**Open Software Project**

**Assignment3 Technical Report**

**경영학과 1517034 정성원**

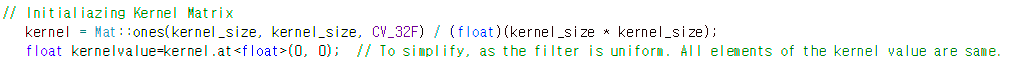
**1. 개요**

해당 과제에서는 Region-wise Operation의 일종인 Uniform Mean Filter, Gaussian Filter, Sobel Filter, Laplacian Filter 그리고 Unsharp Masking를 순서대로 구현한다. 구현한 필터들의 결과를 확인하기 위해서 lena라는 jpg 이미지 파일을 사용했다. 실험 방법에서 나오는 O(i, j), I(i, j), w(s, t), \*는 각각 출력 이미지의 (i, j) 픽셀, 입력 이미지의 (i, j) 픽셀, filtering kernel, convolution 연산을 의미한다. 각 함수, 변수 명 등을 포함한 소스 코드에 대한 설명은 README.txt 파일에 기재하였다.

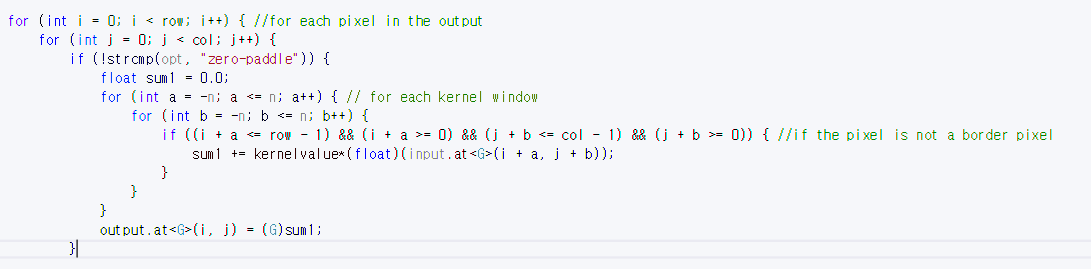
**2. 실험 방법**

**2.1 Uniform Mean Filter**

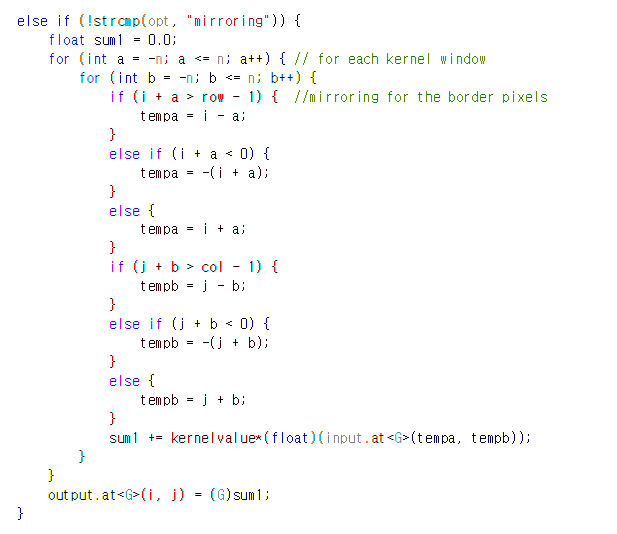
MeanFilterGray.cpp와 MeanFilteRGB.cpp은 Uniform Mean Filter를 구현한 소스코드이다. O(i, j)는 I(i, j)의 Region의 평균 값이다. 따라서 Kernel은 아래와 같다.



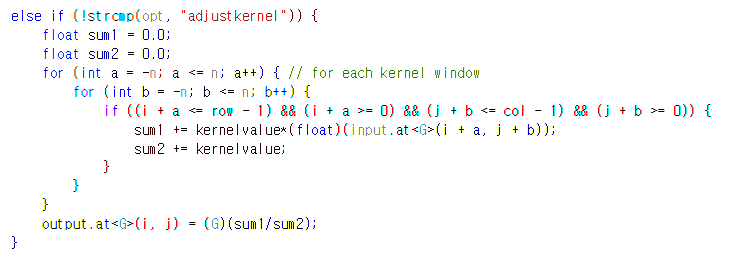
Kernel Operation 도중 커널이 이미지 경계 밖에 적용할 수 있기 때문에 Image Boundary에 위치한 픽셀에 대한 처리가 필요하다. 이미지의 경계에 위치한 픽셀들을 처리하기 위한 방법으로는 Zero Padding, Mirroring 그리고 Adjusting Filter Kernel를 사용했다. 먼저 Zero Padding이란 이미지 경계 바깥에 위치한 픽셀의 Intensity를 0으로 채우는 것이다. 이는 아래와 같은 코드로 작성할 수 있다.



