# CS 474/674 - Image Processing and Interpretation

# Programming Assignment 2

Submitted by-

Andrew Wiltberger and Akshay Krishna

Due Date: October 28th, 2020

Handover Date: October 28th, 2020

**Division of Work:**

Andrew Wiltberger and Akshay Krishna split the programming and report writing equally. Andrew covered the topics, Correlation, and Averaging & Gaussian Smoothing. Simultaneously, Akshay covered the topics Unsharp Masking & High Boost Filtering and Gradient & Laplacian. For the topic Median Filtering, Akshay implemented the image corruption and the Median Filtering part. At the same time, Andrew implemented the Gaussian Filtering for comparison. Both Andrew and Akshay contributed to the sections: Theory, Implementation, Results, and Discussion of their respective topics.

**Theory:**

The first part of this assignment is focused on image correlation. Image correlation is a linear operator that operates on a neighborhood of pixels in an image. The output is a weighted sum of the neighborhood of pixels. To compute a correlation, you must have an input image g(x,y), an output image f(x,y), and a transformation T[ ]. The transformation T will depend on the mask size you pick and the weights in the mask. One important aspect is that the weights of the mask add up to one. To find a pixel value on the output image at (x,y), you would apply the mask to the input image with the center at (x,y). Then the output pixel value is just a weighted sum of nearby pixels in the input. If the mask goes off the image, then the image will be padded with zeros. An application of image correlation is template matching. If there is an object that we want to find in the input image, then we can make a mask of that object and compute a correlation on the image using that mask. This will allow us to find the similarity between the two images and find if the object is in the input image. However, simple template matching does not work in most real-life applications. This is because the desired object could be oriented in any way in the input image, and noise could affect the image.

The next part of the assignment is about average smoothing. Average smoothing is similar to image correlation, however, instead of using a specified mask with weights selected by the user, we will take the average value of the pixels within the mask, and that will be the output. Like with correlation, the mask's weights must add up to one, so for a 3x3 mask, each weight would be 1/9. Like with correlation to compute the output image, apply the mask to every pixel in the input, and we will be able to compute averages to find the output pixel values. When average smoothing is applied to an image, the brightest and largest objects will be extracted because they will have an enormous impact when computing averages. The degree of smoothing will depend on the size of the filter used.

The second part of our assignment also focused on gaussian smoothing. Gaussian smoothing is similar to averaging smoothing, where you apply a mask to every pixel in the input image to compute the output. However, the weights of the mask are determined by the 2D Gaussian function: $G_\sigma(x, y) = \frac{1}{2\pi\sigma} e^{\frac{x^2+y^2}{2\sigma^2}}$ . Therefore, the size of the mask will depend on the standard deviation sigma. This will also determine the strength of smoothing achieved. Because the Gaussian mask gives more weight to pixels in the middle, it provides more gentle smoothing, which will preserve edges better than average smoothing.

The third part of the assignment was about median filtering. Median filtering is where you apply a mask to the input image, and the output pixel is the median value in the mask. This makes the median filtering a nonlinear filter. When it comes to Gaussian noise, median filtering does not have any outstanding performance compared to other methods. Nevertheless, median filtering is beneficial for removing random extreme pixel values known as salty and pepper noise. This is because the "salt and pepper" noise will be pixels close to 0 or 255; therefore, when we take the mask's median value, it is doubtful that the noisy pixels will be selected. It is much more likely that a non-noisy pixel will be selected. Additionally, median filtering will preserve edges more effectively than average or gaussian smoothing. This is because mean filtering only takes pixels in the original image, unlike other methods that may create new pixel values that could be influenced by noise.

The fourth part of the assignment focuses on Unsharp Masking & High Boost Filtering. Till now, we discussed low pass filters. These filters remove high-frequency components present in the image and thus smoothen the image. The high frequencies in the image are represented by edges, sudden and drastic change in the image's pixel values. High pass filters are used to remove the low frequency components present in the image and enhance the edges present in the image. The gradual change in pixel values represents the low-frequency component in the image. The high pass filters are also called Sharpening Filters. These filters are mainly used when we want to highlight fine details in the image. Unsharp Masking and High Boost filtering are examples of high pass filters. The intuition behind Unsharp Masking is quite simple. When we subtract the original image's low-frequency components, we are left out with the high-frequency components. Mathematically it is represented as *High Pass Image = Original Image – Low Pass Image.* The low pass image can be obtained by applying any of the smoothening filters.

