Programming Assignment 2

CS 474

https://github.com/alexander-novo/CS474-PA2

Alexander Novotny	Matthew Page
50% Work	50% Work
Sections 4 and 5	Sections 1 to 3

Due: October 26, 2020
Submitted: October 28, 2020

Contents

1 Correlation

1.1 Theory

Image correlation is a type of spatial filtering, which takes in a set of input pixels along with a number of weights to produce an output pixel at a given location in an image. The input pixels are defined by the location and size of a mask, called the kernel, which consists of a 2D square array of weight values. Typically the mask is an nn array of pixels, where n is odd, and the output pixel corresponds with the center pixel within the mask. When modeling an image as a 2D function f(x, y), the correlation operation can be viewed mathematically by the following equation

$$\mathrm{Corr}_w f(x,y) = (w \star f)(x,y) = \sum_{s=-a}^{a} \sum_{t=-b}^{b} w(s,t) f(x+s, y+t)$$

where w(s, t) is the weight at the relative location s, t of the neighborhood centered at x, y.

The correlation operation is linear, meaning that the output of the correlation function is a linear combination of the input pixels and weights. There are many applications of image correlation, such as smoothing, sharpening, and noise reduction. It is worth noting that a similar operation to correlation exists called convolution, whereby the kernel matrix is flipped both horizontally and vertically.

1.2 Implementation

In order to implement image correlation, a pgm image was first read into the program using command line arguments, along with the output image location and mask. The mask weights were imputed using the pgm image format, allowing other images to serve as the mask. Also, the dimensions of the mask are taken as command line arguments.

The main algorithm for performing correlation consists of iterating through each pixel within the input image and applying the correlation operation at each location. To perform the correlation, a second set of for loops are used to iterate through each pixel within the neighborhood defined by the mask dimensions. The output value is calculated by summing each pixel value multiplied by its corresponding weight value within the mask. Before this value is added to the image, it is added to an integer vector, such that min-max normalization may take place before the pixel is updated in the image. This ensures pixel values remain within the proper range of 0-255.

In order to isolate the possible locations of the mask image within the input image, the normalization was performed such that the largest correlation values were depicted as white pixels and the remaining as black. Discussion of the results of the correlation function is given below.

1.3 Results and Discussion

The correlation function was performed using a patterned input image with the mask corresponding to a single pattern similar to those within the input image. A mask size of 83x55 was chosen, corresponding to the dimension of the mask image. Figure 1 showcases the input images used, along with the resulting (normalized) image of the correlation operation. According to the figure, it is easy to see the locations of the matching patterns found during the correlation process. These correspond to the brightest whte blobs within the resulting image. These blobs are indications of a high correlation value, which indicates that the most of the pixels within the neighborhood are similar to the corresponding mask values, resulting in an overall higher output value. Also, the general direction of the blobs can be used to further narrow the possible locations. For example, although there is a high correlation value in the blob near the top right of the image, the general orientation does not match that of the mask as well as the other three true matches.

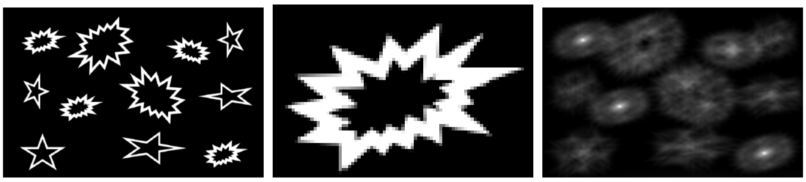


Figure 1: The result of correlation on the patterns image (left) with the given mask. The resulting normalized image (right) demonstrates the effect of correlation. The three white blobs correspond to the location of the mask pattern within the image.

2 Averaging and Gaussian Smoothing

2.1 Theory

Averaging and Gaussian Smoothing are two common techniques for removing fine details from an image, resulting in a smoothed or blurred image. Both types of smoothing are examples of a linear spatial filter. Also, both types of smoothing techniques have similar properties, as well as different advantages and disadvantages.

Smoothing via averaging corresponds to taking the average of each pixel value within the bounds of the kernel and using the result as the output pixel. This operation results in an output value that takes into account all neighboring pixel values equally. Mathematically this operation can be represented as

$$f(x,y) \rightarrow \frac{1}{N}\sum_{s=-a}^{a}\sum_{t=-b}^{b} f(x+s,y+t),$$

where N is the number of elements in the mask. Another method of smoothing is to assign each weight value according to a Gaussian distribution. This results in better smoothing, however it is more difficult to implement compared to using averaging. The values of the mask can be taken from a 2D Gaussian distribution given mathematically as

$$w(s,t) = Ke^{-\frac{s^2+t^2}{2\sigma^2}}.$$

This method of smoothing has the benefit of considering pixels closer to the target pixel as more important or having a higher weight compared to pixels further away from the target image, typically resulting in a better smoothed image.

2.2 Implementation

Average and Gaussian smoothing also used command line arguments to read the input image, output image location, as well as the kernel size and smoothing type. Once the image is read, the smoothing is performed based on the type and size parameter passed by the user. The smoothie was performed by iterating through each pixel of the original image, using a nested for loop. For each pixel in the image, the kernel was applied using another set of nested for loops that iterate through each pixel in the neighborhood defined by the mask size. A bounds check is performed before each pixel access within the image, and if an index falls outside the image boundaries a default (padded) value of zero is assigned. The output pixel of the original image is then updated using the output of the filter.

In the case of smoothing via averaging, the output pixel located at the center of the kernel matrix or mask is defined as the average value of all the pixels within the kernel neighborhood. For Gaussian Smoothing, a normalization constant was calculated by summing all values within the Gaussian mask. Each output pixel was then divided by this constant to ensure that the pixel values remain valid pixel values. A copy of the original image was also used in order to ensure that previously updated pixels do not influence future unaltered pixels during the filtering process.

The main data structures used in the smoothing algorithms include the image class and a 2D array representing the 7x7 and 15x15 Gaussian masks. Each mask was defined as a constant, static 2D array of integers in order to make indexing similar to that of the image pixel values.

2.3 Results and Discussion

The techniques of averaging and gaussian smoothing were applied to two separate images, with two different mask sizes each. The results of the smoothing filters applied to the lenna.pgm image are shown in fig. 2. According to the results, it appears that both filters perform well at removing a large amount of fine detail from the original image, especially near the fur material of the woman's hat. However, it appears that the use of average smoothing using a 7x7 mask leaves the image more blurry compared to the Gaussian smoothed image with a similar sized mask. The average smoothed image also appears slightly darker compared to the others. For the 15x15 masked images, the differences become less evident, with the Gaussian smoothed image having slightly more discernible details and slightly less aliasing compared to the smoothed image via averaging.

