

Coding part :

```
class ConvolutionType(Enum):
    MeanFilter = 1
    Sobel_X = 2
    Sobel_Y = 3
    Laplace = 4

class UnsharpService:
    def __init__(self, threshold):
        self.__threshold = threshold

        self.__mean_mask = [1, 1, 1,
                            1, 1, 1,
                            1, 1, 1]

        self.__sobelx_mask = [-1, 0, 1,
                             -2, 0, 2,
                             -1, 0, 1]

        self.__sobely_mask = [-1, -2, -1,
                             0, 0, 0,
                             1, 2, 1]

        self.__laplace_mask = [-1, -1, -1,
                             -1, 8, -1,
                             -1, -1, -1]
```

```
def averageBlur(self, img):
    rows = img.shape[0]
    cols = img.shape[1]
    mean_img = np.zeros((rows, cols), dtype=img.dtype)
    self.__convolution(img, mean_img, self.__mean_mask,
ConvolutionType.MeanFilter)
    return mean_img
```

這個方法通過對影像的每個像素點周圍的像素值取平均來實現均值過濾。均值過濾是一種簡單的影像平滑技術，用於減少影像噪聲。在實際應用中，這可以使影像看起來更柔和，但可能會使影像細節變得模糊。這個方法使用 `__convolution` 函數來在影像上滑動一個 3x3 的卷積

核（所有元素都是 1），並對核覆蓋的區域進行平均，進行邊緣檢測前，有助於減少由噪聲引起的假邊緣。

```
def __sobelx(self, src, out):
    self.__convolution(src, out, self.__sobelx_mask,
ConvolutionType.Sobel_X)

def __sobely(self, src, out):
    self.__convolution(src, out, self.__sobely_mask,
ConvolutionType.Sobel_Y)
```

應用 Sobel 運算子來計算影像的水平和垂直邊緣，有效地捕捉影像中的邊緣方向和邊緣強度，用於特徵提取，在物體識別、影像追蹤。

```
def firstOrderEdge(self, img):
    rows = img.shape[0]
    cols = img.shape[1]
    sobely_img = np.zeros((rows, cols), dtype=img.dtype)
    sobelx_img = np.zeros((rows, cols), dtype=img.dtype)
    diff_1_img = np.zeros((rows, cols), dtype=img.dtype)

    self.__sobely(img, sobely_img) # sobel y
    self.__sobelx(img, sobelx_img) # sobel x

    for i in range(1, rows - 1):
        for j in range(1, cols - 1):
            sobel = int(sobelx_img[i, j]) + int(sobely_img[i, j])
            diff_1_img[i, j] = self.__check(sobel)

    # 對一階微分結果進行均值濾波
    mean_diff_1_img = self.averageBlur(diff_1_img) # unsharp mask

    return mean_diff_1_img
```

此方法結合了水平和垂直的 Sobel 運算結果，獲得更全面的邊緣影像。將這兩個方向的結果相加，獲得邊緣的總體強度，之後再使用均值過濾來平滑一階微分的結果，這有助於減少由於邊緣檢測引起的噪聲。

```
def secondOrderEdge(self, img):
    rows = img.shape[0]
    cols = img.shape[1]

    diff_2_img = np.zeros((rows, cols), dtype=img.dtype)
```

```

        self.__laplace(img, diff_2_img)

    return diff_2_img

def __laplace(self, src, out):
    self.__convolution(src, out, self.__laplace_mask,
ConvolutionType.Laplace)

```

這個方法通過應用拉普拉斯運算子來執行影像的二階微分，是一種更敏感的邊緣檢測方法。在某些應用中，拉普拉斯運算子可以幫助識別細小的細節或對比較低的邊緣進行更精確的檢測，由於其對噪聲的高敏感性，這種方法通常需要在相對平滑的影像上進行。

```

def __convolution(self, src, out, mask, conv_type):
    rows = out.shape[0]
    cols = out.shape[1]

    for i in range(1, rows - 1):
        for j in range(1, cols - 1):
            conv = mask[0] * src[i - 1, j - 1] + \
                   mask[1] * src[i - 1, j] + \
                   mask[2] * src[i - 1, j + 1] + \
                   mask[3] * src[i, j - 1] + \
                   mask[4] * src[i, j] + \
                   mask[5] * src[i, j + 1] + \
                   mask[6] * src[i + 1, j - 1] + \
                   mask[7] * src[i + 1, j] + \
                   mask[8] * src[i + 1, j + 1]

            if conv_type == ConvolutionType.MeanFilter:
                conv = conv / 9
                conv = 255 if conv > self.__threshold else 0
            elif conv_type == ConvolutionType.Sobel_X or conv_type ==
ConvolutionType.Sobel_Y:
                conv = abs(conv)

            out[i, j] = self.__check(conv)

