

# CSED551 PA#1

## :Image Filtering

20220848 Minsu Sun

September 30, 2024

### Problem 1

(하단의 내용은 *gaussianFilter*를 구현하는 내용에 대한 설명을 기술합니다.)

#### Algorithm - Primitives

```
75 def gaussian1D(sigma: float, a: int) -> float:  
76     """Gaussian distribution function in 1D. This function does not consider constant term of the  
    formula.  
77  
78     Args:  
79         sigma (float): standard deviation of gaussian distribution  
80         a (int): input of gaussian distribution  
81  
82     Returns:  
83         float: desired output of 1D gaussian distribution at given `a` without consideration of  
            constant alpha  
84     """  
85     # ignored constant term: 1 / sqrt(2\pi\sigma^2)  
86     # normalizing will make them useless  
87     return np.exp(-(a**2) / (2 * sigma**2))  
  
                                         Primitive - gaussian1D
```

위 코드는 1차원 Gaussian 분포를 계산하는 함수이다. 2차원 Gaussian 분포는 1차원 Gaussian 분포의 곱으로 나타낼 수 있으며, 이는 아래의 과정에 따라 자명한 결과이다.

$$\begin{aligned} Gaussian_{1D}(x) &= c \cdot e^{\frac{x^2}{2\sigma^2}} \\ Gaussian_{1D}(x) \cdot Gaussian_{1D}(y) &= c \cdot e^{\frac{x^2}{2\sigma^2}} \cdot c \cdot e^{\frac{y^2}{2\sigma^2}} = c^2 \cdot e^{\frac{x^2+y^2}{2\sigma^2}} \\ &= Gaussian_{2D}(x, y) \\ (c &= \frac{1}{\sqrt{2\pi\sigma^2}}) \end{aligned}$$

이에 따라, 2차원 Gaussian 분포는 1차원 Gaussian 분포의 곱을 이용하여 구할 수 있으므로 2차원 Gaussian 분포를 계산하는 함수는 생략한다. 실제 2차원 Gaussian 분포가 필요한 2차원 Gaussian Filter를 계산하는 함수에서도 1차원 Gaussian Filter를 활용하여 계산하기 때문에 더더욱 2차원 Gaussian 분포를 계산하는 함수의 필요성은 더더욱 낮아진다. 위의 계산식 앞에는 상수항  $c$ 가 존재하지만, 이후 Gaussian Filter를 생성하면서 정규화할 것이기 때문에 상수항은 무시하여 생성하도록 설계하였다.

```
90 def build1DGaussianKernel(kernel_size: int, kernel_sigma: float) -> np.ndarray:  
91     """build 1D gaussian kernel with given kernel size and kernel sigma for gaussian distribution  
92  
93     Args:  
94         kernel_size (int): _description_  
95         kernel_sigma (float): _description_  
96  
97     Returns:  
98         np.ndarray: (kernel_size, ) 1D gaussian kernel
```

```

99 """
100 mid = kernel_size // 2
101 kernel = np.arange(mid + 1, dtype=np.float32)
102 kernel = np.vectorize(lambda x: gaussian1D(sigma=kernel_sigma, a=x))(kernel)
103 kernel = np.concatenate((kernel[-1:0:-1], kernel))
104 return kernel / np.sum(kernel)

```

Primitive - build1DGaussianKernel

위에서 기술한 `gaussian1D` 함수를 이용하여 1차원 Gaussian Filter를 만들어 반환하는 함수이다. 주어지는 `kernel_size`가 홀수이며, 필터는 좌우 대칭을 이루기 때문에, 절반에 대해서만 실제 연산을 진행한 후, 나머지는 이를 복사하여 사용하는 식으로 구현하였다. Gaussian Filter는 정규화된 상태이므로, Gaussian 분포를 이용해 계산한 이후 전체에 대해 정규화를 진행하여 결과를 반환한다.

```

107 def build2DGaussianKernel(kernel_size: int, kernel_sigma: float) -> np.ndarray:
108     """build 2D gaussian kernel with given kernel size and kernel sigma for gaussian distribution
109
110     Args:
111         kernel_size (int): kernel size n of n by n matrix
112         kernel_sigma (float): standard deviation of gaussian distribution
113
114     Returns:
115         np.ndarray: (kernel_size, kernel_size) 2D gaussian kernel
116     """
117     # 2D Gaussian kernel is product of 1D Gaussian kernel and its transpose
118     g = build1DGaussianKernel(
119         kernel_size=kernel_size, kernel_sigma=kernel_sigma
120     ).reshape(kernel_size, 1)
121     kernel = np.matmul(g, g.T)
122     # suppose g is normalized already, result of g * g.T should be normalized, too
123     return kernel

```

Primitive - build2DGaussianKernel

2차원 Gaussian Filter를 생성하여 반환하는 함수이다. 2차원 Gaussian Filter는 (`kernel_size`, 1) 크기의 1차원 Gaussian Filter  $f$ 를 이용해  $f \cdot f^T$ 의 연산을 통해 얻을 수 있는 점을 생각하여 위에서 구현한 `build1DGaussianKernel` 함수를 이용해 구현하였다. `build1DGaussianKernel`과 동일하게 정규화를 진행해야 하지만, 정규화된 행렬의 곱 결과 또한 정규화되므로, 정규화는 생략 가능하다.

