Deev Patel & Wei Tong

CS 474: Image Processing & Interpretation

Monday, October 8, 2018

Programming Assignment # 2

Spatial Filtering

(Correlation, Smoothing, Median Filtering, Sharpening)

Division of Work

The smoothing and median filtering sections, of both the code and the report, were done by Wei. The correlation and sharpening sections, of both the code and the report, were done by Deev. Additionally, both Deev and Wei helped in creating the few global helper functions commonly used throughout the assignment. We believe this was an equal division of work because it gave each person responsibility over two of the four major sections of the project.

Technical Discussion

(1) Correlation

Correlation is a linear spatial filter that replaces each pixel value in an image with a linear combination of itself and its neighbors. The weights of the linear combination, which are often normalized to sum to one, are determined by a mask. Essentially, this mask of weights is placed with its center iteratively at every pixel value in a given image. Then, the filtered image is created from the resulting linear combination of each iteration. While this process is relatively straightforward, there is a minor hiccup involved when calculating filtered pixel values near the edges of an image. Here, it is common for the mask to extend outside the image boundaries and make certain weights correspond to an undefined value. We can solve this problem by padding the input image with zeros, extending the image's pixel values, or simply ignoring the image edges. In this assignment, we will use and consider the zero padding method.

Since the mask of weights is simply a matrix of numbers, it is possible to filter a large image using another smaller image as the mask. Doing this can be useful for template matching because the output of the correlation operation is likely to be high when the mask weights and the corresponding pixel values are equal. Thus, by thresholding the filtered image at a high value, we can find the location(s) where the mask image (specifically its center) overlaps with the input image. While this can be useful, it is important to note that the entire process only works when the weights of the mask overlap with same values. So, it would not work when the input image contains a rotated or scaled version of the mask image.

(2) Smoothing

Smoothing filters are low-pass linear spatial filters that eliminate small details and noise in an image. There are two common smoothing filters: averaging and Gaussian smoothing. In averaging, a mask whose values are all one is normalized and used to calculate the new value of the center. Essentially, all the pixel values in a neighborhood are added up (usually zero is used for areas outside the image boundaries) and divided by the number of pixels. The size of the neighborhood, or mask, determines how much detail is lost (larger masks lead to a greater loss in detail). For Gaussian smoothing, instead of averaging out the pixel values in the area corresponding to the mask, the values are weighted based on a sampling of the Gaussian function. The weights are higher towards the center of the mask and decrease as the distance from the center increases. However, the normalized sum of the weights still equals to one. This theoretically helps retain edges and other key characteristics when compared to averaging.

(3) Median Filtering

Median Filtering is a smoothing technique that is non-linear. It is generally used to remove salt and pepper noise because Median Filtering can remove noise involving extreme values while preserving much of the original image. The idea is to go through all the values in a neighborhood around a central value and replace it with the median. This process is then repeated iteratively with each pixel value as the center to obtain the filtered image. The edges and corners of an image are a bit tricky to process because pixels outside of the image are not considered. This can cause the noise to have a greater effect on the median. While the filtering processes works in many cases, it has problems when the noise values are not extreme or too numerous.

(4) Sharpening

Sharpening filters are high-pass spatial filters that seek to highlight details in an image. While there are a few different ways to accomplish this, a common approach is to make use of the fact that details are strongly associated with edges, or areas where pixel values are changing. If we consider an image as a two-dimensional continuous function, say f(x, y), then we can define the first derivative of the image, or its gradient, as shown in (Eq 1) below. Additionally, we can also define the divergence of the gradient, or the laplacian, as shown in (Eq 2). Both the gradient and laplacian can then be used to identify edges.

$$gradient(f) = \nabla f(x,y) = \left\langle \frac{\partial f(x,y)}{\partial x}, \frac{\partial f(x,y)}{\partial y} \right\rangle \quad (Eq\ 1)$$

$$laplacian(f) = \nabla^2 f(x,y) = \nabla \cdot \nabla f(x,y) = \frac{\partial^2 f(x,y)}{\partial^2 x} + \frac{\partial^2 f(x,y)}{\partial^2 y} \quad (Eq\ 2)$$

The gradient is simply a vector of the first derivatives of an image with respect to the x and y directions. Thus, the magnitude of the gradient, defined in (Eq 3) below, can tell us the change in pixel values with respect to location. Visually, these changes can be interpreted as edges. Additionally, the direction of a gradient vector can tell us which direction an edge is going because the direction is always normal to the edge.

$$\left|\nabla f(x,y)\right| = \sqrt{\left(\frac{\partial f(x,y)}{\partial x}\right)^2 + \left(\frac{\partial f(x,y)}{\partial y}\right)^2} \quad (Eq\ 3)$$

Since dealing with images necessarily implies working with the discrete case, we are forced to approximate the gradient. Generally this is best done using two 3x3 masks, one for the x direction and one for the y direction. In each case, we want a mask to find the difference in the pixel values on each side of the center. So, the center area is ignored with zero weights, while the

two sides have opposing weights. Two common such mask sets are the Prewitt and Sobel masks, shown below in Fig. 1. These masks can be used to calculate the two partial derivatives. Then, the two partial derivatives can be combined via (Eq 3) to give us the sharpened gradient magnitude image.

	$\frac{\partial f}{\partial x}$			$\frac{\partial f}{\partial y}$		
Prewitt Masks:	-1	0	1	-1	-1	-1
	-1	0	-1	0	0	0
	-1	0	-1	1	1	1
Sobel Masks:	-1	0	1	-1	-2	-1
	-2	0	-2	0	0	0
	-1	0	-1	1	2	1

Figure 1: The Prewitt and Sobel masks used in this assignment.

The laplacian operator is a second derivative operator that represents the divergence of the gradient vector. Since it is a second derivative operator, it responds to the changes in the gradient of the image. This means that it responds better to sharper edges, which occur at zero crossings in the laplacian image. Just like in the case of the gradient, we can estimate the laplacian using a 3x3 mask. Fig. 2 below shows one such common mask derived by adding one dimensional approximations of $\frac{\partial^2 f}{\partial^2 x}$ and $\frac{\partial^2 f}{\partial^2 y}$. Notice that since the laplacian is not a vector, there is only one mask. So, it is computationally easier to find the laplacian compared to the gradient. However, the laplacian does not give corresponding information about the direction of edges.

0	1	0
1	-4	1
0	1	0

Figure 2: The Laplacian mask used in this assignment.

