Computer Vision

Assignment 3

Alireza Moradi

October 12, 2020

1 Convolution

1.1 Zero-padding

Because we want to convolve the picture with the filter we should rotate the filter 180°.

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 2 & 1 & 6 & 0 \\ 0 & 7 & 1 & 1 & 1 & 0 \\ 0 & 3 & 1 & 2 & 0 & 0 \\ 0 & 1 & 4 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} * \begin{bmatrix} 1 & 2 & 1 \\ 1 & 0 & 1 \\ 1 & 3 & 1 \end{bmatrix} = \begin{bmatrix} 24 & 13 & 13 & 5 \\ 15 & 22 & 19 & 16 \\ 23 & 28 & 11 & 11 \\ 11 & 8 & 11 & 2 \end{bmatrix}$$

1.2 Replicate-padding

$$\begin{bmatrix} 1 & 1 & 2 & 1 & 6 & 6 \\ 1 & 1 & 2 & 1 & 6 & 6 \\ 7 & 7 & 1 & 1 & 1 & 1 \\ 3 & 3 & 1 & 2 & 0 & 0 \\ 1 & 1 & 4 & 0 & 2 & 2 \\ 1 & 1 & 4 & 0 & 2 & 2 \end{bmatrix} * \begin{bmatrix} 1 & 2 & 1 \\ 1 & 0 & 1 \\ 1 & 3 & 1 \end{bmatrix} = \begin{bmatrix} 37 & 19 & 23 & 31 \\ 26 & 22 & 19 & 23 \\ 34 & 28 & 11 & 14 \\ 23 & 21 & 17 & 12 \end{bmatrix}$$

As you can see, border pixels have different values because of different paddings. But center pixels have the same values because the filter doesn't exceed image's borders.

1.3 Zero-padded image

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 2 & 1 & 6 & 0 \\ 0 & 7 & 1 & 1 & 1 & 0 \\ 0 & 3 & 1 & 2 & 0 & 0 \\ 0 & 1 & 4 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

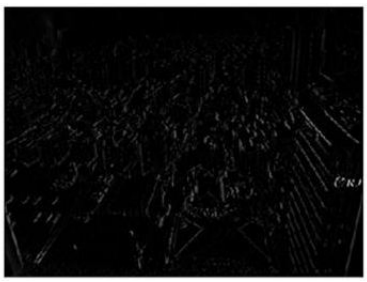
Because the filter size is 3×3, only 1 row of pixels is required for padding in each direction.

2 Kernels

2.1

-1	0	1

This filter is like the Laplace transform but It will show vertical details in the photo. like below:



2.2

1
2
1

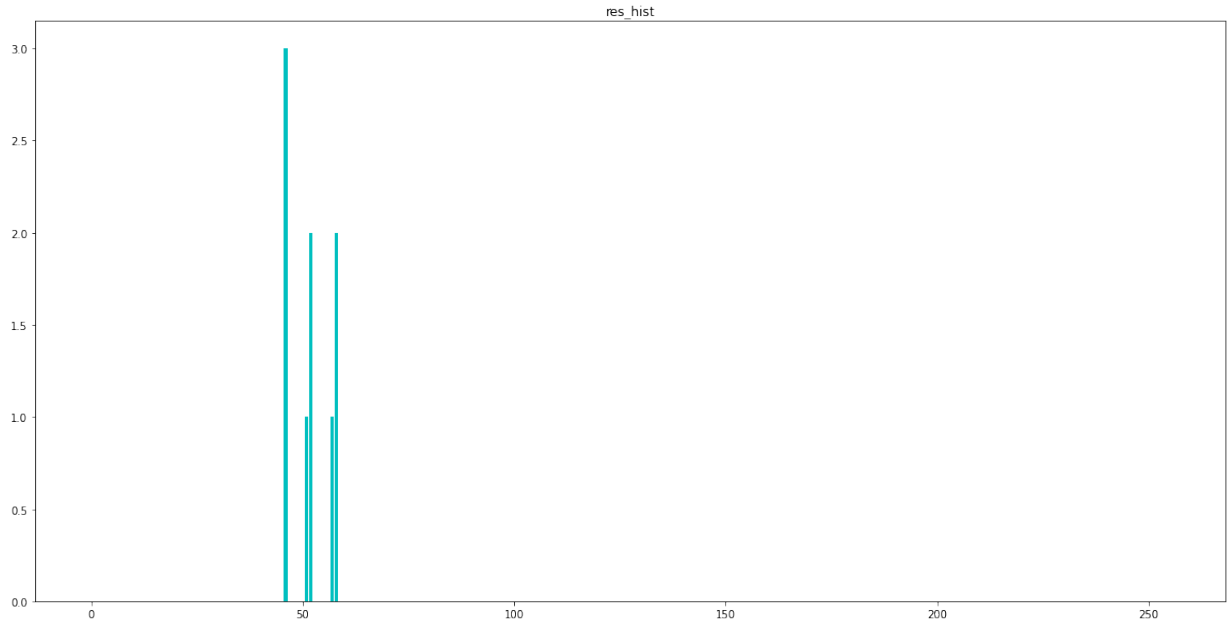
This is the average filter (smooths the photo) which will give more weight to the center pixel in order to not loose too much information e.g. in the borders.

