高等數位影像處理

作業#(4)

姓名：闕楷宸
學號：114318047
指導老師：黃正民

作業說明:

作業架構:

```
├── code
│   ├── CMakeLists.txt
│   ├── flower_512x384.raw
│   ├── HW4.cpp
│   └── parthenon_600x400.raw
├── output image png
│   ├── flower_enhanced.png
│   ├── parthenon_gaussian3x3.png
│   ├── parthenon_gaussian5x5.png
│   ├── parthenon_laplacian_4neighbor_sharpened.png
│   ├── parthenon_laplacian_8neighbor_sharpened.png
│   ├── parthenon_median3x3.png
│   ├── parthenon_median5x5.png
│   ├── parthenon_sobel_sharpened.png
│   ├── parthenon_unsharp_mask.png
│   ├── ROI.png
│   ├── sobel_0_dege.png
│   ├── sobel_90_edge.png
│   ├── sobel_combined_edge.png
│   └── unsharp_mask.png
├── report
│   └── HW4_ADIP.pdf
└── window user
    ├── flower_512x384.raw
    ├── HW4.exe
    └── parthenon_600x400.raw

4 directories, 22 files
```

主程式HW4.cpp

```
----- Homework 4 Menu -----

================ Results Menu ================
 1) task_1_1_i
 2) task_1_1_ii
 3) task_1_1_iii
 4) task_1_2_a
 5) task_1_2_b_i
 6) task_1_2_b_ii
 7) task_1_2_b_iii
 8) task_1_2_c_i
 9) task_1_2_c_ii
 10)task_1_2_c_iii
 11)task_2_1
 12)task_2_2
 0) Exit
Enter the question number:
```

輸入0~12即可輸出結果

建置執行(window,linux)

Linux:

1.使用opencv 4.5.4(sudo apt install libopencv-dev)

2.建置與執行:

```
cd HW#4_114318047/code/
cmake -S . -B build
cmake --build build -j
./build/HW4
```

Window:

```
cd window_user
./HW4.exe
```

Window 建置:

1:Cmake:

```
winget install Kitware.CMake
```

2:OpenCV for window:

C:\opencv\build\x64\vc16\bin(make sure DLLs are in this dir)

3:Confirm CMake config file exists:

C:\opencv\build\x64\vc16\lib\OpenCVConfig.cmake

4:Build and run

```
cd HW#4_114318047\code
```

5:Configure:

```
cmake -S . -B build -G "Visual Studio 17 2022" -A x64
-DOpenCV_DIR="C:\opencv\build\x64\vc16\lib"
```

6:Produce exe:

```
cmake --build .\build --config Release
```

7:Run exe

.\build\Release\HW4.exe

1-1-i:Edge detection

Gaussian smoothing filters with kernel sizes of 3x3 and 5x5 to reduce the salt-and-pepper noise,and save the results as parthenon_gauusian3x3.raw and parthenon_gaussian5x5.raw.

Image with Gaussian 3x3 smoothing filer(parthenon_gaussian3x3.png)

Image with Gaussian 5x5 smoothing filer(parthenon_gaussian5x5.png)

Discussion(1-1-i)

此題使用兩種不同kernel size的Gaussian smoothing filter進行處理，在程式撰寫上我使用此function”**task_1_1_i**”，並呼叫”**apply_gaussian_filter**” 與 ”**apply_replication_padding**”進行處理。

apply_gaussian_filter:

由於每一題皆需要進行replication padding，因此我呼叫**apply_replication_padding**進行padding。
padding 的第一步是先判斷是要padding多少，在使用3x3 mask時需要往外添加一個pixel，5x5則需要往外添加兩個pixel，第二步則是將原始圖像的pixel值assign至padded_image中:

```
    for (int r = 0; r < height; ++r) {
        for (int c = 0; c < width; ++c) {
            padded_input[(r + padding) * padded_width + (c + padding)] = input[r * width + c];
        }
    }
```

第三步則是在padded_image中進行padding，作法則是分別在padded_image的上、下、左、右，以及圖像的角落，及角落的padding方法則是使用離角落最近的pixel進行padding

```
    // Replicate borders
    for (int c = 0; c < width; ++c) {
        for (int p = 0; p < padding; ++p) {
            padded_input[p * padded_width + (c + padding)] = input[c]; // Top rows
            padded_input[(height + padding + p) * padded_width + (c + padding)] = input[(height - 1) * width + c]; // Bottom rows
        }
    }
    for (int r = 0; r < height; ++r) {
        for (int p = 0; p < padding; ++p) {
            padded_input[(r + padding) * padded_width + p] = input[r * width]; // Left columns
            padded_input[(r + padding) * padded_width + (width + padding + p)] = input[r * width + width - 1]; // Right columns
        }
    }

    // Corners
    for (int p = 0; p < padding; ++p) {
        for (int q = 0; q < padding; ++q) {
            padded_input[p * padded_width + q] = input[0]; // Top-left
```

```
            padded_input[p * padded_width + (width + padding +
q)] = input[width - 1]; // Top-right
            padded_input[(height + padding + p) * padded_width +
q] = input[(height - 1) * width]; // Bottom-left
            padded_input[(height + padding + p) * padded_width +
(width + padding + q)] = input[(height - 1) * width + width -
1]; // Bottom-right
        }
    }
```

