Homework 2

Frequency Domain Processing

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. We now read the images with pixel values from 0 to 1.

2 Ideal Lowpass Filter

2.1 Explanation of implementation

After the image is read as a color image, first a border extension is implemented to avoid color bleeding across the images. Therefore cv2.BORDER_REFLECT is used because it creates a natural extension of the image which minimizes the effect of color bleeding. Other methods like cv2.BORDER_CONSTANT and cv2.BORDER_WRAP could create unwanted sharp turns which enhance color bleeding. This is compared later in 6.

```
pad_size = threshold
image = cv2.copyMakeBorder(
    image, pad_size, pad_size, pad_size, pad_size, cv2.
       BORDER_REFLECT
)
```

For the actual filtering, the image split in its channel and later combined back together. The following operations can now be regarded for every channel equivalently. For filtering via the frequency domain, a mask approach is used. We know, the lowest frequencies are in the center of the image in the frequency domain. With the ideal lowpass filter, we do a hard cutoff of frequency. Therefore we create a mask which is 1 in a defined radius to the mask center and 0 everywhere else. In the code, this is done by creating a mask with the size of the image (after boundary extension), initialized with ones. We iterate now over all pixels. If the euclidean distance to the center is bigger than the defined radius, we set this pixel to 0.

```
mask = np.ones(image.shape[:2], dtype=np.uint8)
for i in range(mask.shape[0]):
    for j in range(mask.shape[1]):
        if (
            np.sqrt((i - mask.shape[0] // 2) ** 2 + (j - mask.shape
               [1] // 2) ** 2)
            > threshold_value
        ):
            mask[i, j] = 0
```

Now we can transform each channel via a 2d-fast-fourier-transformation into the frequency domain. After that, it is required to do a shift operation to move the low-frequency information to the center of the image. Since the image is first assumed to be extend periodically the low-frequency information are scattered throughout the image.

The image in the frequency domain can now be multiplied with the mask. This results in an elimination of all high frequencies. Now we can shift the masked channel back to remove frequencies from the center and do a reverse fourier transformation. From this transformation we only take the real part. This is necessary since the Inverse Fourier Transformation produces complex values to represent perodic waves.

Since the imaginary part should ideally be zero, we can just take the real part. With that we now have the ideally filtered image.

```
# fourier transform
channel_f = np.fft.fft2(channel)
channel_f_shifted = np.fft.fftshift(channel_f)

# apply ideal lowpass filter
channel_f_shifted_lowpass = channel_f_shifted * mask

# inverse fourier transform
channel_f_lowpass = np.fft.ifftshift(
    channel_f_shifted_lowpass
)  # reverse the shift
channel_lowpass = np.real(
    np.fft.ifft2(channel_f_lowpass)
)  # reverse the fourier transform
```

The results looks like 1.

Figure 1: Bridge Image Ideal Filter Threshold 50

Before we look at the results in detail with the intermediate steps - lets take a closer look at the code for visualising the channels from the frequency space.

We cannot directly visualise the frequency domain - we also have to deal with the complex numbers for example by taking the absolute and than add 1 to deny a division by zero in the next step. This step is now taking to logarithm. After that it is just required to normalize the values between 0 and 1 to visualise it.

```
channels_f.append(np.fft.fftshift(np.log(np.abs(channel_f) + 1)))

# merge channels to image
image_f = cv2.normalize(image_f, None, 0, 1, cv2.NORM_MINMAX)
```

2.2 Visualisation

In the image 2 and 3, we see all important parts of the image:

- Original Image in Spatial domain
- Original Image after Fourier Transformation and Shift
- Image after Ideal Lowpass Filter in Spatial Domain
- Image after Ideal Lowpass Filter in Frequency Domain

