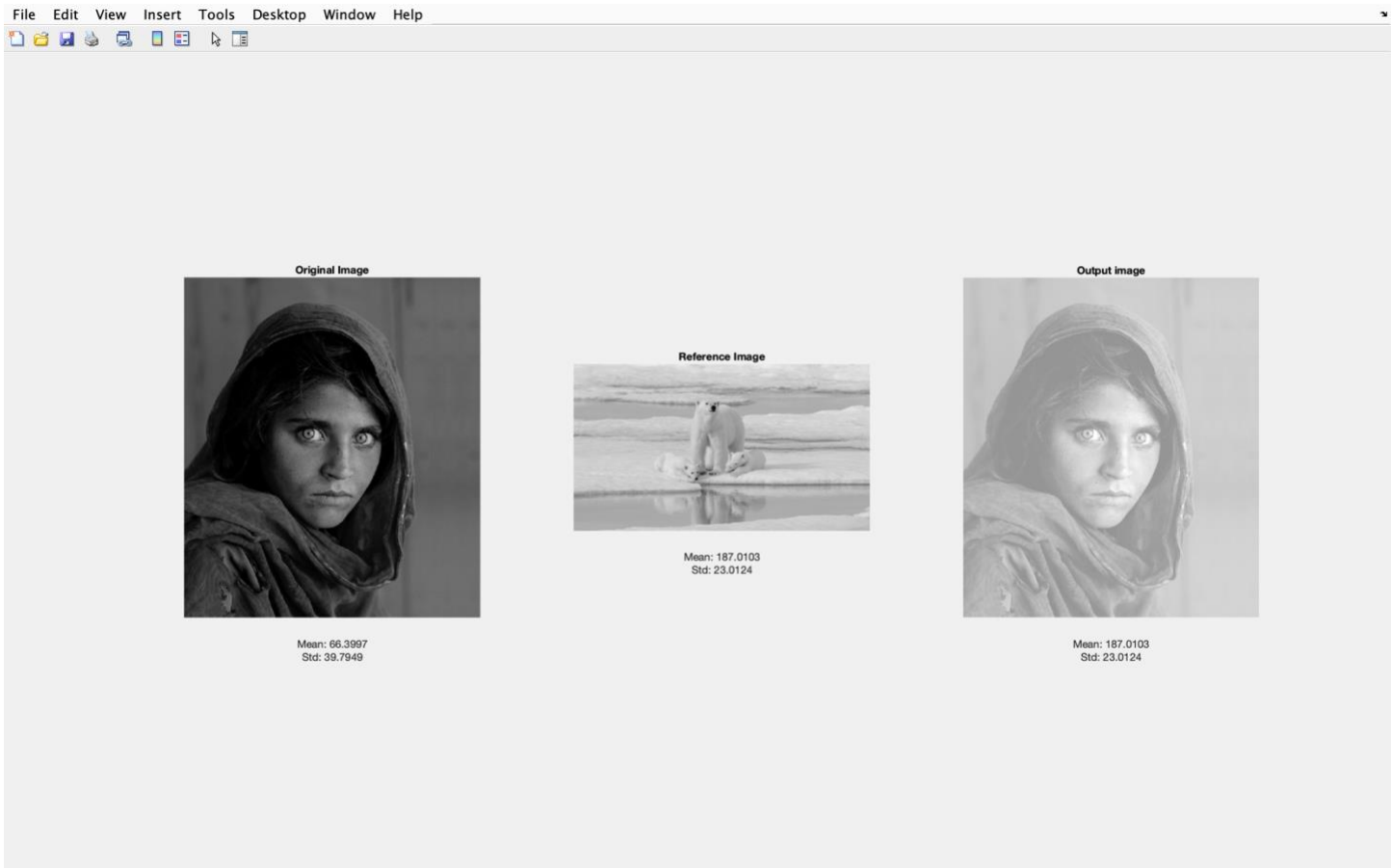
Result of the Implementation:



Figure 6 : the original image



Figure 7 : the reference image



Inferences on the outcome are these;

- Using the grayscaled version of the original image by referencing with grayscale version of the reference image, we obtain the output image. The part that needs to be focused on is the mean and the standard deviation values of reference and output images. As seen, the values are exactly the same since the main purpose was that initially.

1.2) Conditional Scaling:

Description:

Histogram equalization is another example of a point processing operation in which the *contrast of an image* is adjusted using the image's histogram.

Implementation:

“Apply histogram equalization to the original image, given in Figure 1, using MATLAB's built-in “histeq” function, and obtain histograms of both the original and resulting images using “imhist” function.”

```

clc;
clear;
close all;

image = imread('girl.jpg');

%% Histogram Equalization
hisEq_image = histeq(image);

subplot(2,2,1), imshow(image), title('Original Image');
subplot(2,2,3), imshow(hisEq_image), title('Histogram Equalised Image');
subplot(2,2,2), imhist(image), title('Original Image Histogram');
subplot(2,2,4), imhist(hisEq_image), title('Histogram Equalised Histogram');

```

Figure 9 : Histogram equalization in main function.

The embedded function `histeq()` is used in the main function in order to adjust the contrast of the image according to the histogram values (part 1).

Part 2 is to display the difference of histograms of the original image and histogram equalized version of the original image.

Result of the Implementation:

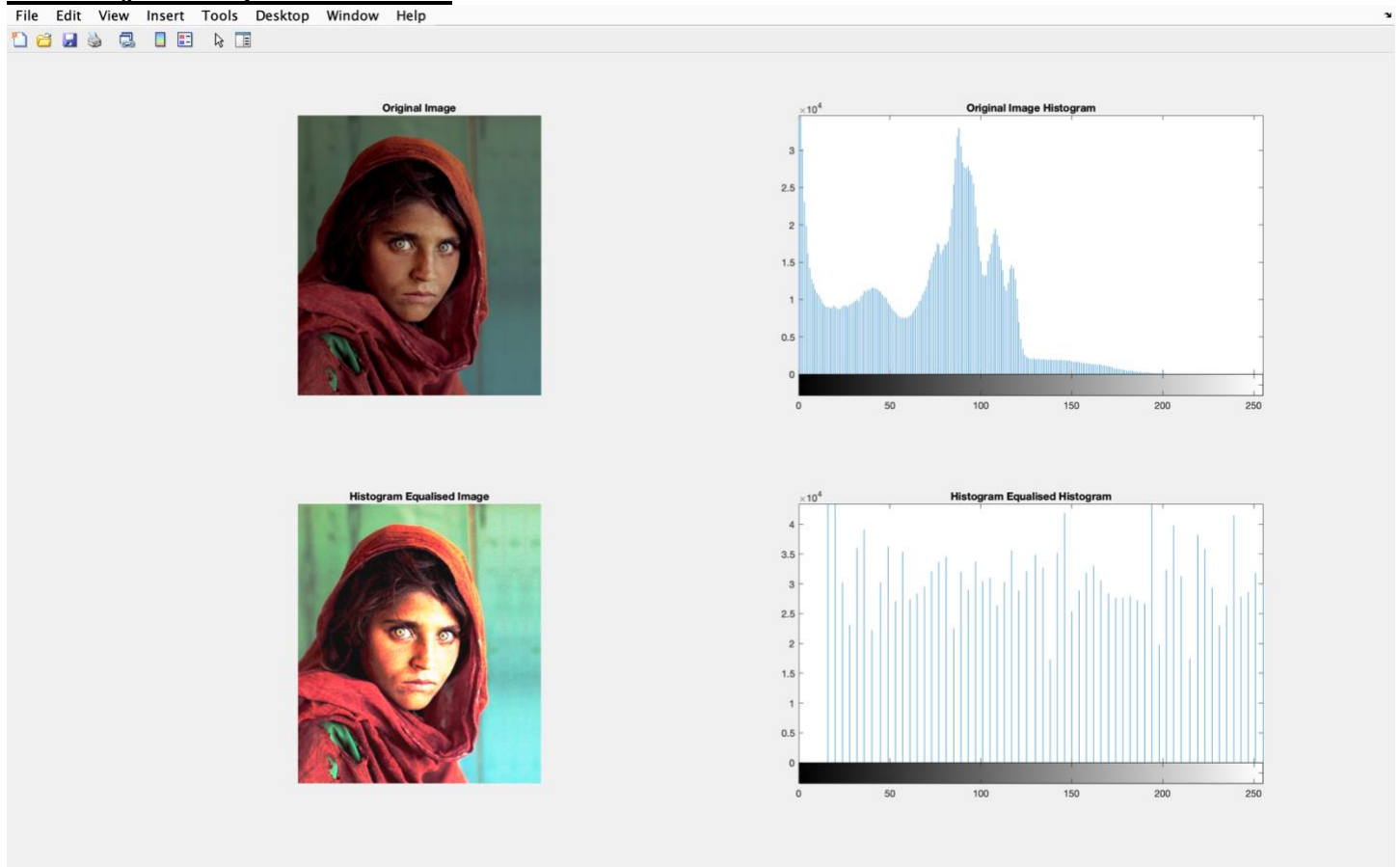


Figure 10 : histograms of the original and `histeq()` function applied images.

Inferences on the outcome are these;

- As described above, Histogram Equalization is a computer image processing technique used to improve contrast in images. It accomplishes this by effectively spreading out the most frequent intensity values, i.e. stretching out the intensity range of the image. And here in outcome of the above codes, it is seen that histogram equalization altered the contrast of the original image. Also, by just looking the histograms of those images, it is seen the intense areas are stretched out in order for equalization.

2) Spatial Filtering

Description:

Spatial Filtering is a method used to modify an image f by replacing the value at each pixel with some function of the values of nearby/neighbor pixels. In this part, Local Mean Filter (Box Filter) which is an example of Spatial Filtering methods will be used in order to eliminate undesirable random variations in intensity values of an image, a.k.a. noise.

Box Filter attenuates noise in the acquired images by convolving it with a sliding window W_p of size $(2k+1) \times (2k+1)$ centered at P . Box filter is realized by replacing each pixel of an image with the average of its neighborhood as follows:

$$\mu_{W_p}(I) = \frac{1}{(2k+1)^2} \sum_{i=-k}^{+k} \sum_{j=-k}^{+k} I(x+i, y+j)$$

$$I_{new}(p) = \mu_{W_p}(I)$$

Implementation:

“Now write a function which takes an image I , given in Figure 3, and a number W (for window size) as inputs and returns I_{new} , the “local mean (box filter) filtered version”. Your function’s name should be “lab1locbox.m”. “

```
1 function I_new = lab1locbox(I, W)
2 [row,col,ch] = size(I);
3
4 if(ch == 3) %if the image id rgb
5     I = rgb2gray(I);
6 end
7
8 I = double(I);
9
10 I_new = zeros(size(I));
11
12 k = (W-1)/2;
13
14 for i = (k+1):(row - k - 1)
15     for j = (k+1):(col - k - 1)
16
17         sub_img = I(i-k:i+k, j-k:j+k); % crops the image by window size centered at I(i,j).
18
19         I_new(i,j) = sum(sum(sub_img)) / ((2*k+1)^2);
20     end
21 end
22
23 figure;
24
25 subplot(1,2,1), imshow(uint8(I)), title('original grayscale image');
26 subplot(1,2,2), imshow(uint8(I_new)), title('filtered image');
27
28
29 end
```

Figure 11 : Box Filtering of an arbitrary image as implemented in lab1locbox.m

In this function, arbitrary image (it is not known whether it is grayscale or not) is taken as input, called I (as shown in 1), and a window size W (as shown in 2) and returns local mean (box filter) filtered version of that image as I_{new} (as shown in 3).

In the part 4, we check if the image is grayscale already or not. If it is, we don't do anything but if it isn't we convert it to the grayscale version.

In part 5, we change the pixel values from integer to double so that when we change the value of it, it will not lose any decimal values.

In part 6, we create new pure black image with same size of input image. Later, we will alter the pure black pixel values to corresponding filtered values of input image.

Part 7 is to get the k value from window size input (W) so that we can use the k value in the below equation. For example, if the window size 7×7 (i.e. $W = 7$), we obtain $k = 3$ and use this value on line 19 while computing the new pixel value of corresponding input pixel value.

Part 8 is the most integral part of this function. Generally, at the end of this bunch of code the new image is altered according to input image's pixel values so that we obtain what we desire.

