Lab 1: Image Processing	
Student ID: 22102304	Student Name: 김나현

Problem 1

A. Implement “smoothing.py”

```python
def create_gaussian_kernel(kernel_size=(3, 3), sigma=1.0):
    # ---------------------------------------------------------------------
    # Create Gaussian filter
    # Args:
    #     kernel_size (tuple): (2,)
    #     -- kernel_size[0]: filter height
    #     -- kernel_size[1]: filter width
    # Return:
    #     kernel (numpy.ndarray): (new_h, new_w, 3)
    # ---------------------------------------------------------------------

    # Fill this
    # 1. 커널의 중심점을 기준으로 x, y 좌표 그리드 생성
    center = kernel_size // 2
    # np.mgrid: 다차원 그리드 좌표를 생성하는 매우 효율적인 방법
    x, y = np.mgrid[-center:center+1, -center:center+1]

    # 2. 2D 가우시안 함수 적용
    # G(x, y) = (1 / (2 * pi * sigma^2)) * exp(-(x^2 + y^2) / (2 * sigma^2))
    exponent = -(x**2 + y**2) / (2 * sigma**2)
    kernel = np.exp(exponent) / (2 * np.pi * sigma**2)

    # 3. 커널의 모든 요소 합이 1이 되도록 정규화(Normalization)
    # 이렇게 해야 이미지 전체의 밝기가 변하지 않음
    kernel = kernel / np.sum(kernel)

    return kernel
```

```python
def padding(source, pad_size):
    # ---------------------------------------------------------------------
    # Pad zero to boundary
    # Args:
    #     source (numpy.ndarray): (H, W, 3)
    #     pad_size (int)
    # Return:
    #     target (numpy.ndarray): (H + 2 * pad_size, W + 2 * pad_size, 3)
    # ---------------------------------------------------------------------

    # 1. 원본 이미지의 높이, 너비, 채널 정보를 가져옴
    h, w, c = source.shape

    # 2. 패딩이 추가될 크기의 0으로 채워진 새 배열(틀)을 만듦
    #    (H + 2 * pad_size, W + 2 * pad_size, C)
    target = np.zeros((h + 2 * pad_size, w + 2 * pad_size, c), dtype=source.dtype)

    # 3. 새 배열의 중앙에 원본 이미지를 복사해 넣음
    target[pad_size : pad_size + h, pad_size : pad_size + w, :] = source

    return target
```

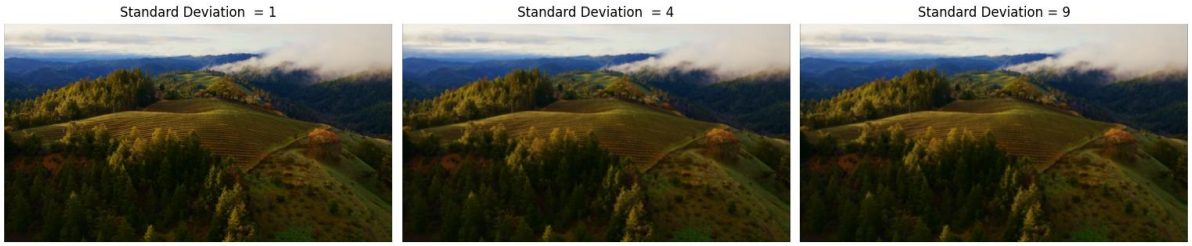
```python
def filtering(source, kernel):
    # Create zero padded images
    kernel_size = kernel.shape[0]
    pad_size = kernel_size // 2
    source_ = padding(source, pad_size)

    # 결과 이미지를 저장할 비어있는 행렬 생성
    target = np.zeros_like(source, dtype=np.float64)

    # 컨볼루션(Convolution) 연산 수행
    # 모든 픽셀 위치(i, j)와 모든 채널(c)에 대해 반복
    h, w, _ = source.shape
    for i in range(h):
        for j in range(w):
            for c in range(source.shape[2]):
                # 패딩된 이미지에서 커널 크기만큼의 영역(patch)을 잘라냄
                patch = source_[i : i + kernel_size, j : j + kernel_size, c]

                # patch와 kernel을 요소별로 곱한 뒤, 그 합을 결과 픽셀값으로 저장
                target[i, j, c] = np.sum(patch * kernel)

    # 이미지 저장을 위해 타입을 uint8로 변환
    # 0보다 작거나 255보다 큰 값들을 잘라냄(clipping)
    target = np.clip(target, 0, 255).astype(np.uint8)

    return target
```

B. Show the results with kernel size of 3, 7, 11. Answer how the size of the Gaussian kernel affects smoothing.

A. 커널 크기가 클수록 이미지가 더 많이 흐려진다. 또 다른 말로는 부드러워진다. 왜냐하면, 커널은 한 픽셀의 색상을 계산할 때 주변의 몇 개 픽셀을 참고할지 결정하는 것과 같다. 커널 크기가 크다는 것은 더 넓은 영역의 픽셀들을 평균내어 값을 정한다는 의미이다. 따라서 주변의 더 많은 픽셀 색상이 섞이게 되어 이미지의 세밀한 디테일이나 날카로운 경계가 사라지고 전체적으로 더 부드럽고 흐릿한 효과가 나타난다. Kernel size를 3,7,11로 했을 때의 결과는 다음과 같다.