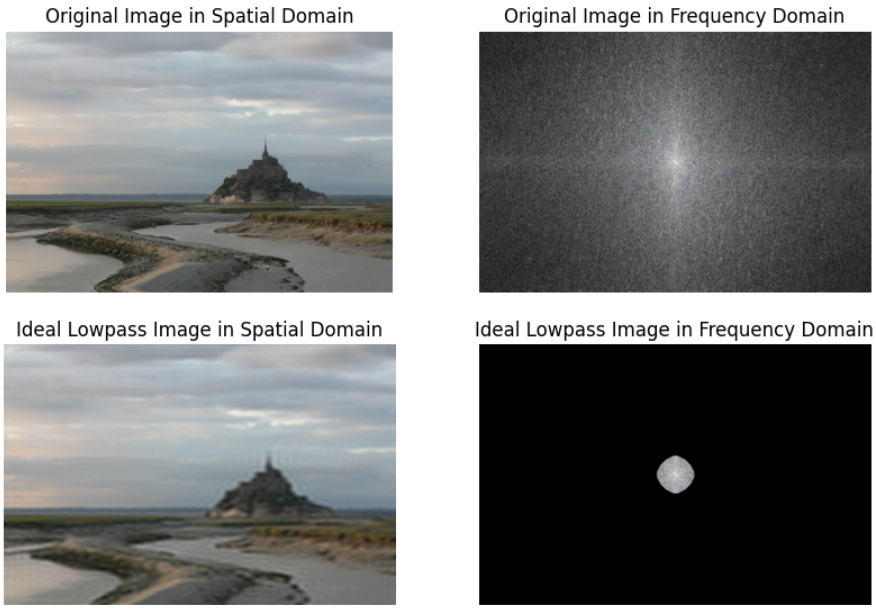

Figure 2: Lecture Image Ideal Filter Threshold 50

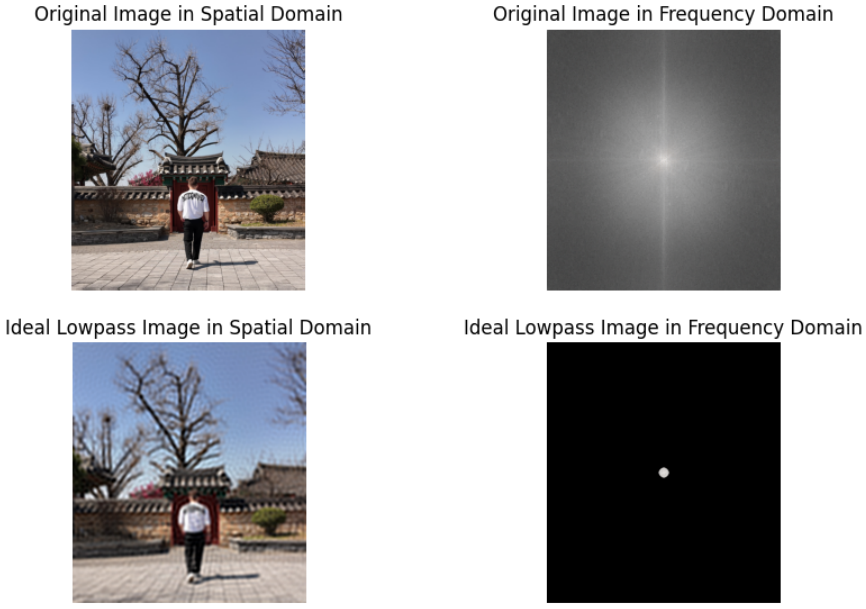

Figure 3: Entrance Image Ideal Filter Threshold 50

2.3 Comparion of different lowpass filter sizes

But how do different lowpass filters affect the result? The bigger the threshold size, the bigger the radius of the circle in the mask. Therefore, we see that we increase the limit of accepted frequencies with rising threshold so it get less unsharped. As 4 and 5 show this.

Figure 4: Lecture Image Ideal Filter Threshold Comparison

Figure 5: Entrance Image Ideal Filter Threshold Comparison

2.4 Discussion of Boundary Handling

With a padding size of at least the threshold size, it prevents operations outside this scope to be zero. Therefore the padding size is set to the threshold size. 6 shows a a comparison with different borders sizes (but instead of ideal lowpass filtering, gaussian filtering is used to mitiage rining effects). The Threshold size used here is 30. Please find attached in 22 and 23 additional examples with different border types.

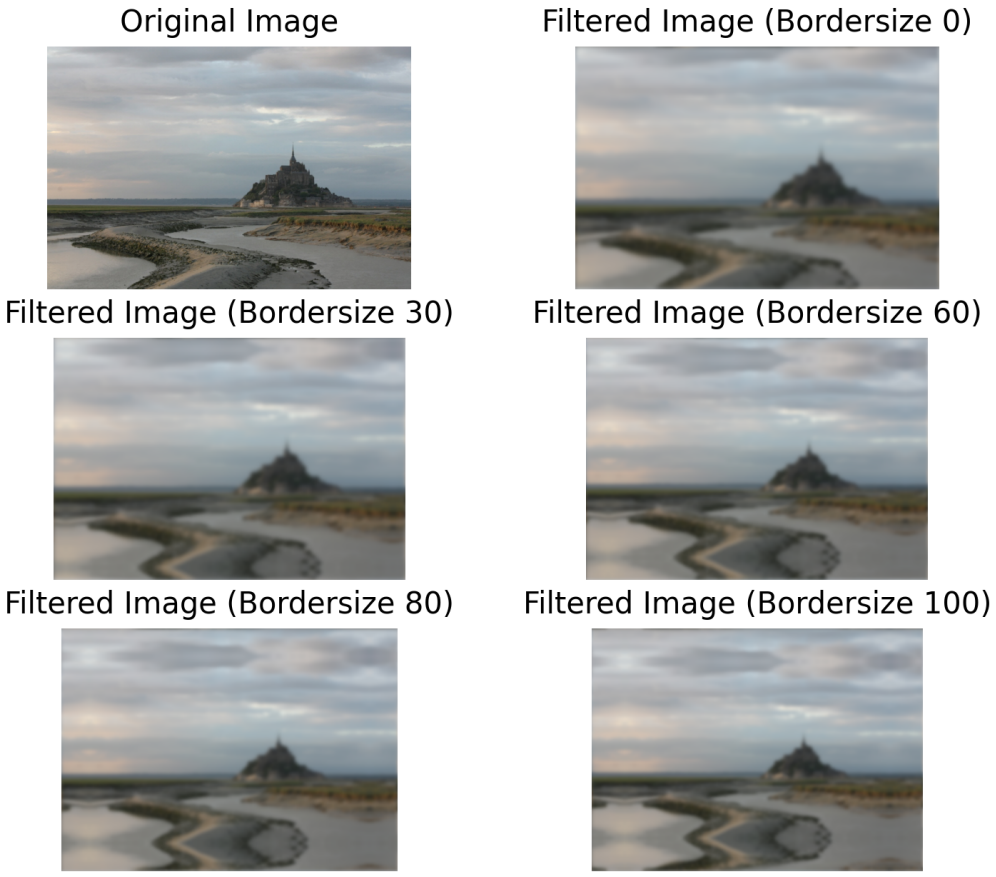

Figure 6: Gaussian Filter Padding Comparison Reflect

3 Gaussian Lowpass Filter

3.1 Explanation of implementation

The implementation of the Gaussian Lowpass Filter is similar to the Ideal Lowpass filter. We also apply a border padding and separate the channel to filter them separately. Lets first create the Gaussian kernel and transform it to the frequency domain. We define the threshold_value as half

the size of the kernel. Then we create the 2d kernel. Therefore we create 2 1-dimensional kernels and multiply them to have a 2d kernel. As kernel size we take the threshold times 2 and add 1 to have to an odd size. Furthermore, we define the sigma as the threshold divided by 6.

```
hs = threshold_value  # half of the filter size
flt = gauss2d((hs * 2 + 1, hs * 2 + 1), hs / 6.0)
```

