Computational Photography Homework 2
– Image Filtering

20170243 SimJaeYoon

1. Overview

This assignment 2 is about unsharp masking and edge-aware filtering. So what I have to do is to implement a bilateral filter and unsharp masking in the spatial domain and frequency domain. I used the same version of the library as python 3.7.11, numpy 1.20.3, opencv 4.5.3, matplotlib 3.4.3 to implement this assignment. And I used visual studio code as IDE. I used my test images as follow.

My test images

2. Unsharp masking

Unsharp masking is an image sharpening technique, first implemented in darkroom photography, but now commonly used in digital image processing software. Unsharp masking is defined as $Y = X + \alpha(X - G * X)$ where Y is the result of sharpened image, X is original input image, G is low pass filter like Gaussian filter, and α is a parameter to control strength of sharpening. The size and shape of G is controlled by the standard variation parameter σ. In this assignment, I must make two types of unsharp masking.

2.1 Using spatial domain filter

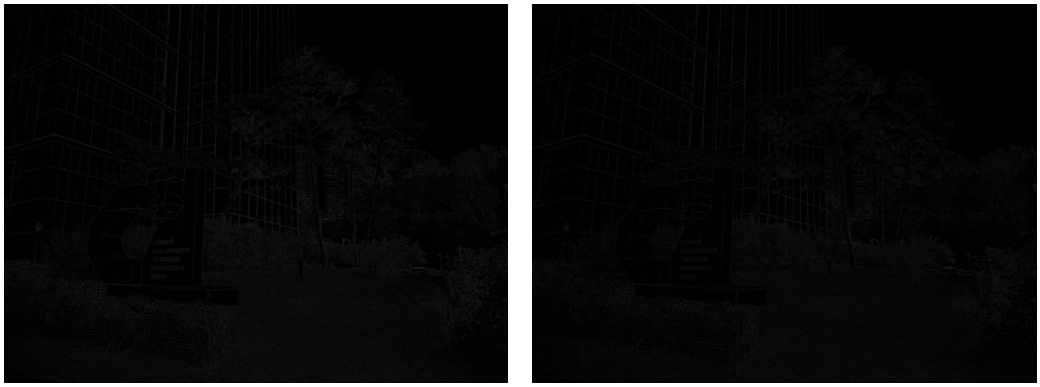
Unsharp masking using spatial domain means that I must use the image pixel values themselves. I used two types of low pass filter. One is cv2.blur() function and another is cv2.GaussainBlur() function. The cv2.blur() function blurs the average of neighboring pixel values within a predetermined kernel size to the pixel value of the image. Unlike the average blurring, which

gave all pixels the same weight, Gaussian blurring, which proceeds using cv2.GaussianBlur() function, gives a high weight to the central pixel. If the third parameter of the cv2.Gaussianblur() function is 0, the sigma is calculated and used according to the specified kernel size. Giving a value other than zero can change the standard deviation.

```
path = "../images/"
input_image_name = sys.argv[1]
input_image = cv2.imread(path + input_image_name, cv2.IMREAD_COLOR)
output_image = spatial_domain_filtering(input_image, 0)
```

```
def spatial_domain_filtering(src, opt):
    if opt == 0:
        blurred_image = cv2.blur(src, (5, 5))
    elif opt == 1:
        blurred_image = cv2.GaussianBlur(src, (5, 5), 1.5)

    dst = src.astype(np.int16)
    mask = dst - blurred_image
    # cv2.imwrite("../images/myimage_mask.jpg", mask)
    dst = dst + 0.5 * mask

    return dst
```

According to the property of Gaussian blur filter, the image can preserve more details than blur filter. So, when the kernel size is the same, I compared the blur filter and the Gaussian blur filter ($X - G * X$). (If you look closely, you can see the difference)



Using blur filter (5X5)(L) / Using Gaussian blur filter (5X5, $\sigma = 1.5$)(R)

And filters were compared according to the size of the kernel G when all other conditions were the same. As the kernel size increases, it becomes difficult to preserve the details of the original image ($G * X$). However, if I perform a negative operation on the original image, on the contrary, more details can be preserved ($X - G * X$).