Figure 2: A comparison of `lenna.pgm` with various smoothed images (From left to right: Original, 7x7 averaging, 15x15 averaging, 7x7 Gaussian, 15x15 Gaussian).

Similarly, the results of averaging and gaussian smoothing for the sf.pgm image are shown in fig. 3. Like with the lenna.pgm images, both smoothed images with a 7x7 mask lose much of the fine grained details of the original image. However, the Gaussian smoothed image again appears slightly less aliased with slightly more discernible details, especially near the bridge trusses. For the 15x15 masks, the Gaussian filter again generates a smoother (less aliased) image compared to the averaging filter, even though both lose considerable detail compared to the 7x7 masks.

Figure 3: A comparison of `sf.pgm` with various smoothed images (From left to right: Original, 7x7 averaging, 15x15 averaging, 7x7 Gaussian, 15x15 Gaussian).

3 Median Filtering

3.1 Theory

Median Filtering is another type of filter commonly used in image processing. Unlike the previous filters discussed, the median filter is an example of a nonlinear filter, whereby its output cannot be expressed as a linear combination of its input pixels. The median filter is performed by finding the median value within the neighborhood of pixels defined by the mask or kernel. The nonlinearity stems from the step of having to sort the pixel values in order to find the median value.

A common application of median filtering is removing certain types of noise from an image, often referred to as salt-and-pepper noise. This type of noise consists of random black and white pixels that become superimposed on an image. Figure 4 shows an example of this type of noise. This filter is able to remove so called salt-and-pepper noise due to the property of the median value m, which is defined as the 50th percentile of an ordered set of values. This implies that values on the extreme end of the scale, such as very dark or bright pixels, will in general not be selected to replace an image's pixel during the filtering process. This results in an enhanced image with the noise removed, especially compared to using other linear spatial filters.

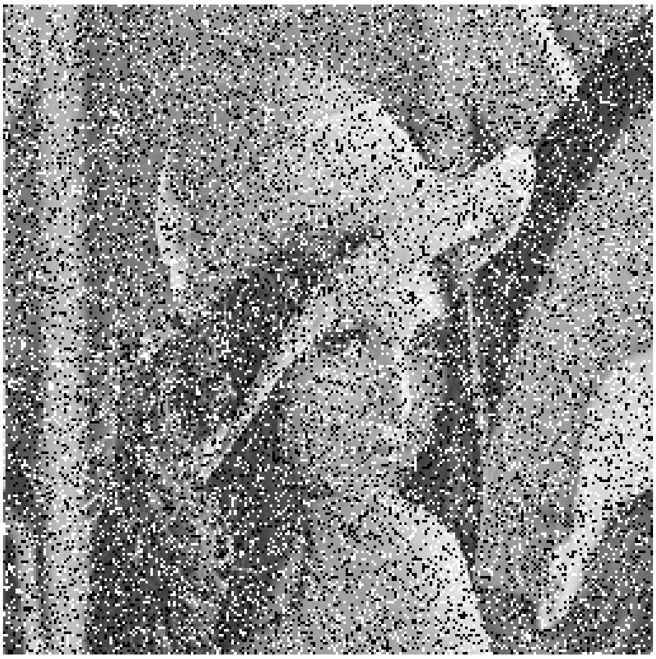


Figure 4: An example of Salt-and-Pepper noise on the lenna.pgm image

3.2 Implementation

In order to implement median filtering a similar process of iterating through each pixel value using a double nested for loop was used. A second set of nested for loops was then used to iterate through each pixel within the kernel neighborhood. Each pixel within the neighborhood was then added to a vector of integer pixel values. If the location of the pixel was out of bounds, a default value of zero was added to the vector instead, acting as a padded zero. To find the median value, the values within the vector were sorted and the middle value was selected. This median value then replaced the original image's pixel value at the location defined by the center of the mask.

In order to test median filtering various noisy images were generated. The process of generating the noise involved creating a vector containing all indices of the input image. This vector was then randomly shuffled, and the first X% of image indices from the random vector were selected to be noise. For each selected index, a random number generator was used to randomly assign either a black of white pixel with equal chance for both.

3.3 Results and Discussion

The median filter was used on two separate images, both of which used varying amounts of noise. In fig. 5 the results of the median filter using the lenna.pgm image are shown. In the case of using a larger mask size of 15x15, it appears that much more smoothing occurred as a result, leaving many of the details lost. However, the main features of the image are much more visible. In the case of a smaller filter size, there remained some artifacts from the noise, however the parts of the image that are not corrupted remain much more detailed compared to the use of larger mask.

Figure 6 showcases similar results using the boat.pgm image. The use of a smaller mask results in less smearing in the final image, however there remains artifacts from the noise. In the case of 50% noise, much of the details become obscured from the substantial amount of artifacts left in the image, possibly due to the overall lighter input image. In general is appears that a tradeoff between the size of the mask and the desired type of image.

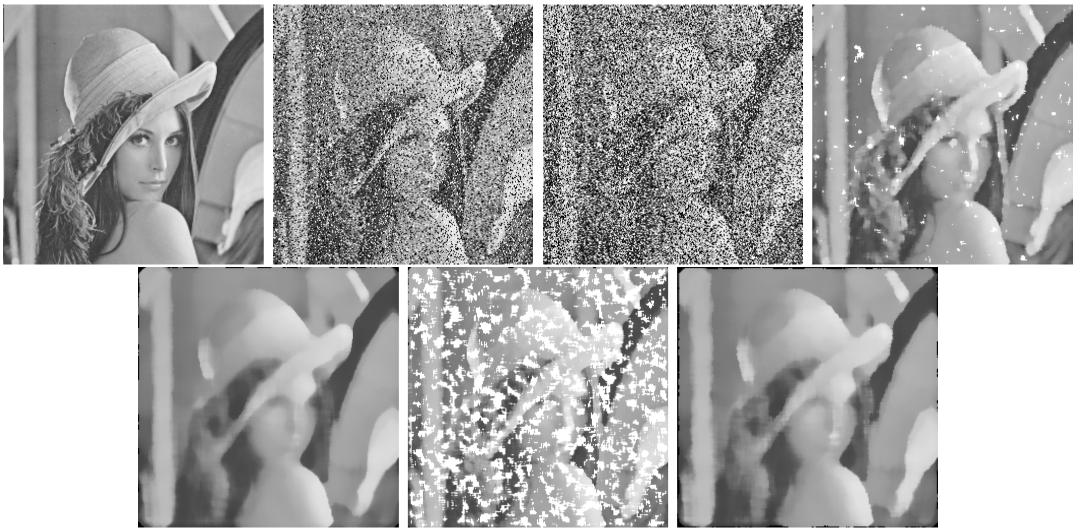


Figure 5: A comparison of `lenna.pgm` with noisy and filtered images (From left to right: Original, 30% noise, 50% noise, 7x7 w/30% noise, 15x15 w/30% noise, 7x7 w/50% noise, 15x15 w/50% noise).