To apply the kernel to the image in the frequency domain - we have to change its size. Therefore we first apply padding (with 0) so the kernel matches the channels shape. After that, we have to shift it, so it is aligned with the image and compute its fourier transformation.

```
flt_f = psf2otf(flt, channel.shape)
```

The channel follows the same procedure as in the ideal kernel. We transform first the channel to the frequency domain. Since the filter is already in the correct domain, doing convolution in frequency domain jsut means multiplying these two matrices. After that, the result can be transformed back to the spatial domain and then we only the to take the real numbers as with the ideal filter.

```
channel_f = np.fft.fft2(channel)
channel_flt_f = channel_f * flt_f
# inverse fourier transform
channel_filtered = np.real(np.fft.ifft2(channel_flt_f))

cropped = channel_filtered[pad_size:-pad_size, pad_size:-pad_size]
```

3.2 Visualisation

As with the Ideal Lowpass Filter, we can again visualize the intermediate steps. The processing of the image in frequency domain is done as before.

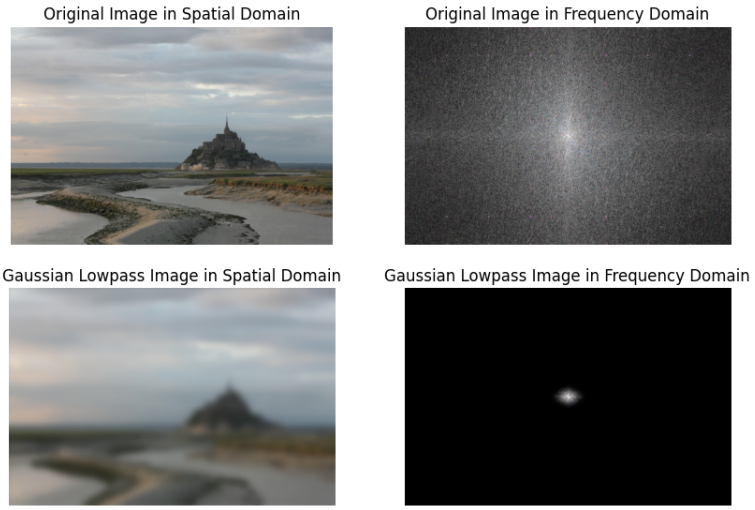

Figure 7: Lecture Image Gaussian Filter Visualisation

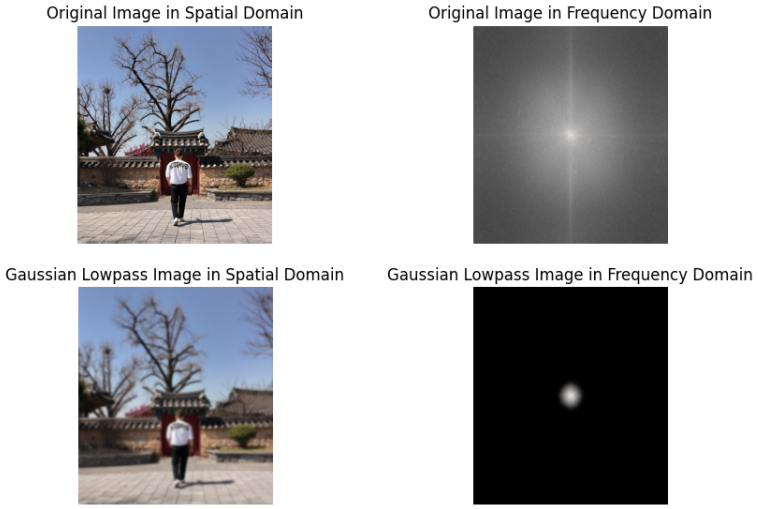

Figure 8: Entrance Image Gaussian Filter Threshold Visualisation

3.3 Comparison of different filter sizes

Lets also compare different filter sizes. The higher the threshold, the bigger the kernel and the blur since the standard deviation increases with increasing threshold. Therefore we cutoff more high frequencies with higher thresholds.

Figure 9: Lecture Image Gaussian Filter Threshold Comparison

3.4 Compare results of ideal and Gaussian Lowpass filter

Now lets compare Ideal and Gaussian Lowpass Filter. Therefore we look at 11 , 10 and 12.

In the images with the ideal lowpass filter we have a clearly visible ringing effect. This is due to the abrupt change in the mask of the ideal lowpass filter in spatial domain. These abrupt changes result in ripples in the mask in frequency domain which lead to ringing. To mitigate these effects, we can change the kernel to gaussian.

We can see the difference if we compare 5 and 2. In the case of the Ideal Filter between black and not black since we see an exact cirlce. In case of the Gaussian Kernel, the circle-borders are more continuous and is not really a circle anymore.

