

# Homework 2.

컴퓨터공학과 20190539 원지윤

## 1. Ideal Lowpass Filter

### 1-1. Implementation

```
def idealLowpassFiltering(image, filter_size, radius, border_type):
    img = cv2.imread('images/input/' + image, cv2.IMREAD_COLOR).astype(np.float32) / 255.
    padding = math.floor(filter_size / 2)
    expanded_image = cv2.copyMakeBorder(img, padding, padding, padding, padding, border_type)
    b, g, r = cv2.split(expanded_image)

    flt = ilf(filter_size, radius)
    flt_f = psf2otf(flt, (expanded_image.shape[0], expanded_image.shape[1]))

    results = []

    for c in [b, g, r]:
        img_f = np.fft.fft2(c)
        img_flt_f = flt_f * img_f
        img_flt = np.real(np.fft.ifft2(img_flt_f))
        img_flt = cv2.normalize(img_flt, None, np.min(c) * 255, np.max(c) * 255, cv2.NORM_MINMAX, -1)
        results.append(img_flt)

    result_image = cv2.merge((results[0], results[1], results[2]))
    output = result_image[padding:-padding, padding:-padding, :]
    cv2.imwrite('images/output/ilf/' + image.split('.')[0] + '_%d_%d_result.jpg' % (filter_size, radius, border_type), output)
    return output
```

#### 1) Input image

- RGB 이미지 데이터를 `cv2.imread()`로 읽어온다. 이때, 컬러 이미지이므로 옵션을 `cv2.IMREAD_COLOR`로 설정하였다.

#### 2) Preprocessing

- boundary handling을 위해, `filter_size`와 `border_type`에 따라 `cv2.copyMakeBorder()`를 이용해 경계를 추가하고 해당 이미지를 사용해 프로세싱을 진행하였다.
- `cv2.split()` 함수를 이용해 r,g,b 채널을 분리하고 이후 각 채널별로 연산을 수행할 수 있게 했다.

#### 3) Fourier Transform(Ideal Lowpass Filter)

- 모든 채널에 동일한 필터를 적용하기 때문에 먼저 Ideal Lowpass Filter를 계산하고, Fourier Transform을 해주었다.
- Ideal Lowpass Filter**

```
def ilf(filter_size, radius):
    # Distance from (u,v) to the center of the mask
    center = filter_size / 2
    filter = np.zeros((filter_size, filter_size))
    for u in range(filter_size):
        for v in range(filter_size):
            D = math.sqrt(pow((u-center), 2) + pow((v-center), 2))
            filter[u][v] = 0 if D > radius else 1.0
    return filter
```

- Ideal Lowpass Filter 정의에 따라, filter의 중심으로부터 떨어진 거리를 계산해 값이 `radius`보다 크면 1(흰색), 작거나 같으면 0(검정색)으로 설정하여 numpy array 형태의 filter를 반환하였다.

#### • Shift & FFT

```
def psf2otf(flt, img_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(img_shape, dtype=flt.dtype)
    # Shift the center to the corner
    flt_padded[:flt_bottom_half, :flt_right_half] = flt[flt_top_half:, flt_left_half:]
    flt_padded[:flt_bottom_half, img_shape[1]-flt_left_half:] = flt[flt_top_half:, :flt_left_half]
```

```

flt_padded[img_shape[0]-flt_top_half:, :flt_right_half] = flt[:flt_top_half, flt_left_half:]
flt_padded[img_shape[0]-flt_top_half:, img_shape[1]-flt_left_half:] = flt[:flt_top_half, :flt_left_half]
# 2D FFT
return np.fft.fft2(flt_padded)

```

- 강의 노트의 코드를 사용하여 구현하였다.
- filter의 크기가 input image의 크기와 같아지도록 만들고, filter의 중심을 가장자리로 이동시킨 뒤(shift), Fourier Transform 을 해주었다.
- shift?
  - 원래 이미지를 Frequency domain으로 transform 했을 때, low frequency 영역이 가장자리 부분에 위치하고 high frequency 영역이 중앙 부분에 위치한다. 따라서 Ideal lowpass filter의 목적처럼 low frequency 영역만 남기고 싶다면, 이미지의 Fourier transform 결과를 FFT shift하여 low frequency 영역이 중앙에 오게끔 하거나, filter의 중앙 부분이 가장자리로 가게끔 shift 해준 뒤 계산하여야 한다. 여기에서는 후자의 방식을 사용해 구현했기 때문에, filter를 shift해주는 과정이 필요하다.

#### 4) Fourier Transform(Image)

- `np.fft.fft2()` 함수를 이용해 채널 별로 Fourier Transform을 수행하였다.

