Homework 1

Image Filtering

Student: Findeis, Frederic Tabo, freddy@postech.ac.kr
Lecturer: SUNGHYUN CHO, s.cho@postech.ac.kr

1 Introduction

This work is done in Python. Some allowed operations (for example reading the image) are done with OpenCV. Therefore images are stored as numpy arrays with the convention $height \times width \times channel$. To understand the code better, please refer to the README.md.

2 Gaussian Filtering

2.1 Algorithms and Code

2.1.1 Kernel Creation

These kernels are created by calculation from the Gaussian Bell curve with specific parameters. For 1 Dimension the formula is $f(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$. This is rewritten in code with the center μ.

```
center = size // 2

for i in range(0, size):
    kernel[i] = (
        1
        / (np.sqrt(2 * np.pi) * sigma)
        * np.exp(-((i - center) ** 2) / (2 * (sigma**2)))
    )

kernel = kernel / np.sum(kernel) # normalize
```

For 2 Dimensions the formula is: $f(i,j) = \frac{1}{2\pi\sigma^2} e^{-\frac{(i-\mu)^2+(j-\mu)^2}{2\sigma^2}}$

```
center = size // 2

for i in range(0, size):
    for j in range(0, size):
        kernel[i, j] = (
            1
            / (2 * np.pi * sigma**2)
            * np.exp(-((i - center) ** 2 + (j - center) ** 2) / (2 *
                (sigma**2)))
        )

kernel = kernel / kernel.sum()
```

The exact same 2d kernel can also be obtained by multiplying 2 1d kernel and getting the outer product.

```
kernel_1d = gaussian_1d(sigma, size)
kernel = np.outer(kernel_1d, kernel_1d)
kernel = kernel / np.sum(kernel)
```

In contrast to normal convolution, we don't have to regard in the following, that the kernel has to be reflected around its middle point since Gaussian kernels are symmetric.

2.1.2 2D Gaussian Kernel Execution (Non-Separable

The Gaussian kernel has the size: $size \times size$. The goal is to apply this for every single window of this size in the image. Therefore we get now a smaller result image. From an image of size: $width \times height$ we get a new image with size $(width - filtersize + 1) \times (height - filtersize + 1)$. To apply this filter, we need to get, for every pixel of the output image, the image window multiplied by the kernel and then summed up. We don't need to divide by an additional factor, since the kernel is normalized. As a result, we have the following loops:

1. k : Loop for the channels (3 or 1 iteration, depending if color or grayscale image)
2. i : Loop for the height (size of output height dimension
3. j : Loop for the width (size of output width dimension

Each execution on the lowest, we will get the resulting value for the pixel (i, j) at the channel k. To calculate this value, we have to calculate

```
output_width = width - filter_size + 1
output_height = height - filter_size + 1

for k in range(3):
    for i in range(0, output_height):
        for j in range(0, output_width):
            output_image[i, j, k] = np.sum(
                kernel * image[i : i + filter_size, j : j +
                    filter_size, k]
            )
```

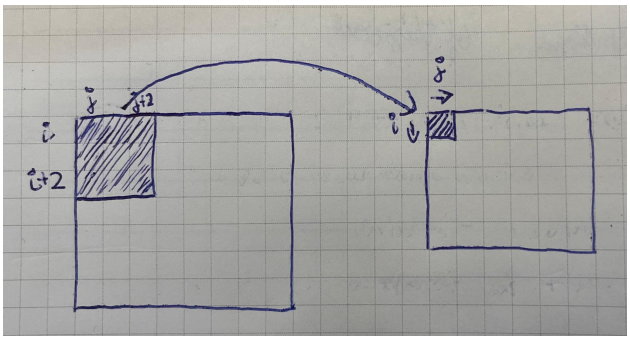


Figure 1: Non-Separable Convolution. Here the "+2" is an exampel for a kernel size of 3

2.1.3 1D Gaussian Kernel - separable

If a kernel $A \in k \times k$ is separable, it means we can also distribute it in $a \cdot a^T, a \in k$.

If we now want to apply the convolution, we can do so stepwise by first applying the horizontal filter and then the vertical filter, or vice versa.

For this, we first calculate the final output dimensions as we did in the previous case. But since we now have two convolution steps, we need an additional matrix to store the intermediate result. Since we first apply the horizontal filter, the width shrinks first, while the height remains the same. When applying the convolution, we always convolve the corresponding slice by indexing the corresponding pixels. This slice automatically becomes the default 1D dimension of the kernel, so the kernel vector does not need to be transposed when switching from horizontal to vertical convolution. 2 shows a small example to illustrate where first horizontal and then vertical convolution is applied.