3 CLAHE

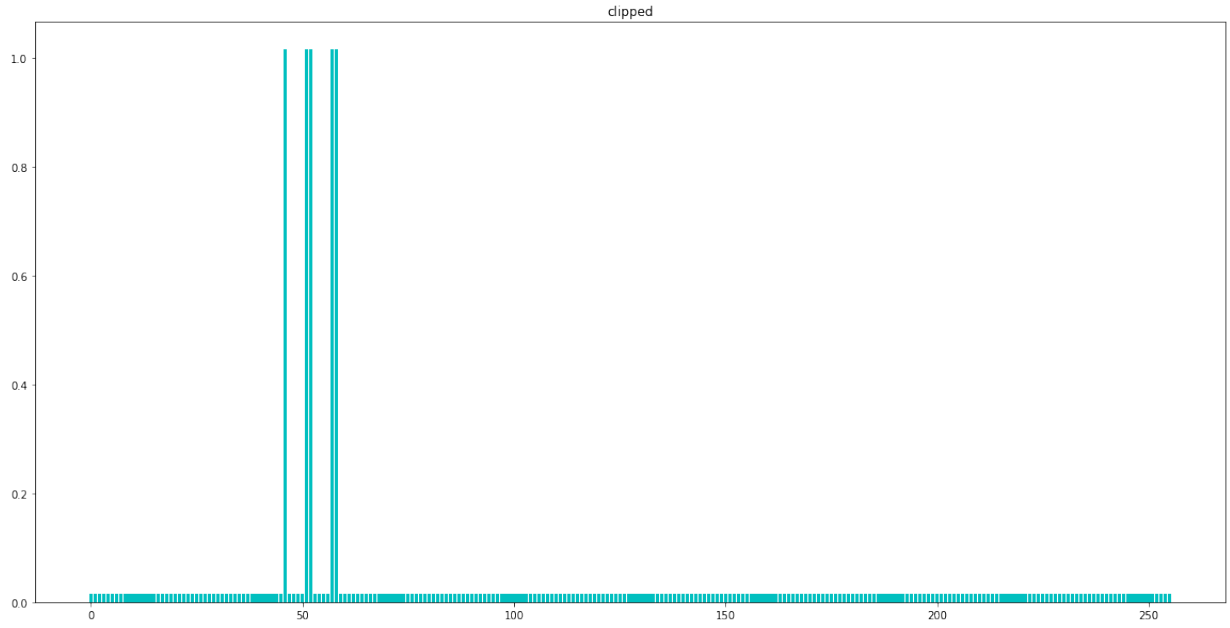
3.1 Manual

46	51	57	59
46	52	58	60
46	52	58	60

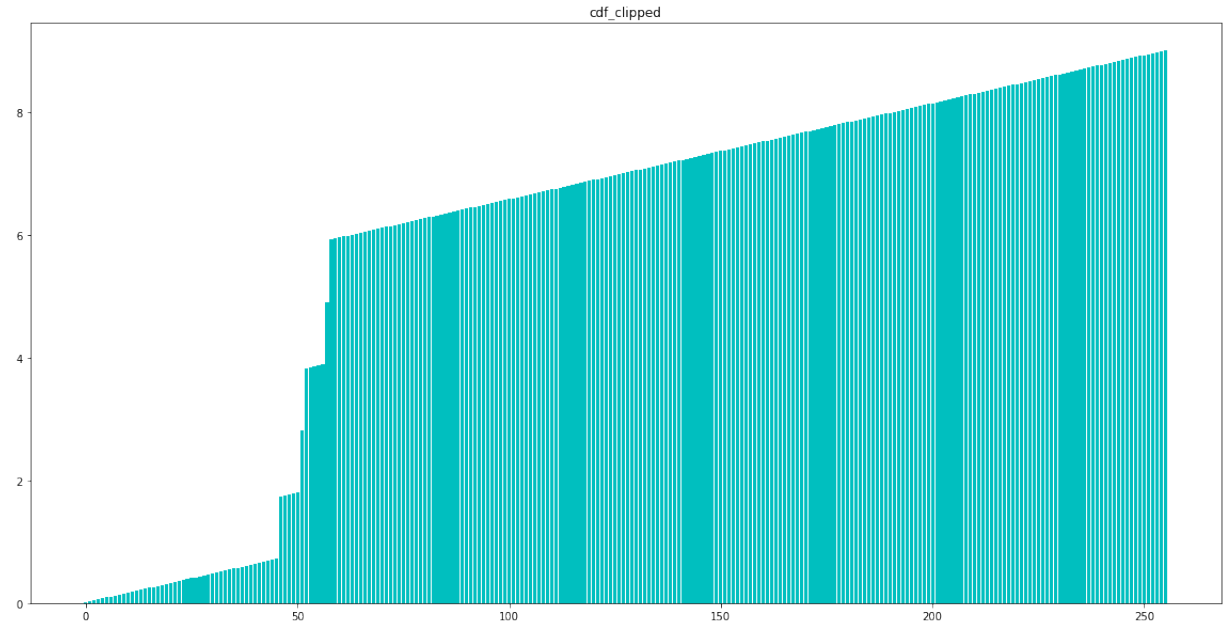
First we have to compute histogram for this window.