#### 5) Multiplication in Frequency Domain

- 앞서 계산한 값을 이용해 frequency domain에서 filter의 Fourier Transform 결과와 이미지의 채널별 Fourier Transform 결과를 곱해주었다.

#### 6) Inverse Fourier Transform & Normalization

- `np.fft.ifft2()` 함수를 이용해 채널 별로 Inverse Fourier Transform을 수행하였다.
- 연산 결과를 다시 이미지의 형태로 저장하기 위해 Normalization을 수행했다. 이때, 기존 색상 분포와 결과물의 색상 분포가 크게 달라지는 것을 막기 위해 범위를 해당 채널에서 픽셀 값의 최솟값과 최댓값 사이, 즉  $(\min(c) * 255, \max(c) * 255)$ 로 설정했다.

#### 7) Postprocessing

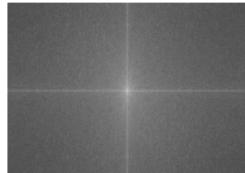
- `cv2.merge()` 함수를 이용해 분리했던 색상 채널들을 다시 합쳐 하나의 이미지로 만들고, expand 했던만큼 다시 가장자리를 crop 해 결과물로 저장했다.

### 1-2. Results

- ideal lowpass filtering results(D=10)



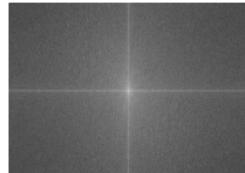
input(color3.jpg)



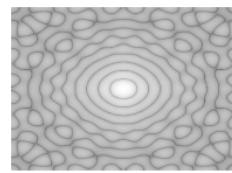
spectrum of an input image(B)



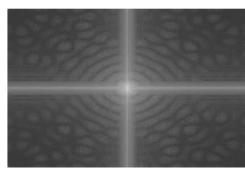
spectrum of an input image(G)



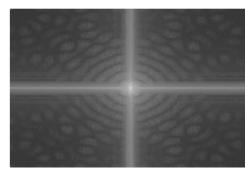
spectrum of an input image(R)



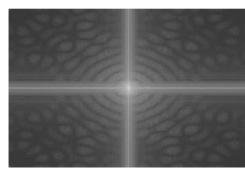
ideal lowpass filter in the frequency domain



filtering result in the frequency domain(B)



filtering result in the frequency domain(G)



filtering result in the frequency domain(R)



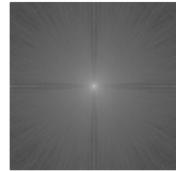
output(color3.jpg)



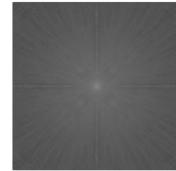
input(shape.jpg)



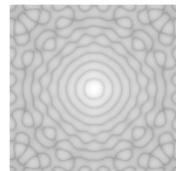
spectrum of an input image(B)



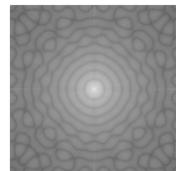
spectrum of an input image(G)



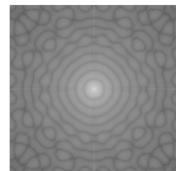
spectrum of an input image(R)



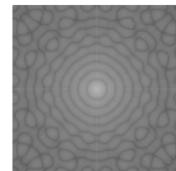
ideal lowpass filter in the frequency domain



filtering result in the frequency domain(B)



filtering result in the frequency domain(G)