## Algorithm - Main Process

```

7 def filterGaussian(
8     image: np.ndarray,
9     kernel_size: int,
10    kernel_sigma: float,
11    border_type,
12    separable: bool,
13 ) -> np.ndarray:
14     """**PROBLEM 1: GAUSSIAN FILTERING**
15
16     Args:
17         image (numpy.ndarray): input RGB image
18         kernel_size (int): an odd integer to specify the kernel size. If this is 5, then the
19                         actual kernel size is 5x5.
20         kernel_sigma (float): a positive real value to control the shape of the filter kernel.
21         border_type (_type_): extrapolation method for handling image boundaries. Possible
22                         values are: `cv2.BORDER_CONSTANT`, `cv2.BORDER_REPLICATE`, `cv2.BORDER_REFLECT`,
23                         `cv2.BORDER_WRAP`, `cv2.BORDER_REFLECT_101`

```

```

21     separable (bool): Boolean value. If `separable == true`, then the function performs
22         Gaussian filtering using two 1D filters. Otherwise, the function performs Gaussian
23         filtering using a normal 2D convolution operation.
24
25     Returns:
26         image (np.ndarray): The function must return a filtered RGB image.
27     """
28
29     assert len(image.shape) == 3 # do not accept GrayScale or ARGB image
30     assert image.shape[2] == 3 # make sure image has RGB channel
31     assert kernel_size % 2 == 1 # kernel size is odd
32     assert kernel_sigma > 0 # kernel sigma is positive real number
33     assert border_type in [
34         cv2.BORDER_CONSTANT,
35         cv2.BORDER_REPLICATE,
36         cv2.BORDER_REFLECT,
37         cv2.BORDER_WRAP,
38         cv2.BORDER_REFLECT_101,
39     ]
40
41     # height, width from original image, not padded image
42     height, width, channel = image.shape
43     result = np.zeros_like(image)
44
45     if kernel_size != 1:
46         padding_size = kernel_size // 2
47         image = cv2.copyMakeBorder(
48             image, padding_size, padding_size, padding_size, padding_size, border_type
49         )
50
51     if separable:
52         kernel = build1DGaussianKernel(kernel_size, kernel_sigma)
53         kernel_row = kernel.reshape(1, kernel_size)
54         kernel_col = kernel.reshape(kernel_size, 1)
55     else:
56         kernel = build2DGaussianKernel(kernel_size, kernel_sigma)
57
58     for i in range(height):
59         for j in range(width):
60             for k in range(channel):
61                 if separable:
62                     # (kernel_size, kernel_size) * (kernel_size, 1) * (1, kernel_size)
63                     result[i][j][k] = np.sum(
64                         image[i : i + kernel_size, j : j + kernel_size, k]
65                         * kernel_col
66                         * kernel_row
67                     )
68                 else:
69                     # (kernel_size, kernel_size) * (kernel_size, kernel_size)
70                     result[i][j][k] = np.sum(
71                         image[i : i + kernel_size, j : j + kernel_size, k] * kernel
72                     )
73
74     return result

```

filterGaussian

주어진 `image`에 Gaussian Filter를 적용하여 반환하는 함수이다. 과제에서 제시한 조건에 따라 `assert`를 이용해 전달받은 매개 변수의 조건을 검사한다.

- 주어진 이미지는 RGB 이미지이므로, `image.shape`은 3개의 원소만을 가져야한다.
- 주어진 이미지는 RGB 이미지이므로, `image`는 3개의 채널만을 가져야한다. 이외의 경우에는 GrayScale 이미지이거나, ARGB 이미지라고 인식해 오류를 발생한다.
- `kernel_size`는 홀수로 제한된다.
- `kernel_sigma`는 양의 실수이다. Python의 타입에 따른 유연성 때문에, 양의 범위에 해당함을 확인한다.
- `border_type`은 제시된 경우에 대해서만 선택할 수 있으므로, 이를 확인한다.

`image`로부터 `height, width, channel`을 따로 저장하고, `np.zeros_like`를 이용해 `image`의 형태가 같은 변수를 만들어 결과를 담는 변수로 사용한다. `kernel_size`가 1일 경우, 패딩을 진행할 필요가 없기 때문에, `kernel_size`가 1이 아닌 경우에만 `cv2.copyMakeBorder`를 이용해 지정된 `border_type`에 따른 패딩을 진행하여 저장한다. 패딩하는 사이즈의 경우, `kernel_size=2n+1`에서 모든 방향으로 `n`만큼의 패딩이 필요하므로, `kernel_size//2`를 사이즈로 사용한다.

이후 `filter`(이후 `kernel`이라 칭함)를 준비하는데, 함수의 매개변수 `separable`에 따라서 `True`일 경우 2차원 Gaussian Kernel, `False`일 경우 1차원 Gaussian Kernel을 각 행과 열에 대해 준비한다. 이 때, `build1DGaussianKernel`의 반환값은 (`kernel_size,` )의 형태이기 때문에, 행과 열 Kernel을 각각 (`1, kernel_size`), (`kernel_size, 1`)의 행렬이 되도록 `kernel.reshape`를 통해 조정한다.

이 단계까지는 다음과 같은 상태를 만족한다.

