HW 02 – REPORT

소속 : 정보컴퓨터공학부

학번 : 202255595

이름 : 임영훈

1. 서론

이번 실습은 gaussian filter를 2D 이미지에 적용하여 blur 효과를 체험해 보고, grayscale 이미지뿐만 아니라 RGB 이미지에도 적용을 해보고, blur 효과를 통해 얻은 low-frequency 이미지를 원본 이미지에서 빼주어 high-frequency 이미지를 얻고 서로 다른 종류의 low-frequency 이미지와 high-frequency 이미지를 서로 더해 hybrid 이미지를 얻어보는 것이다. Blur 효과를 위해 filter를 활용하여 convolution 연산을 수행하여야 한다. 이때 filter를 가로, 세로 방향으로 뒤집으면 cross-correlation 연산으로 바꾸어 좀 더 쉽게 수행 가능하다.

2. 본론

1. Gaussian Filtering

1-1.

Element들의 값의 합이 1이어야 하므로 element 하나의 값은 1/(n*n)이어야 하고 **np.full(배열의 크기, 값)** 함수를 사용하면 크기가 n x n이고 원소 하나의 값이 1/(n*n)인 배열을 얻을 수 있다.

```
def boxfilter(n):
    # 짝수 예외 처리
    assert n%2 == 1, "dimension must be odd"
    # 값이 1/(n*n)이고 n x n 크기의 array return
    return np.full((n,n), 1/(n*n))
```

```
>>> boxfilter(3)
array([[0.11111111, 0.11111111, 0.11111111],
       [0.11111111, 0.11111111, 0.11111111],
       [0.11111111, 0.11111111, 0.11111111]])
```

```
>>> boxfilter(4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in boxfilter
AssertionError: dimension must be odd
```

```
>>> boxfilter(7)
array([[0.02040816, 0.02040816, 0.02040816, 0.02040816, 0.02040816,
        0.02040816, 0.02040816],
       [0.02040816, 0.02040816, 0.02040816, 0.02040816, 0.02040816,
        0.02040816, 0.02040816],
       [0.02040816, 0.02040816, 0.02040816, 0.02040816, 0.02040816,
        0.02040816, 0.02040816],
       [0.02040816, 0.02040816, 0.02040816, 0.02040816, 0.02040816,
        0.02040816, 0.02040816],
       [0.02040816, 0.02040816, 0.02040816, 0.02040816, 0.02040816,
        0.02040816, 0.02040816],
       [0.02040816, 0.02040816, 0.02040816, 0.02040816, 0.02040816,
        0.02040816, 0.02040816],
       [0.02040816, 0.02040816, 0.02040816, 0.02040816, 0.02040816,
        0.02040816, 0.02040816]])
```

1-2.

Sigma 값의 6배를 구한 후 올림을 하여 그 값이 짝수면 1을 더하고 홀수면 그 값을 길이로 사용을 하면 될 것이다., 그 후 길이 만큼의 [... -2, -1, 0, 1, 2, ...] 를 만들고 이 값에 따른 $exp(-x^2/(2 * sigma^2))$ 값을 대응시켜 주고, 전체 element의 합이 1이 되도록 만들어 주기 위해 모든 element의 값들을 다 더하고 이 값을 각각의 element에 나누면 될 것이다.

```
def gauss1d(sigma):
    temp = math.ceil(sigma*6)
    length = temp if temp%2 == 1 else temp+1

    n = int(length/2)
    gaussian = np.array([np.exp(-x**2/(2*sigma**2)) for x in range(-n, n+1)])
    gaussian = gaussian/gaussian.sum()
    return gaussian
```

```
>>> gauss1d(0.3)
array([0.00383626, 0.99232748, 0.00383626])
```

```
>>> gauss1d(0.5)
array([0.10650698, 0.78698604, 0.10650698])
```

```
>>> gauss1d(1)
array([0.00443305, 0.05400558, 0.24203623, 0.39905028, 0.24203623,
       0.05400558, 0.00443305])
```

```
>>> gauss1d(2)
array([0.0022182 , 0.00877313, 0.02702316, 0.06482519, 0.12110939,
       0.17621312, 0.19967563, 0.17621312, 0.12110939, 0.06482519,
       0.02702316, 0.00877313, 0.0022182 ])
```