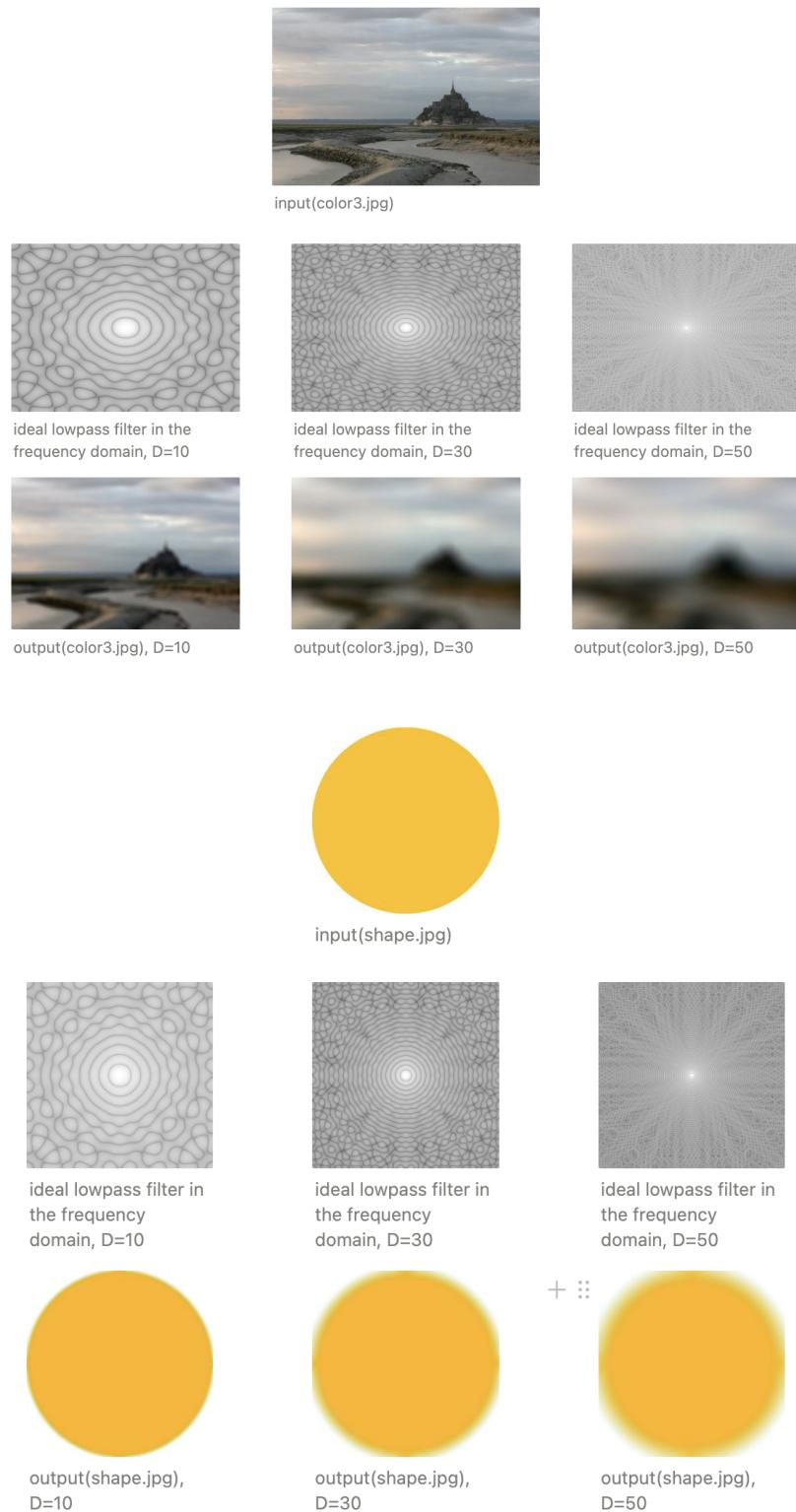


filtering result in the frequency domain(R)



output(shape.jpg)

- **Results of different lowpass filter sizes**



### 1-3. Discussion

- **The effect of the boundary handling**

- Fourier transform은 periodic signal을 가정한다. 따라서 이미지가 아래와 같이 무한히 반복된다고 가정한 상태에서 연산을 진행하게 된다. 이로 인해 Fourier transform이 되면 경계 부분에서 신호가 잘리는 것이 아니라 같은 형태의 신호가 반복되는데, 이때

시작부분과 끝부분의 값이 크게 차이나게 되면 연산 결과가 크게 바뀌면서 결과물이 왜곡되는 현상이 나타난다. 즉, spatial domain으로 다시 변환하였을 때 픽셀 값이 실제 데이터와 크게 차이나면서 경계가 어두워지거나 밝아지는 현상이 나타난다.



- 예시 이미지 color3.jpg를 살펴보면, 기존 이미지의 아랫쪽 중앙부분의 어두운 픽셀들이 결과 이미지의 윗쪽 중앙부분에 영향을 주어 가장자리 부분이 어두워진 것을 확인할 수 있다. 따라서 이를 해결하기 위해 적절하게 boundary를 추가하여 필터링을 수행하고 연산 후에 추가한 만큼의 boundary를 제거해주는 방식을 사용할 수 있다.



output of ideal lowpass filtering /wo boundary handling(color3.jpg)

- 아래 이미지는 `BORDER_REPLICATE` 옵션으로 boundary를 추가한 뒤 계산하여 얻은 결과물이다. 이처럼 기존 이미지의 가장자리 부분과 유사한 값의 픽셀들로 boundary를 추가해주면, frequency domain에서도 frequency의 변화가 작아지므로 왜곡되는 현상을 줄일 수 있다.



output of ideal lowpass filtering /w boundary handling(color3.jpg)

- padding 크기를 filter 크기의 절반 이상으로 설정해주면, 신호의 끝 부분에서 연산에 영향을 주는 부분이 적절한 값으로 대체될 수 있다. filter 크기의 절반보다 커지면 결과에 차이는 없지만 연산 속도가 느려질 수 있다.