The k-iteration is again present for the color.

```
output_width = width - filter_size + 1
output_height = height - filter_size + 1

intermediate_image = np.zeros((height, output_width, 3))

output_image = np.zeros((output_height, output_width, 3))

for k in range(3): # color
    for i in range(0, height): # horizontal filter
        for j in range(0, output_width):
            intermediate_image[i, j, k] = np.sum(
                image[i, j : j + filter_size, k] * kernel
            )

    for i in range(0, output_height): # vertical filter
        for j in range(0, output_width):
            output_image[i, j, k] = np.sum(
                intermediate_image[i : i + filter_size, j, k] *
                   kernel
            )
```

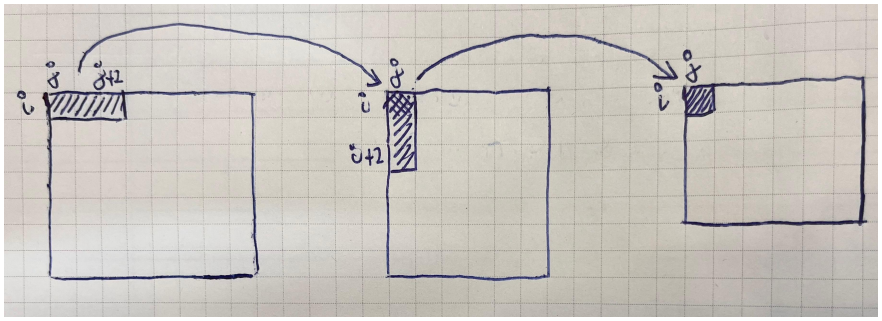


Figure 2: Separable Convolution, kernel size is "2" in this example.

2.2 Examples

All the example images have been processed with the Gaussian kernel with parameters:

- Kernel Size: 5
- Kernel Sigma: 3
- Border Type: cv2.BORDER_CONSTANT in padding with size 1 for top, bottom, left and right

Please find the results attached HERE and HERE.

The Gaussian Filter can be adapted via the kernel_size and the sigma parameter, see 3. With a kernel_size of 1 the image does not get change no matter how big the sigma is, since each pixels gets only convoluted without its neighbors.

The result of convolution with different Gaussian Kernels can be viewed in 3. The corresponding kernels (as a 1d representation) are illustrated 4.