```

此方法實際執行卷積操作，對每個像素周圍的值進行加權求和，從而改變該像素的值。這是所有上述過濾操作的基礎，包括均值過濾、Sobel 運算和拉普拉斯運算。這些方法和運算子結合使用可以實現從基本的影像平滑到複雜的邊緣檢測和特徵提取等多種影像處理任

```
img_path = 'IMG_1.PNG'
```

```
threshold = 128
input_img = cv2.imread(img_path)
cv2.imshow(input_img)

gray_img = cv2.cvtColor(input_img, cv2.COLOR_BGR2GRAY)

service = UnsharpService(threshold=threshold)
diff_1_img = service.firstOrderEdge(gray_img)
diff_2_img = service.secondOrderEdge(gray_img)
```

執行程式碼

```
import matplotlib.pyplot as plt

# 設定畫布大小
plt.figure(figsize=(18, 6))

# 顯示灰度圖像
plt.subplot(1, 3, 1)
plt.imshow(gray_img, cmap='gray')
plt.title('Gray Image')

# 顯示一階微分結果
plt.subplot(1, 3, 2)
plt.imshow(diff_1_img, cmap='gray')
plt.title('firstOrder')

# 顯示二階微分結果
plt.subplot(1, 3, 3)
plt.imshow(diff_2_img, cmap='gray')
plt.title('secondOrder')
plt.axis('off')
plt.show()

# 確保兩張圖片的尺寸相同
if gray_img.shape != diff_1_img.shape:
    print(f'{gray_img.shape}/{diff_1_img.shape}')
    raise ValueError("The images must have the same size to be blended.")
if gray_img.shape != diff_2_img.shape:
    print(f'{gray_img.shape}/{diff_2_img.shape}')
```

```

raise ValueError("The images2 must have the same size to be blended.")

# 將兩張圖片疊加
alpha = 0.5 # 第一張圖片的權重
beta = 0.5 # 第二張圖片的權重
blended_img1 = cv2.addWeighted(gray_img, alpha, diff_1_img, beta, 0)
blended_img2 = cv2.addWeighted(gray_img, alpha, diff_2_img, beta, 0)

plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.imshow(gray_img, cmap='gray')
plt.title('Gray Image')

# 顯示 Sobel 結果
plt.subplot(1, 2, 2)
plt.imshow(blended_img1, cmap='gray')
plt.title('Sobel')
plt.axis('off') # 隱藏坐標軸
plt.show()

plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.imshow(gray_img, cmap='gray')
plt.title('Gray Image')

# 顯示 Laplacian 結果
plt.subplot(1, 2, 2)
plt.imshow(blended_img2, cmap='gray')
plt.title('Laplacian')
plt.axis('off') # 隱藏坐標軸
plt.show()

```

這部分程式碼使用 Matplotlib 和 OpenCV 庫來展示和比較原始灰階圖像、一階微分（Sobel 運算子結果）和二階微分（Laplacian 運算子結果）的視覺效果。首先，它在一個畫布上顯示三種圖像以直觀展示不同處理效果。接著，程式檢查三張圖片尺寸是否一致，並將一階微分和二階微分的圖像各自與原始灰階圖像進行疊加，得到兩種混合圖像，這樣可以在視覺上比較增強前後的對比。最後，這些混合圖像也被顯示出來，以展示邊緣增強的效果。

```

def calculate_psnr(img1, img2):
    mse = np.mean((img1 - img2) ** 2)
    if mse == 0:
        return float('inf') # 避免除以零的情況，如果兩張圖片完全相同

```

```
pixel_max = 255.0
psnr = 20 * np.log10(pixel_max / np.sqrt(mse))
return psnr

# 計算 PSNR
psnr_value1 = calculate_psnr(gray_img, blended_img1)
print(f"PSNR Value between gray_img and blended_img1 is {psnr_value1} dB")
psnr_value2 = calculate_psnr(gray_img, blended_img2)
print(f"PSNR Value between gray_img and blended_img2 is {psnr_value2} dB")
print(f"PSNR1 - PSNR2 = {psnr_value1-psnr_value2}")
```

這段程式碼計算兩張圖片之間的峰值信噪比（PSNR），並使用這個函數來評估原始灰階圖像與其經過邊緣增強處理後的圖像（使用 Sobel 和 Laplacian 運算子生成的混合圖像）之間的差異。

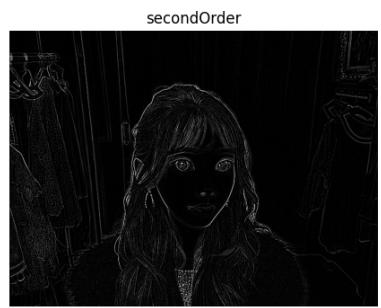
Result:

圖組一：



←原圖

PSNR1 is 27.367793614735664 dB
PSNR2 is 27.367793614735664 dB
PSNR1 - PSNR2 = -0.22510835076186808