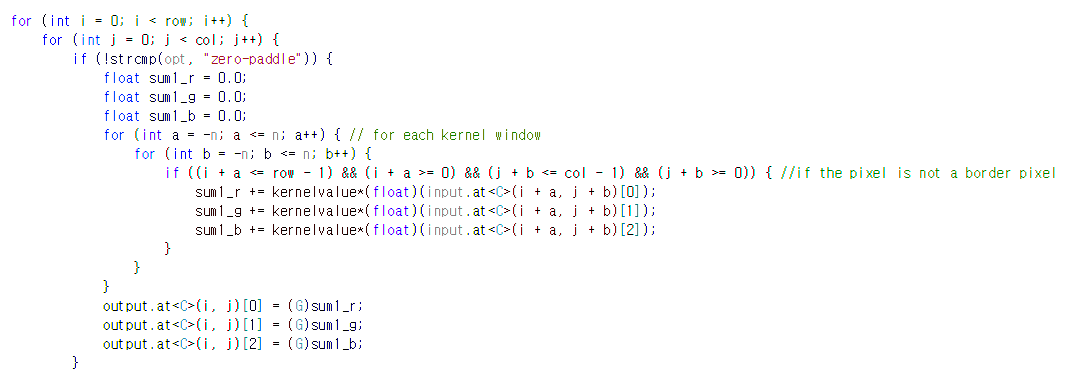
다음으로 Mirroring이란 이미지 경계 바깥에 위치한 픽셀의 Intensity를 경계 근처의 값을 복사해 채우는 것이다. 이는 아래와 같은 코드로 작성할 수 있다.



끝으로 Adjusting filter kernel는 아래의 코드를 통해 구현할 수 있다. 이 세가지 Image Boundary Processing은 Gaussian Filter, Sobel Filter, Laplacian Filter 그리고 Unsharp Masking에도 동일하게 사용할 수 있다.

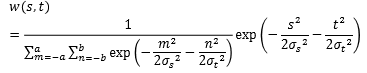


위 코드는 전부 흑백 이미지를 처리하는 코드이다. 만일 RGB 이미지에 Uniform Mean Filter를 적용할 경우, 동일한 Kernel을 사용해 동일한 연산을 입력 이미지의 각 채널 별로 수행한다. 예를 들어, Zero Padding을 사용할 경우 코드는 아래와 같다.

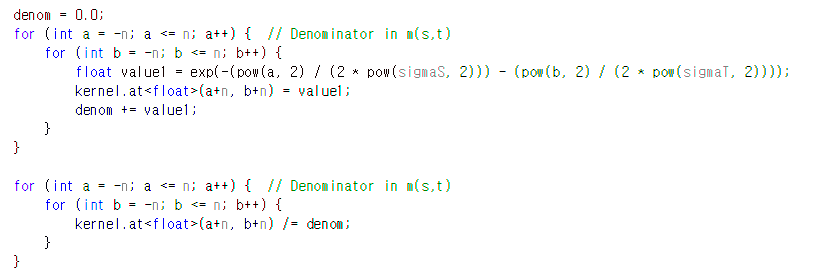


**2.2 Gaussian Filter**

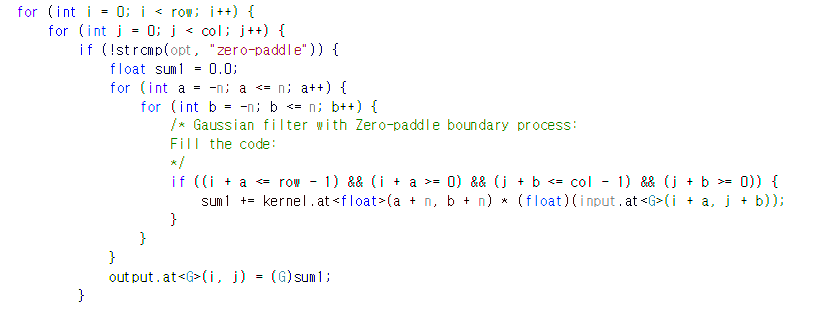
Gaussian Filter란 Uniform Mean Filter과 마찬가지로 low-pass filter의 일종이다. GaussianGraySkeleton.cpp과 GaussianFilterRGB.cpp에서 이를 각각 구현했다. 이미지를 흐리게 하고 싶을 때 사용한다. Gaussian Filter의 Kernel Value은 가우시안 분포를 따라 값이 부드럽게 변하기 때문에 Mean Filter와 비교했을 때 출력 이미지가 자연스럽다.