In part 9, which is a subpart of 8, we create a sub image by extracting from Input image located at (i,j) of given window size. Because of the for 2 for loops nested to each other, we do this process for all the pixel values of input image. Then we will process on this window sized input image.

In part 10, we do the calculation and smoothing operation. Since we iterate the each of every pixel of input image by 2 nested for loops, we set the new value of corresponding output image one by one. We take the sub image that was created 1 line above, and sum all of its pixel values then divide it to the square of the window size. Therefore, we basically take the average of neighbor pixel values of center pixel value and change it accordingly.

In part 11, we show the change between input and output images by displaying them side by side.

Result of the Implementation:

Let set various inputs to our function at main MATLAB file.

Since we desire to get the most optimal output, we have to alter the input values (especially window size) and see the result one by one so that we can decide which window size is most suitable and optimal. It is better to remind ourselves that there is a tradeoff between the fewness of the noises and the blurriness of the image. In other words, when we increase the window size, the noisiness decreases but on the other hand the blurriness of the image increases a lot. This is obvious on above examples. We will investigate the idea by incrementing the input size gradually so that will be easy to realize the difference at each step.

Examination 1: $W = 3 \times 3$. This is the smallest window size that we can use in order to smoothen our image since 1×1 window size is nothing but keeping the pixel value same.

```
%% Local Mean Filter  
image2 = imread("child.png");  
lab1locbox(image2, 3);
```

Figure 12: the code to call the mean filter function

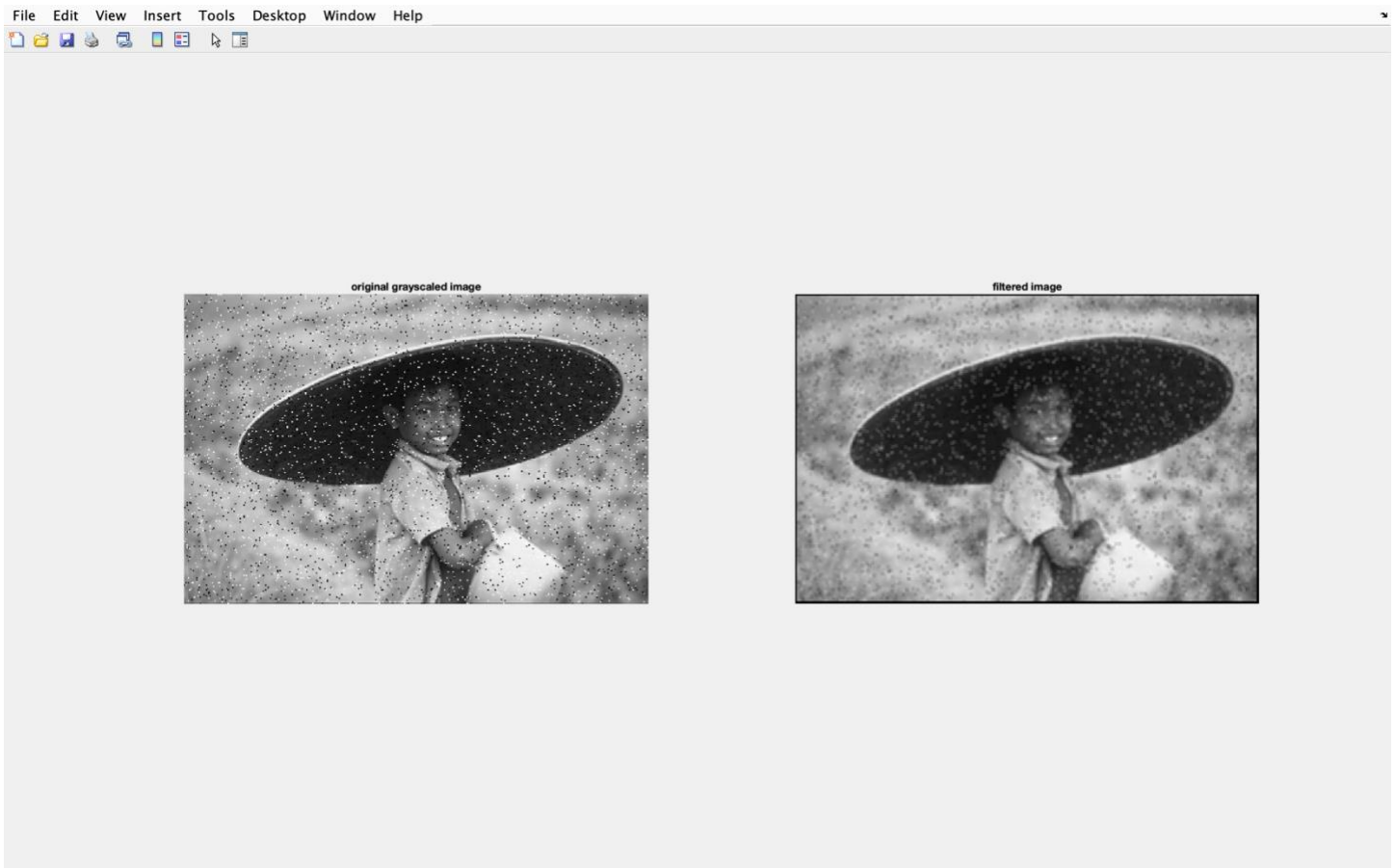


Figure 13: Box filtered image with window size = 3×3

Examination 2: $W = 5 \times 5$.

```
% Local Mean Filter  
image2 = imread("child.png");  
lab1locbox(image2, 5);
```