最後在輸出圖像中可以看出在進行3x3或5x5 Gaussian smoothing filter後，salt-and-pepper有改善，但相對的圖像也變模糊。

原圖

3x3 Gaussian

5x5 Gaussian

1-1-ii:Edge detection

Then,repeat the same procedure using median filters with the same kernel sizes

Image with 3x3 median filter(parthenon_median3x3.png)

Image with 5x5 median filter(parthenon_median5x5.png)

Discussion (1-1-ii)

此題作法大致與上題類似，都會使用3x3及5x5的mask進行處理，並且都會先進行padding的動作。

使用的function則是”**task_1_1_ii**”以及”**apply_median_filter**” 以及”**apply_replication_padding**”，在”**apply_median_filter**”中則是分次將mask的值append至neighborhood的vector中，並且進行使用std::sort進行sorting，並且最後在取中位數的值(neighborhood[(neighborhood.size() /2)])放入mask的中心位置

```
    for (int r = padding; r < height + padding; ++r) {
        for (int c = padding; c < width + padding; ++c) {
            // 3x3 or 5x5 neighborhood
            std::vector<uint8_t> neighborhood;
            for (int kr = -padding; kr <= padding; ++kr) {
                for (int kc = -padding; kc <= padding; ++kc) {
                    // append pixel in padded_input to
neighborhood
                    neighborhood.push_back(padded_input[(r + kr)
* (padded_width) + (c + kc)]);
                }
            }
            // sort and find median
            std::sort(neighborhood.begin(), neighborhood.end());
            output[(r - padding) * width + (c - padding)] =
neighborhood[neighborhood.size() / 2];
        }
    }
```

1-1-iii.Edge detection

Present the resulting images and discuss how Gaussian and median filters differ in their noise suppression performance and visual results

Image with 3x3 Gaussian smoothing filter(parthenon_gaussian3x3.png)

Image with 3x3 median filter(parthenon_median3x3.png)

Discussion (1-1-iii)
此題我使用kernel size 3x3進行比較Gaussian跟median在處理salt-and-pepper上的差異，先說結果，使用median filter的結果有非常好的改善，原因是因為salt-and-pepper noise都是pixel的極值，因此在使用median進行filter的之後可以有效的移除他們，但是Gaussian smoothing filter只是將圖片進行模糊，因此雜訊依舊存在，只是變模糊。

1-2-a:Edge detection

Apply the unsharp masking to sharpen the image **parthenon_median3x3.raw** obtained from the previous step.Output the resulting mask image as **unsharp_mask.raw** and the sharpened image as **parthenon_unsharp_mask.raw**

unsharp_mask(unsharp_mask.png)(不夠明顯)

sharpened_image(parthenon_unsharp_mask.png)

Discussion (1-2-a)

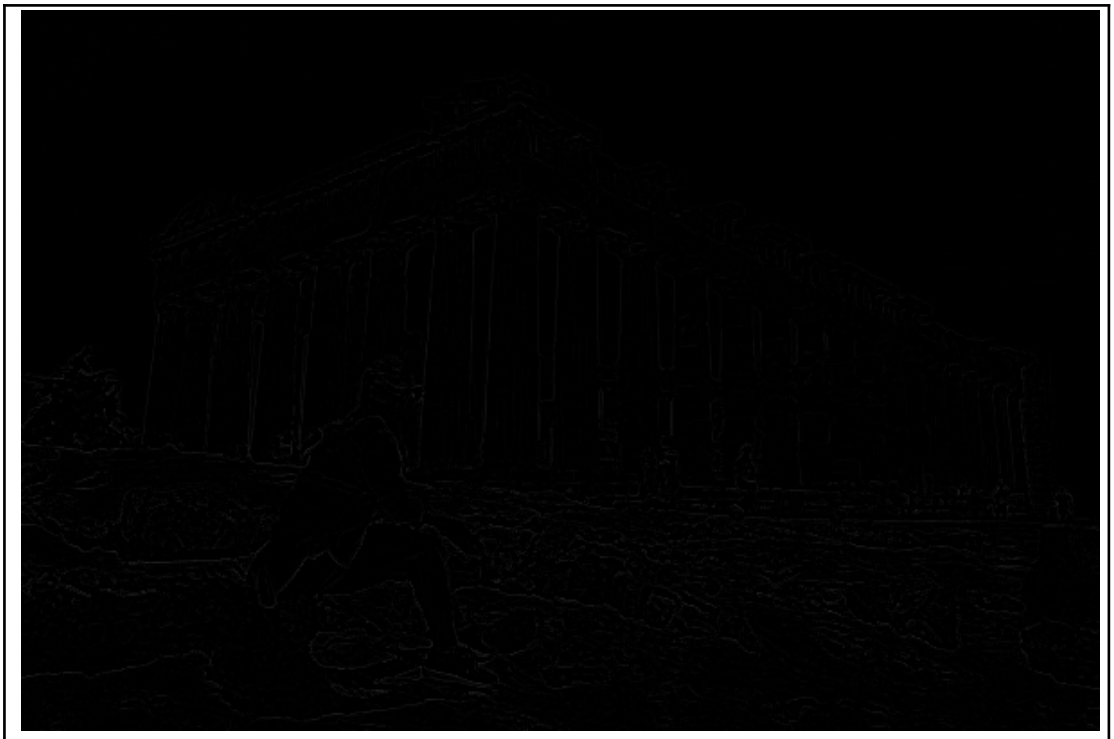
此題使用unsharp masking進行銳化，unsharp masking是使用原圖加上mask(使用原圖減上被模糊的圖片)。