C. the results with standard deviation of 1.0, 4.0, 9.0. Answer how the size of the Gaussian kernel affects smoothing

A. 가우시안 커널의 표준편차(Sigma) 값이 커질수록 이미지가 더 많이 흐려진다. 왜냐하면, 시그마는 가우시안 분포의 퍼짐 정도를 결정한다. 시그마가 작으면 그래프가 좁고 뾰족해진다. 이는 커널의 중심 픽셀에 높은 가중치를 부여하고, 주편 픽셀에는 아주 적은 가중치를 부여한다는 것이다. 따라서, 원본 픽셀의 정보가 많이 유지되어 스무딩 효과가 적다. 하지만 시그마가 크면 그래프가 넓고 평평해진다. 이는 커널 내의 주변 픽셀들에게도 비교적 높은 가중치를 부여해 색상을 골고루 섞는다는 것이다. 따라서, 더 강한 스무딩 효과가 나타난다. Standard deviation을 1,4,9로 했을 때의 결과는 다음과 같다.



Problem 2

A. Implement “upsampling.py”

```python
def upsample_nearest_neighbor(source, target_size):

    old_h, old_w, c = source.shape
    new_h, new_w = target_size

    # 새 이미지의 각 픽셀에 대응하는 원본 이미지의 좌표 계산
    y_coords, x_coords = np.mgrid[0:new_h, 0:new_w]

    # 좌표 스케일링
    scale_h = old_h / new_h
    scale_w = old_w / new_w
    old_y_coords = y_coords * scale_h
    old_x_coords = x_coords * scale_w

    # 가장 가까운 원본 픽셀을 찾기 위해 좌표 반올림
    nearest_y = np.round(old_y_coords).astype(np.int32)
    nearest_x = np.round(old_x_coords).astype(np.int32)

    # 계산된 좌표가 원본 이미지 크기를 벗어나지 않도록 조정
    nearest_y = np.clip(nearest_y, 0, old_h - 1)
    nearest_x = np.clip(nearest_x, 0, old_w - 1)

    # 계산된 좌표 배열을 이용해 원본 이미지에서 픽셀 값을 한 번에 가져옴
    target = source[nearest_y, nearest_x]

    return target
```

```python
def upsample_bilinear(source, target_size):
    # ------------------------------------------------------------------------
    # Implement the upsaming algorithm with bilienar interpolation
    # Args:
    #     source (numpy array): (old_h, old_w, 3)
    #     target_size (tuple): (2,)
    #     -- target_size[0]: new_h
    #     -- target_size[1]: new_w
    # Return:
    #     target (numpy.ndarray): (new_h, new_w, 3)
    # ------------------------------------------------------------------------

    old_h, old_w, c = source.shape
    new_h, new_w = target_size

    # 계산의 정확도를 위해 float 타입으로 빈 결과 이미지 생성
    target = np.zeros((new_h, new_w, c), dtype=np.float64)

    scale_h = old_h / new_h
    scale_w = old_w / new_w

    # 새 이미지의 모든 픽셀(y, x)에 대해 반복
    for y in range(new_h):
        for x in range(new_w):
            # 1. 원본 이미지에 대응하는 float 좌표 계산
            y_f = y * scale_h
            x_f = x * scale_w

            # 2. 보간에 사용할 4개의 주변 픽셀 좌표 계산
            y1 = int(np.floor(y_f))
            x1 = int(np.floor(x_f))
            y2 = min(y1 + 1, old_h - 1) # 경계값 처리
            x2 = min(x1 + 1, old_w - 1) # 경계값 처리

            # 3. 4개 픽셀 사이의 거리 비율(가중치) 계산
            dy = y_f - y1
            dx = x_f - x1

            # 4. 양선형 보간법 (Bilinear Interpolation) 수행
            # 4-1. 윗 줄 두 픽셀을 x축 방향으로 선형 보간
            top_val = (1 - dx) * source[y1, x1] + dx * source[y1, x2]
            # 4-2. 아랫 줄 두 픽셀을 x축 방향으로 선형 보간
            bottom_val = (1 - dx) * source[y2, x1] + dx * source[y2, x2]
            # 4-3. 위아래 두 보간 값을 y축 방향으로 선형 보간
            final_val = (1 - dy) * top_val + dy * bottom_val

            target[y, x] = final_val

    # 이미지 저장을 위해 타입을 uint8로 변환
    return np.clip(target, 0, 255).astype(np.uint8)
```