Using blur filter (3X3)(L) / Using blur filter (11X11)(R)

Using Gaussian blur filter (3X3, $\sigma = 1.5$)(L) / Using Gaussian blur filter (11X11, $\sigma = 1.5$)(R)

Next, the results were compared through the change in σ value in Gaussian flitter. As the s σ value increases, it becomes difficult to preserve the details of the original image ($G * X$). However, if I perform a negative operation on the original image, on the contrary, more details can be preserved ($X - G * X$).

Using Gaussian blur filter (5X5, $\sigma = 0.5$)(L) / Using Gaussian blur filter (5X5, $\sigma = 3.0$)(R)

Finally, I compared the results in spatial domain according to the change in α values. The α value is a parameter to control strength of sharpening. So, when the α value is large, the sharpening effect appears more effectively. It is a C5 building in our university, and if you look

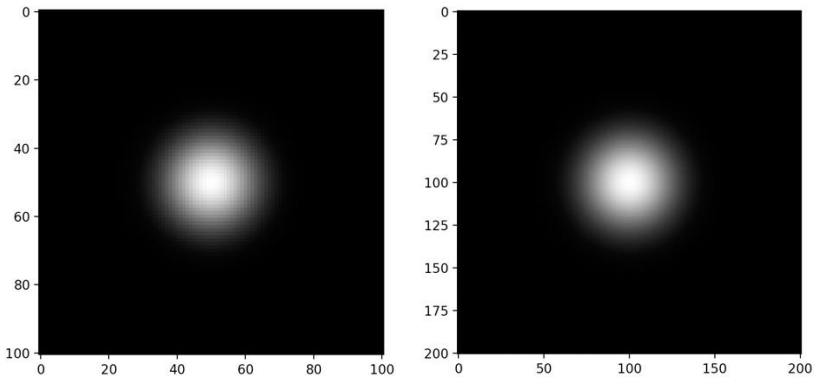
closely such as letters, you can see that the image has become sharper when the σ value is larger.

Unsharp masking in spatial domain using blur filter (11X11, $\alpha = 0.1$)(L) /
Unsharp masking in spatial domain using blur filter (11X11, $\alpha = 1.0$)(R)

2.2 Using frequency domain filter

Unsharp masking using frequency domain means that we must use acquired values through Fourier Transform of pixel values. And we use the algorithm of FFTs in this process. We should make Gaussian filter G to make blur image. In order to make a 2d Gaussian filter, a 1d Gaussian filter was created as follows, and a 2d Gaussian filter was created through their linear combination. The size of the Gaussian filter circle was different and I compared them as follow. The more the size of the circle changes, the more it will affect the subsequent results.

```
# 1D Gaussian filter
def gauss(n = 11, sigma = 1):
    r = np.arange(0, n, dtype = np.float32) - (n - 1.) / 2.
    r = np.exp(-r ** 2. / (2. * sigma ** 2))

    return r / np.sum(r)

# 2D Gaussian filter
def gauss2d(shape = (11,11), sigma = 1):
    g1 = gauss(shape[0], sigma).reshape([shape[0], 1])
    g2 = gauss(shape[1], sigma).reshape([1, shape[1]])

    return np.matmul(g1,g2)
```



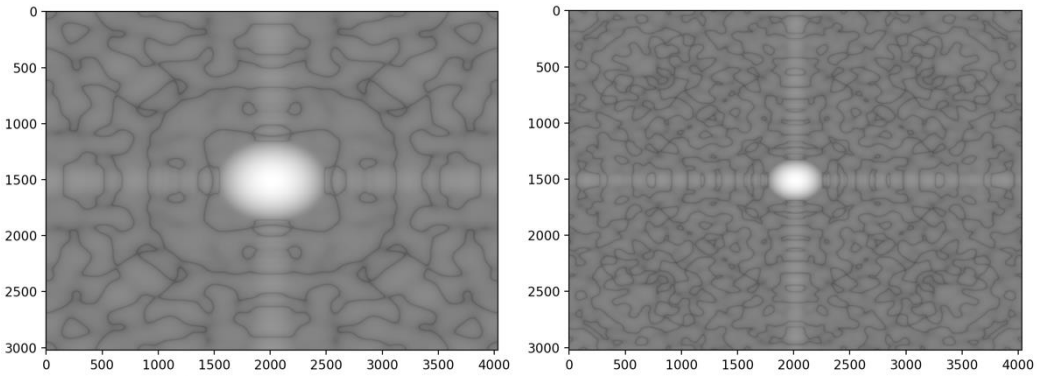
2D Gaussian filter (size = 100)(L) / 2D Gaussian filter (size = 200)(R)