因此我使用先前使用的Gaussian smoothing filter function”**apply_gaussian_filter**“，將**parthenon_median3x3.raw**進行模糊，再使用unsharp masking原理進行處理，並且將pixel進行clamping，由於mask不清楚，因此我將mask value往上加128，使結果較清晰。

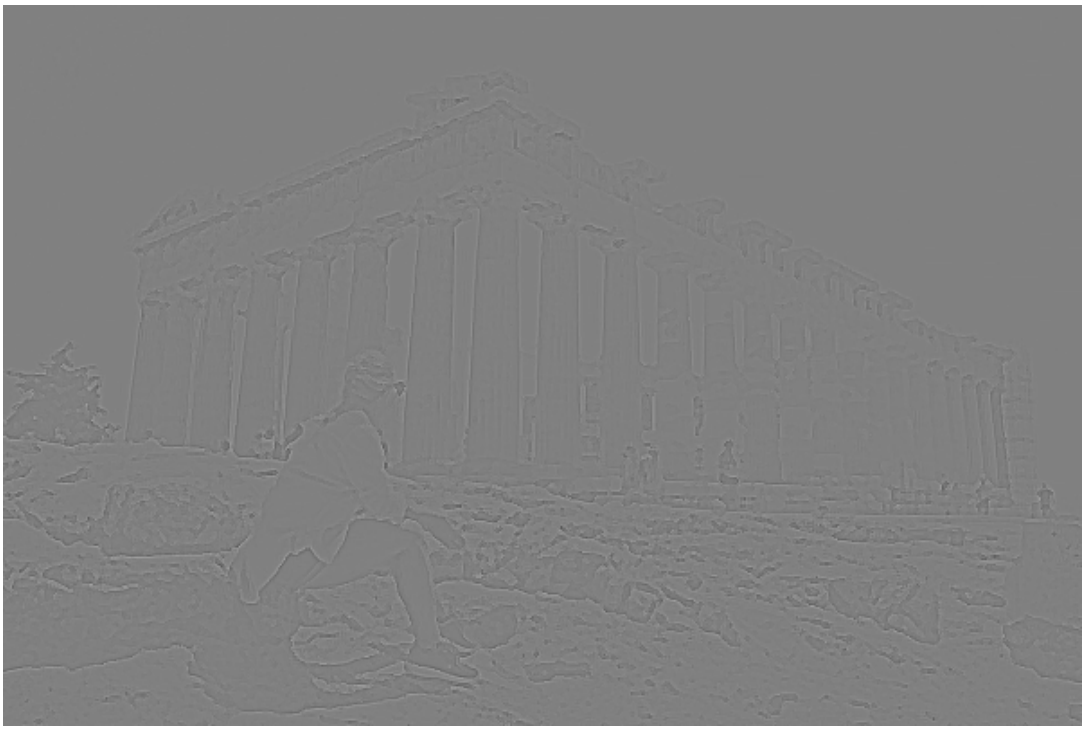
```
   for (int i = 0; i < width * height; ++i) {
       int mask_value = static_cast<int>(image_data[i]) -
static_cast<int>(blurred_image[i]);
       int sharpened_value = static_cast<int>(image_data[i]) +
(mask_value);
       if (sharpened_value > 255)
           unsharp_image[i] = 255;
       else if (sharpened_value < 0)
           unsharp_image[i] = 0;
       else
           unsharp_image[i] = sharpened_value;

       // shift to middle-gray (0 difference = gray)
       int mask_value_clear = mask_value + 128;
       if (mask_value_clear  < 0) mask_value_clear  = 0;
       if (mask_value_clear  > 255) mask_value_clear  = 255;
       mask_image[i] = mask_value_clear ;

   }
```