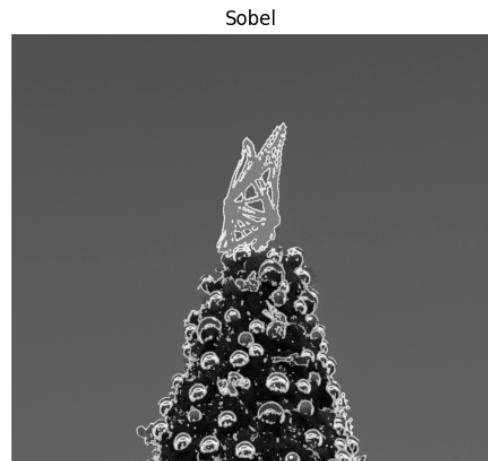
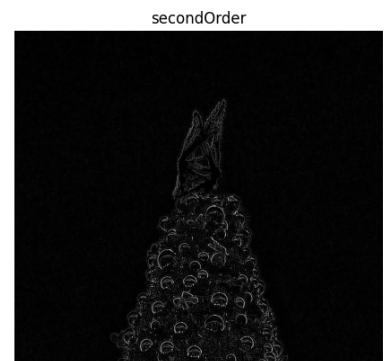
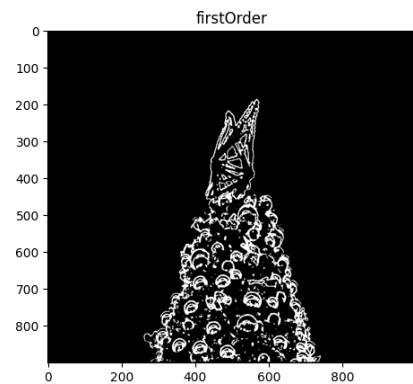
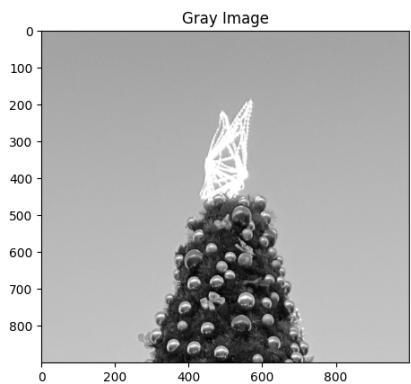


圖組二：



←原圖

PSNR1 is 26.93794727616679 dB
PSNR 2 is 27.102136324688672 dB
PSNR1 - PSNR2 = -0.16418904852188376

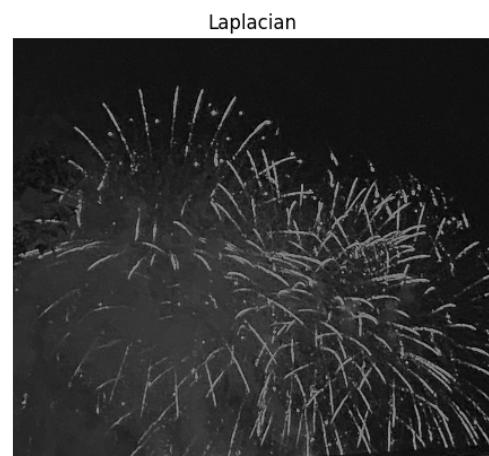
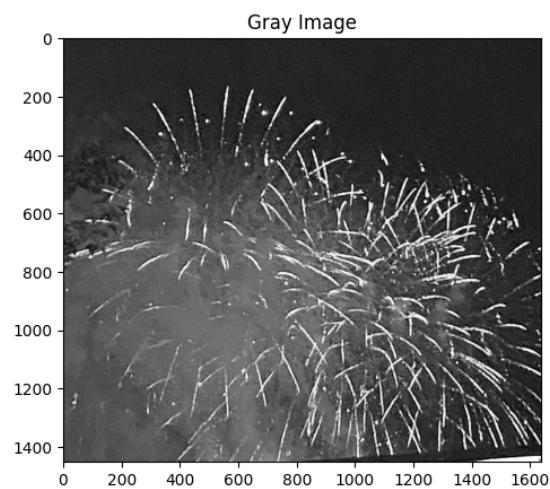
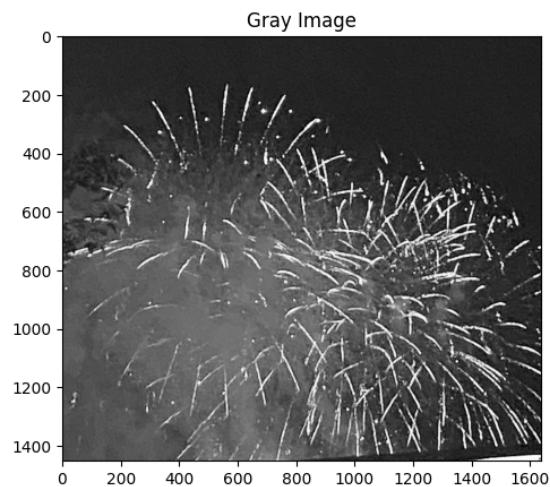
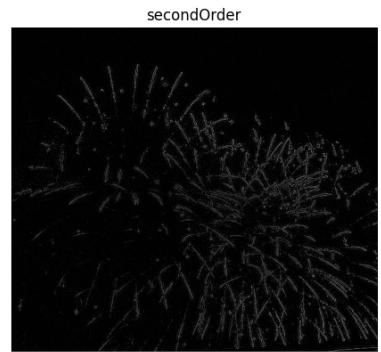
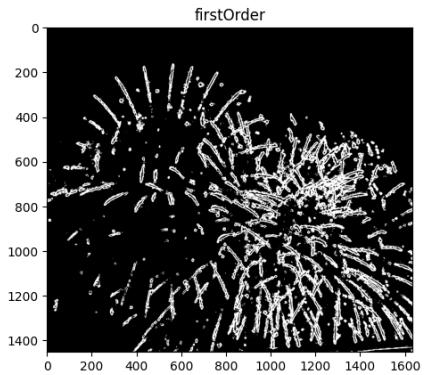
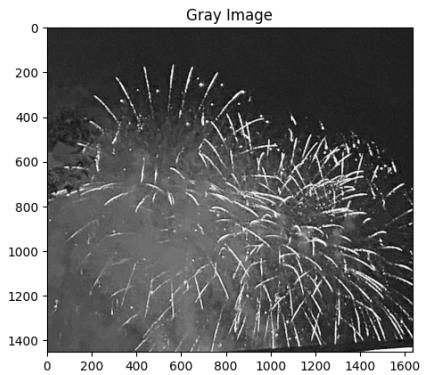