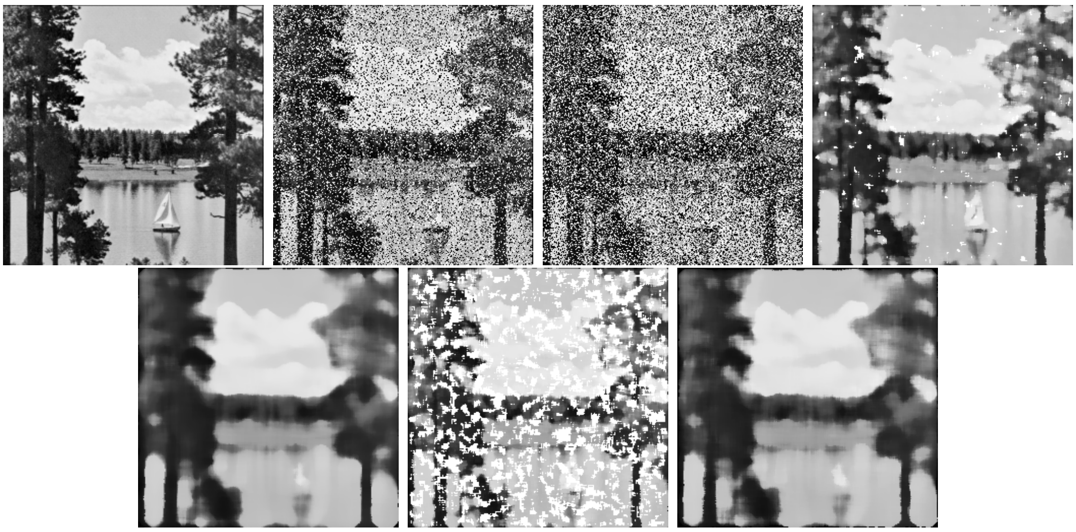


Figure 6: A comparison of `boat.pgm` with noisy and filtered images (From left to right: Original, 30% noise, 50% noise, 7x7 w/30% noise, 15x15 w/30% noise, 7x7 w/50% noise, 15x15 w/50% noise).

Lastly, simple averaging was used in order to compare with the median filtered images. Based on fig. 7, it appears that averaging results in a lower contrast image compared to using median filtering, as well as having an aliased appearance, no matter the amount of noise or mask size. Figure 8 shows similar results using the boat.pgm image. The reason for these results includes the addition of very bright or dark pixels from the noise, which causes all output pixels to tend towards a similar grey value. The random distribution of the noise may also be responsible for the pixelation/aliasing.

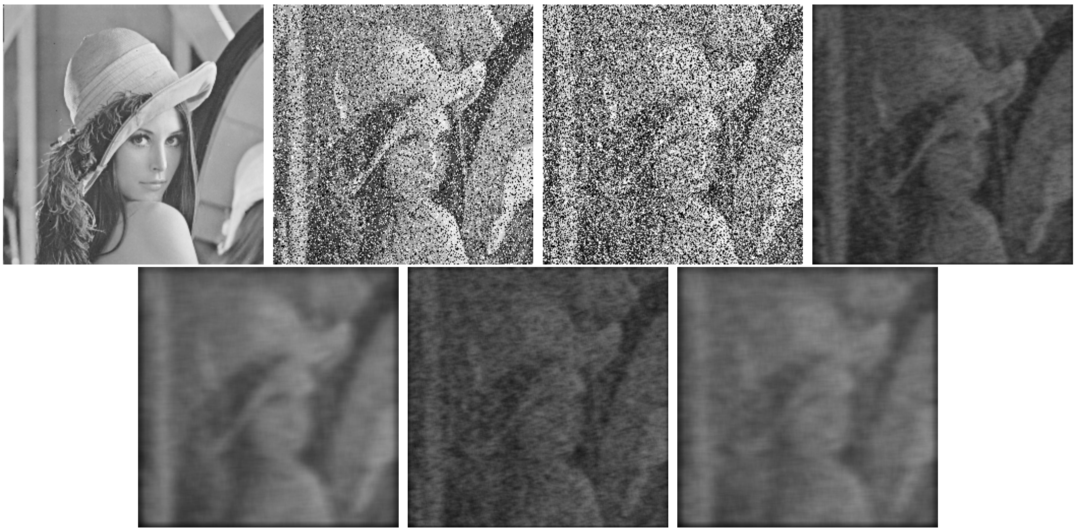


Figure 7: A comparison of `lenna.pgm` with noisy and filtered images (From left to right: Original, 30% noise, 50% noise, 7x7 w/30% noise, 15x15 w/30% noise, 7x7 w/50% noise, 15x15 w/50% noise).

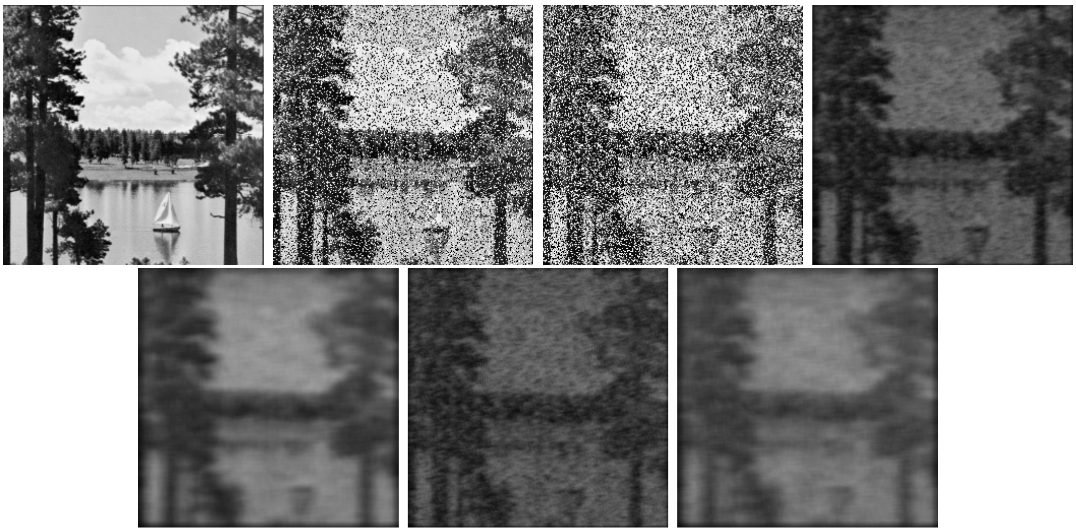


Figure 8: A comparison of `boat.pgm` with noisy and filtered images (From left to right: Original, 30% noise, 50% noise, 7x7 w/30% noise, 15x15 w/30% noise, 7x7 w/50% noise, 15x15 w/50% noise).