Implementation & Results

General Functionality

To help implement the various different spatial filters, we defined two major functions. For reference, they are located in the "Global Helper Functions" section of the Program Listings. The first function, called *remapValues()*, linearly remaps an image's pixel values back to normal range of [0, 255] for visualization/storage purposes. The second function, called *applyMask()*, takes an image and mask, both as ImageType variables, and returns the result of applying the mask at a specified location. We figured this general functionality could be used in the correlation, smoothing, and sharpening parts of the assignment because they all require a mask to be applied to an image at one of more location(s). The function also checks to see if any mask weights correspond to positions outside the given image. If this is the case, it ignores that specific weight. Essentially, the function removes the need to explicitly pad images with zeros when applying a mask. The function also allows one to normalize the mask and use double precision, something that is useful for correlation. While calling this function repeatedly is not ideal computationally, it makes implementing the various parts of assignment more straightforward and understandable.

(1) Correlation

The implementation of correlation simply reads the input and mask images as ImageType variables. Then, it applies the *applyMask()* helper function at every pixel location to create an output image. Finally, the output image is remapped to the default range of [0, 255] and saved. The results of this process are shown in Fig 3. and Fig 4.

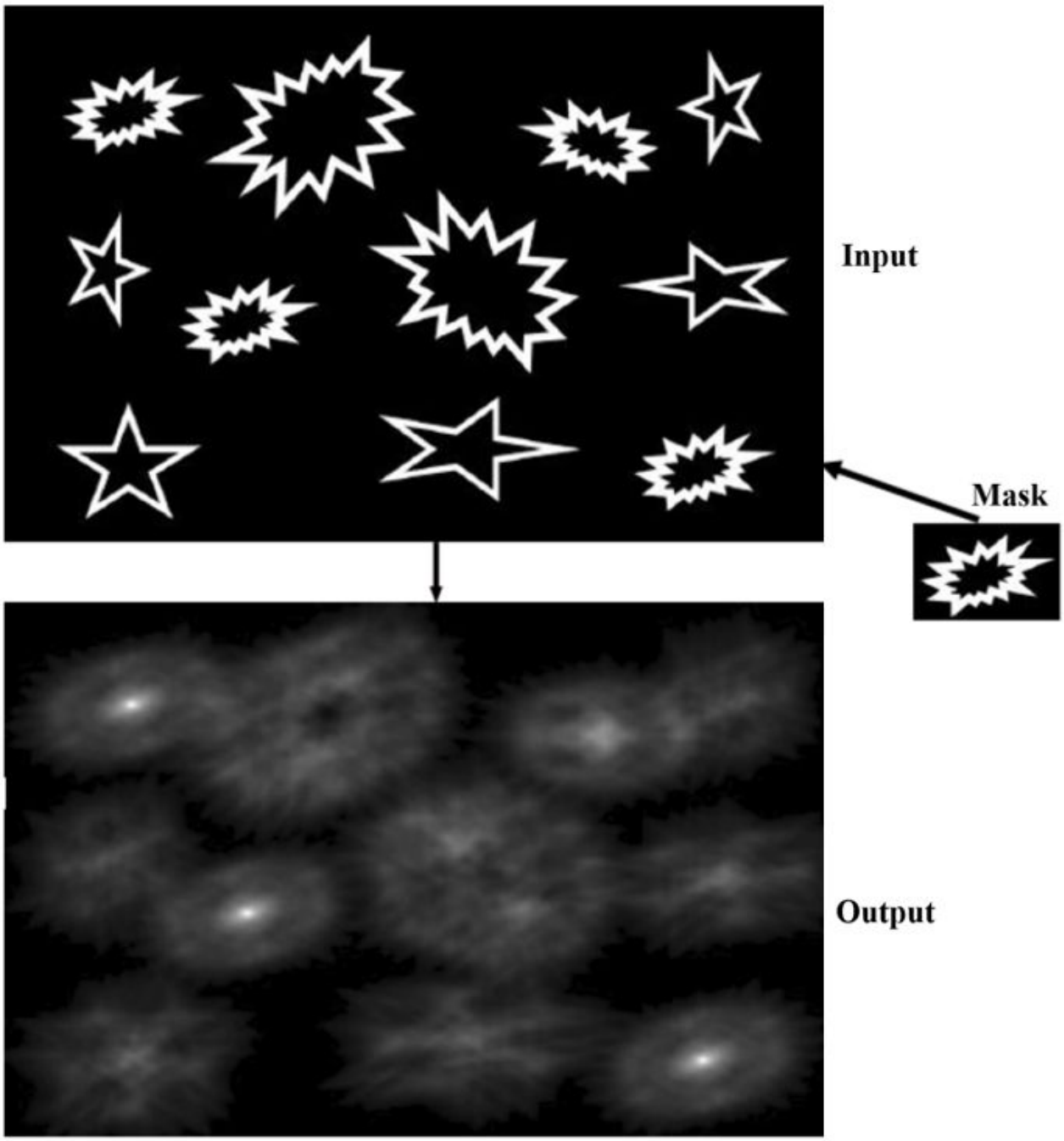

Figure 3: The results of the correlation program. The top image was filtered with the mask image shown in the middle to obtain the resultant bottom image.

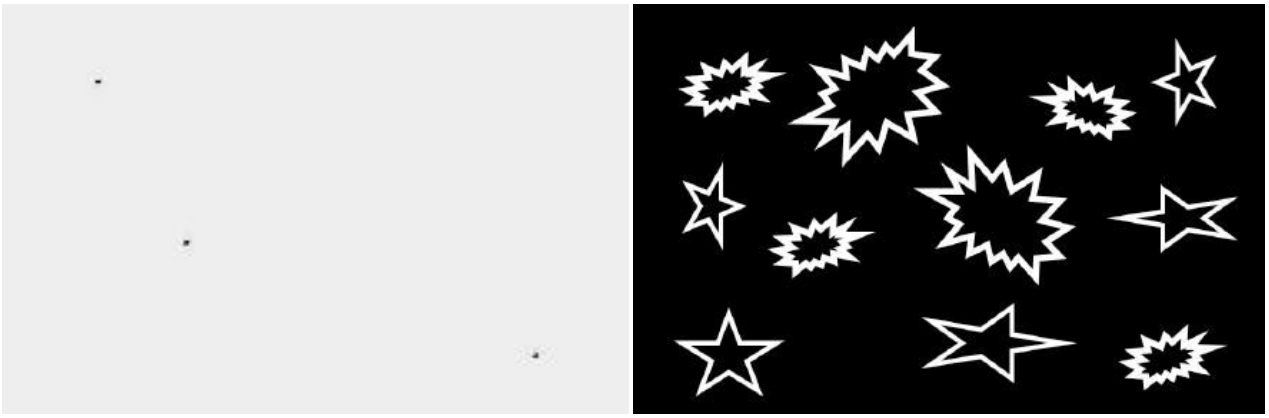

Figure 4: On the left is the output image of correlation (from bottom of Fig. 3) run through a threshold of 220. The threshold image is overlaid by a gray background to make it easier to visually see where the three black dots lie with respect to the original image on the right.