Figure 10: Comparison Ideal vs Gaussian Lowpass Filter Image 1

Figure 11: Comparison Ideal vs Gaussian Lowpass Filter Image 2

Figure 12: Comparison Ideal vs Gaussian Lowpass Filter Image 0

4 Unsharp masking using the convolution theorem

4.1 Explanation of code implementation

The core idea behind increasing the sharpness via unsharping is separated into multiple steps. A highpass filter can be created by subtracting a lowpass-filtered version of the original image from the original image. If the resulting highfrequency component image is now added to the original image, the image looks sharpened since the contrast at edges (high-frequency) is enhanced. By multiplying the high-pass components with a factor, and adapting the previous lowpassfilter, the sharpening can be configured.

For both spatial and frequency domain the image first split into its channels, then the unsharp masking is done and after that, the channels are merged back together.

4.1.1 Implementation Unsharp Masking Spatial Domain

The unsharp masking in the Spatial Domain is straight forward:

We create a Gaussian Kernel in the first step. Then we apply convolution in the spatial domain with the kernel and the image. (It was not stated in the task that opencv filter2D for convolution cannot be used.) After that, to the original image is added the product of alpha (factor for sharpening intensity) times the original image minus the lowpassed image to gain the final result.

```
kernel = gauss2d((kernel_size, kernel_size), kernel_sigma)
GvonvX = cv2.filter2D(image, -1, kernel)
result = image + alpha * (image - GvonvX)
```

4.1.2 Implementation Unsharp Masking Frequency Domain with discussion of FFT results at each step

For the frequency domain it is a little bit different. We first have to transform the image in frequency domain, for the kernel too, but this one has to be shifted also.

The formula of the unsharp masking in the spatial domain is: $y = x + \alpha \cdot (x - g * x)$. If we now transform all variables in the frequency domain, we can use the property of *Linearity* and the *Convolution Theorem*. From Linearty, addition and multiplication with scalars (also negative) stays the same.

With the Convolution Theorem, the convolution turns into a multiplication in frequency domain. Our new formula in is hence: $Y = X + \alpha \cdot (X - G \cdot X)$. After applying the formula in code, we have to to the inverse transformation to get the result in the spatial domain, which we then can visualize again.

```
kernel = gauss2d((kernel_size, kernel_size), sigma=kernel_sigma)
image_f = np.fft.fft2(image) # convert image to frequency domain
kernel_f = psf2otf(kernel, image.shape) # convert to frequency
    domain

GvonvX_f = image_f * kernel_f # apply kernel to image
result_f = image_f + alpha * (image_f - GvonvX_f)
```

```
result = np.real(np.fft.ifft2(result_f)) # convert to spatial domain
```

4.2 Visualisation input images, parameters and corresponding output

Lets look at some examples. 13, 14 and 15. We see easily, that with the unsharp masking the image now looks much sharper since the contrast at the edges is much higher. Furthermore we see that due to the convolution theorem there is no difference whether we Filter in the Spatial or the Frequency Domain.

Figure 13: Comparison Spatial and Frequency Unsharp Masking Image Entrance

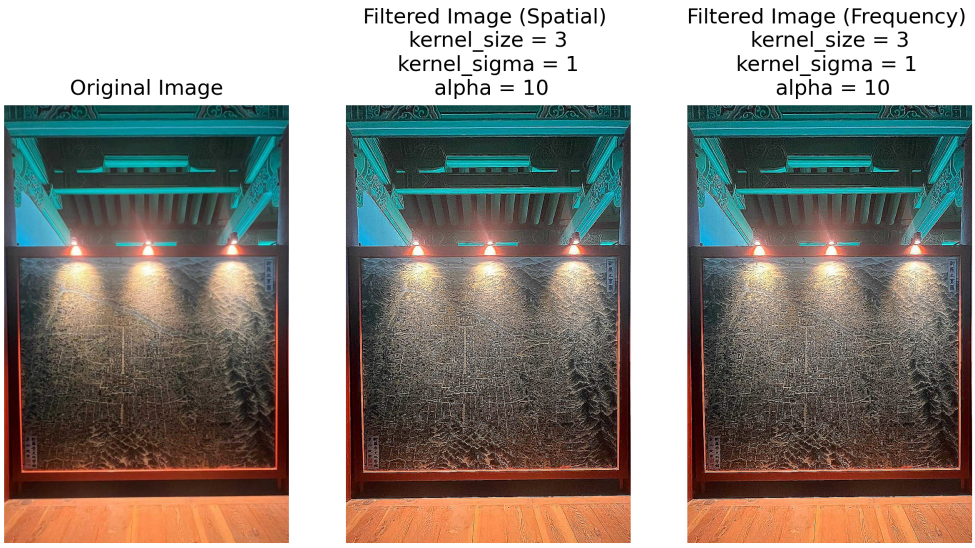

Figure 14: Comparison Spatial and Frequency Unsharp Masking Image Map

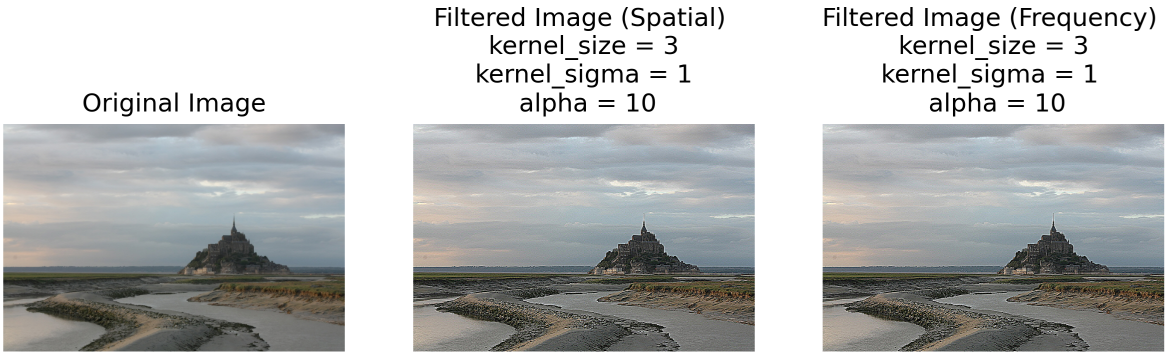

Figure 15: Comparison Spatial and Frequency Unsharp Masking Image Lecture

4.3 Discussion on the effects of parameters

First we analyse the parameters theoretically and then we look at some examples.