The only caveat of Unsharp Masking is, though the image is sharpened and the edges are emphasized, the details are lost. This drawback is overcome by implementing High Boost Filtering. High Boost Filtering's main idea is to amplify the original image and then subtract the low pass component of it. By implementing this, the edges are emphasized, and the details are also not lost. Mathematically it is represented as *High Boost = A . Original Image – Low Pass Image.* Here *A* is the amplification factor, which is multiplied to the input image. This equation can also be written as *High Boost = (A – 1) Original + Original – Low Pass => (A – 1) Original + High Pass.* From this equation, we can see that the Unsharp Masking is a special case of High Boost Filtering when *A* is equal to 1, and when A is greater than 1, High Boost Filtering is performed.

The final part of the project focuses on Gradient and Laplacian filtering. We can also perform sharpening by calculating the image's derivatives, and the derivates are calculated using the gradient function. We can approximate the computation of derivative by implementing filters and performing a correlation on the input image. Two examples of such filters are Prewitt and Sobel filters. These filters consist of both positive and negative weights such that the sum of all the weights in the filter is zero. One of the identification features of the Sharpening filters is, they have negative weights present in the filter. We can compute the gradient along the x and the y-direction. Hence two separate masks are required for computing the gradients along x and y. The gradient image along the x-direction is sensitive to vertical edges, whereas the gradient image along the y-direction is sensitive to horizontal edges. From both of these maps, we can compute the gradient magnitude, which is given by the equation $\sqrt{(\frac{\partial f}{\partial x})^2 + (\frac{\partial f}{\partial y})^2}$. The magnitude of the gradient specifies how significant is the change of intensities. We can also compute the gradient direction, which is given by the equation $\tan^{-1}(\frac{\partial f}{\partial y} / \frac{\partial f}{\partial x})$. The gradient direction specifies the orientation along which the intensities are changing. The Prewitt filters give equal importance to all the pixel values, whereas the Sobel filter gives higher priority to the pixels closer to the center pixel. The Prewitt and Sobel filters are given as follows.

*Prewitt Gradient along x-direction*

| | | |
|---|---|---|
| -1 | -1 | -1 |
| 0 | 0 | 0 |
| 1 | 1 | 1 |

*Prewitt Gradient along y-direction*

| | | |
|---|---|---|
| -1 | 0 | 1 |
| -1 | 0 | 1 |
| -1 | 0 | 1 |

*Sobel Gradient along x-direction*

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

*Sobel Gradient along y-direction*

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

We can also find the sharpened image by computing the second-order derivate of the image, and the second-order derivative is computed by calculating the Laplacian of the image. We can compute the Laplacian of the image by performing the input image's correlation with the Laplacian mask. By performing correlation, we can find the magnitude gradient directly hence decreasing the computational cost. But we cannot find the gradient direction. In this method, the edges can be detected by identifying the zero crossings in the image and thus localizing the edges better. The Laplacian mask is given as follows.

*Laplacian Filter*

| | | |
|---|---|---|
| 0 | -1 | 0 |
| -1 | 4 | -1 |
| 0 | -1 | 0 |

**Implementation:**

**Image correlation** - The steps taken in implementing image correlation are as follows. No specific data structures were used to implement the algorithm. We achieve the functionality using simple if-else blocks.

1. First, we read the image header for the specified mask and store the parameters of the image. Then we allocate memory for an **ImageType mask,** which will hold the mask and read the image into the **mask**. After that, we read the header for the input image, allocate memory for an ImageType object, and read the image into the object. Finally, we use the line **ImageType tempBuffer(N+N1, M+M1, Q)** to allocate memory for a buffer, which will hold the input image when we compute the correlation.
2. Using a nested loop, we will read all pixel values from the input image and put them in the buffer.