1.3

2d array 형식의 gaussian filter를 구하면 된다. 먼저 gaussian 1d filter를 구하고 그후 이것을 np.outer() 에 집어넣으면 된다. 마지막으로 합을 1로 만들면 끝이 난다.

```
def gauss2d(sigma):
    # 1d filter 구하기
    gaussian1d = gauss1d(sigma)
    # np.outer 를 활용하여 2d filter 구하기
    gaussian2d = np.outer(gaussian1d, gaussian1d)
    # normalization
    gauss2d_filter = gaussian2d/gaussian2d.sum()
    return gauss2d_filter
```

```
>>> gauss2d(0.5)
array([[0.01134374, 0.08381951, 0.01134374],
       [0.08381951, 0.61934703, 0.08381951],
       [0.01134374, 0.08381951, 0.01134374]])
```

```
>>> gauss2d(1)
array([[1.96519161e-05, 2.39409349e-04, 1.07295826e-03, 1.76900911e-03,
        1.07295826e-03, 2.39409349e-04, 1.96519161e-05],
       [2.39409349e-04, 2.91660295e-03, 1.30713076e-02, 2.15509428e-02,
        1.30713076e-02, 2.91660295e-03, 2.39409349e-04],
       [1.07295826e-03, 1.30713076e-02, 5.85815363e-02, 9.65846250e-02,
        5.85815363e-02, 1.30713076e-02, 1.07295826e-03],
       [1.76900911e-03, 2.15509428e-02, 9.65846250e-02, 1.59241126e-01,
        9.65846250e-02, 2.15509428e-02, 1.76900911e-03],
       [1.07295826e-03, 1.30713076e-02, 5.85815363e-02, 9.65846250e-02,
        5.85815363e-02, 1.30713076e-02, 1.07295826e-03],
       [2.39409349e-04, 2.91660295e-03, 1.30713076e-02, 2.15509428e-02,
        1.30713076e-02, 2.91660295e-03, 2.39409349e-04],
       [1.96519161e-05, 2.39409349e-04, 1.07295826e-03, 1.76900911e-03,
        1.07295826e-03, 2.39409349e-04, 1.96519161e-05]])
```

1.4(a)

이번에는 이미지 array와 filter를 입력받으면 convolution 연산을 수행한 filtered_array를 return 해주는 것이다. 먼저 입력 받은 array의 모든 element에 대해 convolution 연산을 수행하기 위해 zero-padding을 넣어준다. 그 후 convolution 연산을 쉽게 하기 위해 filter를 상하, 좌우로 두번 뒤집어주고 cross-correlation 연산을 해주면 된다

```python
def convolve2d(array, filter):
    # padding 사이즈 n 구하기
    n = int((filter.shape[0] - 1)/2)

    # numpy 함수를 이용하여 패딩 넣어주기
    padding_array = np.pad(array, ((n,n), (n,n)), "constant", constant_values = 0.)

    # Convolution -> Cross-correlation을 위해 filter 뒤집기
    flipped_filter = np.flip(np.flip(filter, axis=0), axis=1)

    # filter를 통과한 값을 저장하기 위한 array, 기존 이미지 array와 크기가 같음
    filtered_array = np.zeros((array.shape[0], array.shape[1]))
    filter_size = filter.shape[0]

    for y in range(array.shape[0]):
        for x in range(array.shape[1]):
            # Cross-correlation을 해준다
            filtered_array[y, x] = np.sum(padding_array[y:y+filter_size, x:x+filter_size]*flipped_filter)

    return filtered_array
```

1.4(b)

위에서 정의한 gauss2d 함수를 통해 2차원 gaussian filter를 얻고 이것을 convolve2d 함수를 통해 filtered_image를 얻으면 된다.

```
def gaussconvolve2d(array, sigma):
    # gaussian filter 얻기
    filter = gauss2d(sigma)
    # filter 적용
    filtered_array = convolve2d(array, filter)
    return filtered_array
```

1.4(c),(d)

이미지를 불러와 흑백으로 바꾸고 array화 시킨 후, sigma 값 3에 대해 gaussian filter를 적용시키고 이 값을 0~255 사이의 값으로 바꿔 다시 이미지화를 시키면 된다

```
def part1_4(img_name):
    img = Image.open(img_name)              # 이미지 불러오기
    grayscale_img = img.convert('L')        # grayscale로 변환
    img_array = np.asarray(grayscale_img)   # 이미지 -> array 로 변환

    filtered_array = gaussconvolve2d(img_array, 3)  # gaussian convolution
    filtered_array = filtered_array.astype('uint8') # float -> uint(0~255)로 바꾸기

    filtered_img = Image.fromarray(filtered_array)  # array -> 이미지 로 변환

    img.show()          # 원본 이미지
    filtered_img.show() # filter 적용 이미지

    #filtered_img.save(img_name[:-4]+"_gray_low.bpm", 'bmp')    # 저장
    return filtered_array
```