## 2. Gaussian lowpass filter

### 2-1. Implementation

- Gaussian lowpass filtering

```
def gaussianFilteringFFT(image, filter_size, border_type):
    img = cv2.imread('images/input/' + image, cv2.IMREAD_COLOR).astype(np.float32) / 255.
    padding = math.floor(filter_size / 2)
    expanded_image = cv2.copyMakeBorder(img, padding, padding, padding, padding, border_type)
    b, g, r = cv2.split(expanded_image)

    flt = gauss2d((padding*2+1, padding*2+1), padding / 6.0)
    flt_f = psf2otf(flt, (expanded_image.shape[0], expanded_image.shape[1]))

    results = []
    for i, c in enumerate([b, g, r]):
        img_f = np.fft2(c)
        img_flt_f = flt_f * img_f
        results.append(np.abs(np.ifft2(img_flt_f)))
    return np.stack(results, axis=-1)
```

```

    img_flt = np.real(np.fft.ifft2(img_flt_f))
    img_flt = cv2.normalize(img_flt, None, np.min(c) * 255, np.max(c) * 255, cv2.NORM_MINMAX, -1)
    results.append(img_flt)

    result_image = cv2.merge((results[0], results[1], results[2]))
    output_image = result_image[padding:-padding, padding:-padding, :]
    b, g, r = cv2.split(output_image)
    for i, c in enumerate([b, g, r]):
        img_f = np.fft.fft2(c)
    cv2.imwrite('images/output/gaussian/' + image.split('.')[0] + '_%d_%d_result.jpg' % (filter_size, border_type), output_image)
    return output_image

```

- Ideal Lowpass Filtering와 동일한 프로세스로 구현하였다.
  - Preprocessing → Fourier Transform(Gaussian lowpass filter) → Fourier Transform(Image) → Multiplication in Frequency Domain → Inverse Fourier Transform & Normalization → Postprocessing
- Ideal Lowpass Filtering 계산 과정에서 필터 부분만 Ideal Lowpass Filter에서 Gaussian lowpass filter로 바꾸어주었다. 즉, `flt` 만 `ilf(filter_size, radius)`에서 `gauss2d((padding*2+1, padding*2+1), padding / 6.0)`로 수정하였다.

- Gaussian lowpass filter

```

def gauss(n,sigma):
    r = np.arange(0, n, dtype=np.float32) - (n-1.)/2.
    r = np.exp(-r**2./(2.*sigma**2))
    return r / np.sum(r)

def gauss2d(shape, sigma):
    g1 = gauss(shape[0], sigma).reshape([shape[0], 1])
    g2 = gauss(shape[1], sigma).reshape([1, shape[1]])
    return np.matmul(g1,g2)

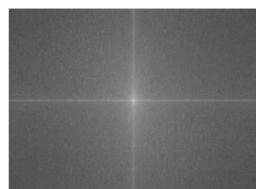
```

- 강의 노트에 있는 코드를 사용하였다.
- 커널 크기(shape)와 sigma 값에 따라 2D gaussian 커널을 계산해 numpy 배열 형태로 반환한다.

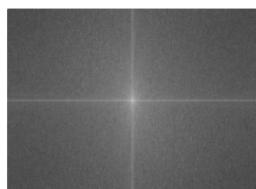
## 2-2. Results



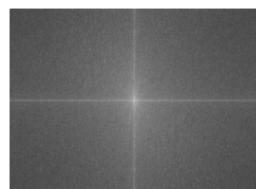
input(color3.jpg)



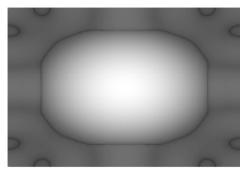
spectrum of an input image(B)



spectrum of an input image(G)



spectrum of an input image(R)



gaussian lowpass filter in the frequency domain



filtering result in the frequency domain(B)



filtering result in the frequency domain(G)



filtering result in the frequency domain(R)



output(color3.jpg)



input(color3.jpg)