圖組三：



←原圖

PSNR1 is 27.44758181881045 dB
PSNR2 is 27.45842072365708 dB
PSNR1 - PSNR2 = -0.010838904846632857

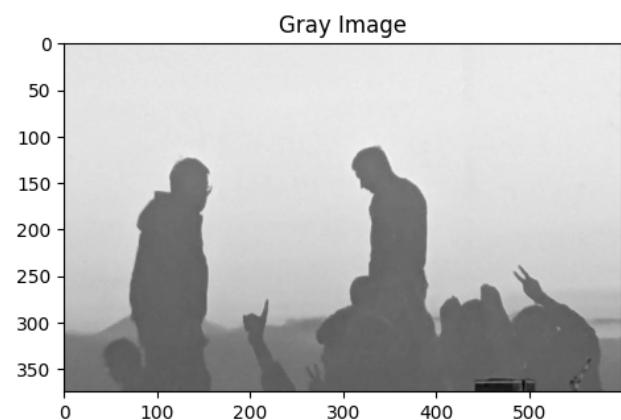
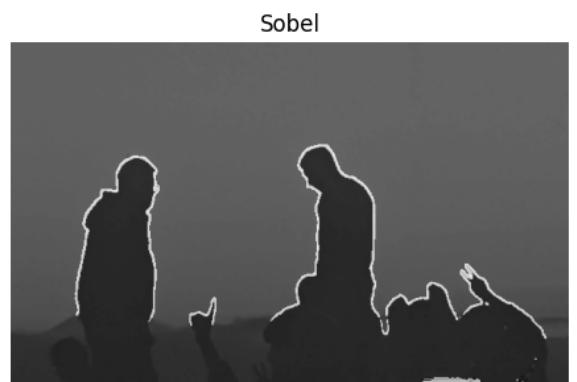
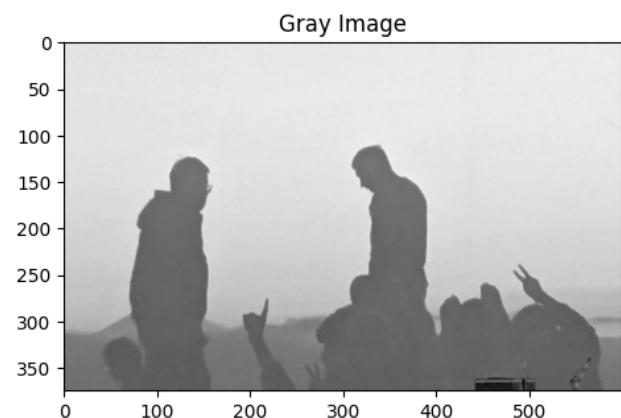
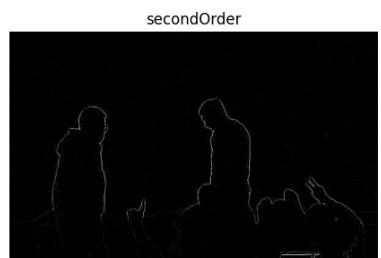
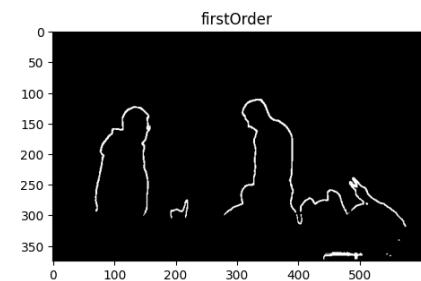
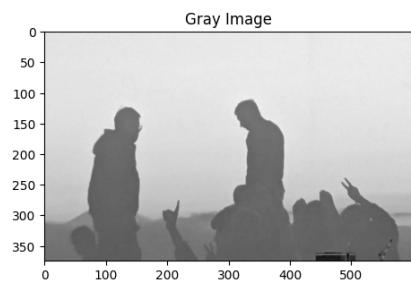


圖組四：



PSNR1 is 27.280030925183674 dB
PSNR2 is 27.32931860817201 dB
PSNR1 - PSNR2 = -0.049287682988335746

←原圖



Discussion:

1. 原始灰階圖像、一階微分(Sobel)圖像和二階微分(拉普拉斯)圖像有著明顯的差異，Sobel 圖像可以增強物體的輪廓和邊緣，而拉普拉斯圖像對細節和高頻噪聲更加敏感。將原始圖像和處理後的圖像進行疊加，可以清晰地看到邊緣增強的效果。
2. 計算峰值信噪比(PSNR)可以量化原始圖像和處理後圖像之間的差異。根據四組圖組結果，Sobel 處理後的圖像與原圖的 PSNR 值通常略低於拉普拉斯處理後的圖像，Sobel 只提取邊緣方向的梯度訊息，主要強調邊緣的方向和強度，其失真度較大，PSNR 小。而拉普拉斯算子不僅捕捉邊緣，還能檢測到圖像中的細節和高頻變化，因為它對圖像中所有突變的靈敏度較高，包括細節和噪聲，使其 PSNR 較大。
3. 除了使用 PSNR 作為客觀指標外，還可以探索其他評估指標，如結構相似性(SSIM)等，甚至嘗試使用人工標註的地圖真實邊緣進行評估。
4. 在執行邊緣檢測之前，可以嘗試其他預處理技術，如直方圖均衡化增強圖像對比度和細節，而在檢測完成後，也可以嘗試不同的後處理方法，如邊緣連接、閾值處理等去進一步改善結果。

補做的部分

Coding part :

```
img_path = 'image04.jpeg'
threshold = 128
input_img = cv2.imread(img_path)
cv2.imshow(input_img)

gray_img = cv2.cvtColor(input_img, cv2.COLOR_BGR2GRAY)

service = UnsharpService(threshold=threshold)
diff_2_img = service.secondOrderEdge(gray_img)

# 將兩張圖片疊加
alpha = 1 # 第一張圖片的權重
beta = 1 # 第二張圖片的權重
blended_img1 = cv2.addWeighted(gray_img, alpha, diff_2_img, beta, 0)

diff_1_img = service.firstOrderEdge(gray_img)
blur_img = service.averageBlur(diff_1_img)

# 將影像正規化到 0.0 到 1.0 之間
normalized_image = blur_img / np.max(blur_img)
multi_images1 = multiply_images(normalized_image, diff_2_img)
multi_images2 = multiply_images(normalized_image, blended_img1)

result1 = cv2.addWeighted(multi_images1, alpha, gray_img, beta, 0)
result2 = cv2.addWeighted(multi_images2, alpha, gray_img, beta, 0)
```