- Alpha: Determinies how much high-frequency detail is added. With small alpha, we get a result close to the original image, with a higher alpha we get more exzessive values.
- Sigma: Determines how much the Gaussian kernel spread. So with a small sigma, we only emphasize small details, with a high sigma, we produce more global sharpening.
- Kernel-Size: Determines the spatial extent of the Gaussian Filter. With a bigger kernel, we get more blurring which results in a stronger sharpening effect and vice versa.

Lets look at this in more detail. In 16

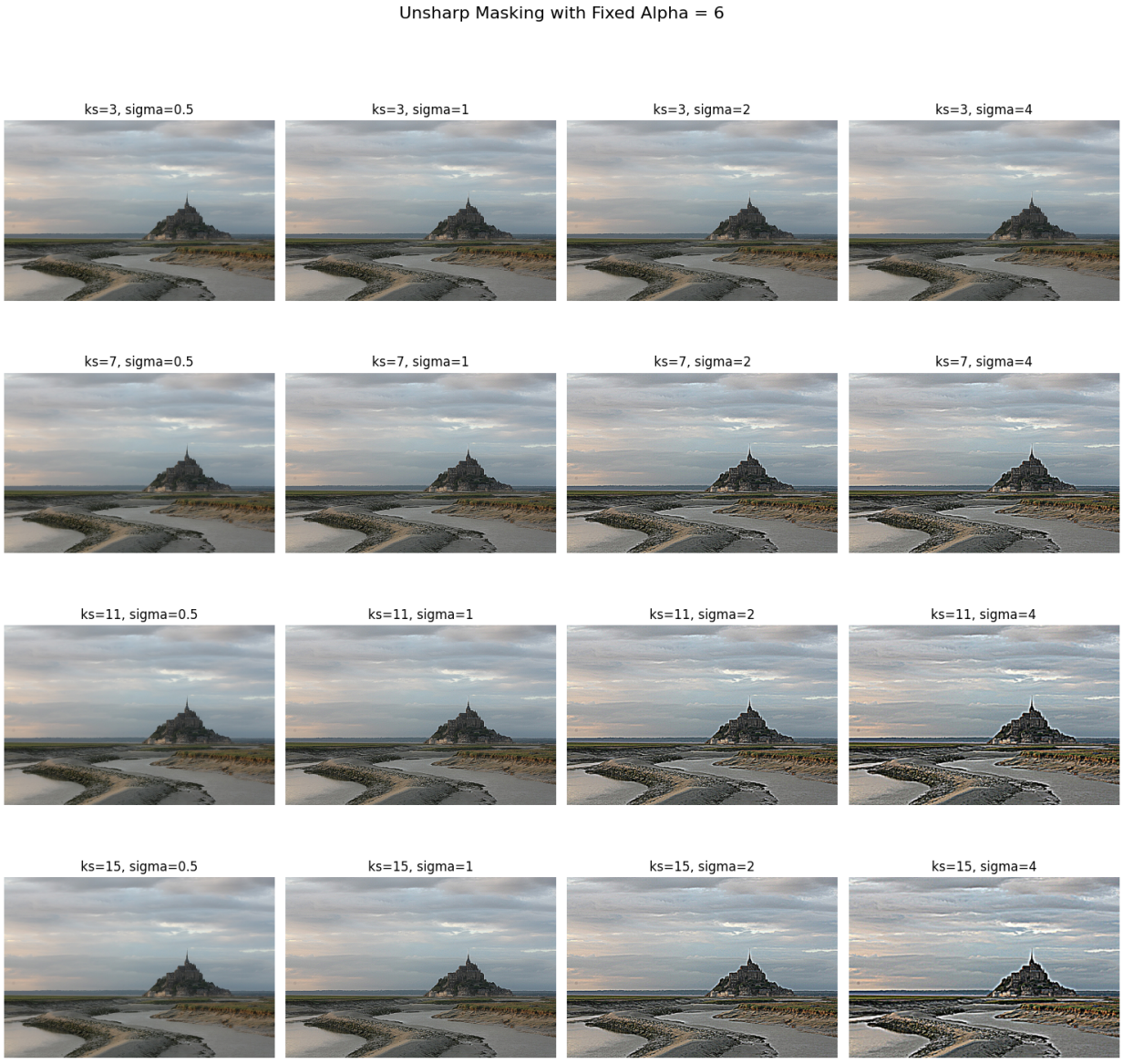

Figure 16: Compare Different Parameters - fixed alpha Image

In 17 we see clearly that an oversharpening happens for a combination of big alphas and a large kernel_size where the kernel size create more blur leading to big sharpened regions.