未經處理之mask



經處理之mask

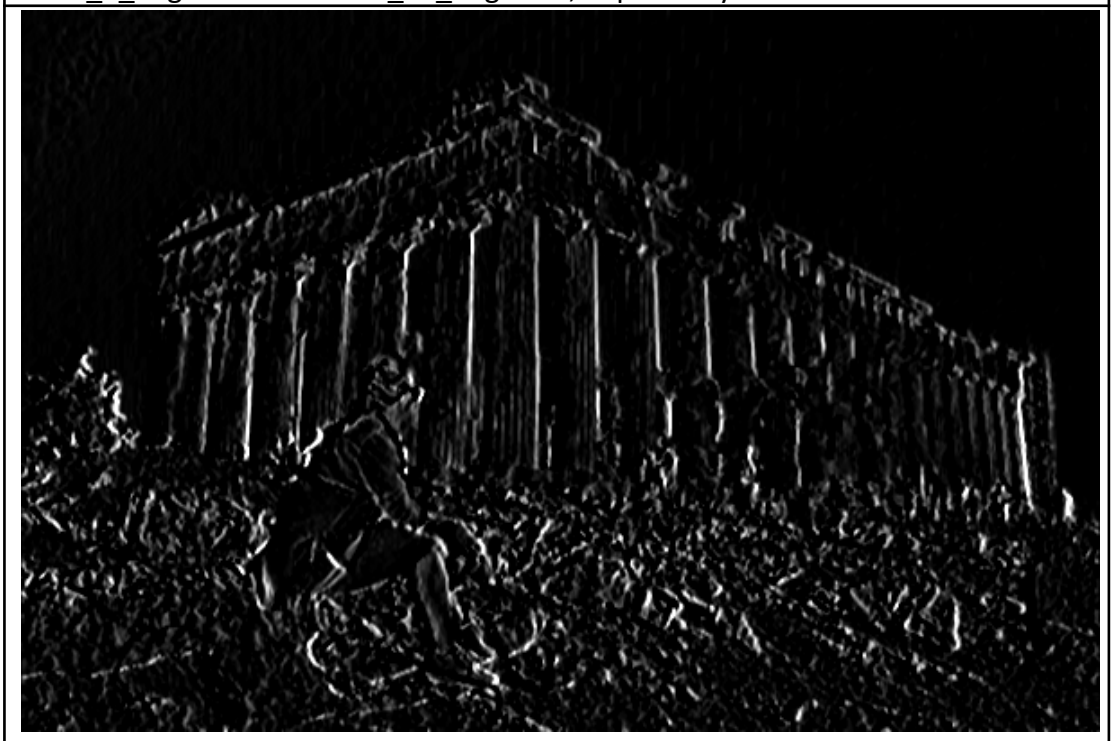
未經過處理之原圖(parthenon_median3x3.png)

經過unsharp masking處理之圖片

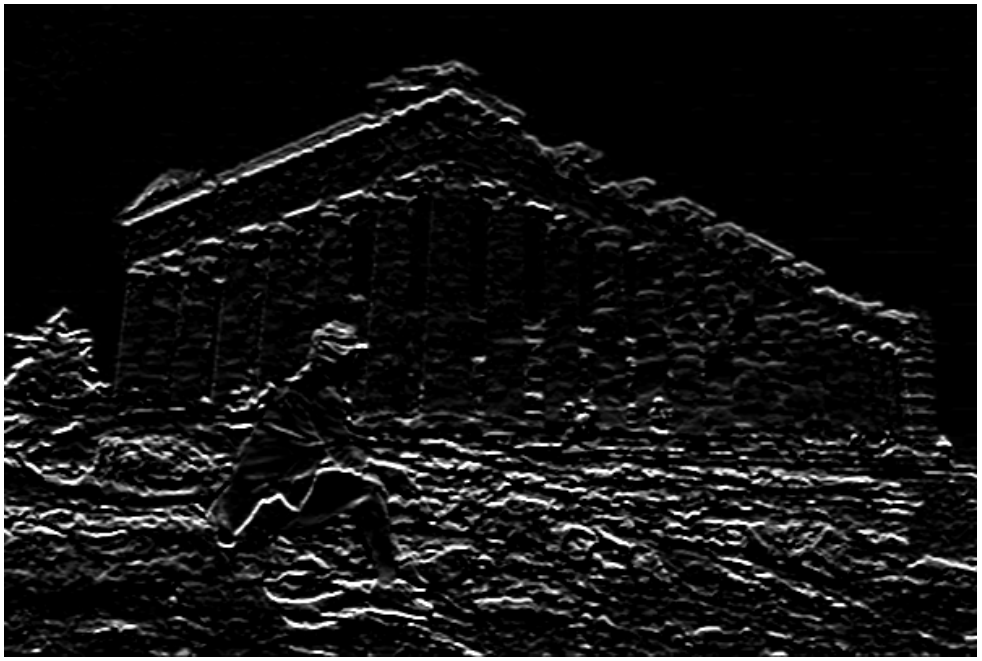
1-2-b-i:Edge detection

Apply 3x3 sobel filter to sharpen the same median-filtered image.