(2) Smoothing

The implementation of smoothing was done by reading in the input image as an ImageType object and then creating a mask from a hard-coded matrix based on user input. The Gaussian mask had to be created by hand while the averaging mask was created by simply looping through the matrix and setting each value to one. The *applyMask()* function was then used at every pixel with the input image and the generated mask. Notably, the function was called with the normalize flag enabled so that the mask would be normalized to sum to 1. The results of the *applyMask()* function were then used to generate and save the output image. The final results after renormalization can be found in Fig. 5.

Figure 5: The output of the smoothed "lenna" and "sf" images. The first and third lines show the result of averaging. The second and fourth lines show the result of Gaussian smoothing

(3) Median Filtering

The implementation of Median Filtering was done by reading in the image specified and creating a mask with a user-specified size. The input images was then corrupted twice, once with 30% of the pixels being corrupted into either white or black, and the second time with a 50% corruption rate. After outputting the corrupted images for verification, as shown in Fig. 6, Median Filtering was applied.

To accomplish Median Filtering, each pixel in the image was iterated through. In each iteration, all the pixel values in the neighborhood were added to a one dimensional array. After sorting the array, the median number was selected to be the new pixel value. The results can be seen in Fig. 7. For comparison, averaging was also done on the corrupted images The averaging, shown in Fig. 8, was done through the same process mentioned earlier.

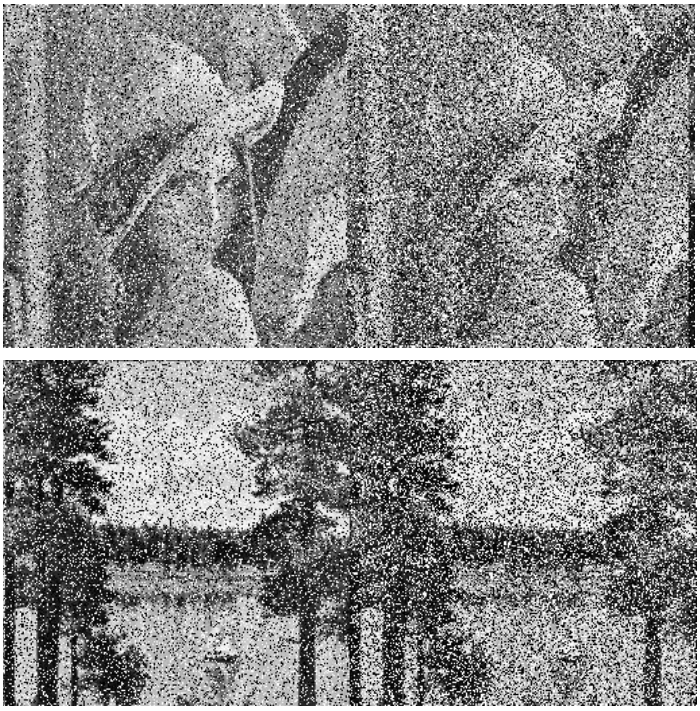

Fig. 6: Results of corrupting the "lenna" (top) and "boat" (bottom) images with salt-and-pepper noise. On the left is 30% corruption. On the right is 50% corruption.

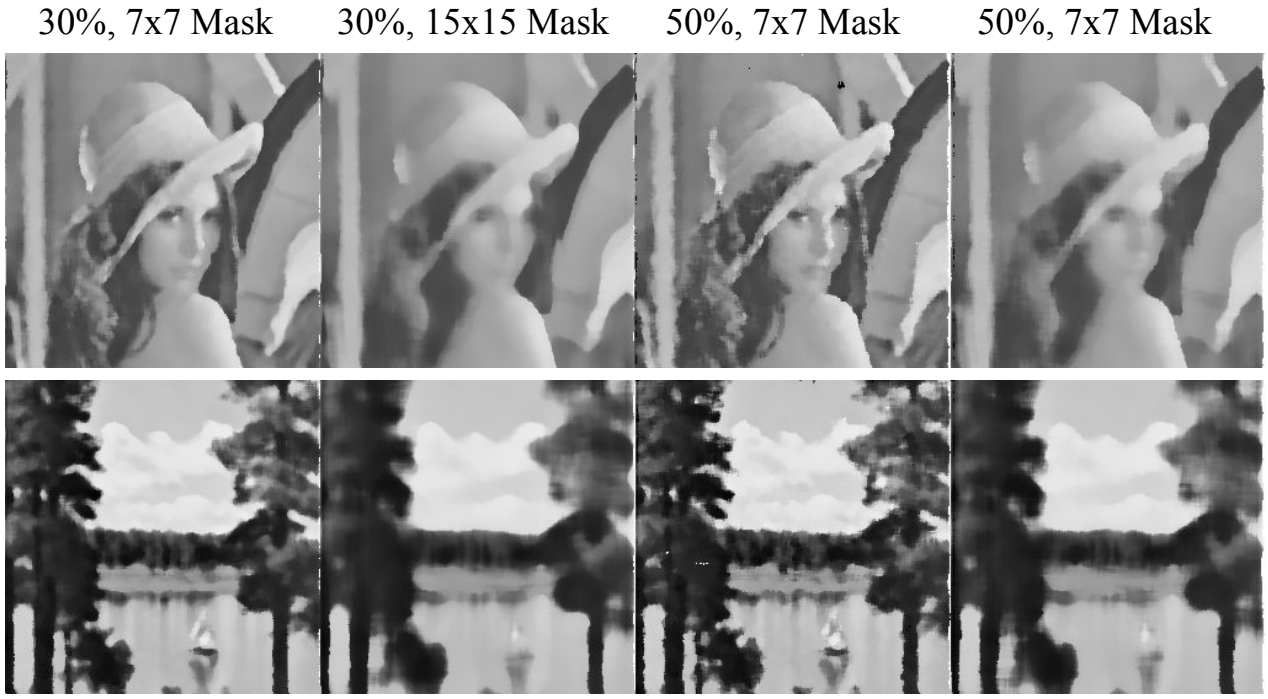

Fig. 7: Results of Median Filtering of the corrupted "lenna" and "boat" images. From left to right: 30% corruption with 7x7 mask, 30% corruption with 15x15 mask, 50% corruption with 7x7 mask, 50% corruption with 15x15 mask.