4 Unsharp Masking and High Boost Filtering

4.1 Theory

When a blurred version of an image is subtracted from the original image, what is left is an image of the edges from the original image. This technique is called unsharp masking, and adding the resulting edge image back to the original image to sharpen the edges is known as high boost filtering. Equation (1) shows how to calculate these images g from the original image f and its blurred version $\bar{f}$ with a constant $A \geq 1$. When $A = 1$, the formula gives us the unsharp mask, and when $A > 1$, we are using high boost filtering.

$$g(x,y) = Af(x,y) - \bar{f}(x,y) \tag{1}$$

4.2 Implementation

The image is first smoothed using a Gaussian kernel, given by table 1, as in section 2. Then eq. (1) is applied directly to each pixel of the output image, and remapped to the interval $[0, 255]$ by first adding 255 (to account for subtracting the blurred image) and dividing by $1 + A$ (to account for subtracting the blurred image and multiplying the original image by A).

The source code for this implementation can be found in listing 5.

1	1	2	2	2	1	1
1	2	2	4	2	2	1
2	2	4	8	4	2	2
2	4	8	16	8	4	2
2	2	4	8	4	2	2
1	2	2	4	2	2	1
1	1	2	2	2	1	1

Table 1: A 7×7 Gaussian kernel.

4.3 Results and Discussion

Figure 9 shows the result of applying the algorithm to `lenna.pgm` with a couple of different A values. Note that since no contrast enhancements are applied, the image loses contrast due to mapping the results to the interval $[0, 255]$. The unsharp mask shows a great amount of detail in the edges.

(a) The original `lenna.pgm`.

(b) The unsharp mask ($A = 1$)

(c) The high boost image ($A = 2$).

Figure 9: Comparison of `lenna.pgm` and its unsharp mask. No contrast enhancements are applied.

Figure 10 shows the result of applying the algorithm to `f_16.pgm` with a couple of different A values. Similarly, the unsharp mask shows a great amount of detail i nthe edges.

(a) The original `f_16.pgm`. (b) The unsharp mask ($A = 1$) (c) The high boost image ($A = 2$).

Figure 10: Comparison of `f_16.pgm` and its unsharp mask. No contrast enhancements are applied.

5 Gradient and Laplacian

5.1 Theory

Thinking of edges as drastically changing values in an image, edges exist precisely where the directional derivative of the image is large. Unfortunately, images aren't continuous functions, so we approximate partial derivatives as finite differences, as in eqs. (2) and (3).

$$f_x(x, y) \approx \frac{f(x+1, y) - f(x-1, y)}{2}, \tag{2}$$

$$f_y(x, y) \approx \frac{f(x, y+1) - f(x, y-1)}{2} \tag{3}$$

If we think of edges as locally maximal changing values, then the second derivative, or laplacian, will be 0. We can use the laplacian to find 0 crossings to find these edges, giving us more locality and faster computation than partial derivatives. Similarly to partial derivatives, we can approximate the laplacian using finite differences.

5.2 Implementation

We use the masks given in table 2 to calculate different partial derivatives using finite differences. For the magnitude, a pair of f_x, f_y masks are used to calculate both partial derivatives, and the magnitude is calculated, taking special care to remap values to the interval $[0, 255]$. For the Laplacian, the mask given in table 3 is used to calculate the laplacian image.

The source code for this implementation can be found in listing 6.

-1	0	1
-1	0	1
-1	0	1

(a) The Prewitt f_x mask

-1	-1	-1
0	0	0
1	1	1

(b) The Prewitt f_y mask

-1	0	1
-2	0	2
-1	0	1

(c) The Sobel f_x mask

-1	-2	-1
0	0	0
1	2	1

(d) The Sobel f_y mask

Table 2: Several masks which can be used to approximate partial derivatives.

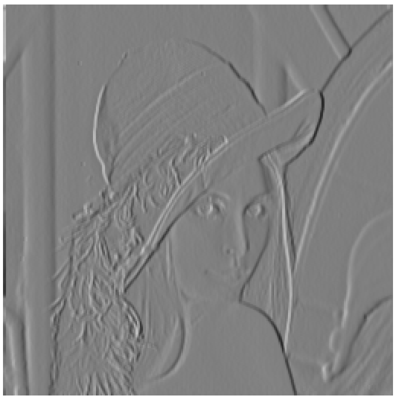
0	1	0
1	-4	1
0	1	0

Table 3: A Laplacian mask.

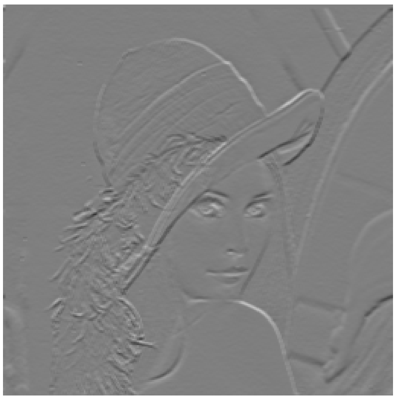
5.3 Results and Discussion

Figure 11 shows the result of applying various derivative-based sharpening algorithms to `lenna.pgm`. The gradient magnitudes give great constrast images of edges, while the laplacian gives really well-defined edges.

(a) The original `lenna.pgm`.

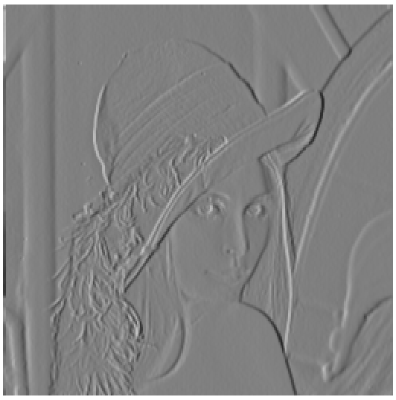


(b) f_x calculated using table 2a.



(c) f_y calculated using table 2b.

(d) The gradient magnitude.



(e) f_x calculated using table 2c.

(f) f_y calculated using table 2d.