- sigma = 3

2. Hybrid images

2.1

이번에 할 것은 RGB 이미지에 Gaussian filter를 적용하여 low frequency 이미지를 얻는 것이다. RGB 이미지를 처리하기 위해 따로 더 특별한 방법이 있는 것은 아니고 R, G, B에 해당하는 3개의 channel로 분리한 후, 각각의 channel에 대해 gaussian filter를 적용하고 다시 하나의 이미지로 합치면 된다.

```
def part2_1(img_name, sig):
    sigma = sig   # 적절한 sigma 값 설정
    img = Image.open(img_name)   # 이미지를 불러온다

    # RGB channel을 분리한 후 filter 적용을 위해 array 형식으로 바꿔준다
    red, green, blue = map(np.asarray, img.split())

    # 각각의 channel에 convolution을 적용한 후, float 값을 uint(0~255) 값으로
    # 바꾸고 Image 형식으로 바꾼다
    red = Image.fromarray(gaussconvolve2d(red, sigma).astype('uint8'))
    green = Image.fromarray(gaussconvolve2d(green, sigma).astype('uint8'))
    blue = Image.fromarray(gaussconvolve2d(blue, sigma).astype('uint8'))

    # RGB 3개의 channel을 합쳐 하나의 이미지로 만든다
    low_freq_img = Image.merge('RGB', (red, green, blue))

    img.show()          # 원본 이미지
    low_freq_img.show() # RGB 사진에 filter를 적용한 이미지
    #low_freq_img.save(img_name[:-4]+"_low.bpm", 'bmp')

    low_freq_array = np.asarray(low_freq_img)   # return을 위한 low_freq_array

    return low_freq_array
```

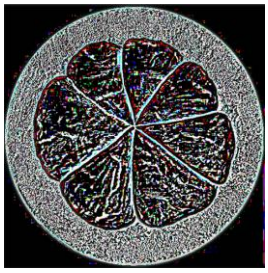
-sigma 값이 3인 경우

- sigma 값이 5인 경우

Sigma 값이 클수록 blur 효과가 더욱 증가하는 것을 알 수 있다.

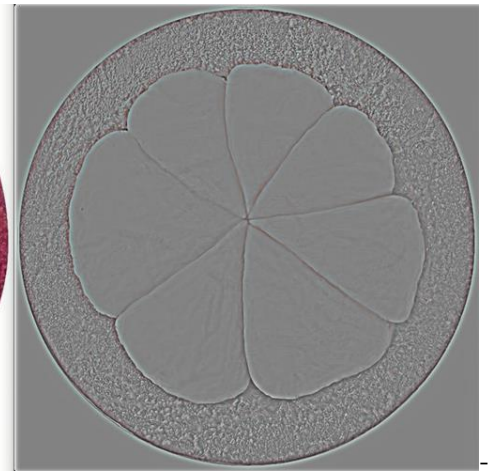
2.2

이번에는 high frequency 이미지를 얻는 것이다. High frequency 이미지는 original 이미지에서 low frequency 이미지를 빼면 된다.part2_1을 이용하여 low-frequency 이미지를 얻고 original 에서 뺀다. 이때 original과 low-frequency 의 intensity가 'uint8'로 되어 있어 1-5 를 해도 -4가 아니라 다른 0~255 사이의 값이 나온다 따라서 이 둘의 데이터 타입을 'int' 형으로 바꿔주고 계산을 진행한다. 그 후 가장 작은 intensity를 가지는 값을 찾아 그 값을 margin 값으로 사용하여 모든 element에 더해준다. 이러한 과정을 진행하면 이미지가 깨지는 현상을 막을 수 있다.



-깨진 이미지(음수 값을 바로 'uint8'로 바꾸어 생성됨)

```python
def part2_2(img_name, sigma, margin_on_off = True):
    original_img = Image.open(img_name)          # 이미지 오픈
    original_array = np.asarray(original_img)    # array 형식으로 바꾸기

    low_freq_array = part2_1(img_name, sigma)    # low-frequency array 얻기

    # high = original - low 인데 array의 값들이 uint8이므로 int16으로 바꾸어 음의 값도 나올 수 있게 한다
    high_freq_array = original_array.astype('int16') - low_freq_array.astype('int16')

    # 가장 낮은 intensity를 margin으로 설정한다
    # mragin을 넣어주지 않으면 음수 값이 존재하고 이를 uint8로 바꾸면 이미지가 이상해짐
    if(margin_on_off):
        margin = np.min(high_freq_array)

        high_freq_array = high_freq_array + margin  # margin 값을 넣어주어 0보다 크게 만들어 준다

        high_freq_array = high_freq_array.astype('uint8')   # uint 형으로 바꿔준다
        high_freq_img = Image.fromarray(high_freq_array)    # 이미지 로 바꿔준다

        high_freq_img.show()    # high-frequency 이미지
        #high_freq_img.save(img_name[:-4]+"_high.bpm", 'bmp')# 저장

    return high_freq_array
```