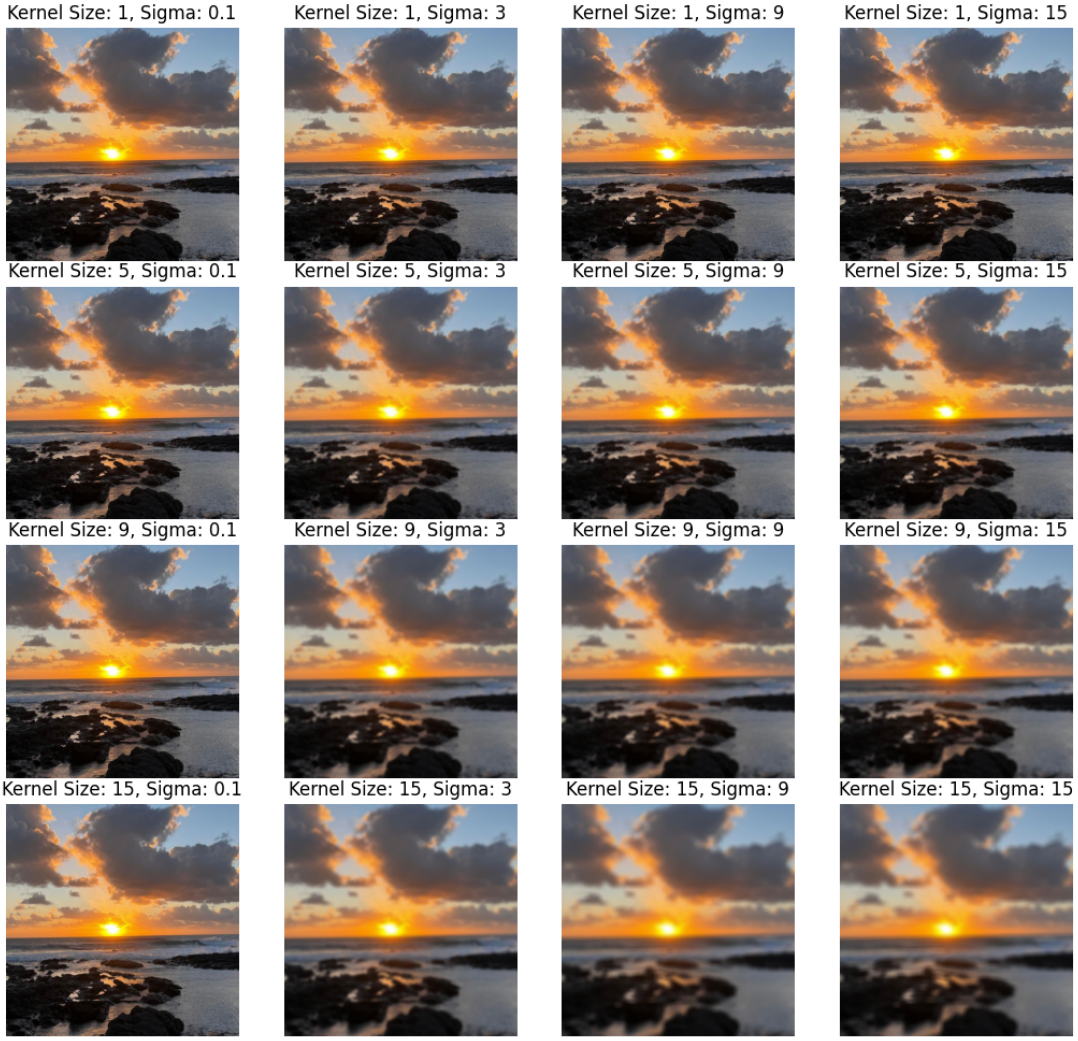


Figure 3: Comparison of Images of different kernel size and sigma

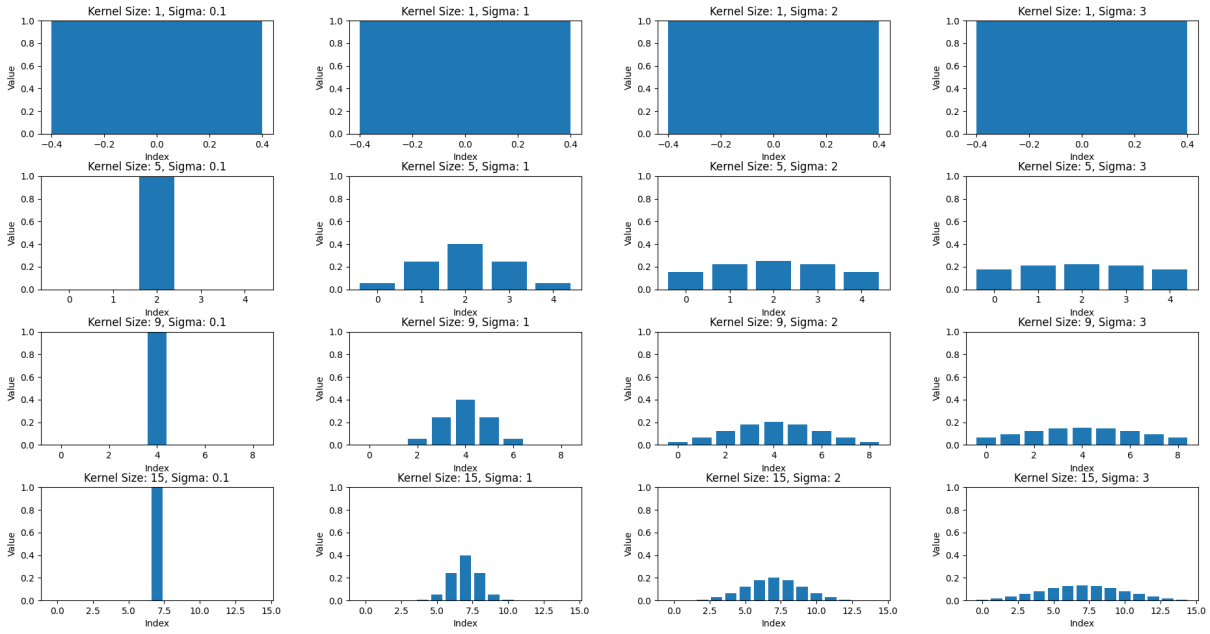


Figure 4: Comparison of Bell Curves of different kernel size and sigma

The bigger the sigma parameter is, the greater the influence of the neighboring pixels and the smaller the influence of the middle pixel. The kernel_size defines how many neighboring pixels are considered. The kernel size must be chosen according to sigma. In the column of sigma 1, it makes no sense to make the kernel size bigger than five, since the influence of the added kernel's field is minimal and can be neglected. A kernel size that is too large unnecessarily shrinks the image and adds unnecessary computation time.