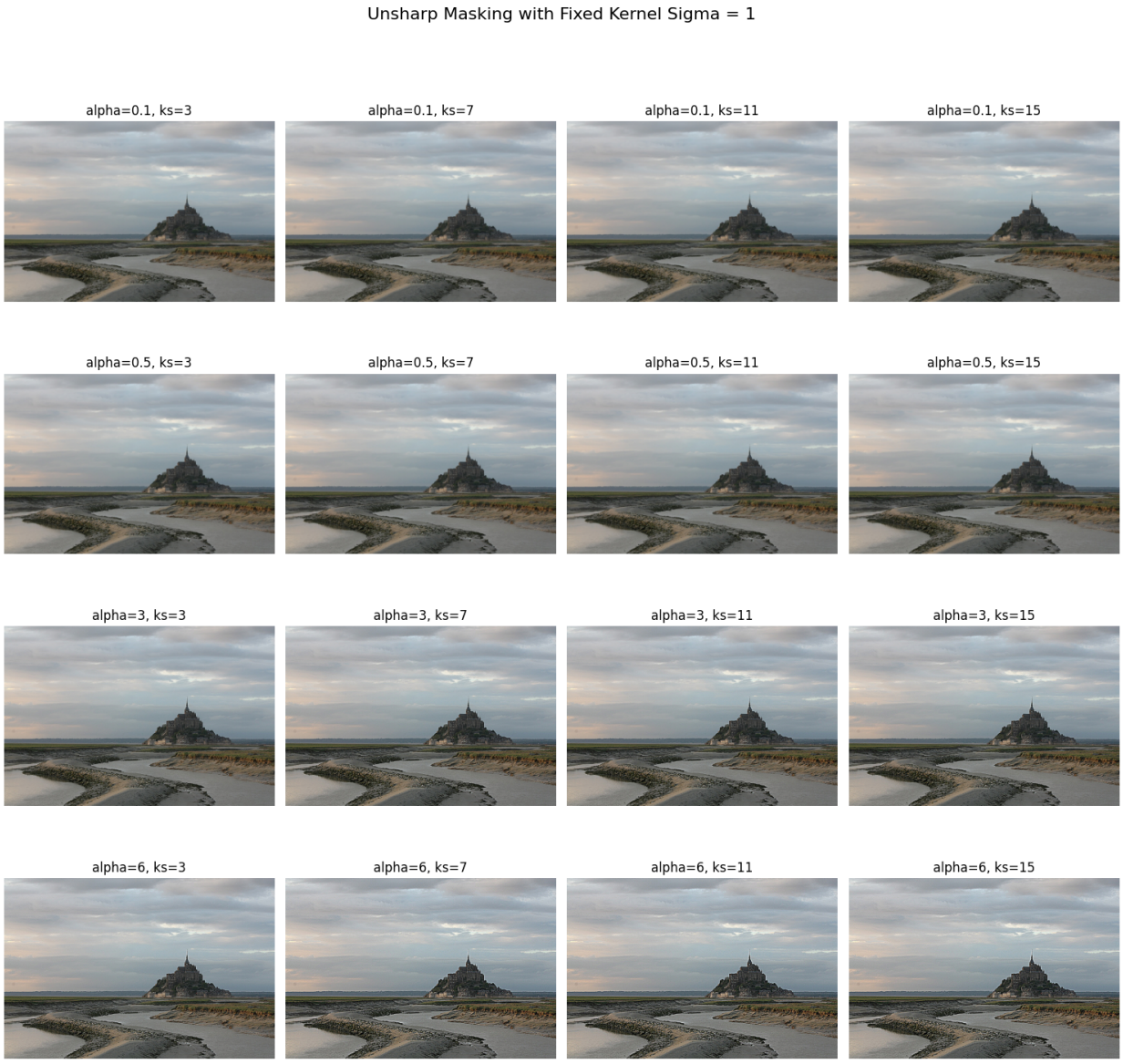

Figure 17: Compare Different Parameters - fixed sigma

In 18 we also see oversharpening due to high alpha. Interesting is hereby the influence of increasing sigma, which leads to global sharpening.