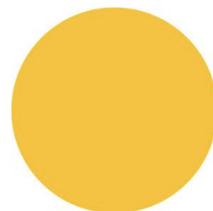
filtering result, filter size=33



filtering result, filter size=77



filtering result, filter size=99



input(shape.jpg)



filtering result, filter size=33



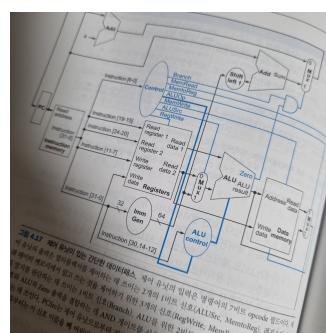
filtering result, filter size=77



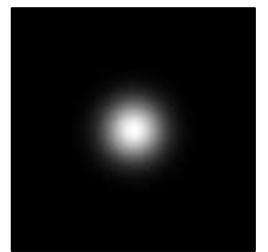
filtering result, filter size=99

### 2-3. Discussion

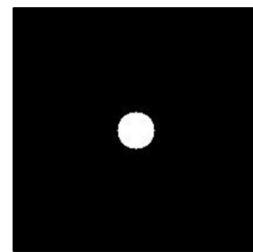
- Compare the results of the ideal and Gaussian lowpass filter



input(color8.jpg)



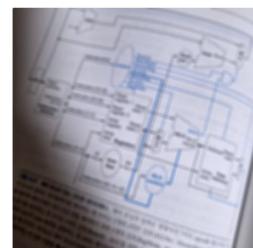
gaussian lowpass filter in spatial domain



ideal lowpass filter in spatial domain

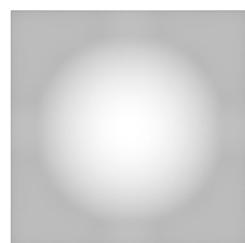


output(gaussian lowpass filter)

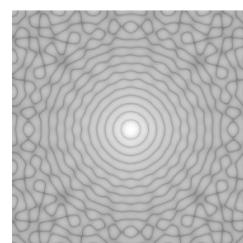


output(ideal lowpass filter)

- Ideal lowpass filter와 Gaussian lowpass filter 모두 이미지에서 high frequency 성분을 제거한다. Ideal lowpass filter는 threshold 이상의 frequency를 제거하는 방식을 사용한다. 그런데 그 경계가 뚜렷하기 때문에 이미지 내의 뚜렷한 경계 부분, 즉 frequency가 급격하게 변하는 부분에서 부드럽게 흐려지지 않고 ringing effect가 발생하게 된다. Gaussian lowpass filter에서는 가우시안 분포를 활용하여 점진적으로 감소하는 형태를 만들어 사용함으로써 이미지 내에 경계가 뚜렷한 부분에서 흐려지는 효과가 부드럽게 나타나게 한다.
- 이 차이는 frequency domain에서 두 필터의 형태를 살펴보면 쉽게 이해할 수 있다. Ideal lowpass filter의 경우에는 Gaussian lowpass filter와 달리 중심으로부터 파동이 퍼져나가는 듯한 형태를 띠고 있으며 이로 인해 ringing effect가 발생하는 것이다.

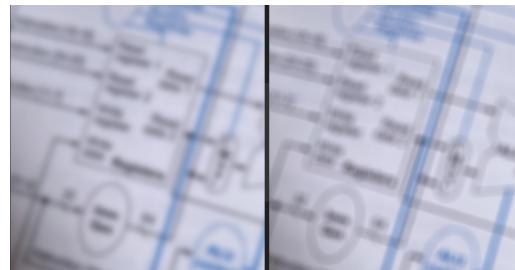


gaussian lowpass filter in frequency domain



ideal lowpass filter in frequency domain

- 결과를 확대해 살펴보면, 파란색 선 주변 부분에서 ringing effect가 나타나는 것을 확인할 수 있다.



comparison of two results(left: gaussian lowpass, right: ideal lowpass)

### 3. Unsharp Masking & Convolution Theorem

#### 3-1. Implement

```

def unsharpMasking(image, alpha, filter_size, domain, border_type):
    img = cv2.imread('images/input/' + image, cv2.IMREAD_COLOR)
    padding = math.floor(filter_size / 2)
    expanded_image = cv2.copyMakeBorder(img, padding, padding, padding, padding, border_type)
    b, g, r = cv2.split(expanded_image)
    flt = gauss2d((padding * 2 + 1, padding * 2 + 1), padding / 6.0)
    results = []
    output_image = np.zeros_like(img)

    if domain == 'frequency':
        flt_f = psf2otf(flt, (expanded_image.shape[0], expanded_image.shape[1]))
        for c in [b, g, r]:
            img_f = np.fft.fft2(c)
            img_flt_f = img_f + alpha * (img_f - flt_f * img_f)
            img_flt = np.real(np.fft.ifft2(img_flt_f))
            results.append(img_flt)

        result_image = cv2.merge((results[0], results[1], results[2]))
        output_image = result_image[padding:-padding, padding:-padding, :]
        cv2.imwrite('images/output/unsharp/' + domain + '/' + image.split('.')[0] + '_%d_%d_result.jpg' % (filter_size, border_type), output_image)
    else:
        width = img.shape[0]
        height = img.shape[1]
        img_flt = np.zeros_like(img).astype(np.float32)
        for i in range(width):
            for j in range(height):
                sub_image = expanded_image[i:i+filter_size, j:j+filter_size].astype(np.float32)
                b, g, r = cv2.split(sub_image)
                for i_c, c in enumerate([b, g, r]):
                    img_flt[i][j][i_c] = sum(sum(flt * c))

        output_image = img + alpha * (img - img_flt)
        cv2.imwrite('images/output/unsharp/' + domain + '/' + image.split('.')[0] + '_%d_%d_result.jpg' % (filter_size, border_type), output_image)

    return output_image