In comparison to the kernel_size which increases the computation with increasing size, the sigma does not change the computation time at all, because it neither adds nor removes any computation time.

Good choices for an example kernel are kernel_size 5 and sigma 1.

2.2.1 Borders

5 shows different types of borders on images.

- cv2.BORDER_CONSTANT: This adds a constant predefined color (for example black) to the image.
- cv2.BORDER_REPLICATE: This just regards the most outwards pixel and continues its color (constant) for the complete border. Example we have abcdefg: we will get aaa—abcedfg—ggg
- cv2.BORDER_REFLECT and cv2.BORDER_REFLECT_101: The reflect the pixels at the edge. Image like putting a mirror at the old border of the image. The difference is the inclusion (cv2.BORDER_REFLECT) and the exclusion (cv2.BORDER_REFLECT_101) of the most outward pixel. Example, we have the image abcdefgh: cv2.BORDER_REFLECT - Border will be mirror reflection of the border elements, like this : fedcba—abcdefgh—hgfedcb cv2.BORDER_REFLECT_101 - Same as above, but with a slight change in which the outtermost pixels (a or h) are not repeated : gfedcb—abcdefgh—gfedcba

  So in short: border_reflect mirrors the pixels at the bordering including the last pixel, border_reflect_101 does not repeat the last pixel
- cv2.BORDER_WRAP: It uses the pixels from the opposite edge to fill the border. Example we have abcdefg: We will get efg—abcdefg—abc

In my opinion, the best method is cv2.BORDER_REPLICATE since it is the closest to a real extrapolation. It tries not to generate new information that could be mismatching, as in the reflect method, and it has the opportunity to continue the sky almost indistinguishably.

cv2.BORDER_WRAP is in my opinion the worst, since it assumes the bottom part of the image would fit the top part of the image. This is in most cases not likely, and leads to unmatching results.

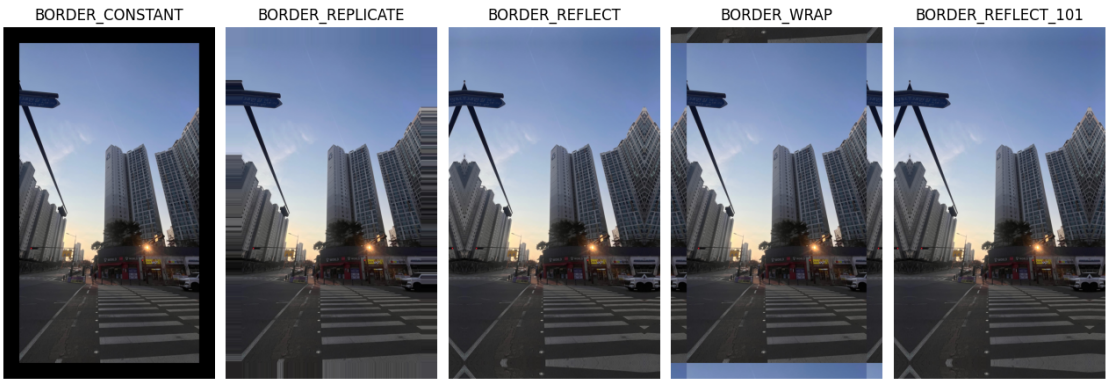


Figure 5: Different types of borders with size 200 pixel

2.3 Computation Time

2.3.1 Theoretical

As already mentioned, the kernel_size increases the computation time with increasing number and the sigma does not change it. The image size has a huge impact on the computation time.

Let h be the height, w the width and k the kernel size.

For a non-separable kernel the runtime is $O(h \times w \times k^2)$. For a separable kernel the runtime is $O(h \times w \times k + h \times w \times k) = O(2 \times h \times w \times k)$.

If we want to solve this, we have to regard

$$k^2 = 2 \cdot k$$

So if $k >= 2$, it is faster to run the separable kernel in theory.