依照講義的步驟實作，

0.灰階原圖經過 1.Laplacian Mask，2.與灰階圖相加

3.將灰階圖用一階微分(Sobel)實作，經過 4.mean filter 模糊處理，

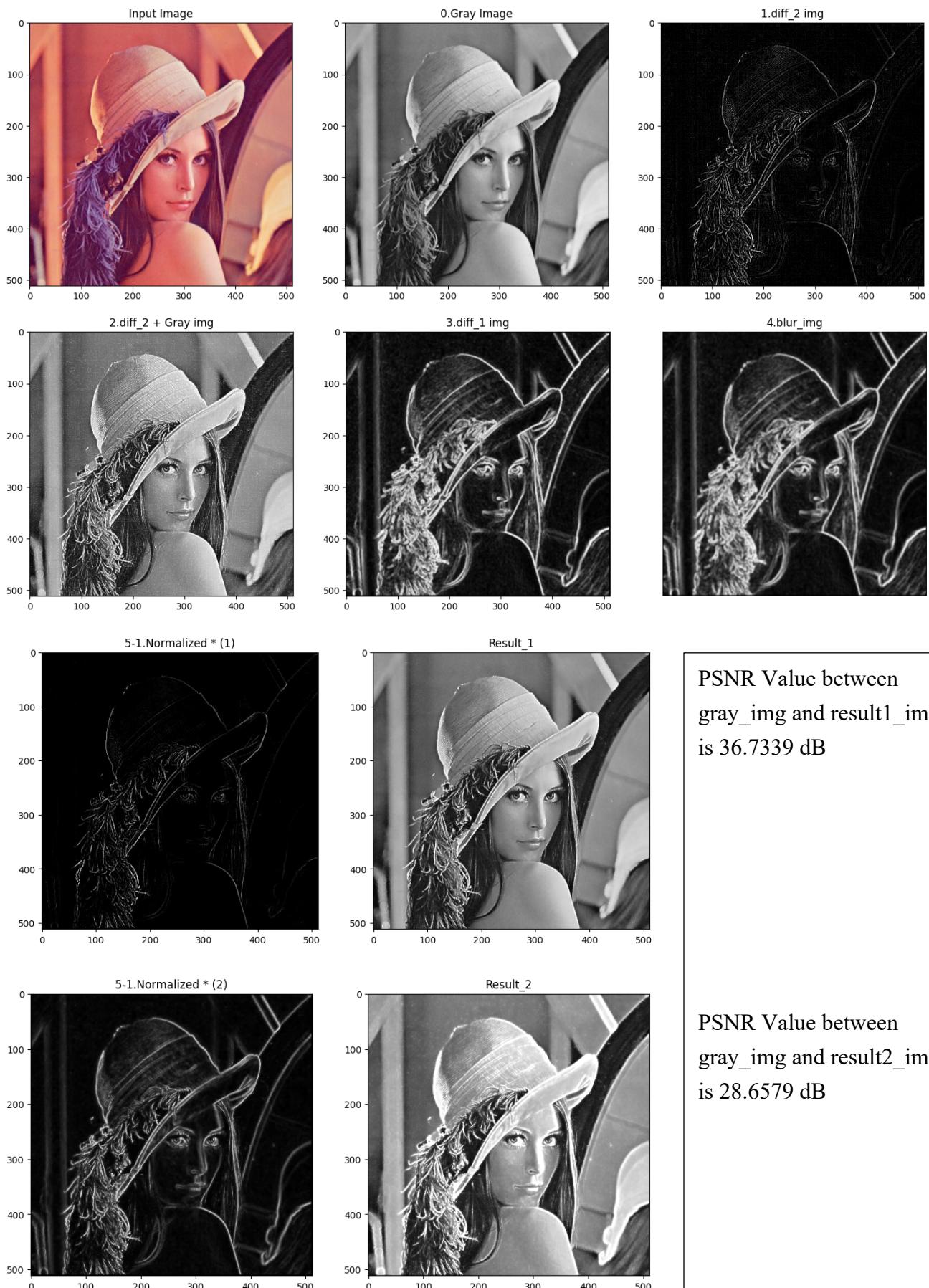
5.1 將 4 正規化後乘上 1，加上原始灰階圖 → Result1

5.2 將 4 正規化後乘上 2，加上原始灰階圖 → Result2

最後得到 2 種銳化的圖片！

圖組一、

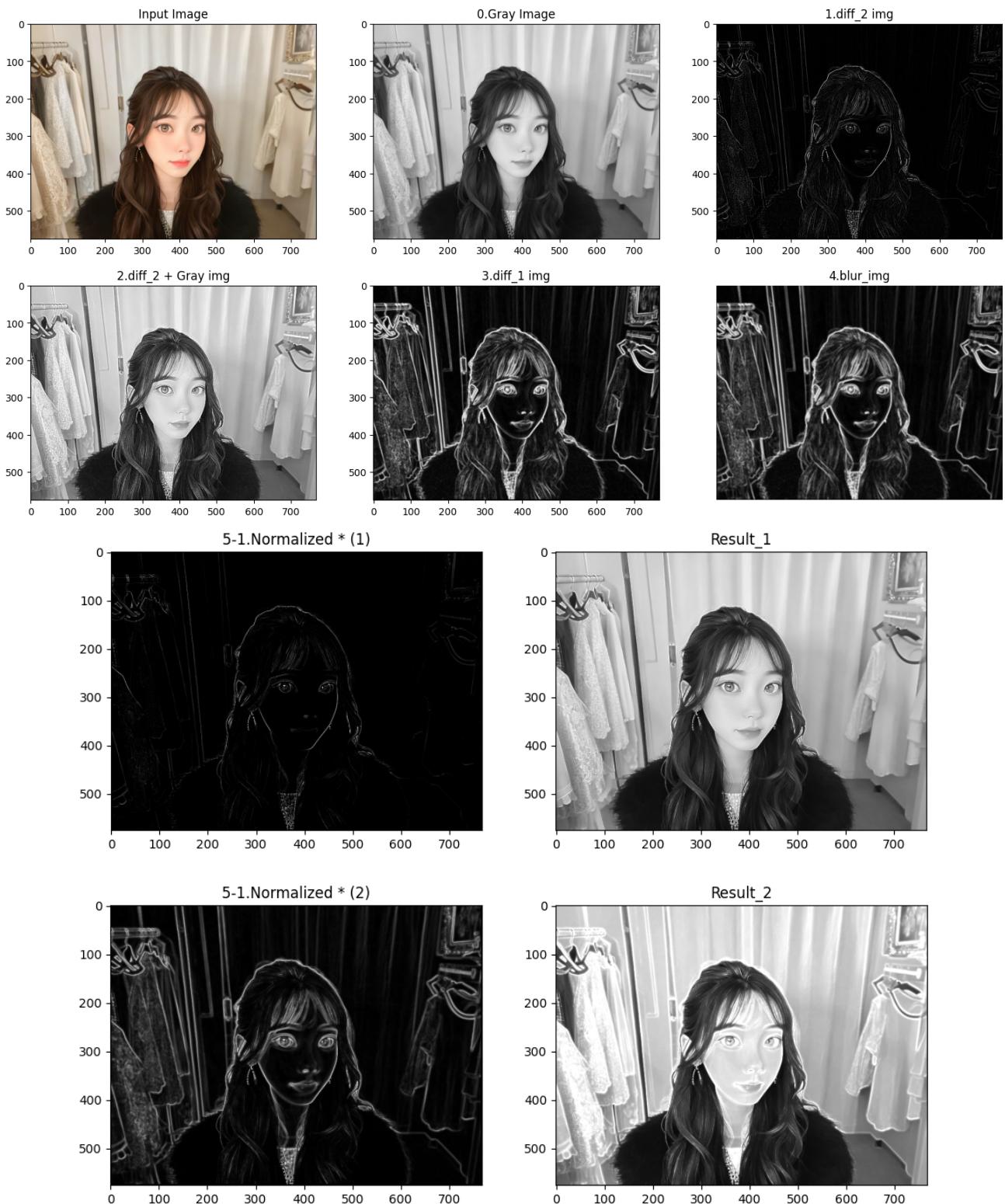
Result:



PSNR Value between
gray_img and result1_img
is 36.7339 dB

PSNR Value between
gray_img and result2_img
is 28.6579 dB

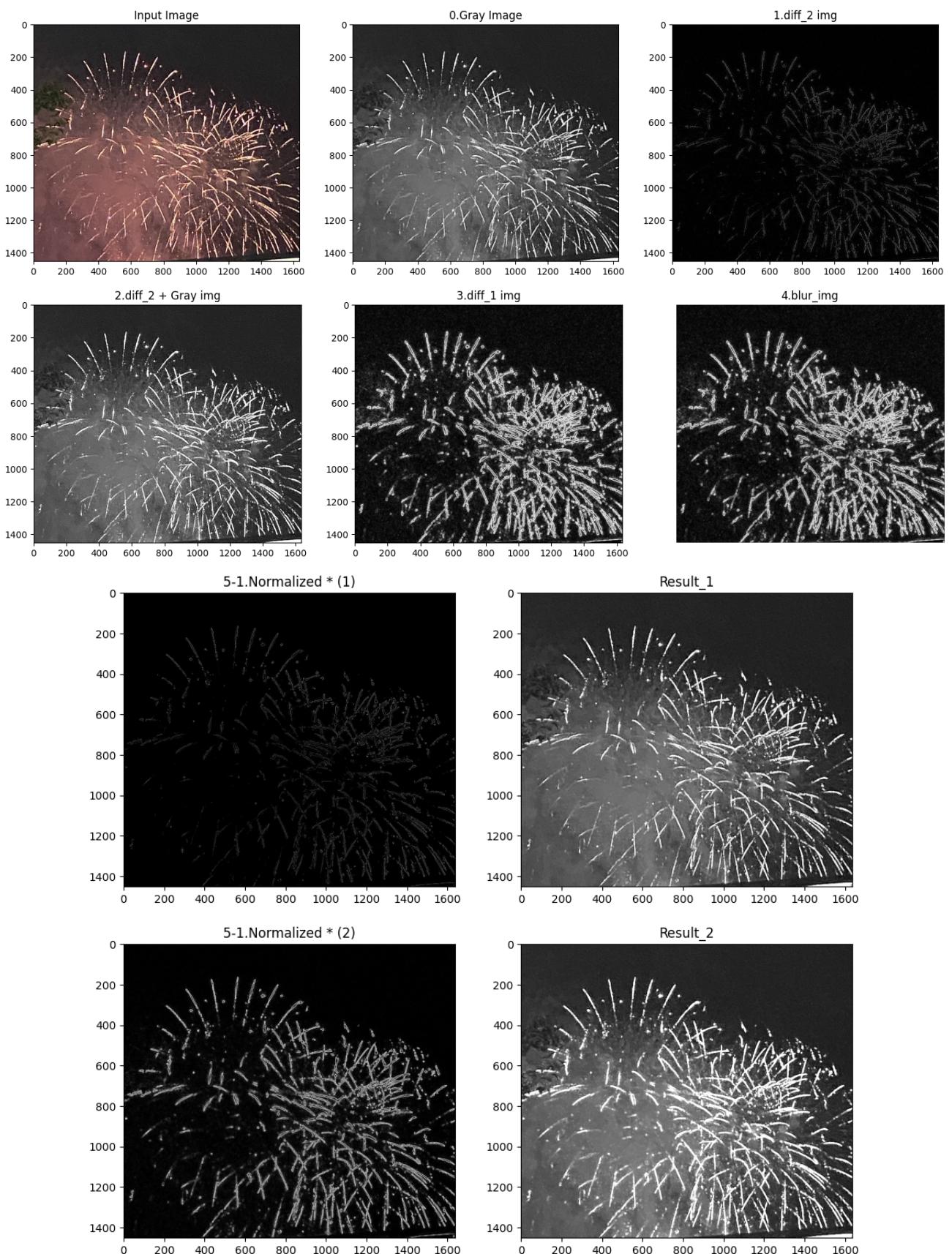
圖組二、



PSNR Value between gray_img and result1_img is 37.8381 dB

PSNR Value between gray_img and result2_img is 29.538 dB

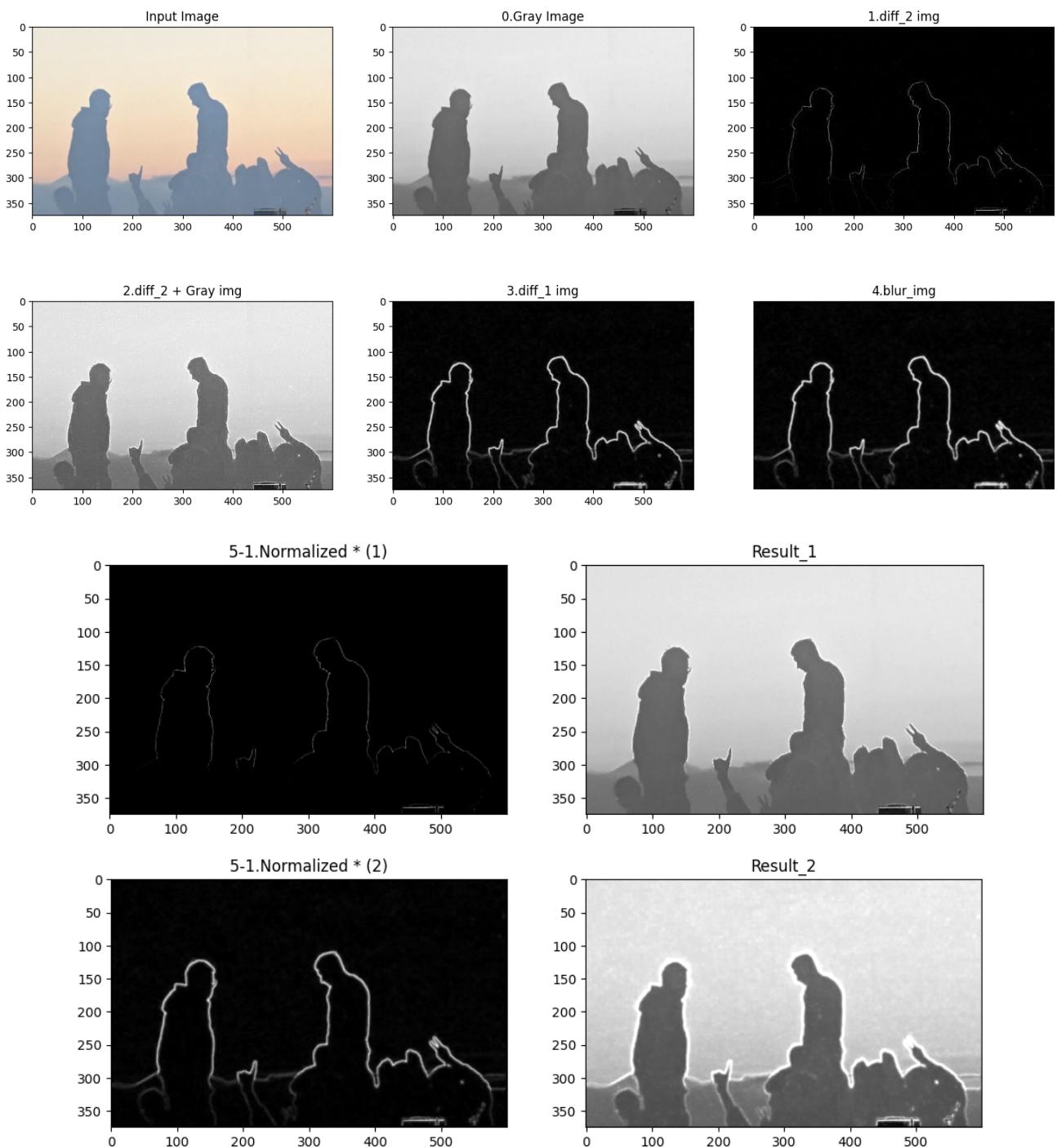
圖組三、



PSNR Value between gray_img and result1_img is 37.9922 dB

PSNR Value between gray_img and result2_img is 30.6431 dB

圖組四、



PSNR Value between gray_img and result1_img is 45.628 dB

PSNR Value between gray_img and result2_img is 33.7243 dB

作業描述：