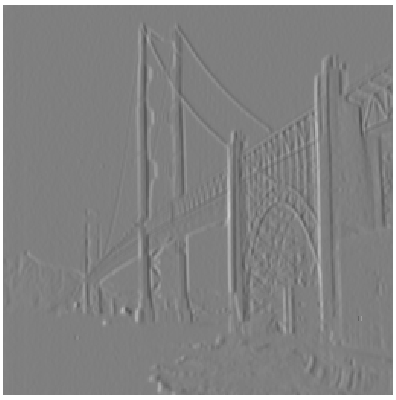
(g) The gradient magnitude.

(h) The Laplacian, calculated with table 3.

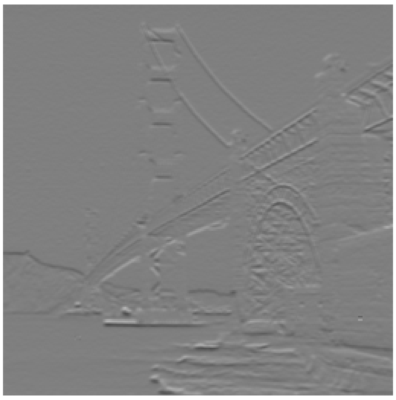
Figure 11: Comparison of `lenna.pgm` and derivatives. No contrast enhancements are applied.

Figure 12 shows the result of applying various derivative-based sharpening algorithms to `sf.pgm`. The gradient magnitudes give great constrast images of edges, while the laplacian gives really well-defined edges.

(a) The original `sf.pgm`.

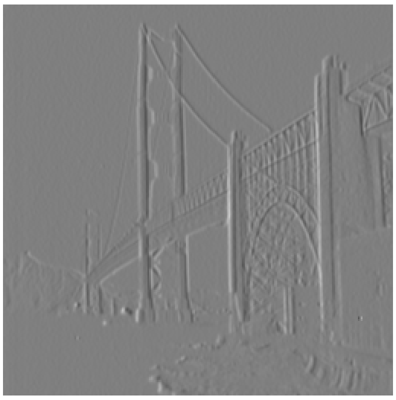


(b) f_x calculated using table 2a.

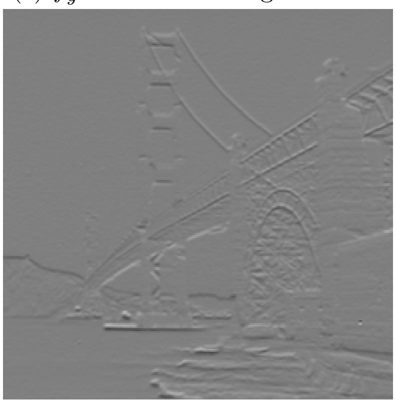


(c) f_y calculated using table 2b.

(d) The gradient magnitude.

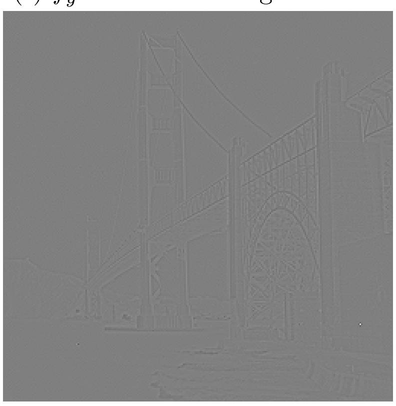


(e) f_x calculated using table 2c.



(f) f_y calculated using table 2d.

(g) The gradient magnitude.



(h) The Laplacian, calculated with table 3.

Figure 12: Comparison of `sf.pgm` and derivatives. No contrast enhancements are applied.

Code Listings

Source code can also be found on the project's GitHub page: `https://github.com/alexander-novo/CS474-PA2`. See previous assignments for common code (such as the `Image` class).

Listing 1: Header file for the common `Mask` class.

```
#pragma once

#include <array>

#include "image.h"

template <typename T, std::size_t N>
class Mask {
  static_assert(N % 2 == 1, "Size of mask must be odd!");

public:
  Mask();
  Mask(const T (&values)[N][N]);

  // Convolution
  Image operator*(const Image& image) const;

  T values[N][N];
  // A sum of all the positive and negative values in the mask, respectively
  // Used for remapping to [0,255] after convolution
  const T& posSum = _posSum;
  const T& negSum = _negSum;

private:
  T _posSum;
  T _negSum;
};

template <typename T, std::size_t N>
Mask<T, N>::Mask() {}

template <typename T, std::size_t N>
Mask<T, N>::Mask(const T (&values)[N][N]) {
  _posSum = _negSum = 0;
  for (unsigned i = 0; i < N; i++) {
    for (unsigned j = 0; j < N; j++) {
      Mask<T, N>::values[i][j] = values[i][j];
      (values[i][j] < 0 ? _negSum : _posSum) += values[i][j];
    }
```

```
  }
}

template <typename T, std::size_t N>
Image Mask<T, N>::operator*(const Image& image) const {
  Image re(image.rows, image.cols, image.maxVal);

  // x and y (and u and v) are interchanged from normal for cache locality purposes
#pragma omp parallel for collapse(2)
  for (unsigned y = 0; y < re.rows; y++) {
    for (unsigned x = 0; x < re.cols; x++) {
      // Keep track of a running sum in a temporary variable instead of in the
      // return image, since we probably have higher precision in T than in
      // pixelT.
      T sum = 0;

      for (unsigned v = 0; v < N; v++) {
        // Clamp the convolution to the image, "extending" the image by
        // its border pixels
        unsigned v_mod = (y + N / 2 < v) ?
                         0 :
                         ((y + N / 2 >= v + re.rows) ? (re.rows - 1) :
                                                       (y - v + N / 2));
        for (unsigned u = 0; u < N; u++) {
          // Clamp u too
          unsigned u_mod = (x + N / 2 < u) ? 0 :
                                           ((x + N / 2 >= u + re.cols) ?
                                                (re.cols - 1) :
                                                (x - u + N / 2));

          sum += values[v][u] * image[v_mod][u_mod];
        }
      }

      // re-map to [0,255]
      re[y][x] = (sum - _negSum * re.maxVal) / (_posSum - _negSum);
    }
  }

  return re;
}
```