2.3.2 Practical

The following tests have been conducted with random images. 6 shows the computation time due to different image sizes. The kernel size in this experiment is 3.

7 shows how the computation time evolves with different kernel sizes. The image size is here 300×300. Both experiments present the difference between separable (1D Gaussian) and non-separable (2D Gaussian).

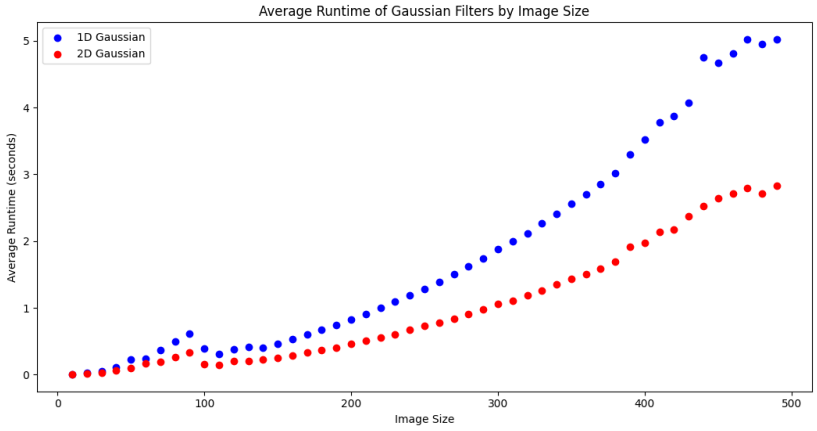


Figure 6: Comparison of different image size

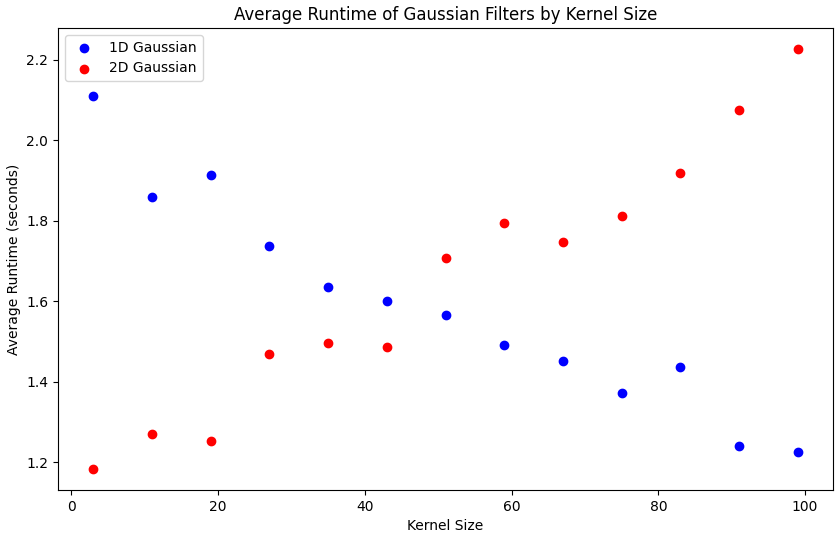


Figure 7: Comparison of different kernel sizes

The Gaussian kernel should in theory be faster for $kernel\_size > 2$ but in these experiments it is only faster if the $kernel\_size > 45$. This is due to the inefficient looping and memory access in python.

3 Histogram Equalization

3.1 Algorithms and Code

3.1.1 Grayscale Histogram Equalization

First, grayscale histogram equalization is explained, and then it is explained for color images.

To equalize a histogram, we first have to compute it. We have up to 256 different gray tones in an image (from 0 to 255) so an array with the length 256 is created. Now there follows an iteration through the whole image. The grayscale of this pixel is then added at its index in the histogram.

```
def get_histogram(image: np.ndarray) -> np.ndarray:
    """
    Compute the histogram of an image.
    """
    height, width = image.shape
    histogram = np.zeros(256, dtype=np.int32)
    for i in range(height):
        for j in range(width):
            histogram[image[i, j].astype(np.uint8)] += 1

    return histogram
```

This histogram has now to be cumulated to be equalized: Therefore we create a new array with a size of 256 (for the intensity levels). The first element is initialized with the histogram value for the zeroth entry, since there is nothing on which this entry can be accumulated. For the remaining elements, we look in the previously created histogram, how often it appears and then add the previous number from the cumulated histogram.

```
cum_histogram = np.zeros(256, dtype=np.int32)
cum_histogram[0] = histogram[0]
for i in range(1, len(histogram)):
    cum_histogram[i] = cum_histogram[i - 1] + histogram[i]
```