B. Show both results. Which interpolation method looks better?

A. 일반적으로 Bilinear Interpolation이 더 좋은 결과를 보여준다. Nearest Neighbor는 계산이 매우 빠르고 원본의 날카로운 경계를 유지하나, 이미지가 확대되면서 픽셀이 그대로 커진 듯한 계단 현상(aliasing)이 발생하고, 전체적으로 각지고 거칠어 보인다. 반면, Bilinear Interpolation은 주변 픽셀들의 색상을 섞어서 중간 값을 만들기 때문에 픽셀 사이의 경계가 매우 부드럽고 자연스럽다. 하지만, 색상을 평균내는 과정에서 원본의 날카로운 디테일이 약간 흐릿해(blurry) 보일 수 있다. 이 고양이 사진에서 각 보간법을 적용했을 때, Bilnear이 픽셀 사이의 경계가 매우 부드럽고 자연스러워 더 좋아 보인다.

Problem 3

A. Implement “downsampling.py”

Lab 1 - Image Processing (Solution) > Lab 1 - Image Processing > 03_downsampling.py > ...

```python
def downsample(source, target_size):
    """

    old_h, old_w, c = source.shape
    new_h, new_w = target_size

    # 1. 작은 이미지의 각 픽셀에 대응하는 원본 이미지의 좌표를 계산합니다.
    #    (upsample_nearest_neighbor와 완전히 동일한 로직)
    y_coords, x_coords = np.mgrid[0:new_h, 0:new_w]

    # 2. 좌표 스케일링
    scale_h = old_h / new_h
    scale_w = old_w / new_w
    old_y_coords = y_coords * scale_h
    old_x_coords = x_coords * scale_w

    # 3. 가장 가까운 원본 픽셀을 찾기 위해 좌표를 반올림합니다.
    nearest_y = np.round(old_y_coords).astype(np.int32)
    nearest_x = np.round(old_x_coords).astype(np.int32)

    # 4. 좌표가 원본 이미지 크기를 벗어나지 않도록 조정합니다.
    nearest_y = np.clip(nearest_y, 0, old_h - 1)
    nearest_x = np.clip(nearest_x, 0, old_w - 1)

    # 5. 계산된 좌표 배열을 이용해 원본 이미지에서 픽셀 값을 가져와 작은 이미지를 생성합니다.
    target = source[nearest_y, nearest_x]

    return target
```

B. Show the result

Problem 4

A. Implement “gaussian_pyramid.py”

```python
def create_gaussian_pyramid(source, target_size, kernel_size, sigma):
    targets = [source]

    kernel = create_gaussian_kernel(kernel_size, sigma)

    current_image = source

    while current_image.shape[0] > target_size[0] and current_image.shape[1] > target_size[1]:

        smoothed_image = filtering(current_image, kernel)

        downsampled_image = smoothed_image[::2, ::2]

        targets.append(downsampled_image)
        current_image = downsampled_image

    return targets
```

B. Show the result

C. Compare the coarsest result of Gaussian pyramid with the result of the problem 3. Which result looks better? Answer the reason.

A. 가우시안 피라미드의 결과가 더 좋아보인다. 이유는 Anti-aliasing 처리 유무의 차이 때문이다. 3번 문제(downsampling)는 원본에서 특정 간격의 픽셀만 '골라내는(subsampling)' 방식이다. 이 과정에서 골라지지 않은 픽셀들의 정보는 모두 버려지기 때문에 이미지의 세밀한 디테일이 깨지거나 왜곡되는 에일리어싱(Aliasing) 현상이 발생한다. 결과적으로 이미지가 거칠고, 원래 없던 패턴이 생기거나 계단 현상이 두드러져 보인다. 하지만 가우시안 피라미드는 이미지를 다운샘플링 하기 전에 반드시 가우시안 블러로 스무딩하는 과정을 거친다. 이 스무딩 과정이 바로 안티에일리어싱 역할을 한다. 픽셀을 버리기 전에 주변 픽셀들의 정보를 평균 내어, 갑작스러운 색상 변화(고주파 성분)를 미리 제거한다. 이로 인해 픽셀 정보 손실을 최소화하여 훨씬 부드럽고 자연스러운 축소 이미지를 얻을 수 있다. 따라서 가우시안 피라미드의 결과가 더 좋아보인다.