Figure 14 : the code to call the mean filter function

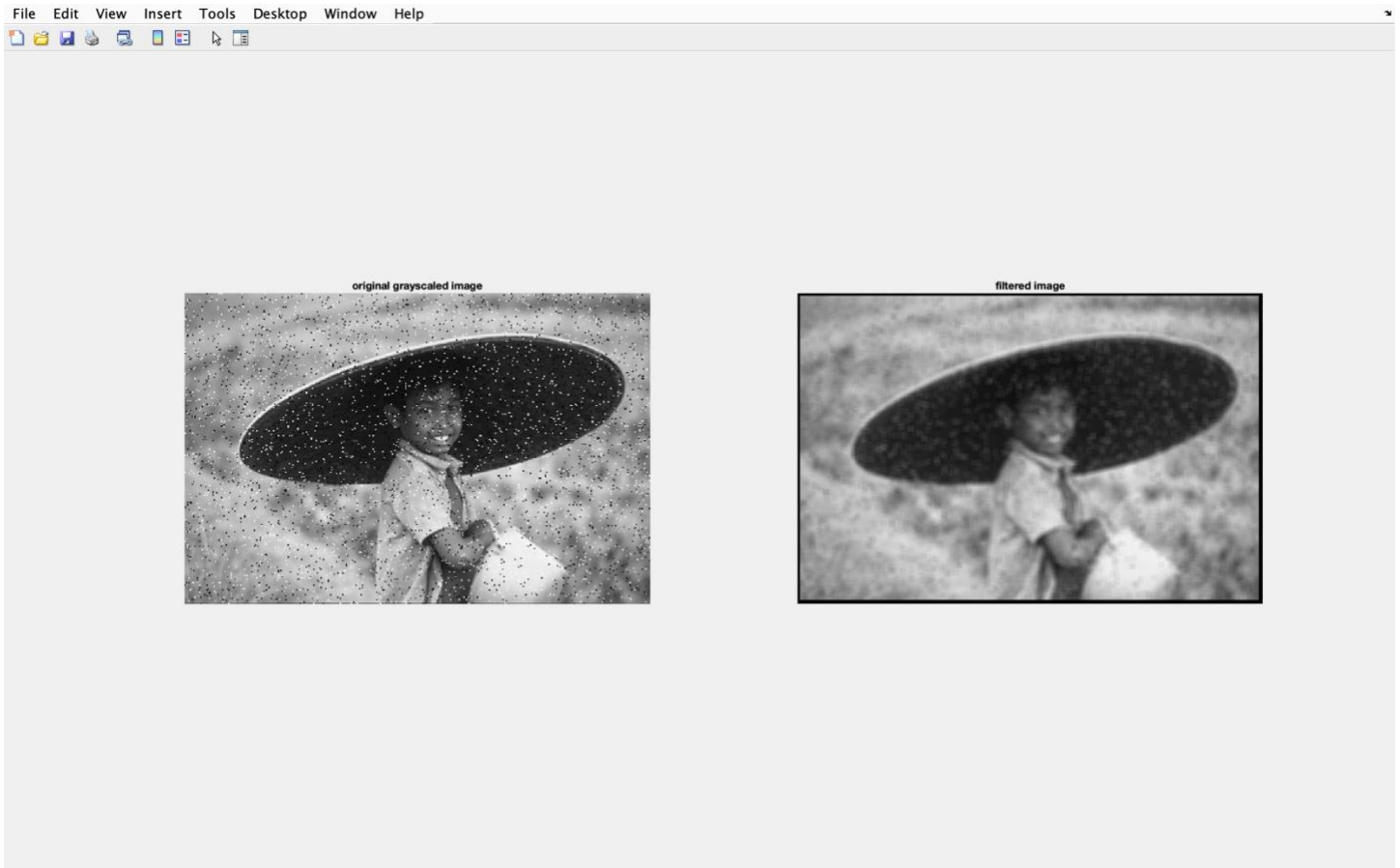


Figure 15 : Box filtered image with window size = 5×5

Examination 3: $W = 7 \times 7$.

%% Local Mean Filter

```
image2 = imread("child.png");  
lab1locbox(image2, 7);
```