Listing 2: Implementation file for the `correlate` program.

```
// Q1-Correlation/main.cpp
#include <iostream>
#include <fstream>
#include <sstream>
#include <vector>

#include "../Common/image.h"
```

```

/*
  Normalizes an image based min-max nomralization
  @Param: image - the input image that will be quantized
  @Param: corr_values - the correlation values
  @Return: void
*/
void normalize_image(Image& image, std::vector<int> corr_values){

  int max = 0;
  int min = 1000000000;
  int value;

  // find min an max correlation values
  for (int i = 0; i < image.rows; i++)
  {
    for (int j = 0; j < image.cols; j++)
    {
      value = corr_values[i * image.cols + j];

      if(value > max){
        max = value;
      }
      if(value < min)
        min = value;
    }
  }

  // scale each pixel and update image
  for (int i = 0; i < image.rows; i++)
  {
    for (int j = 0; j < image.cols; j++)
    {
      double scaled_value = 255.0 * ((corr_values[i*image.cols + j] - min) /
       ↪ (double)(max - min));
      image[i][j] = (int) scaled_value;
    }
  }
}

/*
  Correlates an image based on the image mask
  @Param: image - the input image that will be quantized
  @Param: mask - the image mask
  @Param: size_y - the mask height
  @Param: mask - the mask width
  @Return: void
*/
void correlation(Image& image, Image mask, int size_y, int size_x){


  Image originalImage = Image(image);
  std::vector<int> correlation_values;
```

```

  // iterate through image pixels
    for(int i = 0; i < image.rows; i++){
       for(int j = 0; j < image.cols; j++) {

         int sum = 0;

         // iterate over mask
         for (int k = -size_y/2; k < size_y/2; k++)
         {
           for (int l = -size_x/2; l < size_x/2; l++)
           {

             //check bounds
             if(i + k < 0 || i + k == size_x || j + l < 0 || j + l == size_y)
               sum += 0;
             else
               sum += originalImage[i + k][j + l] * mask[k + size_y/2][l + size_x/2];
           }
         }

         correlation_values.push_back(sum);
       }
     }

     // normalize image
     normalize_image(image, correlation_values);

}


int main(int argc, char** argv) {

   //std::istringstream ss(argv[3]);

  // Read original image
   std::ifstream inFile(argv[1]);
   Image image = Image::read(inFile);

   std::ifstream inMaskFile(argv[3]);
   Image mask = Image::read(inMaskFile);

    // Get mask width
    int mask_size_x;
    std::istringstream ss(argv[4]);
   if(ss >> mask_size_x) {
     if(mask_size_x < 1 || mask_size_x > image.cols){

       std::cout << "Error: Mask size should be greater than 0 and smaller than image size" << std::endl;
         return 1;
     }
```

```
   }

   // Get mask height
   int mask_size_y;
    std::istringstream ss2(argv[5]);
   if(ss2 >> mask_size_y) {
     if(mask_size_y < 1 || mask_size_y > image.rows){

       std::cout << "Error: Mask size should be greater than 0 and smaller than image size" << std::endl;
         return 1;
     }
   }

  std::cout << "Question 1: Correlation." << std::endl;

  // Correlate the image
  correlation(image, mask, mask_size_y, mask_size_x);
  std::cout << "Finished" << std::endl;

  // Save output image
  std::ofstream outFile;
  outFile.open(argv[2]);

  outFile << image;
  outFile.close();

  return 0;
}
```

Listing 3: Implementation file for the `smooth` program.

```
// Q2-Smoothing/main.cpp
#include <iostream>
#include <fstream>
#include <sstream>
#include <string>
#include <vector>

#include "../Common/image.h"

// 7x7 Gaussian mask
static const int mask_7x7[7][7] = {

    {1, 1, 2, 2, 2, 1, 1},
    {1, 2, 2, 4, 2, 2, 1},
    {2, 2, 4, 8, 4, 2, 2},
    {2, 4, 8, 16, 8, 4, 2},
    {2, 2, 4, 8, 4, 2, 2},
    {1, 2, 2, 4, 2, 2, 1},
    {1, 1, 2, 2, 2, 1, 1}

```

```
              };

//15x15 Gaussian mask
static const int mask_15x15[15][15] = {

  {2, 2,  3,  4,  5,  5,  6,  6,  6,  5,  5,  4,  3, 2, 2},
  {2, 3,  4,  5,  7,  7,  8,  8,  8,  7,  7,  5,  4, 3, 2},
  {3, 4,  6,  7,  9, 10, 10, 11, 10, 10,  9,  7,  6, 4, 3},
  {4, 5,  7,  9, 10, 12, 13, 13, 13, 12, 10,  9,  7, 5, 4},
  {5, 7,  9, 11, 13, 14, 15, 16, 15, 14, 13, 11,  9, 7, 5},
  {5, 7, 10, 12, 14, 16, 17, 18, 17, 16, 14, 12, 10, 7, 5},
  {6, 8, 10, 13, 15, 17, 19, 19, 19, 17, 15, 13, 10, 8, 6},
  {6, 8, 11, 13, 16, 18, 19, 20, 19, 18, 16, 13, 11, 8, 6},
  {6, 8, 10, 13, 15, 17, 19, 19, 19, 17, 15, 13, 10, 8, 6},
  {5, 7, 10, 12, 14, 16, 17, 18, 17, 16, 14, 12, 10, 7, 5},
  {5, 7,  9, 11, 13, 14, 15, 16, 15, 14, 13, 11,  9, 7, 5},
  {4, 5,  7,  9, 10, 12, 13, 13, 13, 12, 10,  9,  7, 5, 4},
  {3, 4,  6,  7,  9, 10, 10, 11, 10, 10,  9,  7,  6, 4, 3},
  {2, 3,  4,  5,  7,  7,  8,  8,  8,  7,  7,  5,  4, 3, 2},
  {2, 2,  3,  4,  5,  5,  6,  6,  6,  5,  5,  4,  3, 2, 2},
};

/*
  Smoothes an image based on averaging
  @Param: image - the input image that will be smoothed
  @Param: mask_size - the width and height of the mask
  @Return: void
*/
void smooth_image_average(Image& image, int mask_size){

  Image originalImage = Image(image);

  // Iterate through image pixels
    for(int i = 0; i < image.cols; i++){
       for(int j = 0; j < image.rows; j++) {

         int average = 0;

         //iterate through mask
         for (int k = -mask_size/2; k < mask_size/2; k++)
         {
           for (int l = -mask_size/2; l < mask_size/2; l++)
           {
             // calcualte average
             if(i + k < 0 || i + k >= image.cols || j + l < 0 || j + l >=
              ↪  image.rows)
               average += 0;
             else
               average += originalImage[i + k][j + l];
           }
         }

         image[i][j] = (int) (average / (mask_size * mask_size));
```

```
       }
     }
}

/*
  Smoothes an image based on the Gaussian mask
  @Param: image - the input image that will be smoothed
  @Param: mask_size - the width and height of the mask
  @Return: void
*/
void smooth_image_gaussian(Image& image, int mask_size){

  int normalizion_factor = 0;

  Image originalImage = Image(image);

  // calculate normalizion factor
  for(int i = 0; i < mask_size; i++){
    for (int j = 0; j < mask_size; j++){

      if(mask_size == 7)
        normalizion_factor += mask_7x7[i][j];
      else
        normalizion_factor += mask_15x15[i][j];
    }
  }

  // iterate through image pixels
    for(int i = 0; i < image.cols; i++){
       for(int j = 0; j < image.rows; j++) {

         int output_pixel_value = 0;

         // iterate through mask
         for (int k = -mask_size/2; k < mask_size/2; k++)
         {
           for (int l = -mask_size/2; l < mask_size/2; l++)
           {
             //bounds checking and padding
             if(i + k < 0 || i + k == image.cols || j + l < 0 || j + l == image.rows){
               output_pixel_value += 0;
             }
             //calculate output value
             else if(mask_size == 7)
               output_pixel_value += originalImage[i + k][j + l] * mask_7x7[k + mask_size/2][l + mask_size/2];
             else
               output_pixel_value += originalImage[i + k][j + l] * mask_15x15[k + mask_size/2 - 1][l + mask_size/2 - 1];
           }
         }