Now the histogram can be equalized. It is first shifted to the left-hand side by subtracting the lowest element to every entry of the cumulated histogram. After that, we have to stretch it to the full grayscale range - this is done by multiplying by the highest possible color and dividing by the difference of the maximum value and the minimum value of the histogram. After that, the only remaining thing is to map the pixel colors via the normalized cdf to an equalized image.

```
cdf = cum_histogram
cdf_normalized = (cdf - cdf.min()) * 255 / (cdf.max() - cdf.min())
cdf_normalized = cdf_normalized.astype(np.uint8)

equalized_image = cdf_normalized[image]
```

3.1.2 Color Version

A color image is represented by 3 channel which each consist of $width \times height$. The image is now split into those channel, for which each one is equalized, and then the channels are merged back again.

```
r, g, b = cv2.split(image)

r_equalized, g_equalized, b_equalized = (
    grayscale_histogram(r),
    grayscale_histogram(g),
    grayscale_histogram(b),
)

equalized_image = cv2.merge((r_equalized, g_equalized, b_equalized))
```

3.2 Examples

In the following, the lecture images and some own images in color and gray have been equalized. 8, 9, 10 and 11. You can see the before and after picture, contrast, brightness, and histogram.

Due to the human eye's color reception, for example in 8 in Image 2, new structures can be seen, which were hidden before the transformation.

For images with a distribution which is extremely spiky like 10 Image 3, the Histogram equalization does not work well. It can be seen that the contrast is always increased since the histogram is stretched. The resulting brightness is always around $256/2 = 128$, due to the equalization. This means dark pictures get brighter and vice versa.

In some cases, it can also lead to loss of image details, and as seen below, unnatural images which look strongly edited.

Alternatives to improve this method could be gamma correction and histogram matching.

Another possibility would be to clip first the image to prevent it from over-equalizing.