If frequency domain filtering is performed, padding and shifting should be considered. Usually, the convolution operation of the input image and filter kernel can be performed, but if the sizes are different when performing Fourier transform and multiplication, it cannot be performed. So, first, it is necessary to pad the filter kernel to add 0 to the outside so that it is equal to the size of the input image. And in order to obtain proper results, the center region must be moved up to the left, and at this time, the non-zero values must be shifted to the outside to produce proper results. So, if we proceed with padding and shifting and do Fourier transform at the end, we can get proper results. In the case of a convolution operation in the spatial domain, the operation is usually performed by placing the center of the filter kernel in the center of the image. However, when proceeding with Fourier transform in the frequency domain, it is necessary to consider the center of the filter corresponding to the origin of the image.

```
# Pad and shift the center of filter
def psf2otf(flt, image_shape):
    flt_top_half = flt.shape[0] // 2
    flt_bottom_half = flt.shape[0] - flt_top_half
    flt_left_half = flt.shape[1] // 2
    flt_right_half = flt.shape[1] - flt_left_half

    flt_padded = np.zeros(image_shape, dtype = flt.dtype)
    flt_padded[:flt_bottom_half, :flt_right_half] = flt[flt_top_half:, flt_left_half:]
    flt_padded[:flt_bottom_half, image_shape[1] - flt_left_half:] = flt[flt_top_half:, :flt_left_half]
    flt_padded[image_shape[0] - flt_top_half:,:flt_right_half] = flt[:flt_top_half,flt_left_half:]
    flt_padded[image_shape[0] - flt_top_half:, image_shape[1] - flt_left_half:] = flt[:flt_top_half, :flt_left_half]

    return np.fft.fft2(flt_padded)
```

And in the Gaussian lowpass filter, the size of the circle indicates how much I want to preserve the low frequency component. If the size of circle is larger, then the low frequency components will be preserved more. Conversely, if the size of the circle becomes smaller, it wants to preserve less low frequency components. So, as the size of the circle increases, it creates more

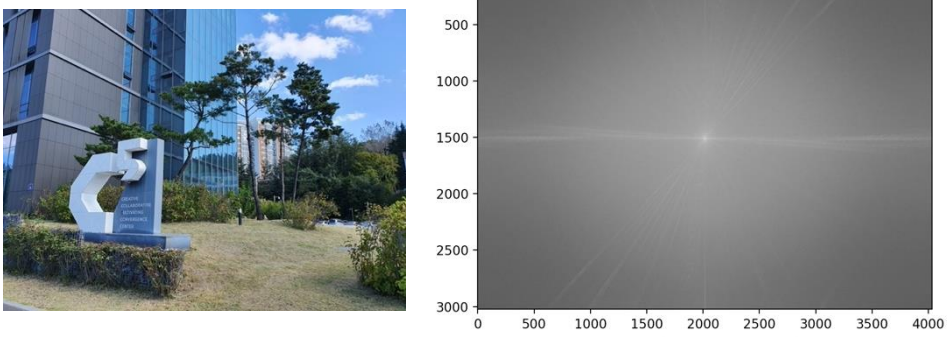
blurrier image. In the case of Gaussian high pass filter, we can think of a situation opposite to Gaussian low pass filter.



Gaussian filter FFT (size = 100)(L) / Gaussian filter FFT (size = 200)(R)

After creating the Gaussian filter, Fourier transform should be performed. This algorithm is provided as a function called fft.fft2() in numpy library. The direction of the white line shown in the result of the FFT is made by the strong edge in the vertical direction of the original image.

```
def calc_spectrum(src):
    f = np.fft.fft2(src)
    spectrum = np.log(np.abs(f))
    dst = np.fft.fftshift(spectrum)

    return dst

input_image_gray = cv2.imread(path + input_image_name, cv2.IMREAD_GRAYSCALE).astype(np.float32) / 255.

plt.imshow(calc_spectrum(input_image_gray), cmap = 'gray')
plt.show()
```