- `image` : (`height + 2 * padding_size, width + 2 * padding_size, channel`)의 `shape`를 가진다.
- `result` : (`height, width, channel`)의 `shape`를 가진다.
- `kernel*` : `separable` 여부에 따라 `kernel_row`와 `kernel_col` 혹은 `kernel`가 정의된다.

준비된 Kernel을 이용하여 이미지와 Convolution 연산을 진행한다. 패딩된 이미지를 각 채널별로 `kernel_size`에 맞추어 잘라 이를 `numpy`의 broadcasting 연산을 통해서 Convolution을 진행한다. `separable`이 `True`일 경우에는 정의된 `kernel`이 `kernel_row`, `kernel_col`로 두 가지이므로, 이를 연속해서 연산에 사용한다.

이후 연산이 모두 완료된 이미지를 반환한다.

## Input Images



(a) color1.jpg



(b) color2.jpg



(c) color3.jpg



(d) color4.jpg



(e) color5.jpg



(f) color6.jpg

## Input Images

기존 제공된 `color1.jpg`, `color2.jpg`, `color3.jpg` 이외에 `color4.jpg`, `color5.jpg`, `color6.jpg`를 추가하여 사용한다. 추가 이미지는 모두 iPhone 13에서 촬영된 이미지이다. 이미지의 해상도는 아래와 같다.

- color1.jpg : 800px x 450px
- color2.jpg : 800px x 612px
- color3.jpg : 800px x 533px
- color4.jpg : 3024 x 3024px
- color5.jpg : 3024px x 4032px
- color6.jpg : 3024px x 4032px

## Input Parameters

위에서 제시한 이미지를 이용해 `fileterGaussian`을 실행할 때, 다음과 같은 파라미터 구성은 이용하여 진행하였다.

- border\_type : cv2.BORDER\_REPLICATE, cv2.BORDER\_REFLECT, cv2.BORDER\_WRAP
- kernel\_size : 1, 25, 45
- kernel\_sigma : 0.1, 1, 10
- separable : True, False

실행에 실제 사용된 설정은 `code/main.py`의 `configs['problem1']`에서 확인할 수 있다.

## Output Images

설계된 바에 의하면 결과 이미지의 총 개수는  $6 \times 3 \times 3 \times 3 \times 2 = 324$ 로, 현 보고서에 모두 포함하기에는 많아 각 이미지별로 Gaussian Filter 효과가 잘 드러나는 이미지들을 첨부한다.

(전체 결과 이미지는 `images/output/problem1`에서 확인할 수 있다. 파일 형식은 `(input_image)_(_kernel_size)_(_kernel_sigma)_(_border_type)_(_separable).jpg`이다.)



(a) color1.jpg



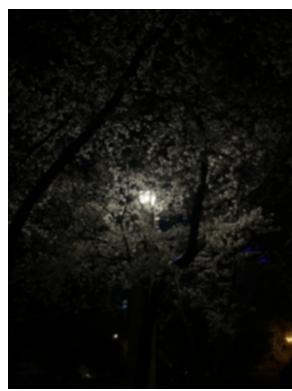
(b) color2.jpg



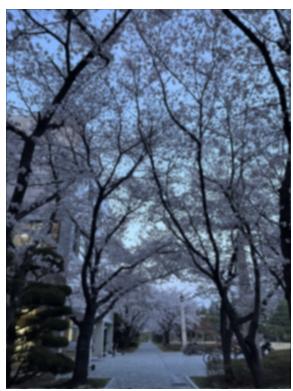
(c) color3.jpg



(d) color4.jpg



(e) color5.jpg



(f) color6.jpg

Output Images

각 이미지에서 사용된 파라미터는 다음과 같다. (`separable`은 해당 파라미터의 여부에 상관없이 동일한 결과를 내므로 이에 대해서는 기술하지 않는다.)

- color1.jpg : `border_type=cv2.BORDER_REPLICATE, kernel_size=25, kernel_sigma=10`

- color2.jpg : border\_type=cv2.BORDER\_WRAP, kernel\_size=25, kernel\_sigma=10
- color3.jpg : border\_type=cv2.BORDER\_REFLECT, kernel\_size=45, kernel\_sigma=10
- color4.jpg : border\_type=cv2.BORDER\_REFLECT, kernel\_size=45, kernel\_sigma=10
- color5.jpg : border\_type=cv2.BORDER\_REPLICATE, kernel\_size=45, kernel\_sigma=10
- color6.jpg : border\_type=cv2.BORDER\_WRAP, kernel\_size=45, kernel\_sigma=10

## Computation Time

### Kernel Size

아래는 Kernel Size에 따른 실행 시간을 표로 나타낸 것이다. 이 때 사용한 파라미터는 cv2.BORDER\_REPLICATE, kernel\_sigma=10, separable=True이다.

Input Image	$T_1$	$T_{25}$	$T_{45}$
color1.jpg	3.30	4.88	<b>6.60</b>
color2.jpg	4.28	6.29	<b>9.05</b>
color3.jpg	3.71	5.50	<b>7.82</b>
color4.jpg	81.64	125.04	<b>173.23</b>
color5.jpg	106.95	159.21	<b>247.01</b>
color6.jpg	106.73	158.94	<b>226.98</b>

Computational Time Depending on Kernel Size

Kernel Size가 커짐에 따라서 실행 시간이 늘어나는 것을 확인할 수 있으며, 입력 이미지의 해상도에 따라서도 실행 시간이 달라지는 것을 확인할 수 있었다.