Compute the horizontal and vertical gradient images and save them as sobel_0_dege.raw and sobel_90_edge.raw,respectively.



Sobel_0_dege.png



Sobel_90_edge.png

Discussion (1-2-b-i)

此題需要使用3x3 Sobel filter，也就是

```
    const int8_t sobel_x[3][3] = {
        {-1,  0,  1},
        {-2,  0,  2},
        {-1,  0,  1}
    };

    const int8_t sobel_y[3][3] = {
        { 1,  2,  1},
        { 0,  0,  0},
        {-1, -2, -1}
    };
```

sobel_x是強化將圖片的水平邊緣
sobel_y是強化將圖片的垂直邊緣

因為sobel_x的變化都是垂直的，也就是他會把水平的邊緣顯示出來，反之亦然。

以下為主要的程式邏輯，與先前幾題都相同，同樣要注意padding 與clamping的問題。

```
    for (int r = padding; r < height + padding; ++r) {
        for (int c = padding; c < width + padding; ++c) {
            int sum = 0;
            for (int kr = -padding; kr <= padding; ++kr) {
                for (int kc = -padding; kc <= padding; ++kc) {
                    sum += padded_input[(r + kr) * (padded_width)
+ (c + kc)] * kernel[(kr + padding) * 3 + (kc + padding)];
                }
            }
            if (sum < 0) sum = 0;
            if (sum > 255) sum = 255;

            output[(r - padding) * width + (c - padding)] = sum;
        }
    }
```

1-2-b-ii:Edge detection

Then,combine the two gradient images to obtain sobel_combined_edge.raw.Compare the characteristic of the 0 and 90 degree of gradient images, and discuss the differences you observe.

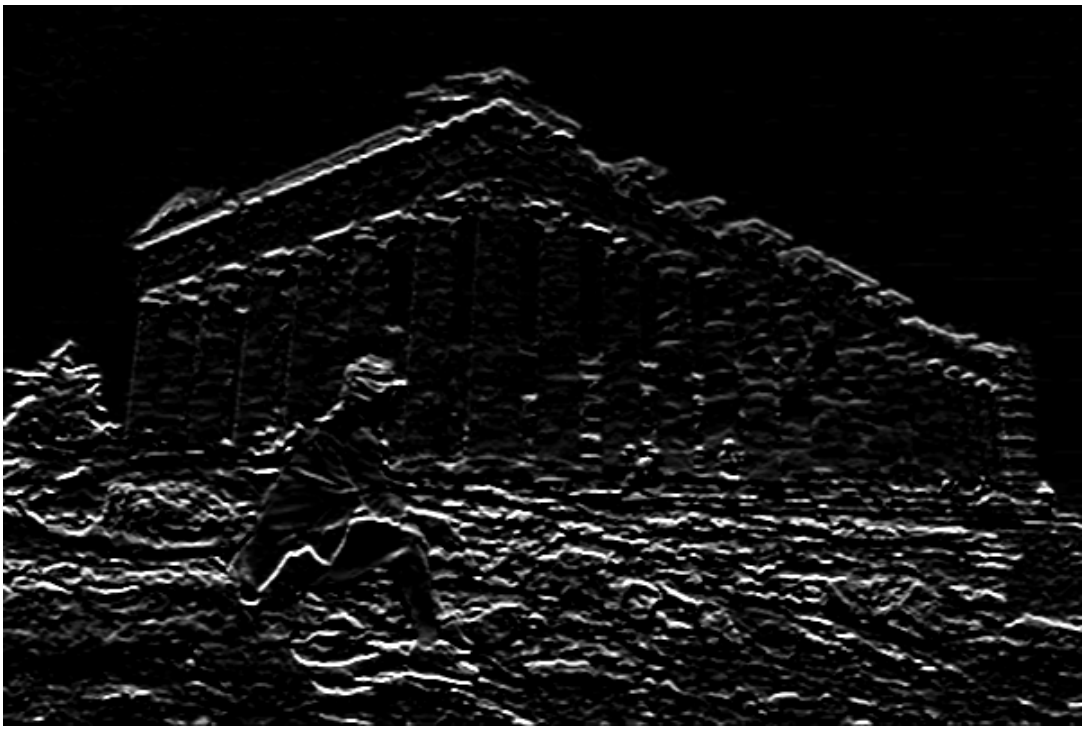
sobel_combined_edge.png

Discussion (1-2-b-ii)