3. To compute the correlation, we will use nested for loops to call the correlation function on every pixel in the input image using the line **int newVal = correlation(i, j, N1, M1, mask, tempBuffer).** The correlation function uses nested for loops to visit every pixel within the mask's width and height. We will take the mask's value and the corresponding values in the original image and multiply them together. We are storing the value in the **newVal** variable. The case where the mask is hitting an edge is dealt with when we allocated the tempBuffer as it is as large as the original image plus the mask image. Once every location in the mask is visited, the newVal variable holds the weighted sum of all pixels in the mask applied to the input image and returned to the main function. Once the function returns, we will keep track of the function's largest value to normalize later. This is done by comparing the new returned value with the previous one, and if the new value is more significant, we will store that value instead. Additionally, once the correlation function returns, we will store the value in **correlationArray** at the same index that the correlation function was called at.
4. Next, we normalize the image. Using nested for loops, we iterate through the **correlationArray** and divide every value by the largest value found by the correlation function. This will ensure that the value of the mask adds up to 1. We will also multiply every value by 255 to get normalized values.
5. Next, we use nested for loops to iterate through the **correlationArray.** At every (i,j) coordinate in **correlationArray,** we will take the pixel value and put in the **inputImage ImageType**.
6. Finally we create the .pgm with the line **writeImage(argv[3], inputImage).**

**Average Smoothing** - The steps taken in the average smoothing correlation are as follows. No specific data structures were used to implement the algorithm. We achieve the functionality using simple if-else blocks and loops.

1. First, we read the image header. Then we allocate memory for an **ImageType** using the header specified parameters. After that, we use the copy constructor to create **averagedImage7** and **averagedImage15,** which will hold the output images. Finally, we will read the input image.
2. To compute the averages, we will use nested for loops to iterate through the input image. At each pixel we call the lines **averagedImage7.setPixelVal(i, j, averageFilter7(i, j, image))**; and **averagedImage15.setPixelVal(i, j, averageFilter15(i, j, image));** which will populate the averaged images with a 7x7 and 15x15 pixel mask respectively. The **averageFilter7** function will take the (i,j) location in the original image where the function was called and use nested for loops to iterate through the mask's width and height. This is done by starting the for loops at i - 3 and j - 3 and ending them at i+3 and j+3. This will give a 7x7 grid around the original pixel the function was called on. Inside the for loops we have the line **if(i >= 0 && i < 256 && j >=0 && j < 256)** if that condition is met we will add the pixel value to the running sum variable, this will deal with mask not being on top of the image. After the for loops are done, we will divide the running sum variable by 49 to compute an average and return it. The **averageFilter15** works identically to **averageFilter7** but with its loops starting at i-7, j-7, ending at i+7 j+7 and dividing by 255. Once the for loops are complete **averagedImage7** and **averagedImage15** will hold the image with the smoothing applied.

3. Finally we will create the .pgm files with the lines **writeImage(argv[2], averagedImage7);** and **writeImage(argv[3], averagedImage15);**

**Gauss Smoothing** - The steps taken in implementing gaussian smoothing are as follows. No specific data structures were used to implement the algorithm. We achieve the functionality using simple if-else and for loop blocks.

1. First, we hardcoded the 7x7 and 15x15 gaussian masks as 2d arrays.
2. Then we read the input image header and allocate memory for the input image. We use the copy constructor to create ImageTypes to hold the output images. Finally, we will read the input image.
3. Next, we will compute the gaussian smoothing. Using nested for loops, we will iterate through all pixels in the input image, and at each pixel, we will run the lines **averagedImage7.setPixelVal(i, j, gauss7(i, j, image, Mask7));** and **averagedImage15.setPixelVal(i, j, gauss15(i, j, image, Mask15));** these lines will populate the output images with pixel values determined by the **gauss7** and **gauss15** functions. The **gauss7** function will use nested for loops starting at i-3 and j-3 ending at i+3 and j+3, giving us a 7x7 mask around the pixel the function was called at. We will also have a second set of counters x and y, which will be our index for the mask array. Inside the for loop we have the line **if(i >= 0 && i < 256 && j >=0 && j < 256)** if that condition is met we will get the pixel value from the original image and multiply it by the value of the mask at location (x,y) the if statement will deal with edge cases. we will keep track of these values with a running sum. Once the whole 7x7 mask has been computed, the running sum is divided by the mask's total value to normalize it, and then it is cast as an integer and returned. The **gauss15** function works identically but with its loops starting at i-7 j-7 ending

at i+7 j+7 and using the 15x15 cade coded mask. Once the for loops are computed, **gaussImage7 and gaussImage15** will hold the output images.

4. To convert the smoothed maps to .pgm format images, we make use of the functionality **writeImage(argv[2], averagedImage7);** and **writeImage(argv[3], averagedImage15);**.

**Median Filtering** - The steps taken in implementing the median filter are as follows. The data structure vector, simple if-else blocks, and loops were used to implement the algorithm.