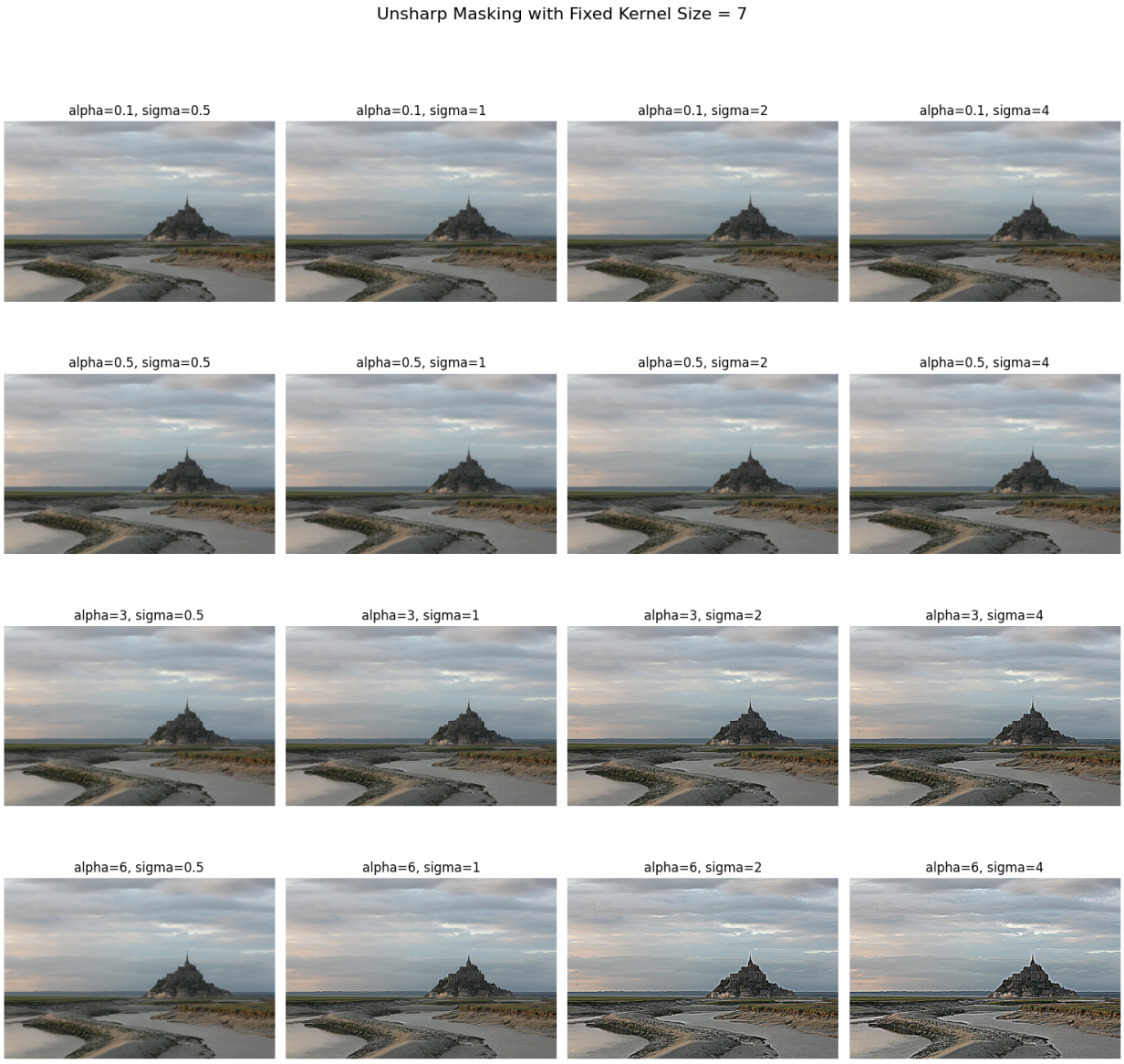

Figure 18: Compare Different Parameters - fixed kernel size

4.4 Comparion between spatial-domain and Frequency domain

Due to the mathematical equivalence of frequency-domain and spatial-domain implementation the both yield the same result.

```
def test_frequency_vs_domain():
    # Test the frequency_vs_domain function with a sample image and
       threshold value
    image = np.random.rand(1000, 1000, 3)  # Sample random image
    alpha = 1.5  # Sample alpha value for unsharp masking
    kernel_size = 5  # Sample kernel size for unsharp masking
    kernel_sigma = 1.0  # Sample kernel sigma for unsharp masking

    image_domain = unsharp_masking(
        image=image,
        domain="spatial",
        alpha=alpha,
        kernel_size=kernel_size,
        kernel_sigma=kernel_sigma,
    )

    image_frequency = unsharp_masking(
        image=image,
        domain="frequency",
        alpha=alpha,
        kernel_size=kernel_size,
        kernel_sigma=kernel_sigma,
    )

    assert np.allclose(image_domain, image_frequency, atol=1e-5), (
        "The␣images␣in␣frequency␣and␣spatial␣domain␣do␣not␣match!"
    )
```

4.4.1 Runtime Comparison

For the runtime comparison we use two images with different sizes to eliminiate this factor in our measurements. In 19, 20 and 21 we see that the big image (1728x1935) takes almost 5 times longer to run in the frequency domain than the small image (800 x 533) in the frequency domain. This is due to the fourier transformation which takes a long time. Unsurprisingly, the alpha value has no influence on the runtime, since it is only used in a scalar multiplication. Same for kernel sigma, which only changes the values of the kernel. As we see in 21, this parameter has an effect on the runtime, but only in the spatial domain. This is due to the convolution in spatial domain, which scales with the kernel size. But in the frequency domain, this makes no difference since we just transform the kernel into a mask, so there is no runtime change.

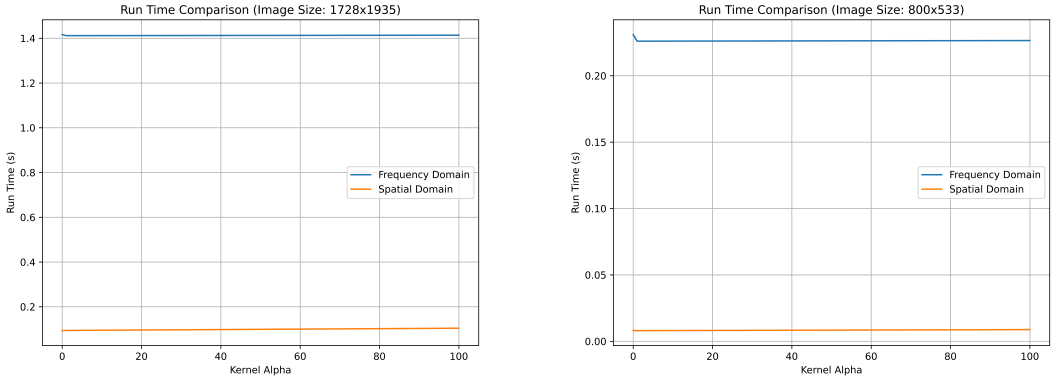

Figure 19: Comparison of images with different alpha values.