此題使用先前算出的sobel_0 跟sobel_90的邊緣圖像，並找出最後的magnitude，所以按照公式，我們必須將Gx跟Gy分別平方後，再取一次平方根求得，最後再進行clamping，程式如下:

```
   std::vector<uint8_t> combined_image(width * height);
   for (int i = 0; i < width * height; ++i) {
       int combined_value =
static_cast<int>(std::sqrt(std::pow(image_data_0_dege[i], 2) +
std::pow(image_data_90_edge[i], 2)));
       if (combined_value > 255)
           combined_image[i] = 255;
       else if(combined_value < 0)
           combined_image[i] = 0;
       else
           combined_image[i] = combined_value;
   }
```

sobel_0:水平邊緣



Sobel_90:垂直邊緣

sobel_combined:水平加上垂直邊緣

1-2-b-iii:Edge detection

Perform image sharpening by adding the sobel_combined_edge.raw image to the original median-filtered image to generate the sharpened result parthenon_sobel_sharpened.raw

parthenon_sobel_sharpened.png

Discussion(1-2-b-iii)

此題使用先前求出的sobel_combined進行銳化，但是如果只是將他加上原圖(parthenon_median3x3.png)，會讓原圖(parthenon_median3x3.png)的邊緣亮度過高，與原圖差距過大，因此需要將sobel_combined加上負數權重(-0.35)再加上原圖(parthenon_median3x3.png)即可。

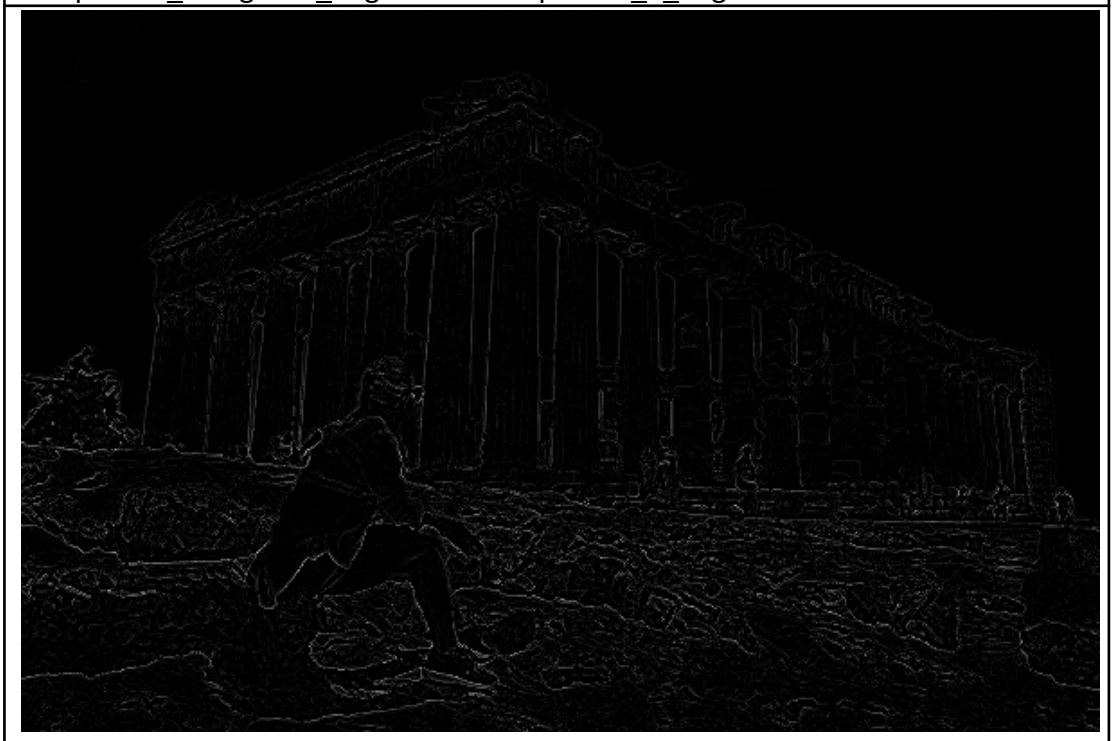
sobel_combined加入parthenon_median3x3.png的結果

上圖為先前使用unsharp masking的結果(parthenon_unsharp_mask.png)

比較之前使用unsharp masking的結果與使用sobel filter的結果，其實作法還蠻類似的，unsharp masking也是透過原圖加上(原圖減上被模糊的圖像=>邊緣)，此題做法也是使用原圖加上(邊緣)，只是使用負數權重避免圖像邊緣過亮(強度太強)。

1-2-c-i:Edge detection

Apply the 3x3 laplacian filter using both 4-neighbor and 8-neighbor configurations to the image parthenon_median3x3.raw.Save the corresponding Laplacian images as laplacian_4neighbor_edge.raw and laplacian_8_edge.raw



laplacian_4neighbor_edge.png

Laplacian_8neighbor_edge.png

Discussion(1-2-c-i)