```

```
            // update image pixel
            image[i][j] = (int) (output_pixel_value / normalizion_factor);
          }
        }
}

int main(int argc, char** argv) {

    int mask_size;
    std::istringstream ss(argv[3]);

  // Get mask size
    if(ss >> mask_size) {
      if(mask_size != 7 && mask_size != 15){
        std::cout << mask_size << std::endl;
        std::cout << "Error: Mask size should be 7 or 15" << std::endl;
        return 1;

      }
    }

    // Get type of smoothing
    std::string filter_type = argv[4];
    if(filter_type != "average" && filter_type != "gaussian"){

      std::cout << "Error: Mask type should be average or gaussian" << std::endl;
      std::cout << filter_type << std::endl;
        return 1;
    }

  // Read original image
    std::ifstream inFile(argv[1]);

    Image image = Image::read(inFile);

  std::cout << "Question 2: Smoothing." << std::endl;

  // Smooth the image
  if(filter_type == "average")
    smooth_image_average(image, mask_size);
  else
    smooth_image_gaussian(image, mask_size);

  // Save output image
  std::ofstream outFile;
  outFile.open(argv[2]);
  outFile << image;
  outFile.close();

  return 0;
}
```

Listing 4: Implementation file for the median program.

```
// Q3-Median/main.cpp
#include <stdlib.h>
#include <stdio.h>
#include <iostream>
#include <fstream>
#include <sstream>
#include <string>
#include <string.h>
#include <vector>
#include <algorithm>
#include <utility>

#include "../Common/image.h"

/*
  Smoothes an image based on the Gaussian mask
  @Param: image - the input image that will be smoothed
  @Param: mask_size - the width and height of the mask
  @Param: noise_percentage - the amount of noise
  @Param: out - the output path
  @Return: void
*/
void smooth_image_average(Image& image, int mask_size, int noise_percentage, char*
↪  out){

  // Iterate though image pixels
  for(int i = 0; i < image.cols; i++){
       for(int j = 0; j < image.rows; j++) {

         int average = 0;

        // Iterate through mask
         for (int k = -mask_size/2; k < mask_size/2; k++)
         {
           for (int l = -mask_size/2; l < mask_size/2; l++)
           {
            // calcualte average
            if(i + k < 0 || i + k >= image.cols || j + l < 0 || j + l >=
            ↪  image.rows)
              average += 0;
            else
               average += image[i + k][j + l];
           }
         }

        // update pixel
         image[i][j] = (int) (average / (mask_size * mask_size));
       }
    }

    // save smoothed image
    std::string path(out);
```

```
    path = path.substr(path.find('/') + 1);
    path = path.substr(0, path.find('-'));
    path += "-smoothed-" + std::to_string(mask_size) + '-' + std::to_string(noise_percentage) + ".pgm";
    std::ofstream outFile;
    outFile.open("out/" + path);
    outFile << image;
    outFile.close();
}

/*
  Filters an image based median filtering
  @Param: image - the input image that will be filtered
  @Param: mask_size - the width and height of the mask
  @Return: void
*/
void median_filter(Image& image, int mask_size){

    Image orignalImage = Image(image);

    // Iterate though image pixels
    for(int i = 0; i < image.cols; i++){
      for(int j = 0; j < image.rows; j++) {

          std::vector<int> pixel_values;

         // Iterate though mask
          for (int k = -mask_size/2; k < mask_size/2; k++)
          {
            for (int l = -mask_size/2; l < mask_size/2; l++)
            {
              //check bounds and pad zeros if necessary
              if(i + k < 0 || i + k >= image.cols || j + l < 0 || j + l >= image.rows)
                pixel_values.push_back(0);
              else
                pixel_values.push_back(orignalImage[i + k][j + l]);

            }
          }

         // sort and get median value
          std::sort(pixel_values.begin(), pixel_values.end());

          image[i][j] = pixel_values[(mask_size * mask_size) / 2 + 1];
      }
    }
}

/*
  Adds noise to an image
  @Param: image - the input image to add noise to
  @Param: noise_percentage - the amount of noise
```

```
  @Param: out - the output path
  @Return: void
*/
void add_noise(Image& image, int noise_percentage, char* out){

    std::vector<int> indicies;
    int num_corrupt_pixels = (int) image.cols * image.rows * (noise_percentage /
    ↪  100.0);

    // save each image location
    for (int i = 0; i < image.cols * image.rows; i++)
    {
      indicies.push_back(i);
    }

    // shuffle indecies
    std::random_shuffle(indicies.begin(), indicies.end());

    // pick first X% of ranodm pixels
    for (int k = 0; k < num_corrupt_pixels; k++)
    {
      int i = indicies[k] / image.cols;
      int j = indicies[k] % image.cols;

      // generate noise value
      int noise_value = 0;
      if(std::rand() % 100 < 50)
        noise_value = 255;

      image[i][j] = noise_value;
    }

    // save noisy image
    std::string path(out);
    path = path.substr(path.find('/') + 1);
    path = path.substr(0, path.find('-'));
    path += "-noise-" + std::to_string(noise_percentage) + ".pgm";
    std::ofstream outFile;
    outFile.open("out/" + path);
    outFile << image;
    outFile.close();
}

int main(int argc, char** argv) {

   int mask_size;
   int noise_percentage;
   std::istringstream ss(argv[3]);

  // Get mask size
   if(ss >> mask_size) {
     if(mask_size != 7 && mask_size != 15){
       std::cout << mask_size << std::endl;
```

```
      std::cout << "Error: Mask size should be 7 or 15" << std::endl;
      return 1;

    }
  }

  std::istringstream ss2(argv[4]);

  // Get noise level
  if(ss2 >> noise_percentage) {
    if(noise_percentage > 100 || noise_percentage < 0){
      std::cout << noise_percentage << std::endl;
      std::cout << "Error: noise_percentage should be between 0 and 100" <<
      ↪  std::endl;
      return 1;

    }
  }

  // Read original image
  std::ifstream inFile(argv[1]);

  Image image = Image::read(inFile);

  std::cout << "Question 3: Median Filtering." << std::endl;

  // Add noise
  add_noise(image, noise_percentage, argv[2]);
  Image imageCopy = Image(image);

  // filter image
  median_filter(image, mask_size);

  // smooth image
  smooth_image_average(imageCopy, mask_size, noise_percentage, argv[2]);

  // Save output image
  std::ofstream outFile;
  outFile.open(argv[2]);
  outFile << image;
  outFile.close();

  return 0;
}
```