Original input image(L) / FFT result of input image(R)

So, in order to perform unsharp masking in the frequency domain, the frequency domain proceeded for each channel as it did in the spatial domain. Both the image and kernel proceeded with the Fourier transform and at the end, the inverse Fourier transform was conducted to resend the result to the spatial domain.

```
src_clone = src.copy()
    src_clone = src_clone.astype(np.uint32)

    hs = 100 # Filter size
    flt = gauss2d((hs * 2 + 1, hs * 2 + 1), hs / 6.) # 2D Gaussian filter

    dst = np.empty(src_clone.shape)

    # Consider color channel
    for color in range(3):
        image_f = np.fft.fft2(src_clone[:, :, color]) # Image FFT
        flt_f = psf2otf(flt, src_clone[:, :, color].shape) # Filter FFT

        spectrum_ = np.log(np.abs(flt_f))
        flt_f_ = np.fft.fftshift(spectrum_)
        #plt.imshow(flt_f_, cmap = 'gray')
        #plt.show()

        ## Unsharp masking
        image_low_f = image_f * flt_f
        image_high_f = image_f - image_low_f

        alpha = 0.5

        image_n_f = image_f + alpha * image_high_f
        image_n = np.real(np.fft.ifft2(image_n_f))

        image_n[image_n < 0.] = 0
        image_n[image_n > 255.] = 255

        image_n = image_n.astype(np.uint8)
        dst[:, :, color] = image_n
```

As the size of Gaussian filter increases, the input image becomes blurred, and more details are included if negative operations are performed on the existing input image and the image to which the filter is applied. Therefore, when a larger filter is used, the result becomes sharper.

Unsharp masking in frequency domain(size = 10, $\alpha = 0.5$)(L) /
Unsharp masking in frequency domain(size = 200, $\alpha = 0.5$)(R)

Finally, I compared the results according to the change in α values. The α value is a parameter to control strength of sharpening. So, when the α value is large, the sharpening effect appears more effectively.

Unsharp masking in frequency domain(size = 100, $\alpha = 0.1$)(L) /
Unsharp masking in frequency domain(size = 100, $\alpha = 1.0$)(R)

3. Edge-aware filtering (Bilateral Filtering)

Low pass filters such as Gaussian filters are often used for removing high-frequency details such as small-scale textures and noise from an image. However, such filters blur entire images, so other structural edges get blurred too. Often, we want to blur out small-scale details and noise while keeping important edges sharp. To this end, many edge- aware filtering methods have been proposed. Bilateral filtering is one of the most popular methods among them.

As much noise as possible from the original image should be removed and the edge should be preserved. Usually, noise is mostly a high frequency component. In the end, our goal is to eliminate these high frequency components. Originally, a low pass filter was mainly used to remove such high frequency components. However, it is not satisfactory because just using a low pass filter results in blurring the entire image. This is because the edge was not preserved even though noise was removed. In the case of a Gaussian filter, edge is not considered. So, using an edge-ware filter instead of a Gaussian filter can prevent the overall smoothness because it recognizes the edge. Bilateral filter is defined as follow.

$$h[m,n] = \frac{1}{W_{mn}} \sum_{k,l} g[k,l] r_{mn}[k,l] f[m+k, n+l]$$

Normalization factor　　Spatial weighting　　Intensity range weighting

If we look at the definition of the bilateral filter, you can see that there are two different weighting functions. One is spatial weighting function, and another is intensity range weighting

function. These two functions are Gaussian functions, and each function is controlled by σ. This value determines the size of the Gaussian function. And bilateral filter is not implemented in frequency domain because it does not use the convolution operation as it is. So, it should be implemented in the Spatial domain. The following functions were created to create a bilateral filter. Each played an important role in the bilateral filter.

```
# Compute spatial weighting function g
def gaussian_spatial_weighting(src, m, n, k ,l, sigma_s):
    r = src[k, l] - src[m, n]
    g = np.exp(- (r ** 2) / (2 * (sigma_s ** 2)))

    return g

# Compute intensity range weighting function r
def gaussian_intensity_range_weighting(m, n, k, l, sigma_r):
    euclidean_dist = np.sqrt((m - k) ** 2 + (n - l) ** 2)
    r_mn = np.exp(- (euclidean_dist ** 2) / (2 * (sigma_r ** 2)))

    return r_mn

# Compute normalization factor W
def normalization_factor(g, r_mn):
    w = g * r_mn

    return w
```