### Separable

아래는 Separable 여부에 따른 실행 시간을 표로 나타낸 것이다. 이 때 사용한 파라미터는 cv2.BORDER\_REPLICATE, kernel\_size=45, kernel\_sigma=10이다.

Input Image	$T_{True}$	$T_{False}$
color1.jpg	6.60	<b>4.54</b>
color2.jpg	9.05	<b>6.17</b>
color3.jpg	7.82	<b>5.73</b>
color4.jpg	173.23	<b>121.22</b>
color5.jpg	247.01	<b>166.04</b>
color6.jpg	226.98	<b>156.56</b>

Computational Time Depending on Separable

Separable이 False인 경우에 실행 시간이 감소하는 것을 확인할 수 있었으며, 이를 통해 1차원 Gaussian Filter를 적용하는 것보다 2차원 Gaussian Filter를 적용하는 것이 훨씬 Computational Cost를 줄인다는 것을 확인할 수 있다.

## Discussion - Parameters & Computation Time

앞선 Computation Time 결과를 통해, Kernel Size에 따라 실행 시간이 오래 걸리는 것을 확인할 수 있었다. Kernel Sigma나 Border Type에 따라서는 실제 연산 횟수가 변화하지 않기 때문에 실행 시간이 달라지지 않는다. Separable의 경우를 통해, 2D Gaussian Filter를 적용하는 것이 1D Gaussian Filter를 적용하는 것보다 더 적은 연산 횟수를 통해 빠른 실행이 가능하다는 것을 확인할 수 있었다. 파라미터 이외에도 입력 이미지의 해상도에 따라서 실행 시간이 오래 걸리는 것 또한 확인할 수 있었다.

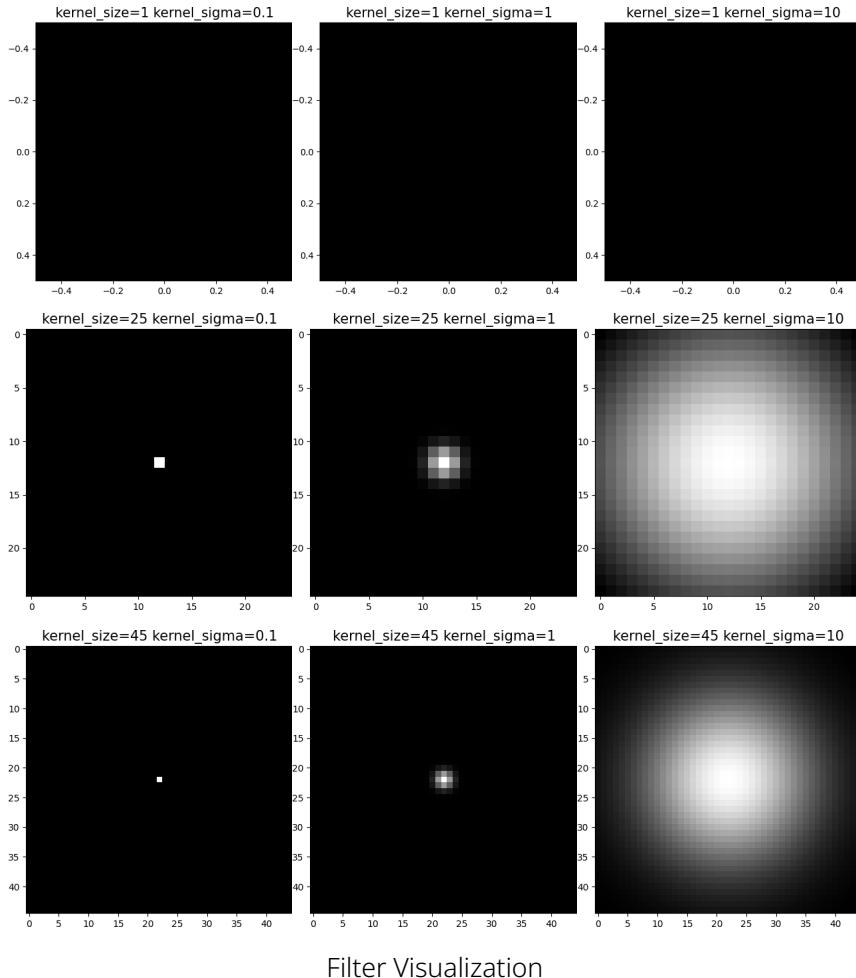
결과를 통해 다음과 같이 연산 시간에 미치는 요소들을 정리할 수 있다.

- Kernel Size: **Kernel Size가 큰 Filter를 사용할수록 연산 시간이 늘어난다.**
- Separable: **2D Gaussian Filter를 적용하는 것이 1D Gaussian Filter를 2번 적용하는 것보다 더 빠른 실행을 보인다.**
- Image Size: 큰 이미지일수록 연산이 더 오래걸린다.

## Filter Visualization

### Visualization

실행에서 사용된 파라미터 조합에 따른 Filter를 시각화한 결과는 아래와 같다. (시각화에 대한 상세 코드는 code/main.py의 problem1\_supplementary 함수에서 확인할 수 있다.)