```

- input으로 이미지 이름, 알파값, 필터 크기, 연산을 수행할 도메인, border 타입을 받도록 하였다.
- domain과 관계없이 필터와 이미지는 같기 때문에 가장 먼저 `cv.copyMakeBorder()` 함수를 이용해 이미지 가장자리에 padding을 추가 해주고 gaussian filter를 numpy 형태로 계산해주었다. `gauss2d()` 는 앞서 사용한 것과 같은 함수를 이용했다. 이때 sigma 값은 필터 크기에 따라 지정되도록 했다.
- **frequency domain**
  - 1) Fourier Transform(Filter)
    - gaussian filter에 대해 `psf2otf()` 함수를 사용해 확장된 이미지만큼 Zero padding 해준 뒤, Shift, Fourier transform를 해주었다.
  - 2) Fourier Transform(Image)
    - `np.fft.fft2()` 함수를 이용해 이미지 또한 채널 별로 Fourier transform을 수행해주었다.
  - 3) Calculation in Frequency Domain
    - 앞서 필터를 처리해준 것을 곱해줌으로써 Gaussian Lowpass Filtering이 이뤄지도록 하고, 이를 기존 이미지에서 빼줌으로써 반대로 High frequency만 남는 값을 얻도록 했다. 이 값에 alpha를 곱하고, 기존 이미지에 더해주어 결과적으로 unsharp masking 이 될 수 있도록 하였다.
  - 4) Inverse Fourier Transform
    - 계산한 결과는 inverse fourier transform을 통해 다시 spatial domain으로 바꿔 픽셀 값으로 저장해주었다.
- 5) Postprocessing
  - `cv2.merge()` 함수를 이용해 분리했던 색상 채널들을 다시 합쳐 하나의 이미지로 만들고, expand 했던만큼 다시 가장자리를 crop해 결과물을 저장했다.
- **Spatial domain**
  - convolution 연산을 수행하여 먼저 gaussian lowpass filtering을 해주었다. 각 채널 별로, 모든 픽셀에 대해 해당 픽셀을 중심으로 한 필터 크기만큼의 영역 값들과 필터와 곱한 뒤 합해줌으로써 필터링 이후 픽셀 값을 계산하여 img\_flt 변수에 저장해주었다.
  - gaussian lowpass filtering 계산이 모두 끝난 뒤, unsharp masking 수식에 따라 계산해줌으로써 결과물을 얻게 하였다.

## 3-2. Results



input(color3.jpg)



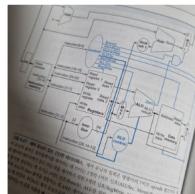
output(filter\_size=33, alpha=1,  
border\_type=BORDER\_REPLIC  
ATE)



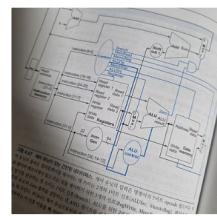
output(filter\_size=77, alpha=1,  
border\_type=BORDER\_REPLIC  
ATE)



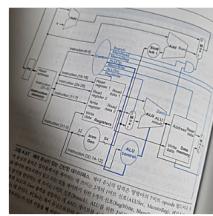
output(filter\_size=99, alpha=1,  
border\_type=BORDER\_REPLIC  
ATE)



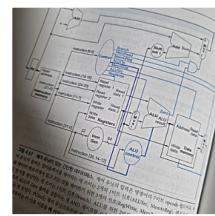
input(color8.jpg)



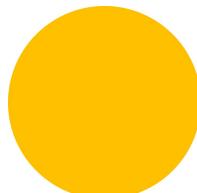
output(filter\_size=33, alpha=1,  
border\_type=BORDER\_REPLIC  
ATE)



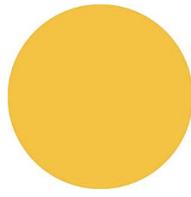
output(filter\_size=77, alpha=1,  
border\_type=BORDER\_REPLIC  
ATE)