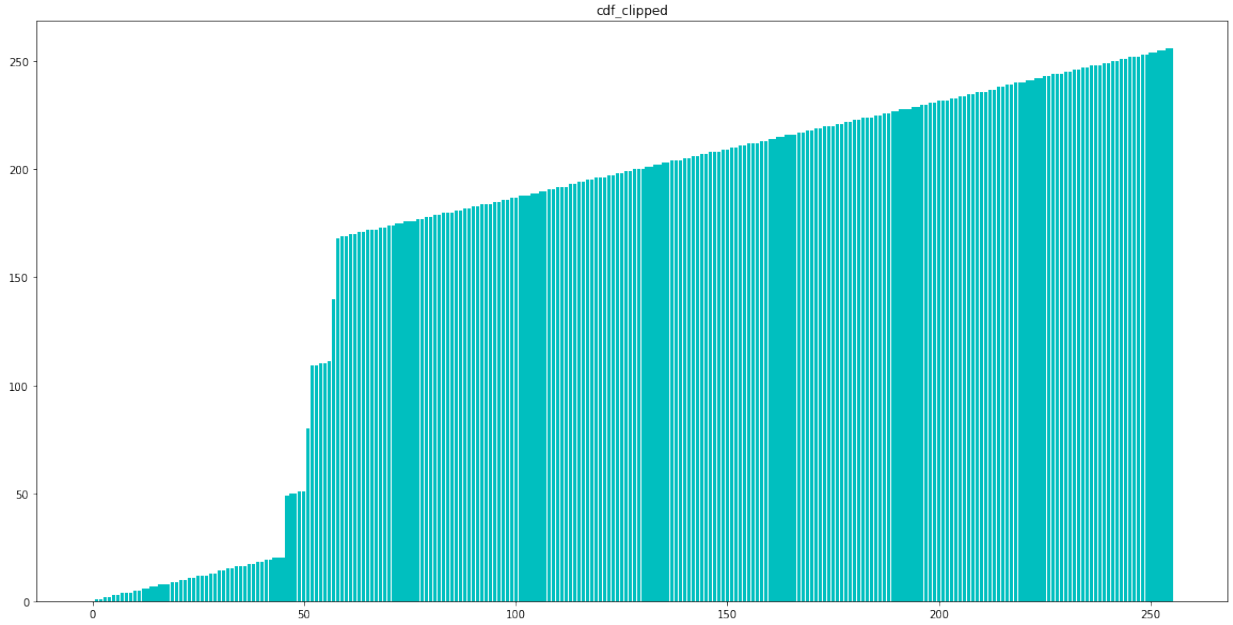
Now we apply clip limit 1 and spread the value among all the bins uniformly.



And now we compute CDF of the clipped image.



And now we do $\frac{L-1}{n}$ and round the results.



And if we look at the array containing these values, We see that 52 is mapped to 109.

46	51	57	59
46	52	58	60
46	52	58	60

Similarly for this window, 58 is mapped to 133. And I don't use padding and use the same transform for the neighbors of a border pixel. So the final result is:

49	80	105	162
49	109	133	191
49	109	133	191

For clip limit = 2 the result is:

62	91	114	199
62	148	171	255
62	148	171	255

3.2 OpenCV Implementation

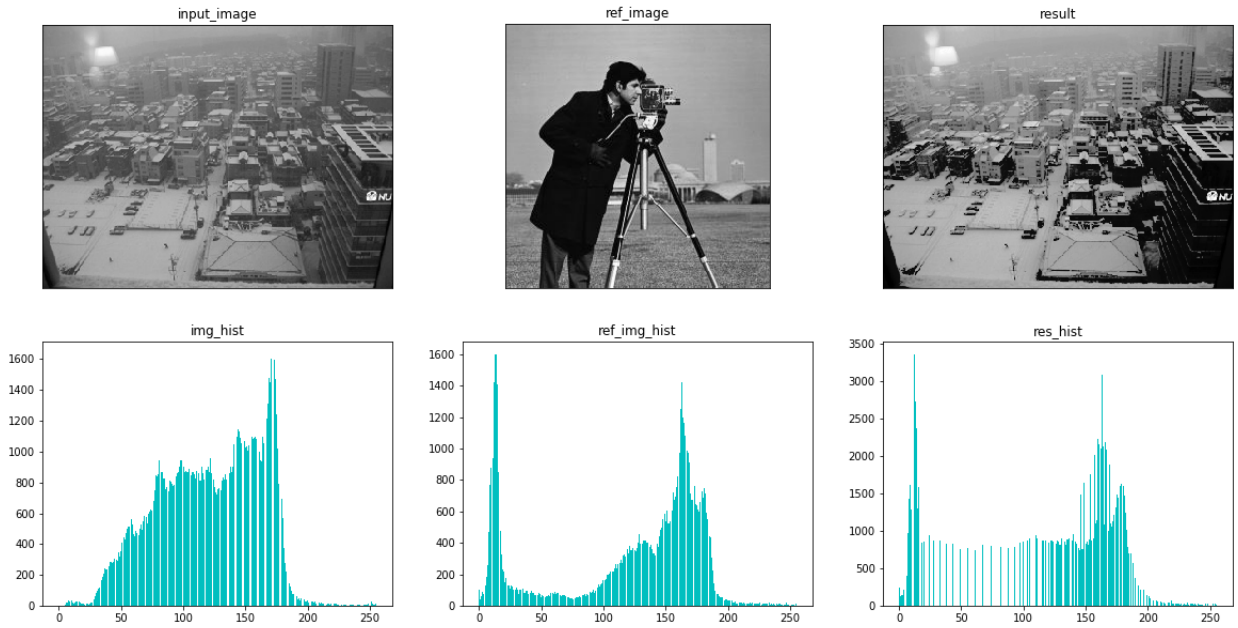
128	191	160	191
128	255	192	255
128	223	176	223

So because openCV uses `REFLECT_101` and I didn't, and also because openCV's CLAHE uses bilinear interpolation (to remove artifacts) for border pixels, the final results are different.

4 Implementations

4.1 Histogram Matching

I have 2 implementations for this question. The first one is fully manual where I equalize the histogram of both images and find the mappings. And finally using $T_1^{-1}T_2$ I match the histogram of the first image to the second one. Also bins that are not present are mapped to the nearest larger number.(This implementation is commented below.)
The second implementation is using `numpy's` linear interpolation function, where there's no need to fully equalize the histogram of each image, Instead we compute the normalized CDF of each image and give the results to the interp function and it does the matching.




```
def histogram_matching(img, ref_img):
    '''
    Matchs the histogram of the input image to the histogram of reference image.

    Parameters:
        img (numpy.ndarray): The input image.
        ref_img (numpy.ndarray): The reference image.

    Returns:
        numpy.ndarray: The result image.
```

```
    '''

    out_img = img.copy()
    L = 256
    n = img.shape[0] * img.shape[1]

    cdf_ref = compute_cdf(ref_img)
    cdf_img = compute_cdf(img)

    new_pixels = np.interp(cdf_img, cdf_ref, np.arange(256))
    out_img = (np.reshape(new_pixels[out_img.ravel()], out_img.shape)).astype(np.uint8)

#     for i in range(len(out_img)):
#         for j in range(len(out_img[i])):
#             for k in range(len(cdf_ref)):
#                 if cdf_img[out_img[i][j]] <= cdf_ref[k]:
#                     out_img[i][j] = k
#                     break

    return out_img
```

4.2 Gaussian Kernel

Here I used the below formula to fill in each pixel of the filter:

$$G(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

Then the filter is convolved with the image. This procedure is repeated with `OpenCV's GaussianBlur` function. `OpenCV's` implementation is faster and does a little better job in smoothing the noises.


```python
def gaussian_filter(size, std):
    '''
    Creates the Guassian kernel with given size and std.

    Parameters:
        size (int): The size of the kernel. It must be odd.
        std (float): The standard deviation of the kernel.

    Returns:
        numpy.ndarray: The Guassina kernel.
    '''

    kernel = np.zeros((size,size), np.float)
    pi = np.pi
    e = np.exp(1)
    r = size//2

    for i in range(size):
        for j in range(size):
            kernel[i][j] = (1/(2*pi*(std**2))) * e**-(((i-r)**2 + (j-r)**2)/(2*(std**2)))
    return kernel
```

```
def opencv_filter(img):
    '''
    Applys the OpenCV's guassian blur function on input image.

    Parameters:
        img (numpy.ndarray): The input image.

    Returns:
        numpy.ndarray: The result image.
    '''

    out = cv2.GaussianBlur(img,(5,5),0)
    return out
```