### Discussion - Kernel Size & $\sigma$

같은 `kernel_sigma`에서 `kernel_size`에 따른 필터를 확인하였을 때, 필터별로 영향을 받는 부분(값이 높은, 시각화 결과에서 하얀색이 많이 보이는)의 분포가 달라지는 것을 확인할 수 있다. `kernel_sigma=10`인 경우 `kernel_size=45`에서는 골고루 필터의 전체 범위에 영향을 주는 모습을 볼 수 있지만, `kernel_size=25`에서는 다소 너무 많은 부분에서 영향을 주는 것을 확인할 수 있다.

이에 따라서, 주어진 `kernel_size` 혹은 `kernel_sigma`에 따라서 올바른 파라미터를 결정하여 필터의 전체 영역에 골고루 영향을 미칠 수 있도록 선택하는 것이 좋아보인다. 다만, `kernel_size`가 큼에도 Gaussian Filter 효과가 두드러지지 않는 케이스가 존재하는데(`color4.jpg`, `color5.jpg`, `color6.jpg`), 이는 이미지의 해상도에 따른 차이이다. 따라서, `kernel_size`는 입력 이미지의 크기에 맞추어 적당한 값을 선택하고, 이를 토대로 필터의 전반적인 부분에 영향을 미칠 수 있는 적당한 `kernel_sigma`를 찾는 것이 최적의 방법으로 생각된다.

위에서 제시한 시각화 결과 중에서는 `kernel_size=45` `kernel_sigma=10`인 경우의 필터가 이를 잘 나타내고 있다고 생각되며, `kernel_size=25`인 경우에는 `kernel_sigma=7` 부근에서 적절한 결과를 보일 것으로 생각된다.

## Border Types

### Result

아래는 `border_type`에 따른 결과 비교이다. 다른 파라미터의 경우, `kernel_size=45`, `kernel_sigma=10`을 사용하였다.



Result of Different Border Types

(추가로 사용한 `color4.jpg`, `color5.jpg`, `color6.jpg`의 결과는 위의 3가지 이미지에 비해 차이가 뚜렷하지 않아 생략한다. 이는 입력 이미지의 크기가 나머지에 비해 너무 큰 것으로 판단된다.)

## Discussion - Best & Worst Border Type

우선 각 Border Type에 따른 패딩 내용은 다음과 같다.

- **Reflect** : 가장자리의 이미지를 좌우반전 혹은 상하반전을 통하여 패딩한다.
- **Replicate** : 가장자리의 픽셀들을 복제하여 패딩한다.
- **Wrap** : 가장자리의 정반대편에 위치하는 이미지를 이용하여 패딩한다.

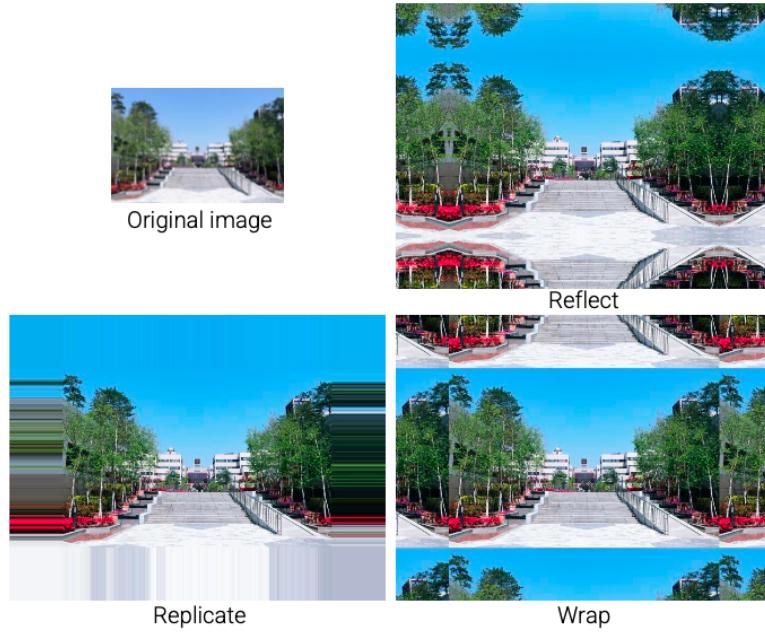
각 Border Type별 패딩 결과 예시는 아래와 같다.

위의 Border Type에 따른 Gaussian Filtering 결과를 보면, Wrap을 이용한 Border Type의 경우 이미지에 따라서 결과 이미지의 테두리에 검은 음영이 지는 것을 확인할 수 있으며 이는 특히 `color1 - Wrap`에서 잘 드러난다(이미지 상단 모서리). 이는 반대편 테두리에 상대적으로 어두운 색이 존재할 경우 어두운 색이 이미지를 패딩하는 과정에 포함되기 때문에, 결과적으로 Gaussian Filtering을 진행할 때도 해당 색이 반영되어 나타나는 결과이다. 반대로 반대편에 상대적으로 밝은 색이 존재할 경우에는 하얀 음영이 생기며, 이 또한 `color1 - Wrap`에서 확인할 수 있다(이미지 하단 모서리).

이와 같은 생각을 바탕으로 아래와 같이 Best와 Worst를 선정하였다.