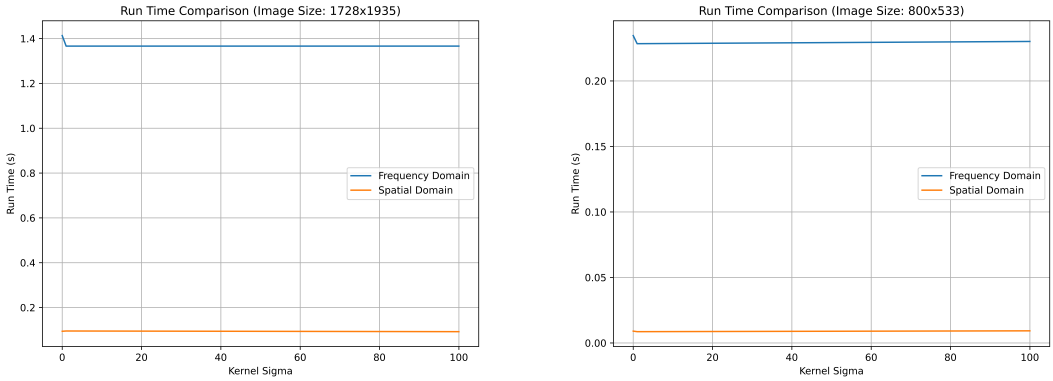

Figure 20: Comparison of images with different kernel sigmas.

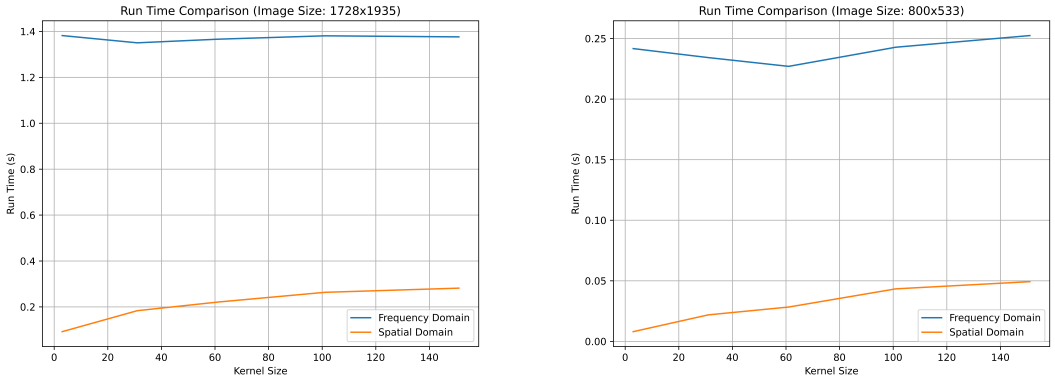

Figure 21: Comparison of images with different kernel sizes.

5 Images Attached

5.1 Border Comparison

Figure 22: Gaussian Filter Padding Comparison Constant Black Threshold 30

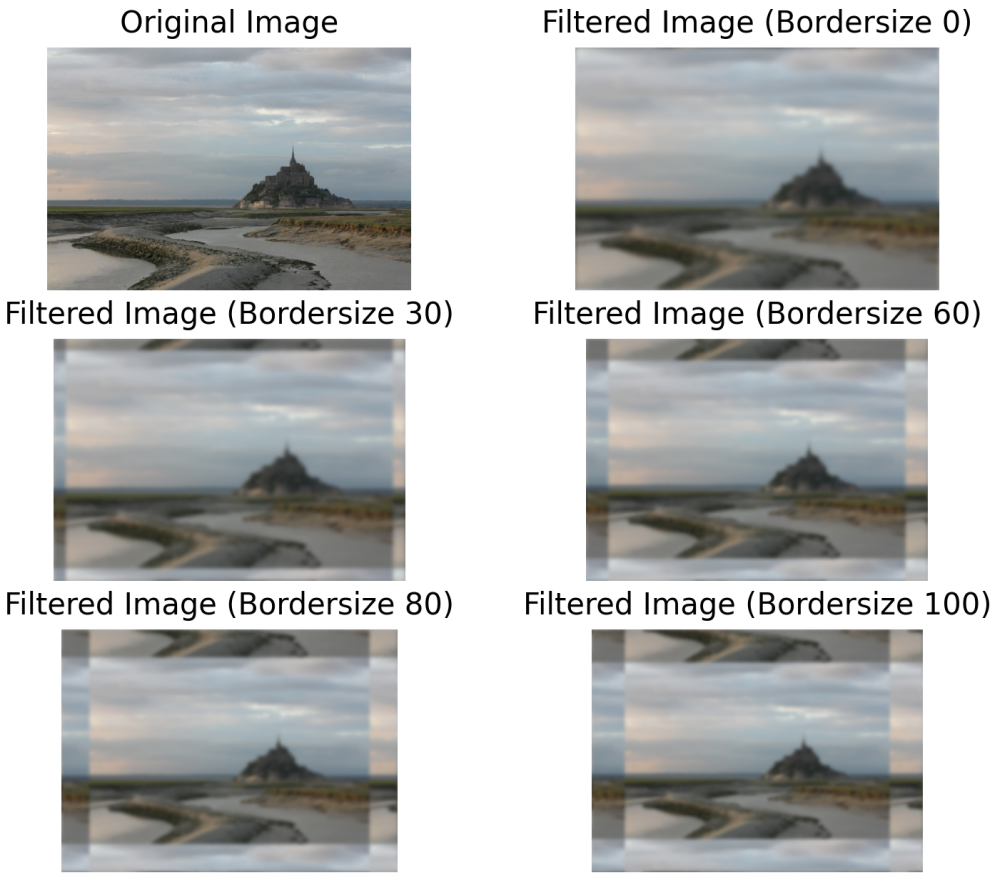

Figure 23: Gaussian Filter Padding Comparison Wrap, Threshold 30