위 식은 w(s, t)의 Kernel Value이다. Kernel Value는 아래와 같은 코드로 초기화할 수 있다.



Kernel 연산은 Uniform Mean Filter와 동일하게 수행한다. 아래는 흑백 이미지에 Zero-padding을 이용해Gaussian Filter 연산을 수행하는 코드이다.



**2.3 Sobel Filter and Laplacian Filter**

Sobel Filter와 Laplacian Filter는 High-pass Filter의 일종으로 Intensity 변화를 감지한다. 입력 이미지의 종류에 따라 SobelGraySkeleton.cpp, SobelRGB.cpp, LaplacianGray.cpp, LaplacianRGB.cpp 에 나누어 구현했다.

먼저 Sobel Filter의 크기는 3으로 고정하였으며, Kernel은 Figure1과 같다.

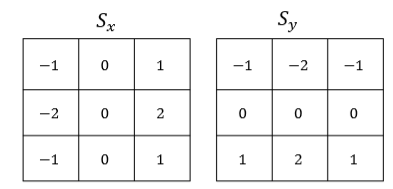
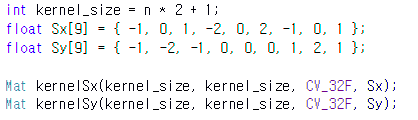
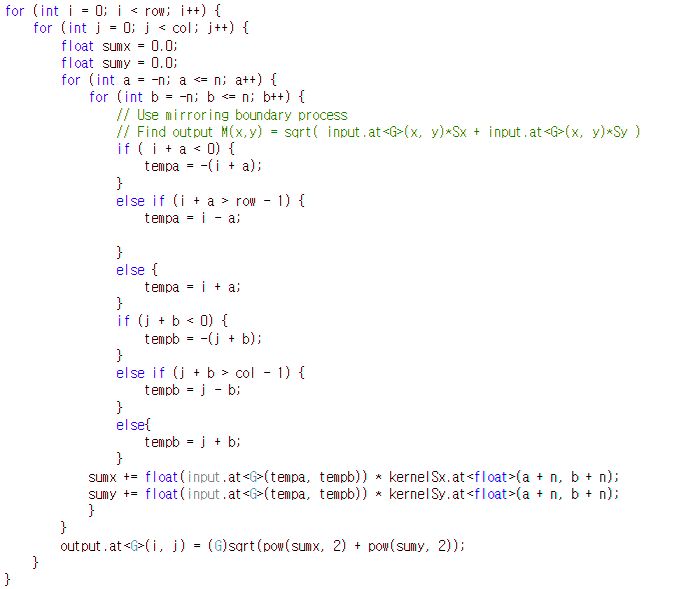


Figure . Sobel Filter

아래는 Figure 1을 코드로 구현한 결과이다. 위 값을 이용해 Sx Kernel과 Sy Kernel을 각각 초기화하였다.



I(i, j)에 Filter를 적용시킨 결과를 , Filter를 적용시킨 결과를 라고했을 때, O(i, j)는 이다. 흑백 입력 이미지에 Mirroring을 사용했을 때 코드는 아래와 같다.



Laplacian Filter의 크기 또한 3으로 고정하였으며, Kernel은 Figure 2과 같다.

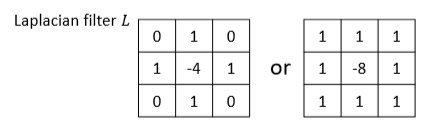
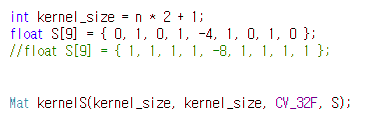
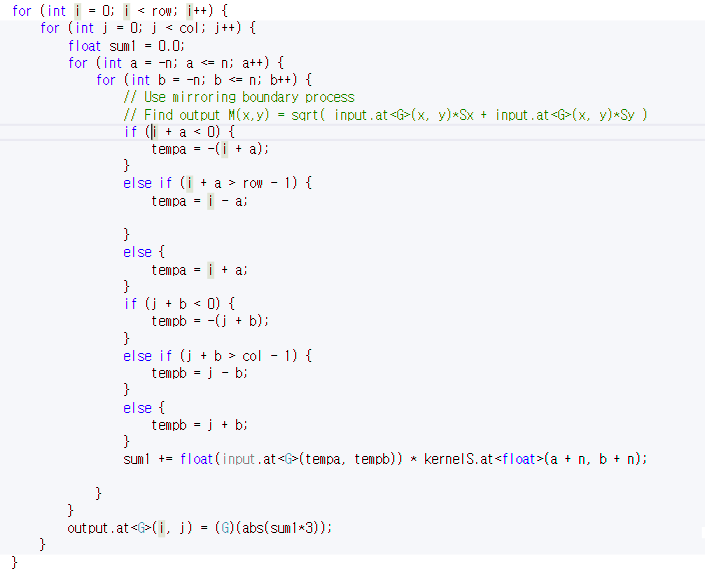