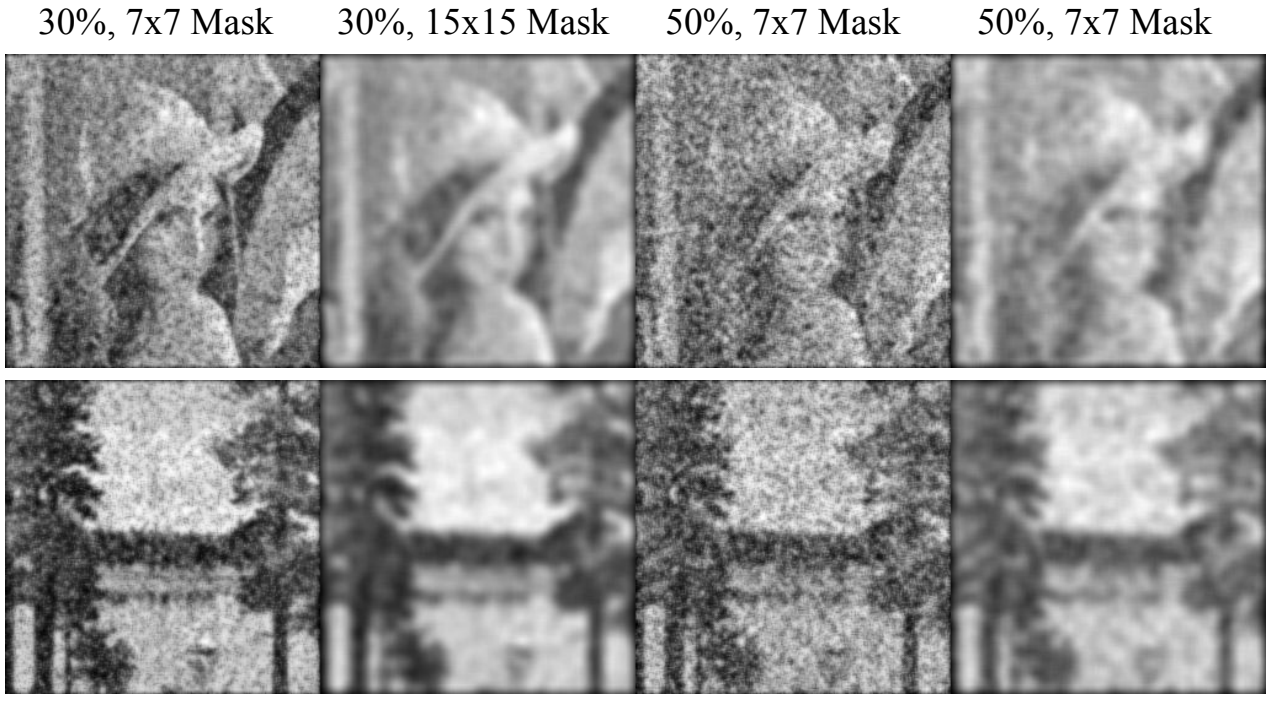

Fig. 8: Results of averaging of the corrupted "lenna" and "boat" images. From left to right: 30% corruption averaged with a 7x7 mask, 30% corruption averaged with a 15x15 mask, 50% corruption averaged with a 7x7 mask, and 50% corruption averaged with a 15x15 mask.

(4) Sharpening

The Gradient part of the sharpening program was implemented by creating functions to generate the Prewitt and Sobel masks from Fig. 1 as ImageType variables. In each case, the *applyMask()* helper function was applied at every point in the input image. This was done twice, once with each mask, creating two images. Then, the magnitude image was found via (Eq 3). Finally, this magnitude image was renormalized. The results are shown in Fig. 9 through Fig. 11.

Figure 9: The result of the Prewitt and Sobel gradient operators on the “lenna” image

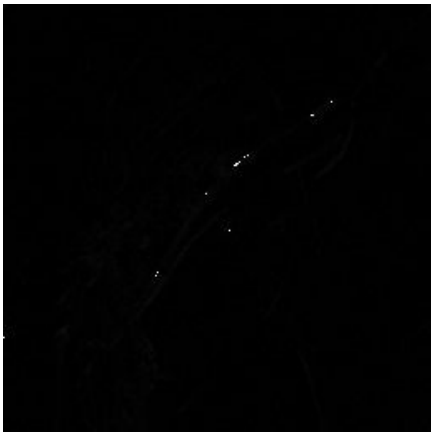

Figure 10: Image showing the minor differences between the Prewitt and Sobel gradient magnitude images for the "lenna" image. The picture was generated by subtracting the Sobel gradient image from the Prewitt gradient image. The white areas represent differences.