此題使用3x3 laplacian 4-neighbor and 8-neighbor filter，兩者的差別就是四鄰只會注重在上下左右，並沒有對角的鄰居，因此八鄰的細節會更密集，因為考慮的鄰居更多，使用的kernel則是下圖，使用中心點為正的值，使邊緣為亮

```
    const int8_t laplacian_4_neighbor[3][3] = {
        { 0, -1,  0},
        {-1,  4, -1},
        { 0, -1,  0}
    };

    const int8_t laplacian_8_neighbor[3][3] = {
        {-1, -1, -1},
        {-1,  8, -1},
        {-1, -1, -1}
    };
```

而後續laplacian filter與先前幾題類似，進行replication padding後再進行處理

```
    // Padding replication
    padded_input = apply_replication_padding(input_image, width,
height, padding);

    // Apply Laplacian filter
    for (int r = padding; r < height + padding; ++r) {
        for (int c = padding; c < width + padding; ++c) {
            int sum = 0;
            for (int kr = -padding; kr <= padding; ++kr) {
                for (int kc = -padding; kc <= padding; ++kc) {
                    sum += padded_input[(r + kr) * (padded_width)
+ (c + kc)] * kernel[(kr + padding) * 3 + (kc + padding)];
                }
            }
            if (sum < 0) sum = 0;
            if (sum > 255) sum = 255;
            output_image[(r - padding) * width + (c - padding)] =
sum;
        }
    }
```

1-2-c-ii:Edge detection

Use each laplacian result to sharpen the image and produce parthenon_laplacian_4neighbor_sharpened.raw and parthenon_laplacian_8neighbor_sharpened.raw

Parthenon_laplacian_4neighbor_sharpened.png

Parthenon_laplacian_8neighbor_sharpened.png

Discussion(1-2-c-ii)

此題使用先前求得的4-neighbor與8-neighbor的邊緣圖像進行銳化，我們可以從結果看到經過8-neighbor銳化後的圖像有完整的細節呈現，因為8-neighbor的細節圖已經比4-neighbor呈現更多細節。

程式如下：

```
   for (int i = 0; i < width * height; ++i) {
       int sharpened_value_4 = static_cast<int>(image_data[i] +
static_cast<int>(image_data_laplacian_4[i]));
       int sharpened_value_8 = static_cast<int>(image_data[i] +
static_cast<int>(image_data_laplacian_8[i]));
       if (shapened_value_4 > 255)
           sharpened_image_4[i] = 255;
       else if (sharpened_value_4 < 0)
           sharpened_image_4[i] = 0;
       else
           sharpened_image_4[i] =
static_cast<uint8_t>(std::min(std::max(sharpened_value_4, 0),
255));
       if (sharpened_value_8 > 255)
           sharpened_image_8[i] = 255;
       else if (sharpened_value_8 < 0)
           sharpened_image_8[i] = 0;
       else
           sharpened_image_8[i] =
static_cast<uint8_t>(std::min(std::max(sharpened_value_8, 0),
255));
   }
```

1-2-c-iii:Edge detection

Present the edge detection results and the sharpened images,and discuss the difference between the two filter configurations in terms of characteristics and sharpening effects.

Laplacian_4neighbor_edge.png

Laplacian_8neighbor_edge.png

Parthenon_laplacian_4neighbor_sharpened.png

parthenon_laplacian_8neighbor_sharpened.png

Discussion(1-2-c-iii)

先前已比較過差異，這裡比較一下與先前幾種filter的差異，unsharp masking、sobel、laplacian:

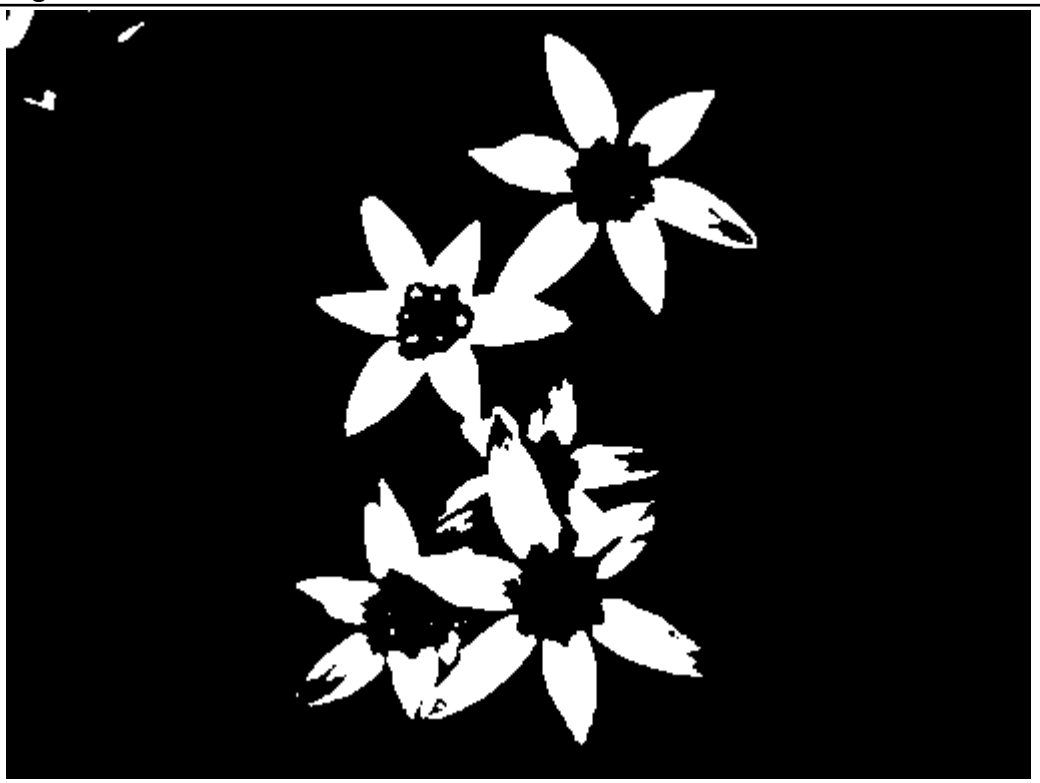
Unsharp masking:
(因為是使用Gaussian filter進行模糊並減掉的，因此效果較自然，邊緣強度較弱，雜訊少)

sobel filter:
(因為是使用sobel_0與sobel_90的處理，最後合併，因此結果圖的邊緣會較粗，且方向明顯，並且銳化比unsharp強烈)

Laplacian filter:
(強烈的銳化處理，對雜訊較敏感)

2-1:Image smoothing and sharpening

The image flower_512x384.raw is used in this part.The first task is to extract the flowers as a region of interest(ROI) through a combination of blurring and threshold.The goal is to obtain a binary mask image where the flowers appear as white foreground regions and the leaves as black background.Try to make your ROI as accurate as you can.Show the resulting binary image and discuss how you designed the mask and determined the threshold value.



ROI.png

Discussion(2-1)

此題使用 flower_512x384.raw 進行前幾題一直使用的Gaussian smoothing function”**apply_gaussian_filter**”，並且使用7x7的kernel size，再使用threshold=200進行thresholding，將花朵變成255，並且將樹葉變成0，但是左上角的樹葉有些pixel值跟花朵相似，因此無法完成去除，這也是global thresholding的問題之一，在原本的基礎再使用形態學應該可以有效去除。

以下是使用Gaussian進行模糊再進行ROI分割之部份程式

```
    std::vector<uint8_t> blurred_image(width * height);
    blurred_image = apply_gaussian_filter(image_data,
blurred_image, width, height,7);

    // Thresholding to create binary mask
    std::vector<uint8_t> binary_mask(width * height);
    const uint8_t threshold = 200; // Adjust threshold value as
needed
    for (int i = 0; i < width * height; ++i) {
```

```
        if (blurred_image[i] > threshold) {
            binary_mask[i] = 255; // White for foreground
(flowers)
        } else {
            binary_mask[i] = 0;   // Black for background
(leaves)
        }
    }
```

2-2:Image smoothing and sharpening

Using the binary ROI image from the previous step, perform different filtering operations on the foreground and background.Specifically,apply a 3x3 high-boost filter to enhance the foreground(flowers), and a 7x7 Gaussian smoothing filter to soften the background(leaves).Use logical operations to combine these two results based on the ROI mask and produce the final output image.Present your method,show the results,and discuss the observed effects of the two filters on the difference regions.

flower_enhanced.png

Discussion(2-1)

此題使用先前的ROI，區分花朵與樹葉(背景)，並使用high-boost filter對花朵進行強化，並且使用7x7的Gaussian filter對於樹葉背景進行模糊，以下為使用high-boost filter的方法，也就是原圖乘上權重(1.7)再減去模糊圖像:

```
   std::vector<uint8_t> blurred_image(width * height);
   blurred_image = apply_gaussian_filter(input, blurred_image,
width, height, kernel_size);

   for (int i = 0; i < width * height; ++i) {
       int high_boost_value = static_cast<int>(A *
static_cast<int>(input[i]) -
static_cast<int>(blurred_image[i]));
       if (high_boost_value > 255)
           output[i] = 255;
       else if (high_boost_value < 0)
           output[i] = 0;
```

```
        else
            output[i] = high_boost_value;
    }
```

以下則是使用ROI區分花朵與樹葉:

```
    std::vector<uint8_t> final_image(width * height);
    for (int i = 0; i < width * height; ++i) {
        if (roi_data[i] == 255) { // Foreground
            final_image[i] = high_boosted_image[i];
        } else { // Background
            final_image[i] = blurred_image[i];
        }
    }
```