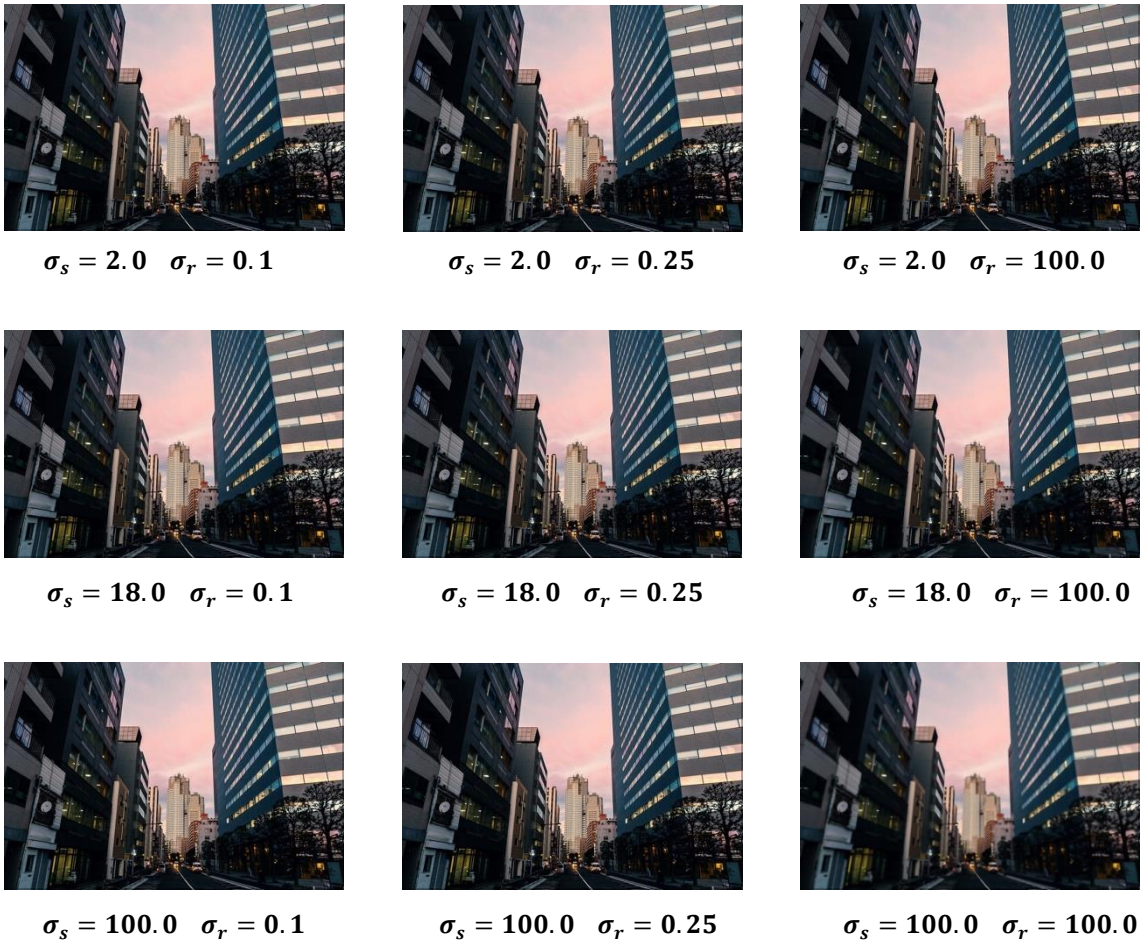
And using this, I made a function that applies a bilateral filter to the original image. Parameters include input images, kernel sizes, and sigma values.

```
# Bilateral filtering process
def bilateral_filtering(src, kernel_size, sigma_s, sigma_r):
    b, g, r = cv2.split(src)

    dst_blue = np.zeros(b.shape)
    dst_green = np.zeros(g.shape)
    dst_red = np.zeros(r.shape)

    height = src.shape[0]
    width = src.shape[1]

    half_kernel_size = kernel_size // 2

    left_x = half_kernel_size
    right_x = width - half_kernel_size
    up_y = half_kernel_size
    bottom_y = height - half_kernel_size

    # Inside of image
    for m in range(up_y, bottom_y):
        for n in range(left_x, right_x):
            W_blue = 0
            W_green = 0
            W_red = 0

            kernel_left_x = n - half_kernel_size
            kernel_right_x = n + half_kernel_size + 1
            kernel_up_y = m - half_kernel_size
            kernel_down_y = m + half_kernel_size + 1

            # Kernel filtering: g[k,l]r_mn[k,l]f[m+k,n+l]
            for k in range(kernel_up_y, kernel_down_y):
```

```
                for l in range(kernel_left_x, kernel_right_x):
                    g_blue = gaussian_spatial_weighting(b, m, n, k, l, sigma_s)
                    g_green = gaussian_spatial_weighting(g, m, n, k, l, sigma_s)
                    g_red = gaussian_spatial_weighting(r, m, n, k, l, sigma_s)

                    r_blue = gaussian_intensity_range_weighting(m, n, k, l, sigma_r)
                    r_green = gaussian_intensity_range_weighting(m, n, k, l, sigma_r)
                    r_red = gaussian_intensity_range_weighting(m, n, k, l, sigma_r)

                    w_blue = normalization_factor(g_blue, r_blue)
                    w_green = normalization_factor(g_green, r_green)