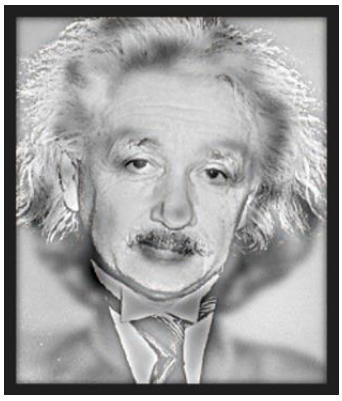
- sigma=3

2.3

마지막 파트는 hybrid 이미지를 만드는 것이다. Hybrid 이미지는 low-frequency 이미지와 high-frequency 이미지를 더하면 생성할 수 있다. Hybrid 이미지를 만들 때 high-frequency 이미지에는 low-frequency 이미지를 더해줄 것이기 때문에 margin 값을 더해주지 않아도 된다. 두 이미지를 더하고 난 후 여전히 남아있는 음수 값과 255를 초과하는 값은 각각 0과 255로 바꿔주면 된다.

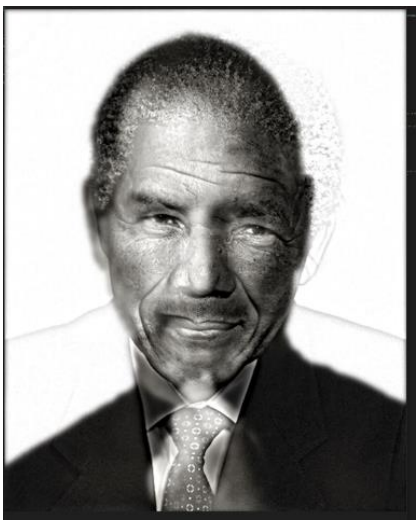
```python
def adjustValue(n):
    # 0보다 작은 값은 0으로 255보다 큰 값은 255로 바꾸기
    if n < 0:
        return 0
    elif n > 255:
        return 255
    else:
        return n
```

```python
def part2_3(low_img, high_img, low_sigma, high_sigma):
    low_freq_array = part2_1(low_img, low_sigma)
    high_freq_array = part2_2(high_img, high_sigma, False)   # 마진 추가 하지 말기


    hybrid_array = low_freq_array + high_freq_array # 두 이미지 더하기

    vectorized_func = np.vectorize(adjustValue)
    hybrid_array = vectorized_func(hybrid_array)    # 각 pixel의 intensity를 0~255 사이의 값으로 맞춰준다

    hybrid_array = hybrid_array.astype('uint8')     # Image로 바꾸기 위해 uint로 형 변환

    hybrid_img = Image.fromarray(hybrid_array)      # Image로 바꾸기
    hybrid_img.show()
    #hybrid_img.save("hybrid.bmp", 'bmp')    # 저장

    return hybrid_img
```



Low-frequency Image: Marilyn, sigma=3

High-frequency Image: eistein, sigma=3



Low-frequency Image: steve, sigma=3

High-frequency Image: mandela, sigma=5

Low-frequency Image: mangosteen, sigma=3

High-frequency Image: orange, sigma=3

Low-frequency Image: lion, sigma=3

High-frequency Image: tiger, sigma=3

3. 결론

이번 hw에서는 gaussin filter를 적용한 이미지를 만들어 볼 수 있었다. 이때 sigma 값이 클수록 blur 효과가 더욱 더 증가한다는 것을 알 수 있었고 high-frequency 이미지를 만들 때에도 sigma 값이 클수록 좀 더 선명한 high-frequency 이미지를 만들 수 있었다. 하지만 sigma 값이 너무 과하면 별로 좋지 않은 이미지가 얻어진다. 다라서 적정한 sigma 값이 필요 하였는데 일반적으로 sigma값이 3인 이미지가 가장 괜찮았다. 마지막으로 hybrid 이미지를 만들어 보았는데, hybrid 이미지를 만들 때 역시 적절한 sigma 값으로 조정해주어야 하고 괜찮은 hybrid 이미지를 만들기 위해서는 두 이미지의 적절한 위치 조정이나 크기 조절이 필요해 보였다. (샘플로 받은 이미지는 따로 조절하지 않아도 잘 맞아서 문제가 없었다)|

이렇게 python의 pillow library와 numpy를 이용하여 이미지에 blur 효과 및 detail 이미지를 얻고 이 둘을 활용하여 hybrid 이미지를 만들 수 있었다.