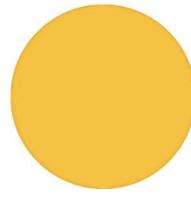
output(filter\_size=99, alpha=1,  
border\_type=BORDER\_REPLIC  
ATE)



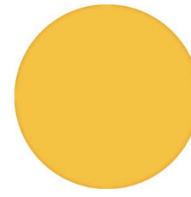
input(shape.jpg)



output(filter\_size=33,  
alpha=1,  
border\_type=BORDE  
R\_REPLICATE)



output(filter\_size=77,  
alpha=1,  
border\_type=BORDE  
R\_REPLICATE)



output(filter\_size=99,  
alpha=1,  
border\_type=BORDE  
R\_REPLICATE)

### 3-3. Discussion

- The effects of the parameters

- filter size

- 필터 크기가 커질수록 sigma 값이 커지고, sigma가 클수록 이미지가 더욱 흐려지게 된다. 따라서 흐림 정도가 더 큰 이미지를 원본 이미지에서 뺀 값을 사용하게 되면서, 경계에서의 대비 효과는 더 커지는 것을 확인할 수 있다.



filter\_size=33, sigma~=2.66, border\_type=BORDER\_REPLICATE



filter\_size=77, sigma~=6.33, border\_type=BORDER\_REPLICATE



filter\_size=99, sigma~=8.16, border\_type=BORDER\_REPLICATE

- alpha

- alpha 값이 커질수록 원본 이미지에 high frequency 영역이 더 많이 혼합되게 된다. 따라서 alpha 값이 커질수록 경계가 더욱 선명한 이미지를 얻을 수 있다.



alpha=1, filter\_size=77, border\_type=BORDER\_REPLICATE



alpha=2, filter\_size=77, border\_type=BORDER\_REPLICATE



alpha=3, filter\_size=77, border\_type=BORDER\_REPLICATE

- **The results of each step**

- Input 이미지를 불러와 border\_type에 따라 가장자리에 padding을 추가해준다.

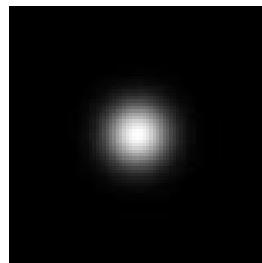


input(color3.jpg)



expanded image(border\_type=BORDER\_REPLICATE)

- filter size에 따라 가우시안 필터를 생성한다.

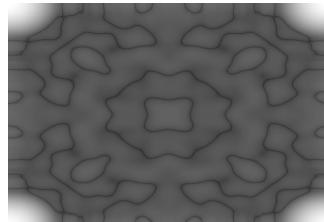


gaussian kernel (size=77)

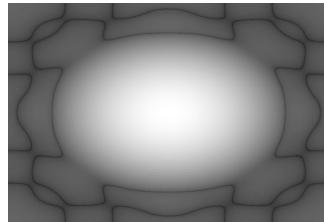
이 단계까지는 도메인에 관계 없이 동일하지만, 이후 연산의 경우에는 도메인에 따라 다른 방식을 사용한다.

1) frequency domain일 때

- 만들어진 가우시안 필터를 이미지 크기만큼 zero padding 해주고 shift한 뒤, Fourier transform으로 변환해준다.

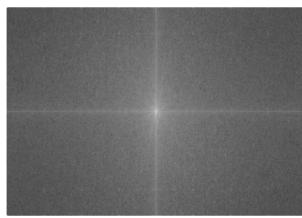


gaussian kernel after padding&shift&fft

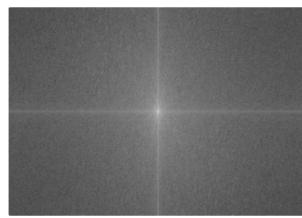


spectrum of gaussian kernel after padding&shift&fft

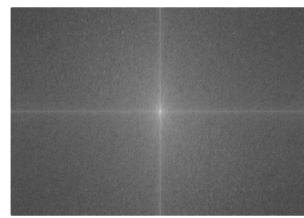
- 이미지를 채널 별로 나누고, 각각을 Fourier transform으로 변환한 필터와 곱해준다.