Listing 5: Implementation file for the `unsharp` program.

```
// Q4-Unsharp/main.cpp
#include <cmath>
#include <cstring>
#include <fstream>
#include <iomanip>
```

```cpp
#include <iostream>

#include "../Common/image.h"
#include "../Common/mask.h"

// Struct for inputting arguments from command line
struct Arguments {
  char *inputImagePath, *outImagePath;
  double A;  // High Boost option
  Image inputImage;
  std::ofstream outFile;
};

static const Mask<unsigned, 7> gauss7 = {{{1, 1, 2, 2, 2, 1, 1},
                                          {1, 2, 2, 4, 2, 2, 1},
                                          {2, 2, 4, 8, 4, 2, 2},
                                          {2, 4, 8, 16, 8, 4, 2},
                                          {2, 2, 4, 8, 4, 2, 2},
                                          {1, 2, 2, 4, 2, 2, 1},
                                          {1, 1, 2, 2, 2, 1, 1}}};

int unsharp(Arguments& arg);
bool verifyArguments(int argc, char** argv, Arguments& arg, int& err);
void printHelp();

int main(int argc, char** argv) {
  int err;
  Arguments arg;

  if (!verifyArguments(argc, argv, arg, err)) { return err; }

  return unsharp(arg);
}

int unsharp(Arguments& arg) {
  Image out = gauss7 * arg.inputImage;

  // Apply High Boost filter from original and low pass.
  // Then re-map values to [0, 255]
#pragma omp parallel for collapse(2)
  for (unsigned y = 0; y < out.rows; y++) {
    for (unsigned x = 0; x < out.cols; x++) {
      out[y][x] =
          (arg.A * arg.inputImage[y][x] - out[y][x] + out.maxVal) / (1 + arg.A);
    }
  }

  arg.outFile << out;
  arg.outFile.close();

  return 0;
}
```

Listing 6: Implementation file for the `gradient` program.

```
// Q5-Gradient/main.cpp
#include <cmath>
#include <cstring>
#include <fstream>
#include <iomanip>
#include <iostream>
#include <string>

#include "../Common/image.h"
#include "../Common/mask.h"

// Struct for inputting arguments from command line
struct Arguments {
  char *inputImagePath, *outImagePath;
  enum MaskType { PREWITT, SOBEL, LAPLACIAN } maskType;
  enum MaskDir { X, Y, MAGNITUDE, UNSPECIFIED } maskDir = UNSPECIFIED;
  Image inputImage;
  std::ofstream outFile;
};

static const Mask<int, 3> prewitt[2] = {{{{1, 0, -1}, {1, 0, -1}, {1, 0, -1}}},
                                          {{{1, 1, 1}, {0, 0, 0}, {-1, -1, -1}}}};
static const Mask<int, 3> sobel[2]   = {{{{1, 0, -1}, {2, 0, -2}, {1, 0, -1}}},
                                        {{{1, 2, 1}, {0, 0, 0}, {-1, -2, -1}}}};
static const Mask<int, 3> laplacian  = {{{0, 1, 0}, {1, -4, 1}, {0, 1, 0}}};

int gradient(Arguments& arg);
bool verifyArguments(int argc, char** argv, Arguments& arg, int& err);
void printHelp();

int main(int argc, char** argv) {
  int err;
  Arguments arg;

  if (!verifyArguments(argc, argv, arg, err)) { return err; }

  return gradient(arg);
}

int gradient(Arguments& arg) {
  Image out;
  switch (arg.maskType) {
    case Arguments::PREWITT:
      switch (arg.maskDir) {
        case Arguments::X:
          out = prewitt[0] * arg.inputImage;
          break;
        case Arguments::Y:
          out = prewitt[1] * arg.inputImage;
          break;
```

```
        case Arguments::MAGNITUDE:
          Image x = prewitt[0] * arg.inputImage;
          out     = prewitt[1] * arg.inputImage;
// Compute magnitude
#pragma omp parallel for
          for (unsigned i = 0; i < out.cols * out.rows; i++) {
            // Subtract out.maxVal / 2.0, since we have negative gradient
            // values, but we mapped to [0, 255]. Divide by 2.0 to map to
            // [0, 255] again
            out.pixels[i] = sqrt(((out.pixels[i] - out.maxVal / 2.0) *
                                      (out.pixels[i] - out.maxVal / 2.0) +
                                  (x.pixels[i] - out.maxVal / 2.0) *
                                      (x.pixels[i] - out.maxVal / 2.0)) /
                                 2.0);
          }
          break;
      }
      break;
    case Arguments::SOBEL:
      switch (arg.maskDir) {
        case Arguments::X:
          out = sobel[0] * arg.inputImage;
          break;
        case Arguments::Y:
          out = sobel[1] * arg.inputImage;
          break;
        case Arguments::MAGNITUDE:
          Image x = sobel[0] * arg.inputImage;
          out     = sobel[1] * arg.inputImage;
          // Compute magnitude
#pragma omp parallel for
          for (unsigned i = 0; i < out.cols * out.rows; i++) {
            // Subtract out.maxVal / 2.0, since we have negative gradient
            // values, but we mapped to [0, 255]. Divide by 2.0 to map to
            // [0, 255] again
            out.pixels[i] = sqrt(((out.pixels[i] - out.maxVal / 2.0) *
                                      (out.pixels[i] - out.maxVal / 2.0) +
                                  (x.pixels[i] - out.maxVal / 2.0) *
                                      (x.pixels[i] - out.maxVal / 2.0)) /
                                 2.0);
          }
          break;
      }
      break;
    case Arguments::LAPLACIAN:
      out = laplacian * arg.inputImage;
      break;
  }

  arg.outFile << out;
  arg.outFile.close();

  return 0;
```

```
}
```