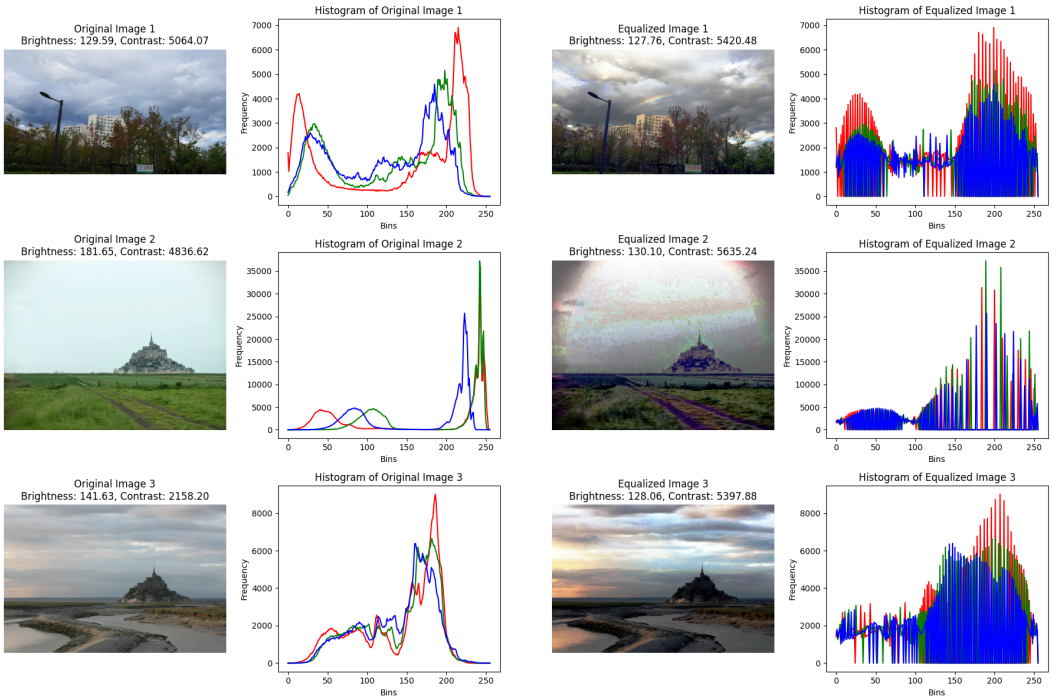


Figure 8: Histogram equalization lecture color images

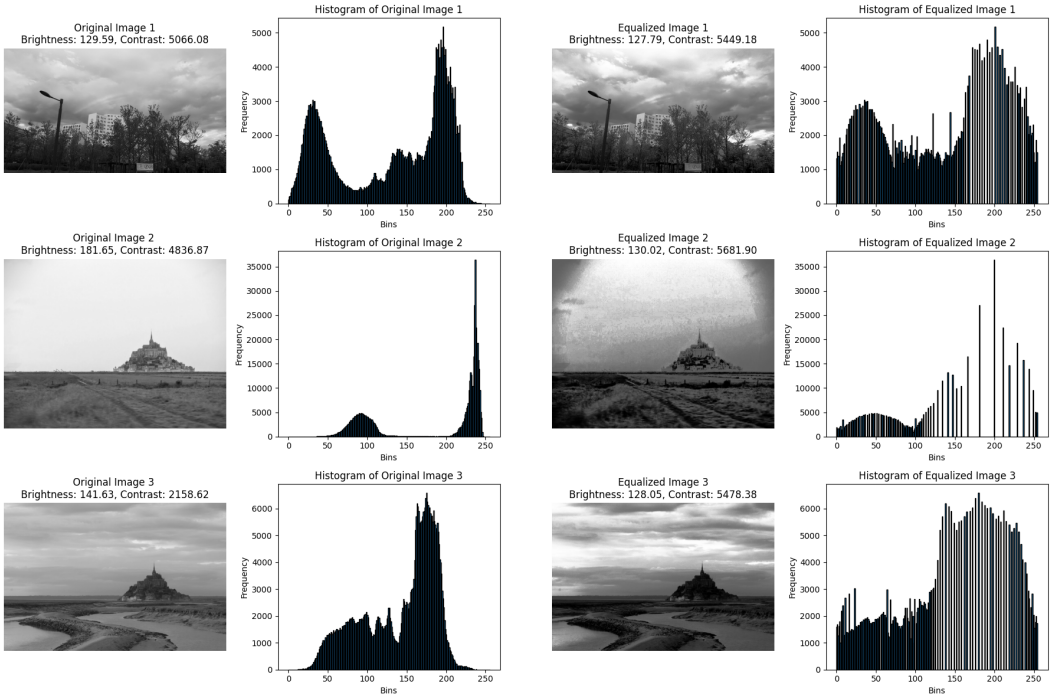


Figure 9: Histogram equalization lecture gray images

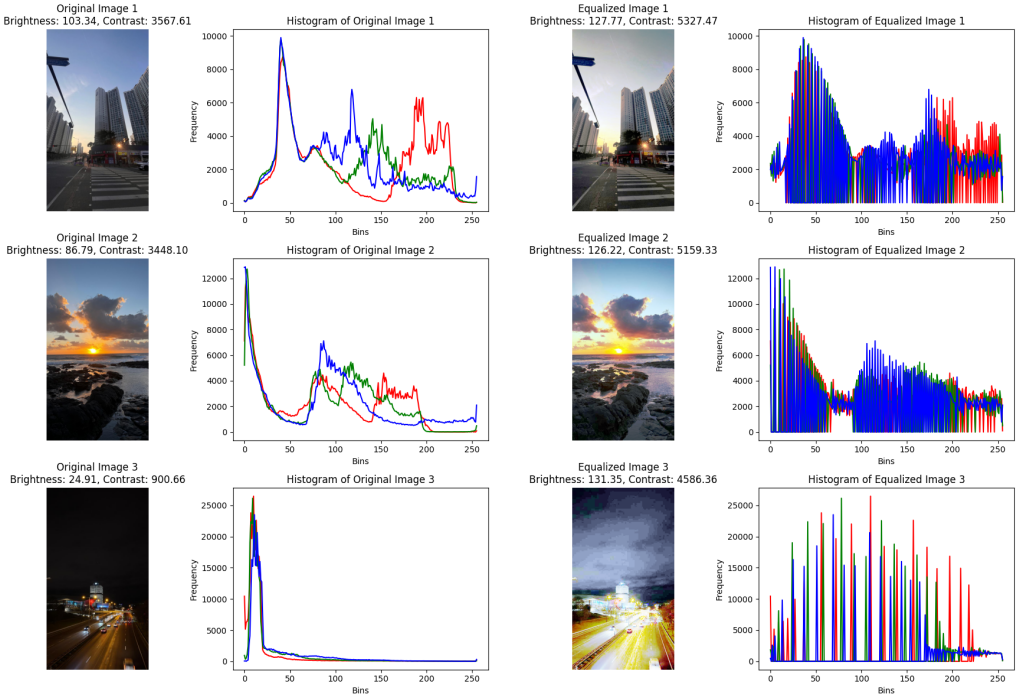


Figure 10: Histogram equalization own color images

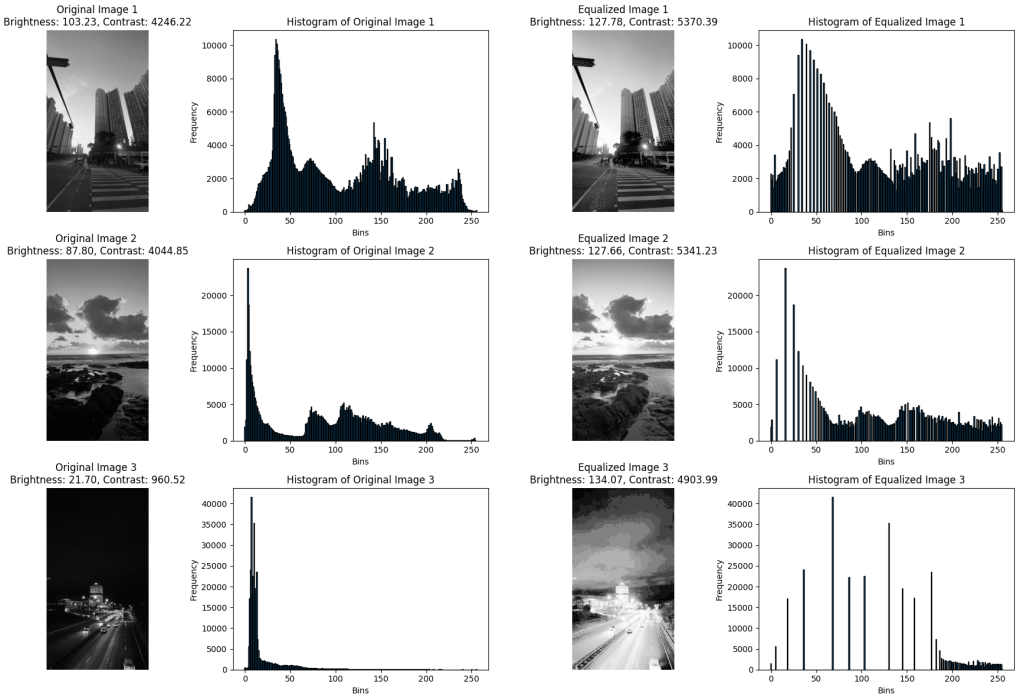
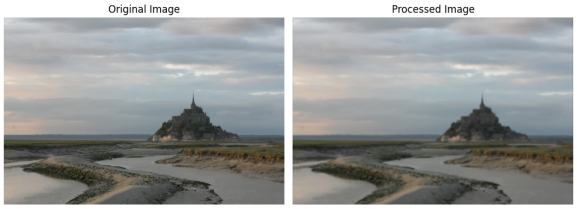


Figure 11: Histogram equalization own gray images

4 Images Attached

(a) Image 1

(b) Image 2



(c) Image 3

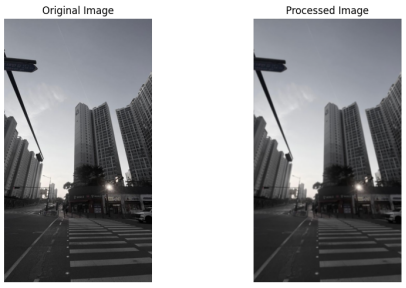
(d) Image 4

(e) Image 5

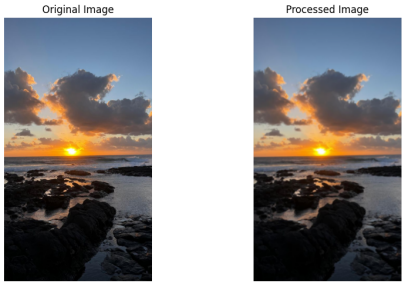
(f) Image 6

Figure 12: Images from gaussian_lecture.

(a) Image 1



(b) Image 2



(c) Image 3

(d) Image 4

(e) Image 5

(f) Image 6

Figure 13: Images from gaussian_own_small.