1. First, the user is asked to input a median filter size. If the user enters an even number, we keep asking until an odd number is entered.
2. Next, we read the input image header and allocate memory for an **ImageType output_image** and **ImageType input_image** of the same size as the input image**.** Finally, we read the user-specified input image into **input_image.**
3. Next, the line **corrupt_image** adds salt and pepper noise to the image. The **corrupt_image** first initializes a seed for a random number generator. Next, the function will run a for loop for an amount determined by the number of rows times the number of column times the percent of pixels you want to corrupt. For example, if there are 16 pixels in an image, we want to randomly corrupt 50% of the image; we have to corrupt 8 pixels. In each iteration of the loop, we will randomly select a row and column as well as a pixel value. The pixel value will either be 0 or 255. We will then set the value of the randomly selected pixel to either white or black. After this, the function has returned **input_image,** which holds the corrupted images. During the corruption process, we also execute a while loop, which keeps track of the coordinates generated. If the random coordinates generated are repeated, then we compute the coordinates again until we find unique locations in the image to corrupt.

4. After that, we write the corrupted image to a .pgm file with the line **writeImage.**
5. Then we apply the median filter to the corrupted image with the line **median_filtering**. The **median_filtering** function will iterate through the input image coordinates using nested for loops, and at every pixel value, we will call **compute_median** and store the return value in **median_pixel.** We will then set the pixel value in the output image to be **median_pixel.** The **compute_median** function used nested for loops to iterate through the filter window's width and height in the image. The outer for loop will start at **row = i-(filter_size-1)/2** and end at **i+(filter_size-1)/2.** That means for a filter size of 7x7, It would start at i-3 and end at i+3. The nested for loop does the same functionality as the outer loop but for the columns. Inside the for loops there is the command **if (row < 0 || col < 0 || row >=N || col >= M);**. If this condition is met, that means we are iterating outside the image's bounds, and we do nothing, indicating we are not performing zero padding. If the condition is not met, then we are within the image's bounds, and we can retrieve the pixel value from the input image. The value is then stored in the **pixel_vector**. This is a vector because the size of the filter is unknown at compile time. Once the for loop has iterated through the dimensions of the filter, the **pixel_vector** is sorted using the command **std::sort(pixel_vector.begin(), pixel_vector.end());** and the median value is returned. Once all pixels in the input image have had the median filter applied, **median_filter** returns to main, and the **output_image** now holds the image with a median filter applied.
6. Finally, the image is converted into .pgm format with the command **writeImage.**

**Unsharp Masking & High Boost Filtering –** The steps taken in implementing Unsharp Masking and High Boost Filtering are as follows. No specific data structures were used. We attained this functionality using simple if-else blocks and loops.

1. First, we read the image header of our input image and allocate memory to store our **input_image.** The **ref_image** is the low pass version of our input image, and hence it has the same dimensions as the input image. We allocate the same amount of memory to the reference image using the copy constructor. Once the memory is allocated, the input image and the reference image are read, respectively.
2. Then we prompt the user to enter the amplification factor **A.** If the value of **A** is > 1, then High Boost filtering is performed. If **A** = 1, then Unsharp Masking is performed on the input image.
3. Then we call the **amplification** function. In this function, we iterate through the input image dimensions and multiply each pixel with the amplification factor **A.** Thus, our image is amplified by the value specified by the user. If the value of **A** is 1, then no amplification is performed.
4. Then we call the **subtraction** function. This function accepts the amplified input image, the low pass reference image, iterates through each pixel values, and stores the pixel values' difference. In this function, we keep track of the minimum and maximum difference values.
5. After the subtraction is performed, we call the mapper function. The functionality of the mapper function has been taken from the ImageQuantization.cpp from Project 1. This mapper function maps the pixel values to the range of [0, 255] for visualization.
6. Then we finally convert the image to .pgm format using the command **writeImage.**

**Gradient and Laplacian –** The steps taken in implementing Gradient and Laplacian based are sharpening are as shown below. We have utilized the data structure vector and have implemented using simple if-else blocks and loops.