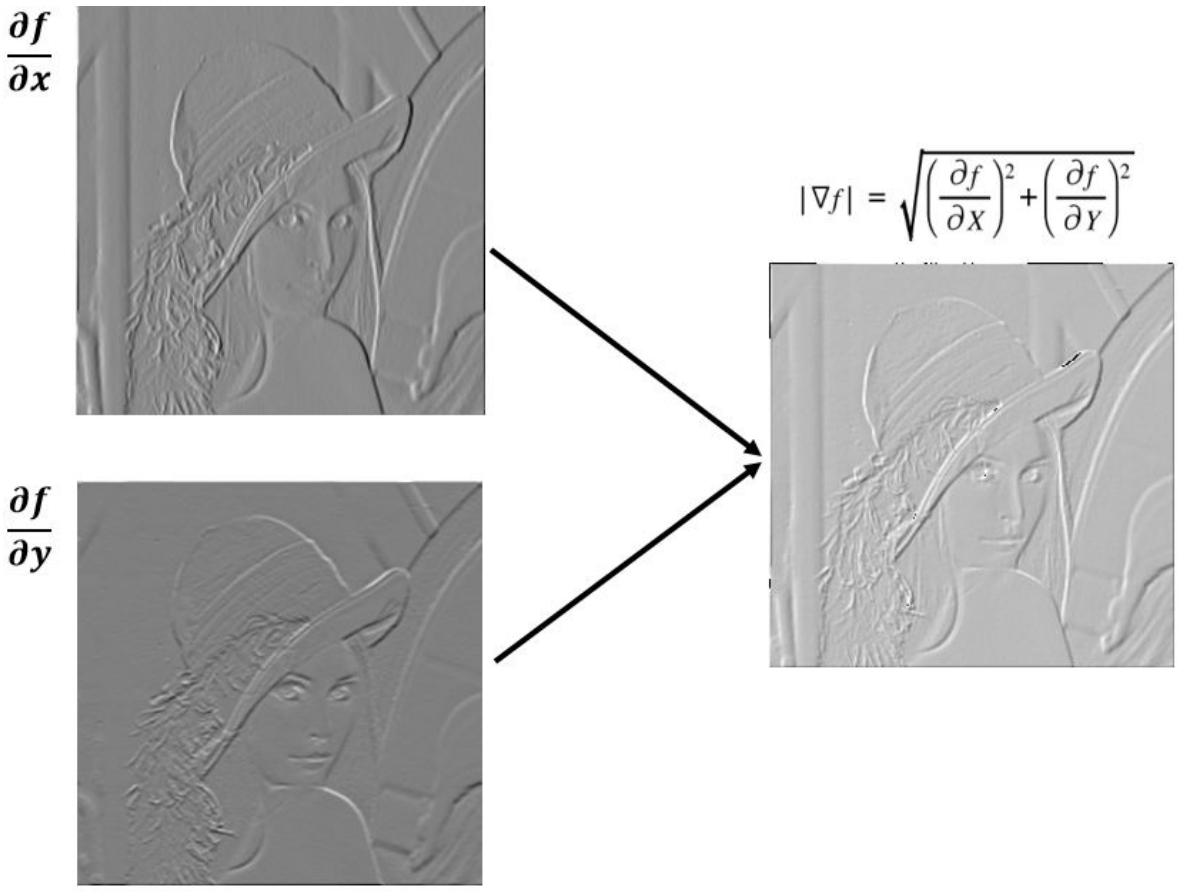

Figure 11: On the left are the x and y gradient images created by the Prewitt masks for the "lenna" image. On the right is the combined gradient magnitude image.

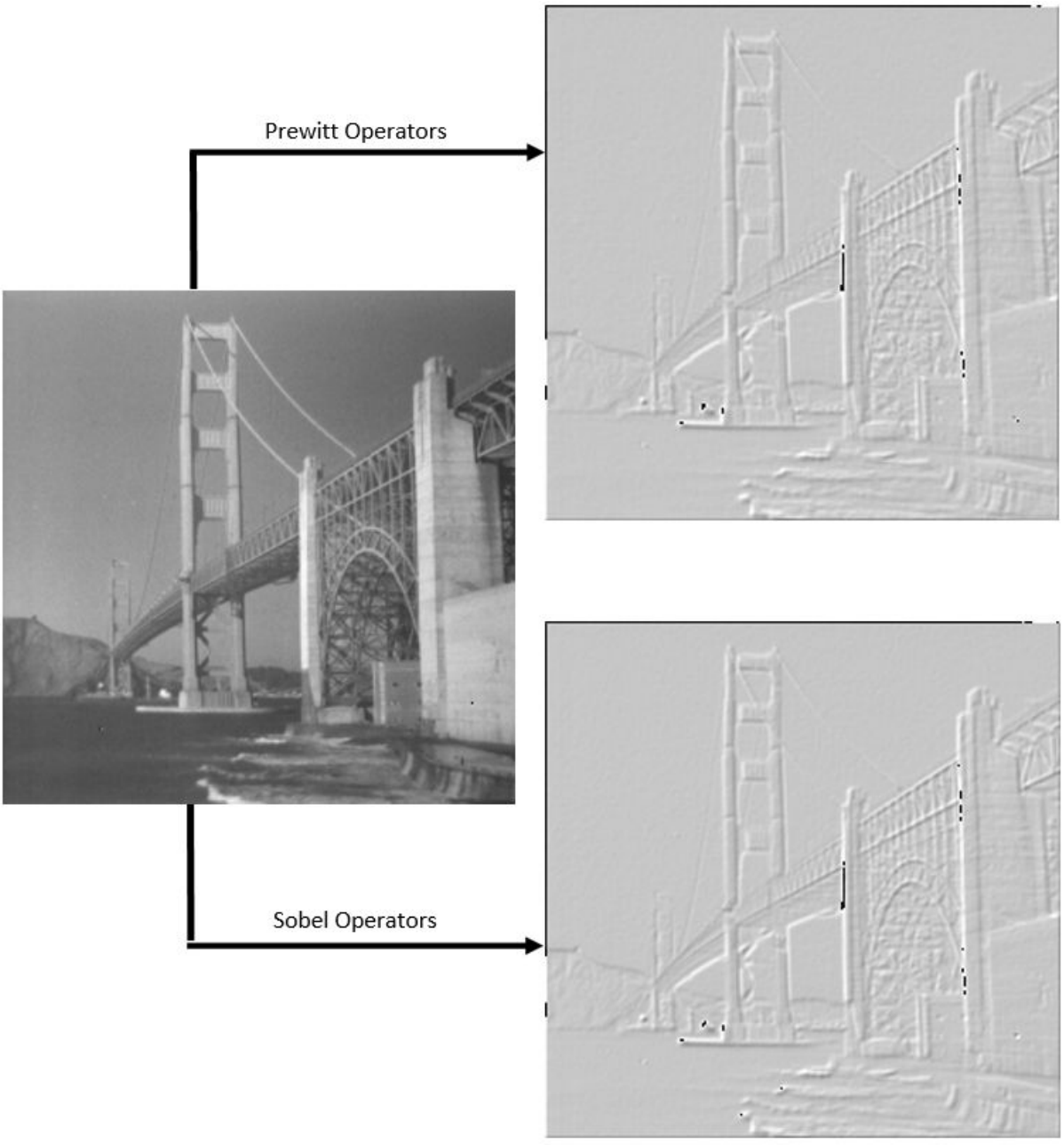

Figure 12: The result of the Prewitt and Sobel gradient operators on the "sf" image.

The Laplacian part of the sharpening program was implemented by creating a function to generate the laplacian mask from Fig. 2. The mask was applied at all locations via the helper function *applyMask()*. Finally the output was renormalized. The results are shown below.