- **Best**: `Replicate` - 다른 `Wrap`이나 `Reflect`에 비해, 가장자리의 색깔을 가장 잘 패딩할 수 있기 때문에, 의도하지 않은 영향을 최소화 할 수 있어 이를 Best로 선정한다.
- **Worst**: `Wrap` - 위에서 언급한 대로, 반대편의 상대적으로 다른 색깔이 존재할 경우 의도치 않게 영향을 받게 되기 때문에, `Wrap`을 Worst로 선정한다.

`Reflect`의 경우, 가장자는 아니더라도, 가장자리에 상대적으로는 근접한 픽셀이 결국 큰 사이즈의 패딩을 진행할 때 영향을 주게 되어, `Wrap`의 부작용과 같은 결과를 낼 수 있어 Best로 선정하지 않았다. 이에 따라, 가장자리 근처에서 급격한 색의 변화가 존재할 경우 부작용이 발생할 것으로 생각된다.



Example of Padding on Various Border Type

## Problem 2

### Algorithm - Primitives

```

150 def histogramEqualizationSingleChannel(image: np.ndarray) -> np.ndarray:
151     """histogram equalization on single channel
152
153     Args:
154         image (np.ndarray): input image of (height, width), this should be single channel image
155
156     Returns:
157         np.ndarray: histogram equalized image, same size as input image
158     """
159     assert len(image.shape) == 2
160
161     hist, _ = np.histogram(image.flatten(), 256, [0, 256])
162     chf = hist.cumsum() # cumulative histogram function
163     coef = 255 / chf[-1] # transformation coefficient
164     return np.vectorize(lambda p: int(coef * chf[p]))(image)

Primitive - histogramEqualizationSingleChannel

```

1개의 채널에 대하여 Histogram Equalization을 진행한다. `assert`를 이용하여 주어진 `image`의 형태가 1개 채널의 이미지인지 검사한다. 이후, `numpy.histogram`을 이용하여 이미지의 히스토그램을 계산하여 저장한다. 각 픽셀의 최대값은 256이므로 히스토그램의 구간 또한 `[0, 256]`으로 설정하여 계산한다. 구한 히스토그램을 기반으로 Cumulative Sum을 계산하여, Cumulative Histogram Function을 구한다. 이때, Cumulative Histogram Function의 마지막 값은 전체 픽셀의 개수이므로, 이는 `width x height` 결과와 같다. 이미 Cumulative Function을 통해 구하였으므로, 이를 이용하여 Equalization 계수로 사용한다. 주어진 `image`에 위에서 구한 Equalization 계수를 이용하여 Transform 함수를 `numpy.vectorize`로 적용시켜 결과를 반환한다.

### Algorithm - Main Process

```

129 def histogramEqualization(image: np.ndarray) -> np.ndarray:
130     """**PROBLEM 2: HISTOGRAM EQUALIZATION**
131
132     Args:

```

```

133     image (np.ndarray): input image, it should be grayscale or (a)rgb image
134
135     Returns:
136         np.ndarray: histogram equalized image, same size as input image
137         """
138     assert (
139         len(image.shape) == 3 or len(image.shape) == 2
140     ) # image should be (A)RGB or GRAYSCALE
141
142     if len(image.shape) == 2: # GRAYSCALE
143         return histogramEqualizationSingleChannel(image)
144
145     for c in range(image.shape[2]):
146         image[:, :, c] = histogramEqualizationSingleChannel(image[:, :, c])
147
148     return image

```

Primitive - histogramEqualization

주어진 image의 shape를 통해 채널이 존재하는 이미지이거나 채널이 존재하지 않는 흑백 이미지일 수 있도록 assert로 검사한다. shape이 (height, width)인 흑백 이미지의 경우, 단일 채널이며, 이에 대한 Histogram Equalization은 위의 histogramEqualizationSingleChannel을 이용하여 계산 후 반환한다. 이외에는 채널이 1개 이상 존재하는 이미지이므로, 각 채널에 대한 이미지를 추출하여 각각에 대해 Histogram Equalization을 진행한 후, 이를 각 채널에 다시 저장한다. 이후 계산된 결과를 반환하여 마무리한다.