Figure 16 : the code to call the mean filter function

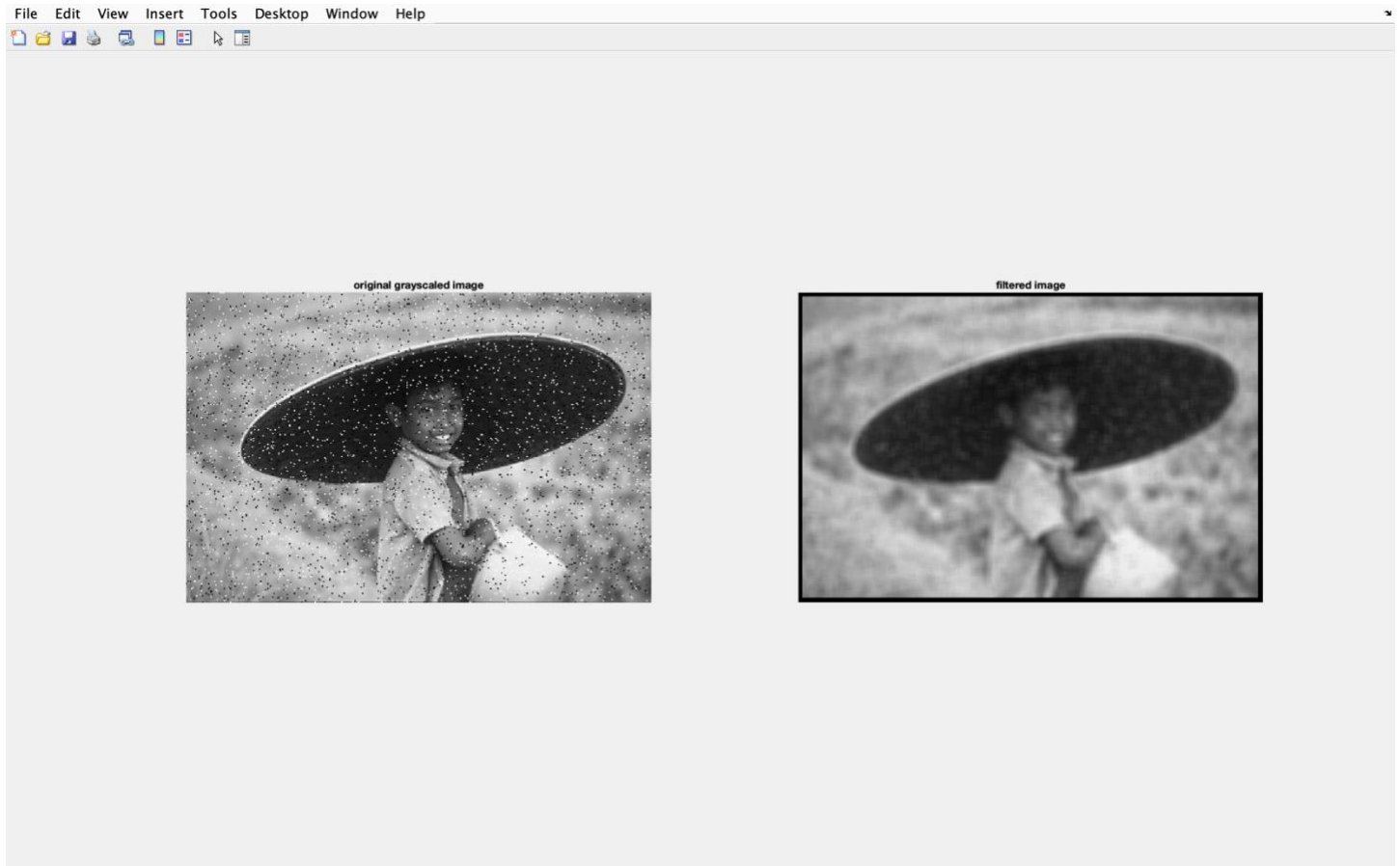


Figure 17 : Box filtered image with window size = 7×7

Examination 3: $W = 9 \times 9$.

```
%% Local Mean Filter  
image2 = imread("child.png");  
lab1locbox(image2, 9);
```

Figure 18 : the code to call the mean filter function

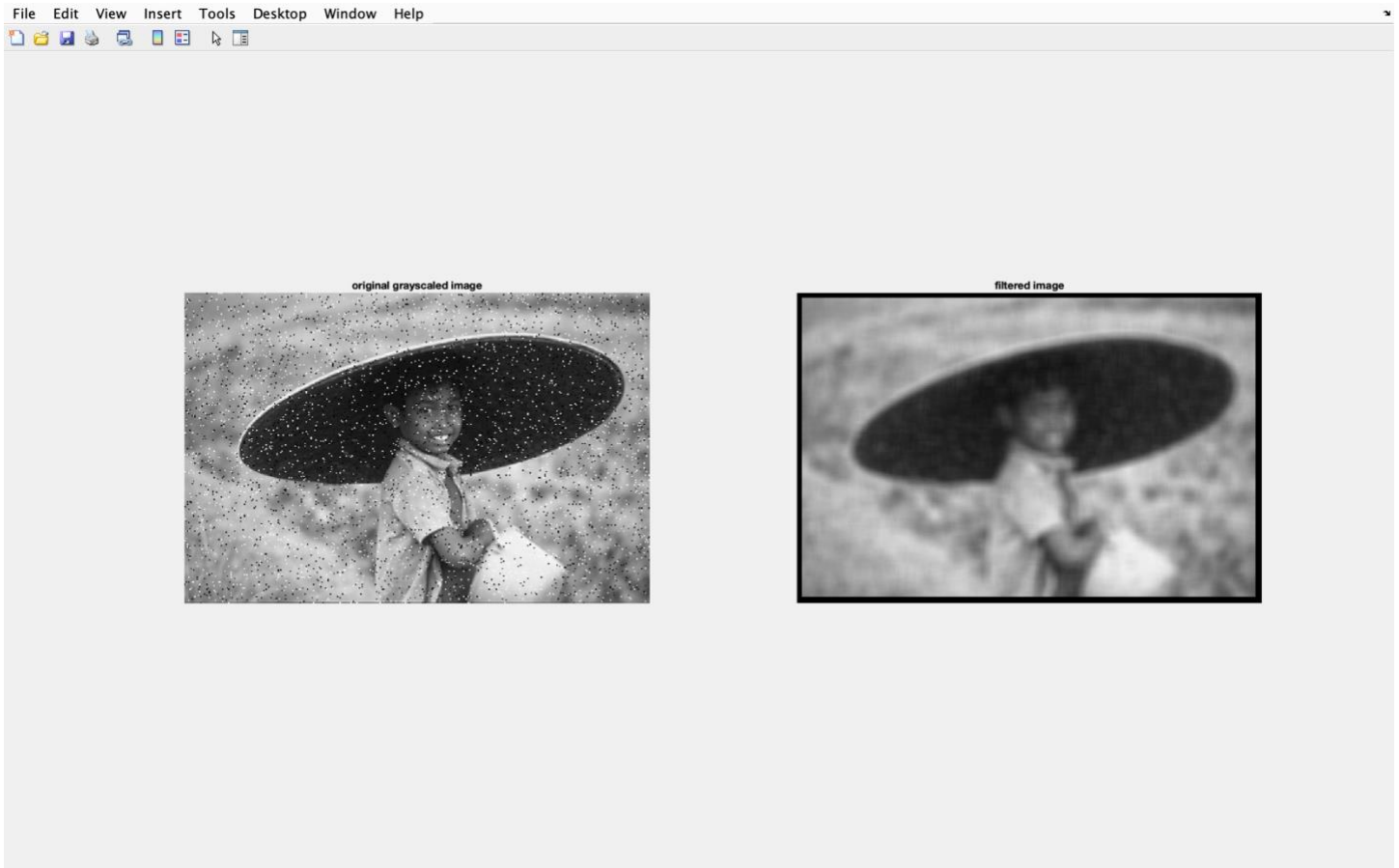


Figure 19 : Box filtered image with window size = 9×9

RESULT: As we increase the window size from 3×3 up to 9×9 , it is seen that our image is diminishing the noises more and more, but the blur of the image increases apparently as well. After this point, it is implementors choice to choose which window size is the best to use in order to smooth and filter the image. Also notice that the side edges of the images get thick while increasing the window size. In my opinion, I would use 5×5 filter since the blur is acceptable and the noise is decreased enough.