                    w_red = normalization_factor(g_red, r_red)

                    W_blue = W_blue + w_blue
                    W_green = W_green + w_green
                    W_red = W_red + w_red

                    dst_blue[m, n] = dst_blue[m, n] + b[k, l] * w_blue
                    dst_green[m, n] = dst_green[m, n] + g[k, l] * w_green
                    dst_red[m, n] = dst_red[m, n] + r[k, l] * w_red

            # Normalization: 1/W_mn
            dst_blue[m, n] = int(round(dst_blue[m, n] / W_blue))
            dst_green[m, n] = int(round(dst_green[m, n] / W_green))
            dst_red[m, n] = int(round(dst_red[m, n] / W_red))

    # Outside of image
    for m in range(0, up_y):
        for n in range(0, width):
            dst_blue[m, n] = b[m, n]
            dst_green[m, n] = g[m, n]
            dst_red[m, n] = r[m, n]

    for m in range(bottom_y + 1, height):
        for n in range(0, width):
            dst_blue[m, n] = b[m, n]
            dst_green[m, n] = g[m, n]
            dst_red[m, n] = r[m, n]

    for m in range(0, height):
        for n in range(0, left_x):
            dst_blue[m, n] = b[m, n]
            dst_green[m, n] = g[m, n]
            dst_red[m, n] = r[m, n]

    for m in range(0, height):
        for n in range(right_x + 1, width):
            dst_blue[m, n] = b[m, n]
            dst_green[m, n] = g[m, n]
            dst_red[m, n] = r[m, n]

    # Create bilateral filtered image
    dst = np.zeros(src.shape)
    dst[:, :, 0] = dst_blue
    dst[:, :, 1] = dst_green
    dst[:, :, 2] = dst_red

    dst = dst.astype(np.uint8)

    return dst
```

```
path = "../images/"
input_image_name = sys.argv[1]
input_image = cv2.imread(path + input_image_name, cv2.IMREAD_COLOR).astype(np.float32)
output_image = bilateral_filtering(input_image, 5, 100.0, 0.1)
```