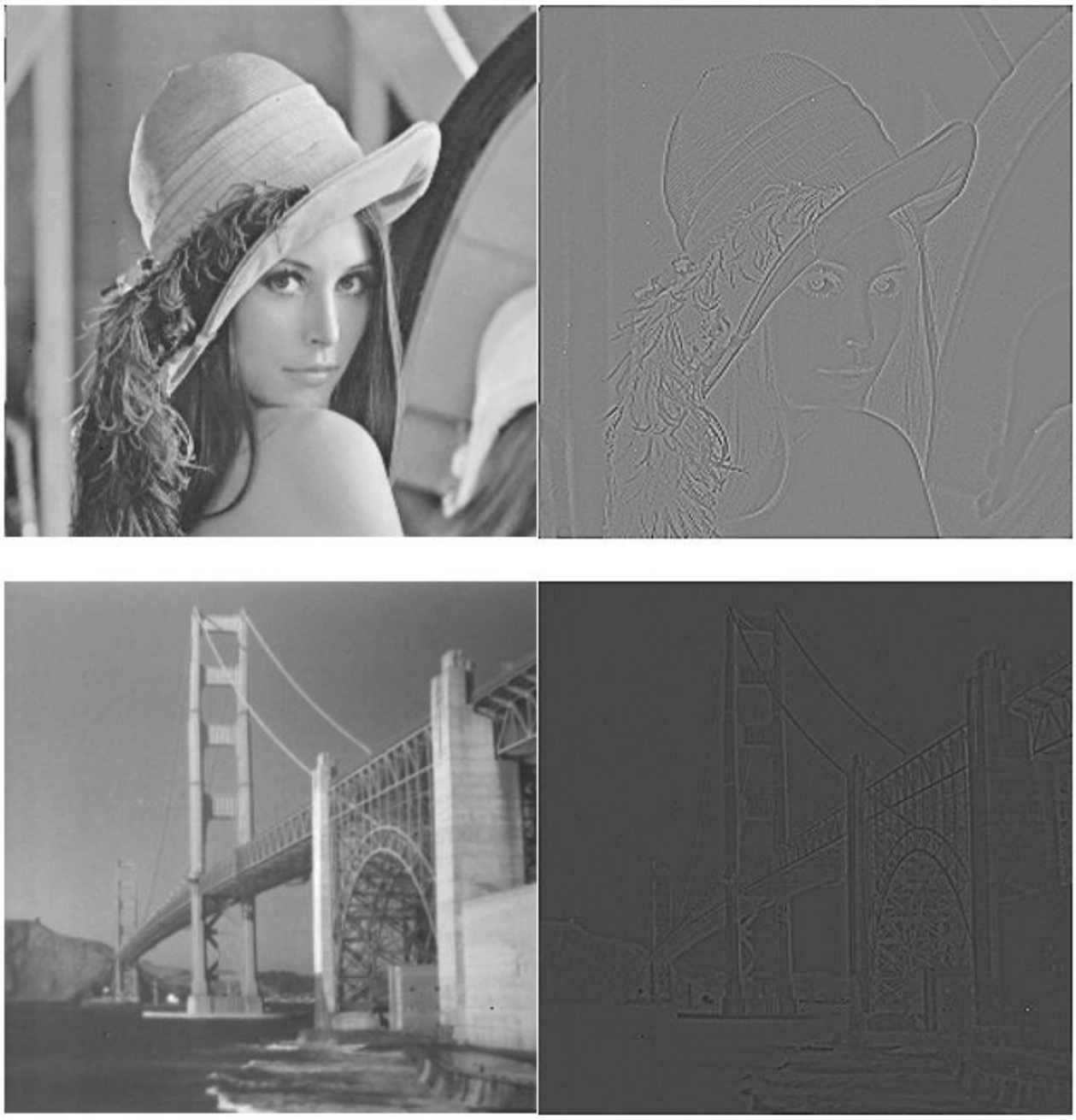

Figure 13: The results of the laplacian operator. The top shows the result for “lenna” image, while the bottom shows the result for the “sf” image.

Discussion

(1) Correlation

In the resultant image of the correlation process, shown in the bottom of Fig. 3, three relatively bright spots can be seen. These spots indicate the three unique locations of the mask in the original image. This is especially evident when the dots are isolated via thresholding as shown in Fig. 4. The fact that the black dots are so precise and accurate as to the location of the mask image shows the power of correlation followed by thresholding in template matching.

However, beyond these three bright locations, the output image of the correlation process is quite hard to visually decipher. Moreover it is nearly impossible to reverse the process and tell what input pixels caused the output. Thus, the visual result of the correlation process leaves us with little absolute understanding of areas that do not match the template accurately. Notably, areas that contain a scaled or rotated version of the mask are nearly indistinguishable from areas that contain a completely different object. This is evidenced by the fact that, in the resultant image, it is hard to find differences between the locations of the 5-pointed stars and the unmatched multi-pointed stars.

(2) Smoothing

In the output images for the smoothing process shown in Fig. 5, it can be seen that a larger mask causes more blur. Specifically, many of the edges are much smoother in the images processed with a 15x15 mask compared to the images processed with a 7x7 mask. For the “lenna” image, the hair details in the 15x15 image are almost completely gone, as are the details of the

hat. The same thing happens in the “sf” image, where the thin wires and beams of the bridge become indiscernible.

There are also some minor differences between the images processed by averaging and Gaussian smoothing. They are not very noticeable in the “lenna” image, but can be seen in the “sf” image. When comparing the bridges, there is more detail in the cables and beams in the images processed by the Gaussian mask compared to the images processed via averaging.

(3) Median Filtering

The function to corrupt images by adding salt-and-pepper noise did its job very well. It can easily be seen in Fig. 6 that the images have a large amount of noise, something that is especially bad in the images with 50% noise. Despite that, a person can still figure out roughly what the original image looks like. Fig. 8 shows the abilities of the averaging filter discussed previously in restoring the images. From the images, it is clear that large amounts of salt-and-pepper noise remain, although the larger mask did seem to remove more of it. However, the 15x15 mask also removed so much detail from the images that the original images could only be guessed at. Essentially, averaging is not a good technique to use on images with salt-and-pepper noise, because too much detail is removed before the noise goes away.