2) Sharpening

Description:

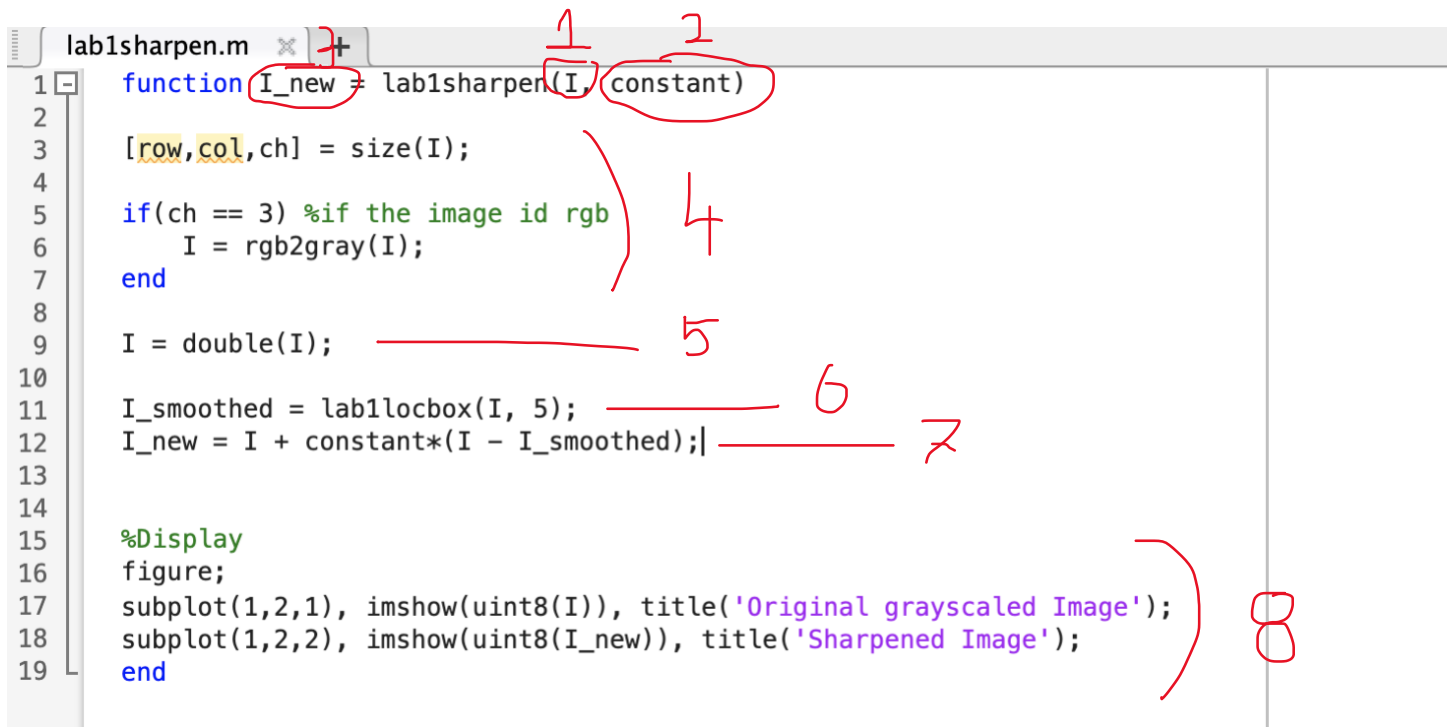
What sharpening does is increasing the contrast of the input image along edges without adding too much noise within homogenous regions. As in the case of box filtering, this process has tradeoff as well. The tradeoff is between the noises occurs at homogenous regions on the image and the amount of sharpening along the edges. The mathematical description of implementation is as follows;

$$I_{new}(p) = I(p) + \lambda[I(p) - S(p)]$$

where S is the smoothed version of the given image I
and $\lambda > 0$ is a scaling factor which controls the influence of the correction signal.

Implementation:

“Now write a function which takes an image I , given in Figure 4, and a constant λ as inputs and returns I_{new} , the “sharpened” version of the image. Use a box filter to smooth the image. Your function name should be “lab1sharpen.m”. “



```
lab1sharpen.m
1 function I_new = lab1sharpen(I, constant)
2
3 [row,col,ch] = size(I);
4
5 if(ch == 3) %if the image id rgb
6     I = rgb2gray(I);
7 end
8
9 I = double(I);
10
11 I_smoothed = lab1locbox(I, 5);
12 I_new = I + constant*(I - I_smoothed);
13
14
15 %Display
16 figure;
17 subplot(1,2,1), imshow(uint8(I)), title('Original grayscaled Image');
18 subplot(1,2,2), imshow(uint8(I_new)), title('Sharpened Image');
19 end
```

Handwritten annotations in red:

- 1: circled around the function name `lab1sharpen`
- 2: circled around the input `constant`
- 3: circled around the output `I_new`
- 4: circled around the `if` statement
- 5: circled around the `I = double(I);` line
- 6: circled around the `I_smoothed = lab1locbox(I, 5);` line
- 7: circled around the `I_new = I + constant*(I - I_smoothed);` line
- 8: circled around the `subplot` lines

Figure 20 : the lab1sharpen.m function that sharpens the image.

This function takes an arbitrary image as I (shown in 1) and scaling factor as ‘constant’ (shown in 2) and returns the sharpened version of the input image as I_{new} (shown in 3).

In the part 4, as always, we check if the image is grayscaled already or not. If it is, we don’t do anything but if it isn’t we convert it to the grayscaled version.

In part 5, we convert the each pixel values from integer to double so that when we change the value of it, it will not lose any decimal values.

Part 6 is to smooth the input image and save the smoothed version as I_smoothed by utilizing the lab1locbox function which was explained detailly in previous chapter.

Part 7 is basically the main part of implementing the given mathematical equation. It creates new image by using the smoothed version of the. Input image and input 'constant'. Notice that depending on the input 'constant', the resulting image differs. At the end, we have output image as I_new.

And finally, the part 8 is to display the outcome of the function by showing the input and the output image next to each other in a figure.