## Input Images



(a) gray1.jpg



(b) gray2.jpg



(c) gray3.jpg



(d) gray4.jpg



(e) gray5.jpg



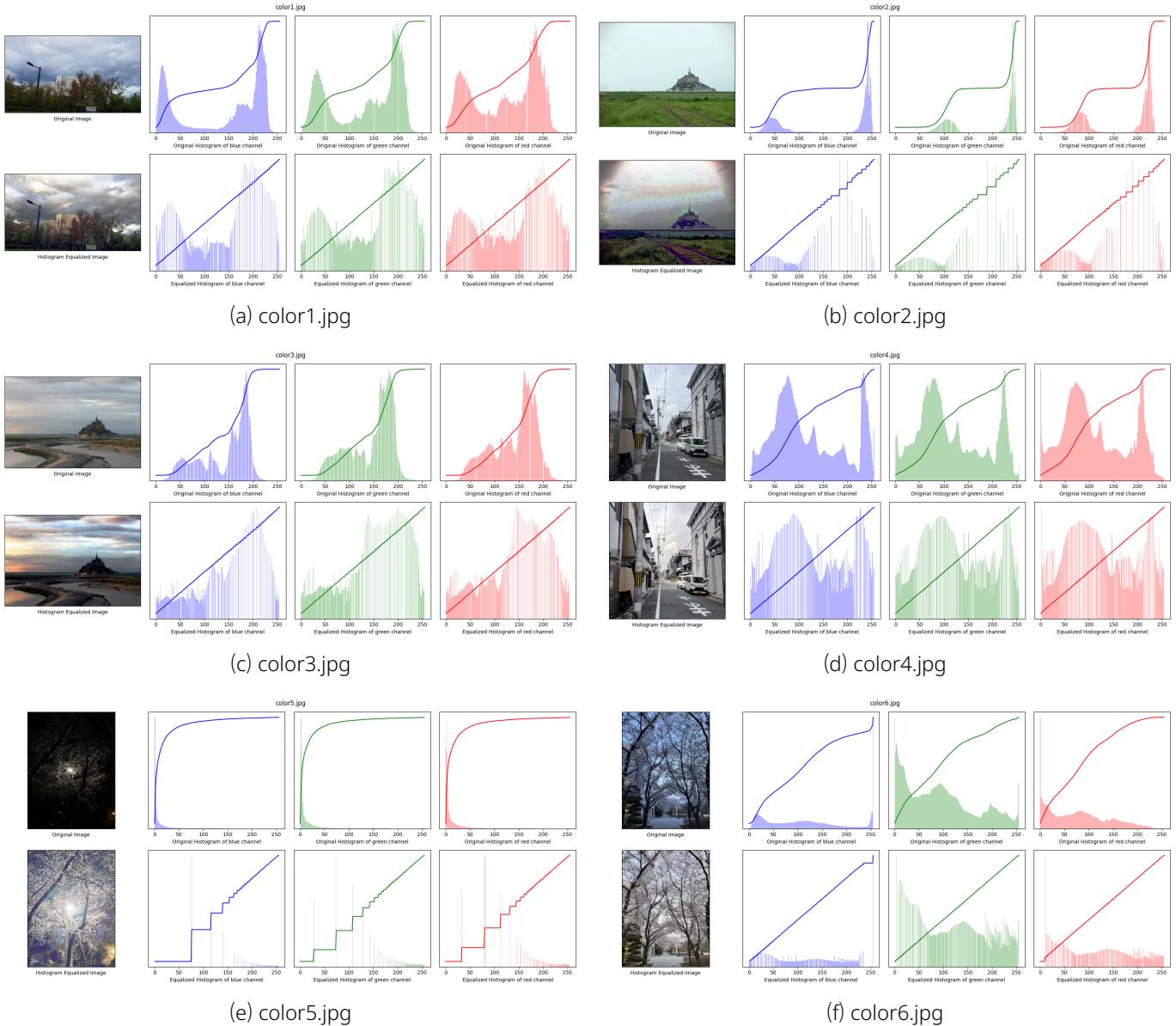
(f) gray6.jpg

Additional Input Images

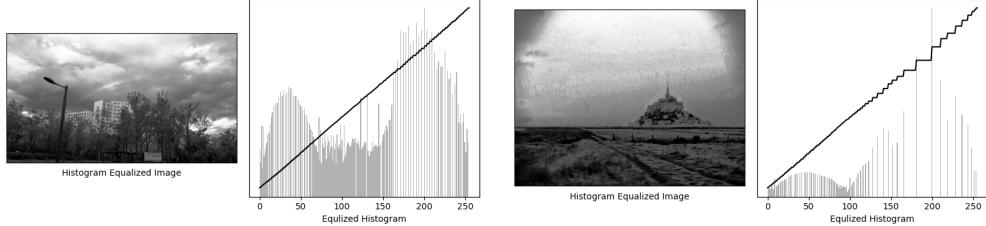
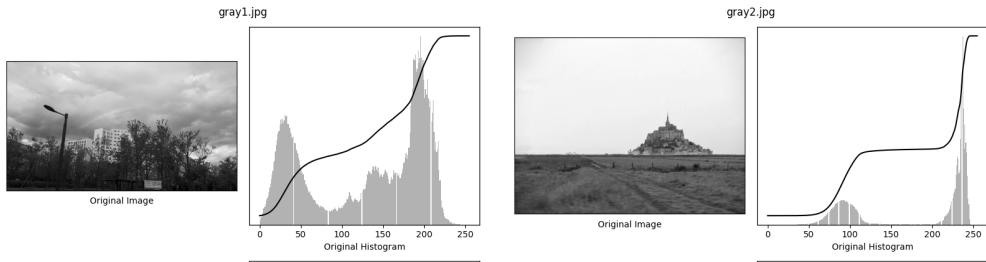
Problem 1에서 사용한 Color 이미지들과 이를 흑백으로 변환한 이미지들을 추가로 사용한다. 추가로 사용하는 흑백 이미지는 위와 같다. Problem 1에서 기존 사용하였던 이미지들은 위에서 확인 가능하다. 직접 촬영하여 추가한 사진의 경우 OpenCV를 통하여 GrayScale로 변환 후 저장하여 사용한다.

## Output Images

다음은 실행 결과 이미지와 해당 이미지의 전후 각 채널에 대한 시각화를 진행한 결과이다. (시각화되지 않은 실행 결과 이미지들은 images/output/problem2에서 확인할 수 있다.)