The results of Median Filtering can be seen in Fig. 7. In the images that were only corrupted at 30%, the 7x7 mask is more than enough to remove the majority of the noise. However, at 50% corruption, there is a white streak on the left tree when using the 7x7 mask. This is most likely due to a heavy concentration of white noise in that area. This white streak does not appear when the 15x15 mask was used. In the results of median filtering, the general shape of the

images is kept, along with some detail. In both the “lenna” and “boat” images, it seems that using the 7x7 mask for Median Filtering results in an image of similar quality, even when the corruption was raised from 30% to 50%. Using the 15x15 mask also produces similar images, although they are much fuzzier than their 7x7 counterparts. It seems that the size of the mask used for Median Filtering affects the quality of the resultant image. This can most likely be attributed to the fact that a larger mask is affected by more variability from the original image, since a larger portion of the image is included in the mask.

(4) Sharpening

The gradient operators worked well in bringing out the details and highlighting the edges. In Fig. 9, we can see that both the Prewitt and Sobel operators created similar outputs for the “lenna” image. The areas where the pixel values change quickly with regards to location are the most notable in the sharpened images. For instance, the curved black band in the top right heavily stands out in the output images. This is because the gradient operators respond to changes in pixel values and the black band indicates an area of abrupt change with respect to the background.

One thing that was surprising was how little difference there was between using the Prewitt and Sobel masks. In fact, from Fig. 10, we can see that the only real differences between the two operators for the “lenna” image occurred along the crease in hat. In general the crease along the hat seems to be an area where the gradient is very high, something that is probably fueled by the thin nature of the crease line. In addition to the crease line, the gradient was very high along the edges. This is to be expected as padding the image with zeros can cause a relatively high gradient when the edges of the image contain high values. It probably would have

been better for visualization process to simply ignore the edges so that the normalization process wouldn't be skewed by the high values near the edges.

In Fig. 11, we can see how the two directional derivatives responded to the "lenna" image. Through the prominent nature of the bar of the left, it is obvious that the mask estimating the partial derivative with respect to x responded well to vertical changes. Similarly, it is evident that the mask for the partial derivative with respect to y responded to horizontal changes.

In Fig. 12, we can see many of the same results mentioned above for the "lenna" image occur with the "sf" image. Here, the Prewitt and Sobel operators once more created similar images. Moreover, there again seemed to be a high gradient near thin lines and the along the sides of the image. For this image specifically, the gradient operator had a hard time with the framing below the right side of the bridge, where the mesh of wires created many abrupt changes in pixel values.

The laplacian operator seemed to be more concise in finding edges. In Fig. 13, we can see that the laplacian operator mapped edges to much smaller areas for both the "lenna" and "sf" images. This is mainly due to the fact that the laplacian is a second derivative operator that responds to changes in the gradient. So, the original edge lines were displayed more thinly because they were derived from the thicker versions in the gradient. This resulted in a much better output for areas with many edges, such as underneath the right portion of the bridge in the "sf" image. Additionally, the laplacian seemed to be visually better in areas where the gradient magnitude was very large. For instance, the laplacian created a much more consistent output along the crease line of the hat in the "lenna" image.

Program Listings

Global Helper Functions

Below are the two main functions of Helper namespace. They are used throughout the other sections of the code. Some minor functions, such as those to find minimum and maximum pixel values in an image, are not shown due to their simplicity.

```
//Function to linearly remap values in an image to the range [0-255]
void Helper::remapValues(ImageType & img) {
    int currMin = findMinPixelVal(img); //find minimum value
    int currMax = findMaxPixelVal(img); //find maximum value

    int rows, cols, levels;
    img.getImageInfo(rows, cols, levels); //get image information

    int currVal, newVal;
    for (int r = 0; r < rows; ++r)
        for (int c = 0; c < cols; ++c) {
            img.getPixelVal(r, c, currVal);
            //find new value using linear transformation from class
            newVal = (((double) currVal - currMin)
                      / (currMax - currMin)) * 255;

            img.setPixelVal(r, c, newVal); //change value
        }
}

//Function to apply given mask at given location in image.
//Will treat locations outside the image as having a 0 pixel value.
//maskCenterCol & maskCenterRow help define center of mask with (0,0)
//at the top left corner of the mask.
double Helper::applyMask(const ImageType & img,
                         const ImageType & mask, int row, int col,
                         unsigned int maskCenterR,
                         unsigned int maskCenterC,
                         bool normalizeMask = true) {
```

```
    int imgRows, imgCols, maskRows, maskCols, levels;
    img.getImageInfo(imgRows, imgCols, levels); //get image info
    mask.getImageInfo(maskRows, maskCols, levels); //get mask info

   //ensure given mask center is a valid location
    assert(maskCenterR <= maskRows && maskCenterC <= maskCols);

    int currVal;
    img.getPixelVal(row, col, currVal); //get current value
    double newVal = 0;

    //iterate through mask weights and get matching image value
    int weight, value, rowLoc, colLoc, rowOffset, colOffset;
    for (int i = 0; i < maskRows; ++i)
         for (int j = 0; j < maskCols; ++j) {
              mask.getPixelVal(i, j, weight); //get weight value

              //find offsets from location in image
              //used to find corresponding image value of weight
              rowOffset = i - maskCenterR;
              colOffset = j - maskCenterC;

              //find location in image to use current weight on
              rowLoc = row + rowOffset;
              colLoc = col + colOffset;

              //add weighted value if location is inside the image
              if (rowLoc >= 0 && rowLoc < imgRows &&
                  colLoc >= 0 && colLoc < imgCols) {
                   img.getPixelVal(rowLoc, colLoc, value);
                   newVal += (value * weight);
              }
    }
    if (normalizeMask)
         newVal /= getSumOfPixels(mask); //normalize if requested

   return newVal;
}
```

(1) Correlation

Below is the relevant part of the correlation code. Right before this section of code, the original and mask images were read in as ImageType variables. Right after this section, the output image was stored.

```
//find center of mask - could be uneven for image with even rows/cols
int maskCenterRow = maskRows / 2;
int maskCenterCol = maskCols / 2;

//apply mask to every pixel of image and store result in output
double newVal;
for (int r = 0; r < imgRows; ++r)
     for (int c = 0; c < imgCols; ++c) {
          //apply mask with normalization to prevent overflow
          //use global helper function to find each specific result
          newVal = Helper::applyMask(img, mask, r, c,
                                     maskCenterRow, maskCenterCol);

          outputImg.setPixelVal(r, c, (int) newVal); //save value
     }

Helper::remapValues(outputImg); //remap image values to [0 255]
```