To consider the color image and prevent overflow problems, the data type was changed from uint8 to float type and separated into three channels b, g, and r to filter each. σ_s can control the size of spatial weighting function and σ_r can control the size of intensity range weighting. The small size of σ_r means that the size of the circle of Gaussian kernel is small, which means that they want to blend using intensity values that differ very little from each other. Conversely, the large size of σ_r means that the circle size of the Gaussian kernel is large, which means that it will give a large weight even if the intensity is very different from each other. If σ_r becomes to infinity, we will give constant weight to all intensity values. This is the same as the meaning of not using intensity range weighting function anymore. So, increasing the value of σ_r can eliminate edges with a large difference in intensity values. On the other hand, increasing the value of σ_s widens the range of spatial weight, effectively removing small details. In the process of executing my code, the problem of bilateral filtering was revealed. Time complexity was not good due to the large amount of computation. So I used low-resolution images instead of high-resolution images taken with my cell phone.



$\boldsymbol{\sigma_s = 2.0 \quad \sigma_r = 0.1}$ $\boldsymbol{\sigma_s = 2.0 \quad \sigma_r = 0.25}$ $\boldsymbol{\sigma_s = 2.0 \quad \sigma_r = 100.0}$

$\boldsymbol{\sigma_s = 18.0 \quad \sigma_r = 0.1}$ $\boldsymbol{\sigma_s = 18.0 \quad \sigma_r = 0.25}$ $\boldsymbol{\sigma_s = 18.0 \quad \sigma_r = 100.0}$

$\boldsymbol{\sigma_s = 100.0 \quad \sigma_r = 0.1}$ $\boldsymbol{\sigma_s = 100.0 \quad \sigma_r = 0.25}$ $\boldsymbol{\sigma_s = 100.0 \quad \sigma_r = 100.0}$

To take a closer look at the changes, we will expand and compare the following two cases.

$\sigma_s = 2.0 \quad \sigma_r = 0.1$

$\sigma_s = 100.0 \quad \sigma_r = 100.0$

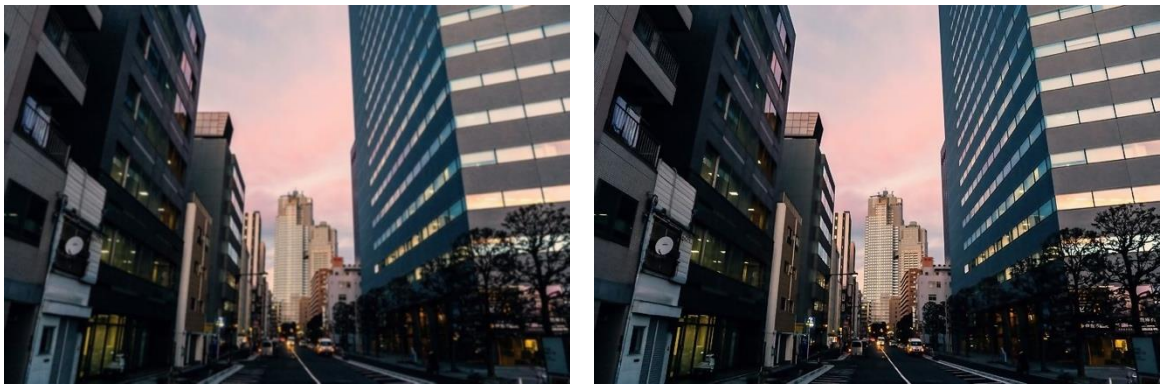
And I compared the function I made with the bilateral filter function provided by OpenCV with the sigma value of 50.

```
output_image = bilateral_filtering(input_image, 5, 50.0, 50.0)
output_image = cv2.bilateralFilter(input_image, 5, 50.0, 50.0)
```

OpenCV bilater filtered image(L) / My own bilater filtered image(R)

Now, we will compare the results using Gaussian filter and bilateral filter. The closer the Gaussian filter is to the current pixel, the greater the weight, and the farther it is, the smaller the weight. Gaussian filters were created based on the fact that nearby pixels have similar values because images change slowly spatially. Therefore, since the noise value is relatively small in correlation with the neighboring pixel values, it may be mitigated by a weighted average of the neighboring pixel values. However, there are disadvantages in areas that change rapidly spatially, such as edges. The difference between the distance between the current pixel and the neighboring pixel and the pixel value is simultaneously reflected in the weight. It can be thought of as a more advanced version of the Gaussian filter that reflects only the distance between pixels in the weight.

```
output_image = bilateral_filtering(input_image, 5, 50.0, 50.0)
output_image = cv2.GaussianBlur(input_image, (5, 5), 1)
```



OpenCV Gaussian filtering(L) / My own bilateral filtering(R)

The results of Gaussian filtering were more affected by the sigma value and kernel size. Therefore, the edge was not easily preserved in the results of Gaussian filtering. It seems that it is more difficult to preserve the edge than the bilateral filter that adds weight that considers the distance between pixels.