Figure . Laplacian filter

Laplacian Filter는 아래와 같이 초기화했다.



O(i, j)의 경우 L \* I의 결과가 음수일 수도 있으므로 L \* I의 결과에 절대값을 취한 값을 이용한다. 단, 출력 이미지를 더 잘 시각화하기 위하여 임의의 상수를 O(i, j)에 곱했다. 흑백 입력 이미지에 Mirroring을 사용한 코드는 다음과 같다.



**2.4 Gaussian filter in a separable manner**

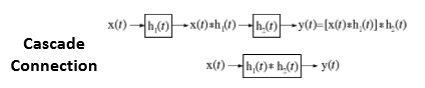
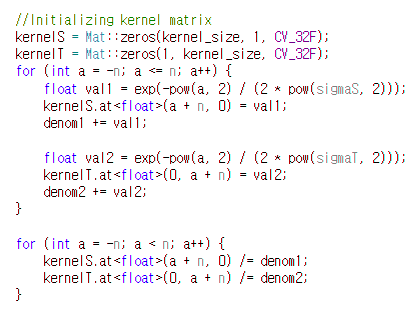


Figure . Separable Filtering

Gaussian Filter는 Figure 3과 같이 분리하여 적용시킬 수 있다. 기존 Kernel을 KernelS와 KernelT로 각각 나누어 아래와 같이 초기화했다.



연산은 입력 이미지에 kernelS를 이용해 임시 결과(temp\_output)를 얻은 후 kernelT를 이용해 최종 결과(output)를 얻는 방식으로 수행했다. 아래는 흑백 이미지에 Zero Padding을 이용해 가우시안 연산을 수행한 코드이다.



**2.5 Unsharp Masking**

Unsharp Masking은 high-frequency components를 강화하여 입력 이미지를 더욱 Sharp하게 만드는 기법이다. 알고리즘은 Figure 4와 같다.

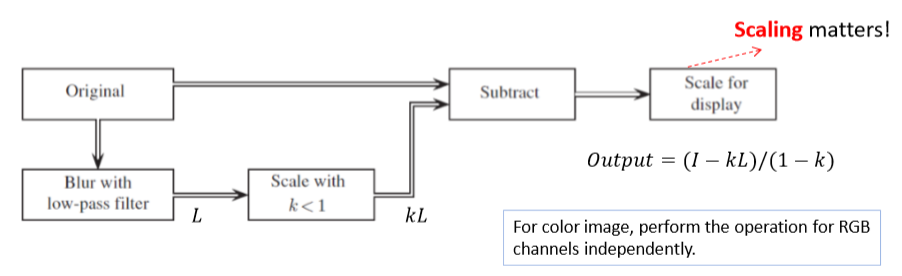
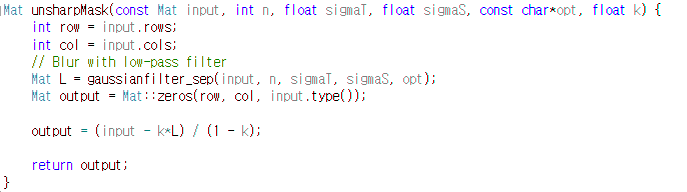


Figure . Unsharp Masking



**3. 결과 및 토의**

**3.1 Uniform Mean Filter**

MeanFilterGray.cpp과 MeanFilterRGB.cpp의 결과를 살펴본다. 실험 결과 Kernel 크기가 클수록 흐려지는 정도가 강해짐을 확인할 수 있었다. 아래 Figure 5는 Kernel 크기가 10 x 10인 Uniform Mean Filter를 적용한 결과이다. 사진은 시계방향 순서로 원본, Zero padding, Mirroring, Adjusting Filter Kernel를 적용한 결과이다. Zero Padding을 사용했을 때는 이미지의 경계가 검은 것을 확인할 수 있다.