(2) Smoothing

On the next page is the relevant part of the smoothing code. Right before this section, the mask was hard-coded in, depending on whether the user specified a 7x7 or 15x15 mask and if they wanted to do averaging or Gaussian smoothing. The different choice selections would simply change what mask was passed to the *applyMask()* helper function (e.g. am_7x7 for 7x7 mask averaging). Right after this section, the output image was stored with the desired name.

```
ImageType outputImg(N, M, Q);
double newVal;
for(int i = 0; i < N; ++i)
      for(int j = 0; j < M; ++j){
            // Apply mask with normalize to obtain new value
            newVal = Helper::applyMask(image, am_7x7, i, j, 3, 3);
            outputImg.setPixelVal(i, j, (int) newVal);
      }
Helper::remapValues(outputImg); // Remap image values to [0, 255]
```

(3) Median Filtering

Below is the salt-and-pepper corrupting function. The percentage of the image to be corrupted is passed in and used to calculate if a pixel needs to be changed before storing in the new image object.

```
void saltPepperCorruption(const ImageType &og_image,
                          ImageType &new_image, int percentage){
    int numRows, numCols, numLevels, roll, tmpVal;
    og_image.getImageInfo(numRows, numCols, numLevels);
    srand(time(NULL));

    // Iterate through all pixel values
    for(int i = 0; i < numRows; ++i)
        for(int j = 0; j < numCols; ++j){
            roll = rand() % 100;
            if(roll < (percentage / 2))
                new_image.setPixelVal(i, j, 255); // Make it white
            else if(roll < percentage)
                new_image.setPixelVal(i, j, 0); // Make it black
            else{
                og_image.getPixelVal(i, j, tmpVal);
                new_image.setPixelVal(i, j, tmpVal); // Keep value
            }
      }
}
```

Below is the function used to apply the median filtering technique to the image. The user specified mask size is passed to the function. A one dimensional array is used to store each value in the neighborhood. Then, the median of the array is used as the pixel value in the new image.

```
void medianMask(const ImageType &og_image, ImageType &new_image,
                int maskSize){
   int values[maskSize * maskSize]; // Array to store mask values
   int imgRows, imgCols, imgLevels, count = 0;
   int rowOffset, colOffset, rowLoc, colLoc, sortedMedianValue;
   og_image.getImageInfo(imgRows, imgCols, imgLevels);

   // Iterate through all pixels in input image
   for(int i = 0; i < imgRows; ++i)
      for(int j = 0; j < imgCols; ++j){
        // Iterate in neighborhood
         for(int k = 0; k < maskSize; ++k)
            for(int l = 0; l < maskSize; ++l){
               rowOffset = l - (maskSize / 2);
               colOffset = k - (maskSize / 2);
               rowLoc = i + rowOffset;
               colLoc = j + colOffset;

               // only consider values in valid locations
               if(rowLoc >= 0 && rowLoc < imgRows &&
                  colLoc >= 0 && colLoc < imgCols){
                 og_image.getPixelVal(rowLoc, colLoc, values[count]);
                 count++;
               }
            }

      std::sort(values, values + count);
      // Median is found and set to new image
      sortedMedianValue = values[count / 2];
      new_image.setPixelVal(i, j, sortedMedianValue);
      count = 0;
       }
}
```

(4) Sharpening

For the sharpening section of program, two major helper functions, shown below, were defined in addition to the ones already shown in the "global helper functions" section. There were also functions defined to generate the Prewitt, Sobel, and Laplacian masks. These functions simply generated the masks required by the assignment as ImageType variables.

```
//function to apply mask at every point and save result to a given
//output image after remapping back to viewable range
void applyMaskEverywhere(const ImageType & img,
                         const ImageType & mask,
                         ImageType & outputImg) {

   int imgRows, imgCols, maskRows, maskCols, levels;
   img.getImageInfo(imgRows, imgCols, levels); //get image info
   mask.getImageInfo(maskRows, maskCols, levels); //get mask info

   //apply mask at every pixel location and store result in output
   double newVal;
   for (int r = 0; r < imgRows; ++r)
      for (int c = 0; c < imgCols; ++c) {
         //get new value with global helper function
         //do not normalize mask since it is sharpening mask
         newVal = Helper::applyMask(img, mask, r, c,
                                    (int) (maskRows / 2),
                                    (int) (maskCols / 2), false);

         outputImg.setPixelVal(r, c, (int) newVal); //save result
      }

   Helper::remapValues(outputImg); //remap values into [0,255]
}
```

```
//helper function to find gradient magnitude given the partial
//derivatives with respect to X and Y. Only use for gradient.
void calculateGradientMag(const ImageType & gradientX,
                          const ImageType & gradientY,
                          ImageType & gradientMag) {

   int imgRows, imgCols, levels;
   gradientMag.getImageInfo(imgRows, imgCols, levels); //get info

   //calculate magnitude at every pixel value
   int x, y, mag;
   for (int r = 0; r < imgRows; ++r)
      for (int c = 0; c < imgCols; ++c) {
         gradientX.getPixelVal(r, c, x); //get x value
         gradientY.getPixelVal(r, c, y); //get y value

         //use magnitude formula find magnitude of gradient
         mag = (int) std::sqrt(xVal * xVal + yVal * yVal);
         gradientMag.setPixelVal(r, c, mag); //store magnitude
      }
}
```

Below is the major part of the code for the application of the two gradient (Prewitt and Sobel) and laplacian operators. It makes heavy use of the functions already shown above.

```
ImageType outputImg(imgRows, imgCols, Q); //final output image

#if PREWITT
ImageType maskX(3, 3, 255), maskY(3, 3, 255);
ImageType outputX(imgRow, imgCol, Q), outputY(imgRow, imgCol, Q);

generatePrewitt(maskX, maskY); //generate Prewitt masks

applyMaskEverywhere(img, maskX, outputX); //apply X mask
applyMaskEverywhere(img, maskY, outputY); //apply Y mask
```

```
//calculate magnitude to get output image
calculateGradientMag(outputX, outputY, outputImg);

#elif SOBEL
ImageType maskX(3, 3, 255), maskY(3, 3, 255);
ImageType outputX(imgRow, imgCol, Q), outputY(imgRow, imgCol, Q);

generateSobel(maskX, maskY); //generate Sobel masks

applyMaskEverywhere(img, maskX, outputX); //apply X mask
applyMaskEverywhere(img, maskY, outputY); //apply Y mask

//calculate magnitude to get output image
calculateGradientMag(outputX, outputY, outputImg);

#elif LAPLACIAN
ImageType mask(3, 3, 255);

generateLaplacian(mask); //generate Laplacian mask

//apply Laplacian mask to get output image
applyMaskEverywhere(img, mask, outputImg);
#endif
```