這份作業要實作 Unsharp Masking 的圖像銳化方法，主要步驟包括對灰階圖進行 Laplacian Mask 及 Sobel 一階微分，將一階微分結果模糊處理並正規化，分別與 Laplacian Mask 結果及其與灰階圖相加的結果相乘，最後與原始灰階圖相加得到兩種不同程度銳化的結果。

Discussion 補充：

1. 在所有圖組中，Result_1 的 PSNR 值都明顯高於 Result_2，表示 Result_1 的銳化效果相對較溫和，與原始灰階圖的差異較小，而 Result_2 的 PSNR 值較低說明其銳化效果更強烈，與原圖差異更大。
2. 從肉眼視覺上觀察，Result_2 的銳化效果確實比 Result_1 更加明顯，邊緣和細節在 Result_2 中更加突出和清晰，這與 PSNR 值的結果是一致的。
3. 造成 Result_1 和 Result_2 差異的主要在第 5 步，Result_1 是將正規化後的模糊圖乘上 Laplacian Mask 的結果，而 Result_2 是將正規化後的模糊圖乘上 Laplacian Mask 與灰階圖相加的結果，原來乘上相加後的結果導致 Result_2 的銳化效果更加強烈。
4. 最後統整，在選擇 Result_1 還是 Result_2 時，可以根據實際應用的需求來決定。

希望得到較為自然、細微的銳化效果 \mapsto Result_1

需要更強烈的邊緣增強和細節突出 \mapsto Result_2