Figure . Uniform Mean Filter 결과 (Gray / RGB)

**3.2 Gaussian Filter**

GaussianGraySkeleton.cpp과 GaussianRGB.cpp의 결과를 살펴본다. 실험 결과 Kernel 크기가 클수록, Sigma가 클수록 흐려지는 정도가 강해짐을 확인할 수 있었다. 아래는 Kernel 크기 21, Sigma T 2, Sigma S 2의 Gaussian Filter를 적용한 결과이다. 사진은 시계방향 순서로 원본, Zero padding, Mirroring, Adjusting Filter Kernel를 적용한 결과이다. Uniform Mean Filter를 적용했을 때와 비교해 결과 이미지가 자연스럽게 흐려진 것을 확인할 수 있다.



Figure . Gaussian Filter (Gray / RGB)

**3.3 Sobel Filter and Laplacian Filter**

SobelGraySkeleton.cpp, SobelRGB.cpp, LaplacianGray.cpp, LaplacianRGB.cpp의 결과를 살펴본다. Kernel의 크기는 Sobel Filter와 Laplacian Filter 모두 3 x 3으로 고정하였으며, 이미지의 경계는 Mirroring을 이용해 처리했다. Figure 7은 순서대로 SobelGraySkeleton.cpp, SobelRGB.cpp, LaplacianGray.cpp, LaplacianRGB.cpp의 결과이다.

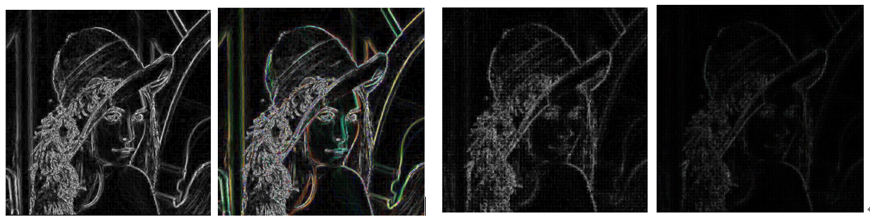
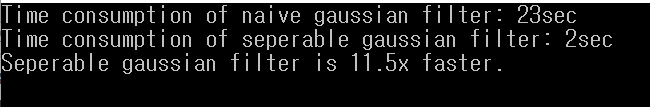


Figure . Sobel Filter and Laplacian Filter

**3.4 Gaussian filter in a separable manner**

GaussianFilterGray.cpp과 GaussianFilterRGB.cpp의 결과를 살펴본다. 해당 소스 코드는 3.2 에서 구현한 Gaussian Filter를 separable한 방식으로 구현해 연산의 효율성을 높였다. 결과 이미지는 3.2와 동일했다.



Figure

위 Figure 8 은 3.2에서 구현한 Gaussian Filter와 3.4에서 구현한 Separable Gaussian Filter의 연산 속도 차이이다. 크기 21 x 21, sigma S = 2, sigma T = 2인 가우시안 필터를 동일한 흑백 이미지에 적용하였다. Separable Gaussian Filter가 Naïve 버전의 Gaussian Filter에 비하여 11배 빠르게 연산을 수행하는 것을 확인할 수 있다.

**3.5 Unsharp Masking**

UnsharpMaskingGray.cpp와 UnsharpMaskingRGB.cpp의 결과를 살펴본다. 아래 Figure 9는 순서대로 원본, Unsharp Masking(k = 0.1) 수행 결과, Unsharp Masking(k= 0.5) 수행결과이다. k 값이 높아질수록 이미지가 더욱 선명해지는 것을 확인할 수 있었다. Unsharp Masking에는 Kernel 크기 10, Sigma T 2, Sigma S 2의 Gaussian Filter를 적용했다.

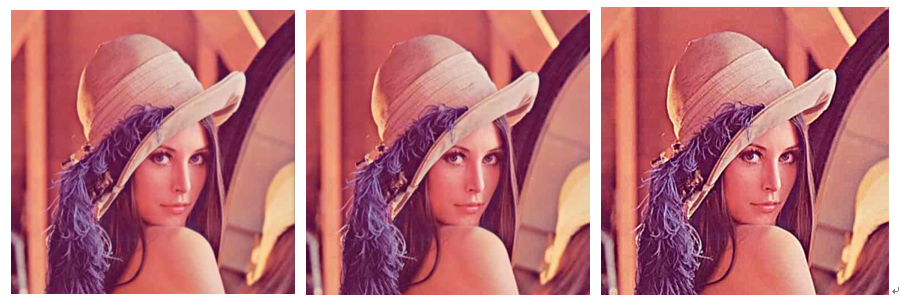


Figure . Unsharp Masking

**3.6 Convolution**