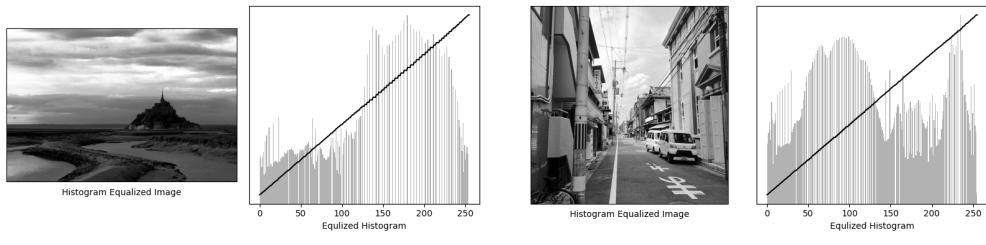
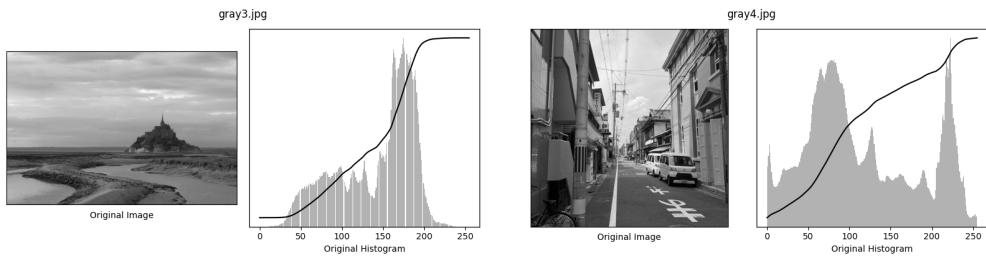


Output Color Images



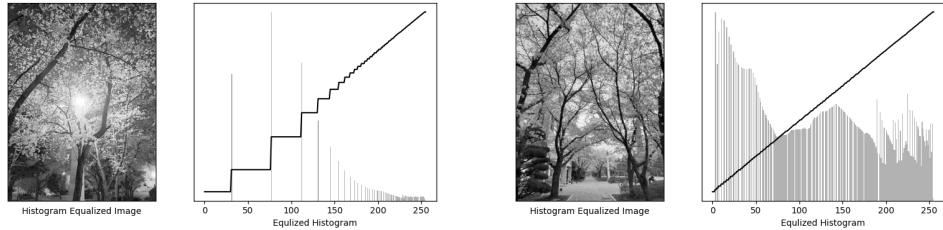
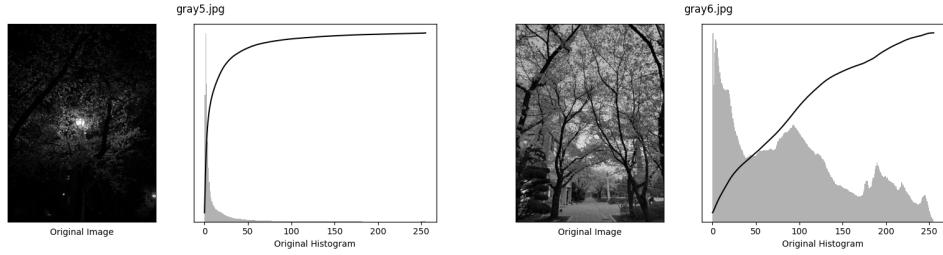
(a) gray1.jpg

(b) gray2.jpg



(c) gray3.jpg

(d) gray4.jpg



(e) gray5.jpg

(f) gray6.jpg

### Output Gray Images

## Discussion - Brightness & Colors

Histogram Equalization은 Linear한 Cumulative Histogram Function을 만드는 Transformation이다. 밝기는 픽셀값이 높을 수록 밝다고 인식하게 되는데, Linear하도록 Transformation이 진행되다 보니, 원래 밝았던 이미지는 어두워지는 경향을, 어두웠던 이미지는 밝아지는 경향을 보인다. 이는 Cumulative Histogram Function의 오목함과 불록함을 통해 해석할 수 있을 것으로 생각된다.

색감의 경우, 밝기와 마찬가지로 각 RGB 채널의 픽셀값에 의하여 해당 픽셀이 더 불어보이거나, 초래보이거나, 푸르러보인다. 앞서 이야기했듯이 Linear하도록 만들다 보니, 결국 덜 밝었던 부분은 더 밝도록, 덜 초래보이던 부분은 더 초래보이도록, 덜 푸르르던 부분은 더 푸르게 보이도록 만들게 된다.

결론적으로, Histogram Equalization은 이미지를 전반적으로 모든 부분에서 균일하게 보일 수 있도록 처리하는 방법이라고 생각할 수 있다. 이와 더불어, 전체 이미지의 대비가 올라가 이미지 내 물체들의 구분성도 높아지는 효과를 가져오게 된다.

## Discussion - Improvement

color5.jpg나 gray5.jpg의 시각화 결과를 볼 경우, 어두운 밤에 찍힌 이미지이다보니 낮은 픽셀값의 분포가 높아, Equalization 이후에는 낮은 픽셀값에서 비교적 너무 Discrete한 경향을 보인다. 이미지로 보았을 때, 지나치게 밝기가 올라가면서 전체적인 이미지를 확인하기 힘든데, 이런 Discrete한 구간에 대한 완화된 Equalization을 적용해, Equalization 이후 이를 개선할 수 있을 것으로 생각된다.