Result of the Implementation:

```
%% Sharpening
image3 = imread('bird.png');
lab1sharpen(image3, 1);
```

Figure 21 : sharpening with scaling factor = 1

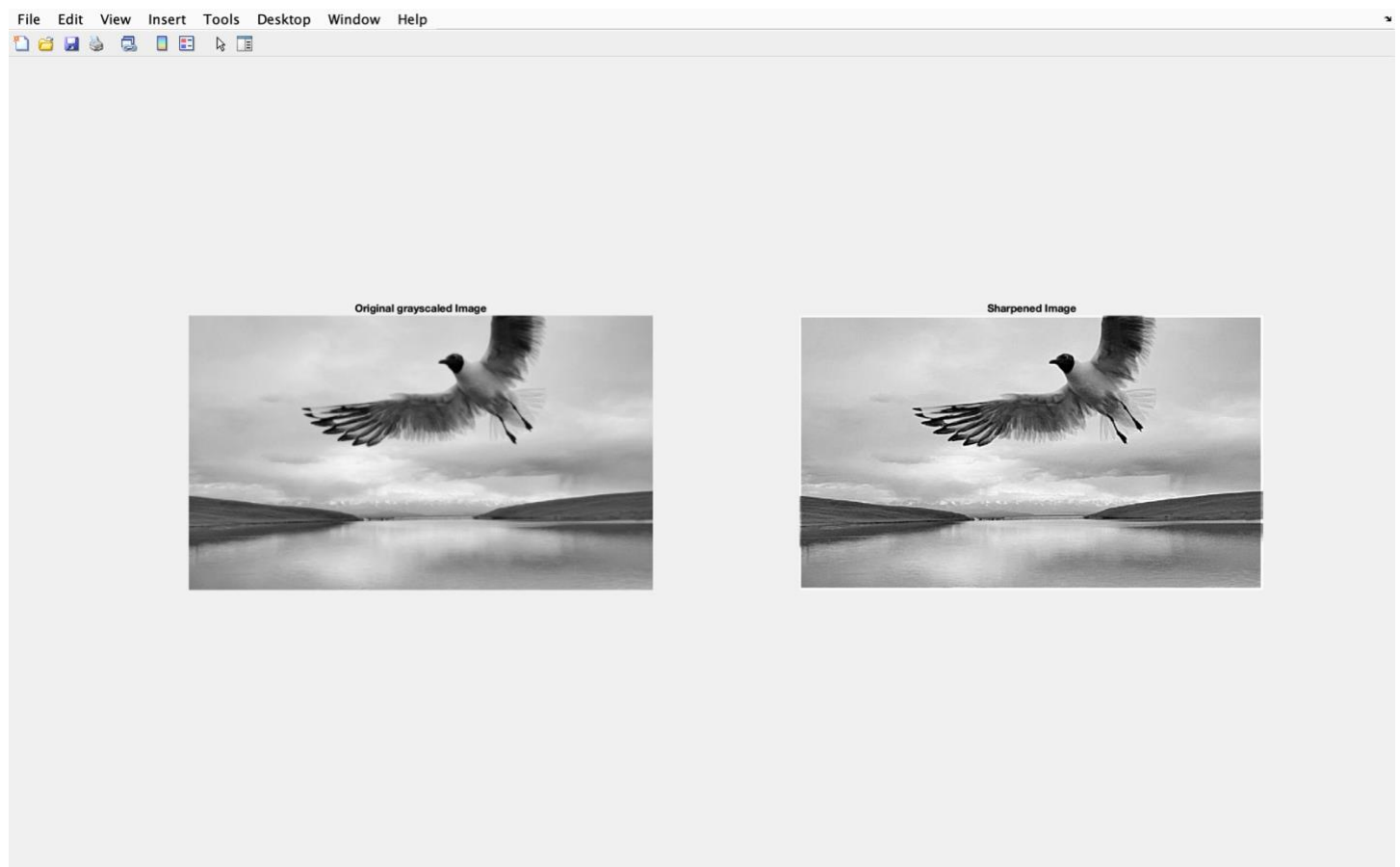


Figure 22 : outcome of the sharpening with scaling factor = 1

%% Sharpening

```
image3 = imread('bird.png');  
lab1sharpen(image3, 2);
```

Figure 23 : sharpening with scaling factor = 2

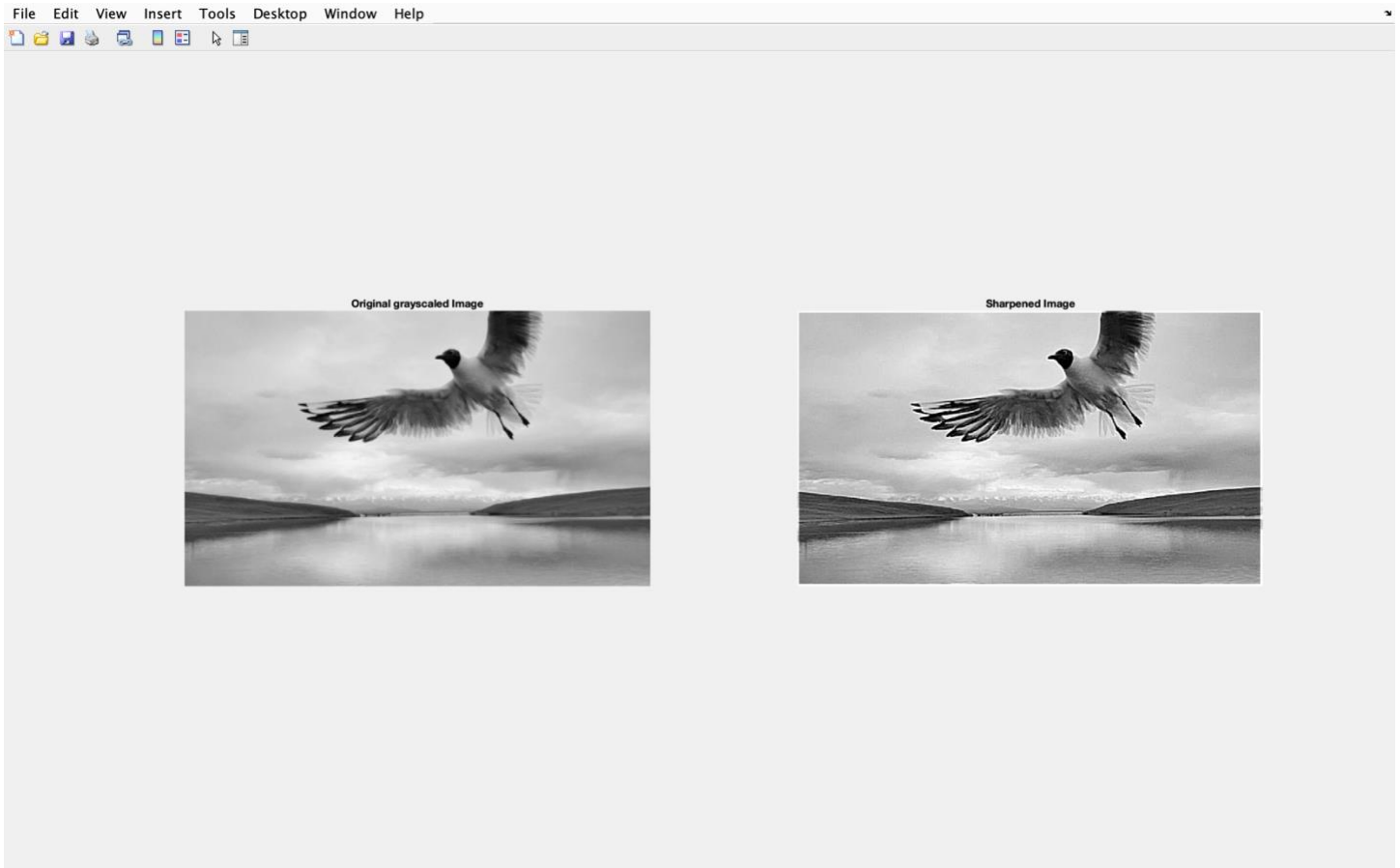


Figure 24 : outcome of sharpening with scaling factor = 2

%% Sharpening

```
image3 = imread('bird.png');  
lab1sharpen(image3, 3);
```

Figure 25 : sharpening with scaling factor = 3



Figure 26 : output of the sharpening with the scaling factor = 3

%% Sharpening

```
image3 = imread('bird.png');  
lab1sharpen(image3, 10);
```

Figure 27 : sharpening with scaling factor = 10

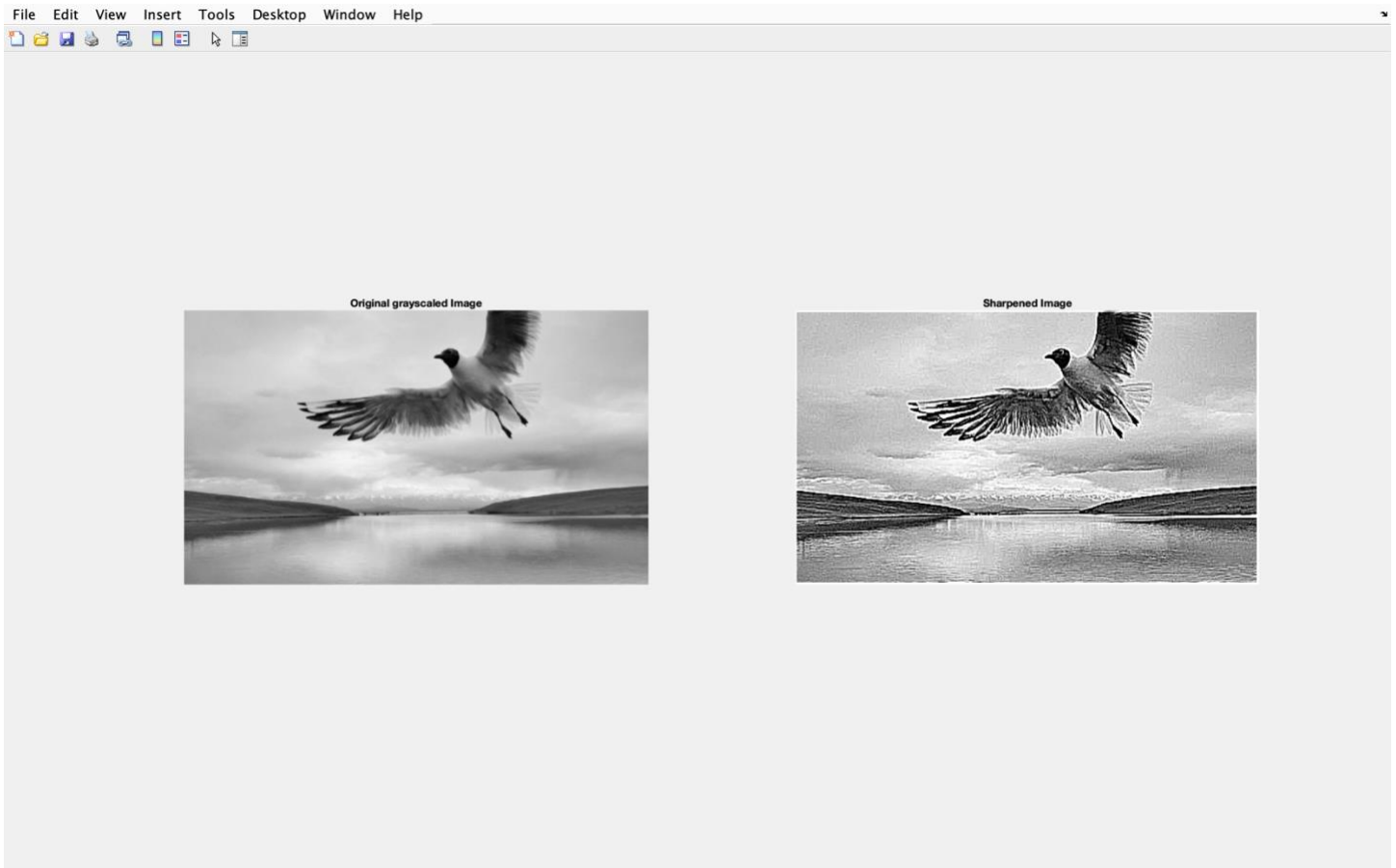


Figure 28 : the outcome of the sharpening with scaling factor = 10

So, as seen above, by increasing the scaling factor starting from 1 up to 10, we see the sharpness of the wings of the bird increases dramatically. Especially, in the output of sharpening with scaling factor = 10, the wing edges are so obvious. However, as you might have noticed, on the same figure, the noise is dramatically a lot. Of course, the tradeoff exists here and personally, I wouldn't choose to trade sharpness with this much noise on homogenous regions. To sum up, my preference would be using scaling factor 3 since the noise is not that much on homogenous areas and also the sharpness of the wings of the bird is enough for me. Of course, one might have chosen different scaling factor according to their requirements from the image.

-----THE END OF THE LAB#1 POST LAB REPORT-----