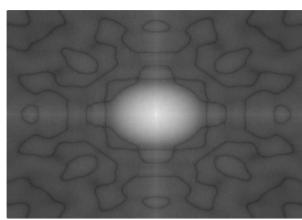
spectrum of input(G)



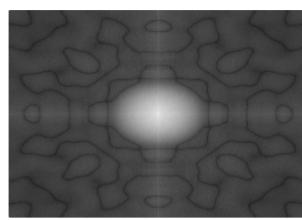
spectrum of input(G)



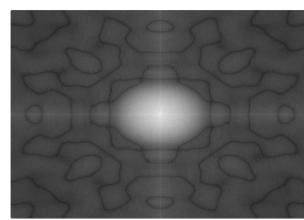
spectrum of input(R)



spectrum of gaussian lowpass filtering result(B)

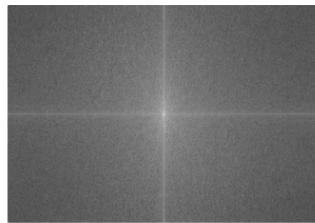


spectrum of gaussian lowpass filtering result(B)

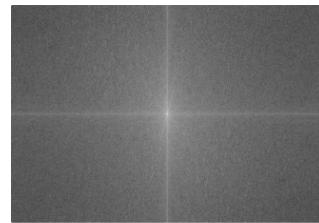


spectrum of gaussian lowpass filtering result(R)

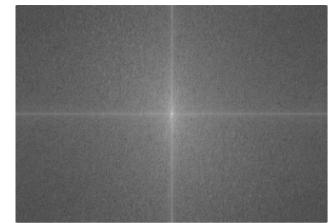
- 곱한 값을 기존 이미지에서 빼주어 high frequency만 남도록 하고, 이에 다시 alpha 값을 곱해 원본 이미지와 더해줌으로써 unsharp masking을 수행한다.



spectrum of unsharp masking  
result(B)



spectrum of unsharp masking  
result(G)



spectrum of unsharp masking  
result(R)



output(alpha=1, filter\_size=77, border\_type=BORDER\_REPLICATE)

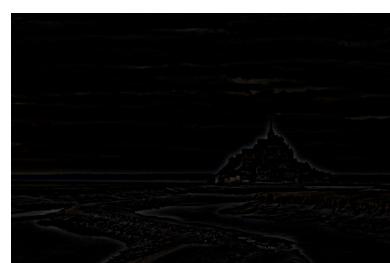
## 2) spatial domain

- 만들어진 필터의 중심이 픽셀의 중심에 오도록 설정한 뒤, 필터와 해당 영역의 픽셀 값들의 곱을 수행한 뒤 합을 구한다. 즉 convolution 연산을 수행해준다. 이를 통해 lowpass filtering이 이뤄진다.



filtered image

- 필터링된 이미지를 원본 이미지에서 빼주어 high frequency로 이뤄진 이미지를 얻는다.



input - filtered image

- high frequency 이미지에 alpha 값을 곱한 뒤 원본 이미지와 더해줌으로써 unsharp masking이 이뤄지도록 한다.



output(alpha=1, filter\_size=77, border\_type=BORDER\_REPLICATE)

- Computation time

- spatial domain과 frequency domain 모두 필터를 생성하는 단계까지는 동일하기 때문에 이후 연산에 대해서만 비교해보고자 한다.

(전체 픽셀 개수를 N, 필터 크기를 k라 가정)

- Spatial domain

- convolution 연산

- N개의 픽셀에 대해 모두 sub\_image\* filter를 수행하는데 이는  $k \times k$  크기의 두 행렬을 곱하는 것으로 총  $N * k^2$  번의 곱을 계산한다. 이후 곱한 값을 모두 더해주어야 하고, 이 역시  $k \times k$  번 수행하므로, convolution 연산에서는 채널당  $2Nk^2$  번 연산이 이뤄진다.  $\Rightarrow O(Nk^2)$

- unsharp masking

- 모든 픽셀에 대해 unsharp masking을 위한 연산  $Y = X + a(X - X')$ 을 수행해주어야 한다.  $\Rightarrow O(N)$

- 따라서 time complexity를 계산하면,  $O(Nk^2) + O(N) = O(Nk^2)$ 가 된다.

- Frequency domain

- fast fourier transform

- 파이썬 numpy에서 제공하는 fft, ifft 함수를 사용하였는데, 이들은 모두  $O(N \log N)$ 의 time complexity를 갖는다. 따라서 zero padding, shift를 거친 filter를 fourier transform하는 데  $O(N \log N)$  이미지를 fourier transform하는 데  $O(N \log N)$ , 이미지를 다시 inverse transform 하는 데  $O(N \log N)$ 이다. 따라서 모든 fourier transform에 대한 time complexity는  $O(N \log N)$ 이 된다.  $\Rightarrow O(N \log N)$

- Calculation in frequency domain

- 모든 픽셀에 대해 unsharp masking을 위한 연산  $Y = X + a(X - G^*X)$ 을 수행해주어야 한다. 이는 채널당 N번 수행한다.  $\Rightarrow O(N)$

- 따라서 time complexity를 계산하면,  $O(N \log N) + O(N) = O(N \log N)$ 이 된다.