1. First, we read the image header and allocate memory for the **input_image** based on the image header's information. Using the copy constructor, we allocate memory for **prewitt_x, prewitt_y, prewitt_magnitude, sobel_x, sobel_y, sobel_magnitdue,** and finally **laplacian**. These images will have the same dimensions as that of the input image. Then we read in the input image using the command **readImage(argv[1], input_image);.** We hard code the Prewitt, Sobel, and the Laplacian masks by initializing the variables, **prewitt_x_mask, prewitt_y_mask, sobel_x_mask, sobel_y_mask, and laplacian_mask** with their respective values, as specified in the theory section.
2. Then we call in the **correlation** function. This function computes the gradient maps based on the filter sent as an argument. This function accepts the input image, the destination image where the computed values should be stored, the dimensions of the input image, and the mask with which the correlation should be performed. In this function, we iterate through each pixel value, as the center pixel, and perform correlation. The actual correlation is performed by the **sliding** function, which is called inside the **correlation** function.
3. This **sliding** function accepts the center pixel's coordinates, the input image, the mask, and the dimensions of the input image. If the mask is out of the bounds of the image dimensions, we perform zero padding. If the mask is within the image's dimensions, then we multiply each of the pixel overlapping with the mask and do a simultaneous summation. The final result is divided by the mask's size to keep the range of pixel values between [0, 255].
4. First, we call the **correlation** function by sending the **prewitt_x_mask,** and the computed gradients are stored in **prewitt_x.** Correlation is repeated by sending the **prewitt_y_mask,**

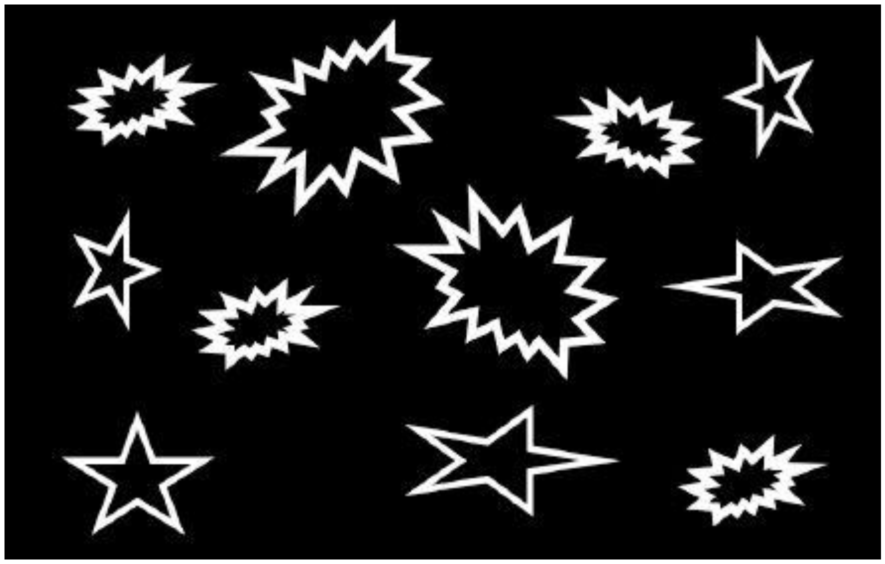
and the gradients are stored in **prewitt_y**. Then using these gradient maps, we compute the magnitude by calling the **magnitude** function. The computed magnitude is stored in **prewitt_magnitude**. This function iterates through each of the locations in both the x and y gradient maps and computes the magnitude using the formula specified in the theory section. Then the magnitude map is converted to .pgm format using the **writeImage** function. To better visualize, we add the magnitude map to the original input image by calling the **adder** function.

5. The **adder** function accepts the magnitude map and the input image. Then each of the corresponding pixels is added together. The adder function also keeps track of the minimum and maximum sum. The added map and the minimum & maximum pixel values are then sent to the **mapper** function so that the range of the added map is between [0, 255]. Then finally, the added map is converted to .pgm format using the **writeImage** function. Steps 4 and 5 are repeated by passing the **sobel_x_mask** to compute the gradients along x-direction and are stored in **sobel_x.** Similarly, the y-direction gradients are computed by passing the **sobel_y_mask** and stored in **sobel_y.** Then the magnitude is computed and added back to the input image for better visualization.
6. Now we will be generating the high pass image using the Laplacian mask. By using the Laplacian mask and performing correlation, the magnitude map is generated automatically. Hence, we call the **correlation** function by sending the **laplacian_mask,** and the magnitude is stored in **laplacian**. We call the adder function for better visualization and add the input image to the Laplacian magnitude map. Finally, both the magnitude and the added map are converted to .pgm format using the **writeImage** functionality.

## Results and Discussion:

**Image correlation-** The procedure we used to image correlation worked as expected. The output was a blurred version of the original image, with a few bright sports dispersed in the image. Those bright sports correspond to a high correlation between the image and the mask applied at that location. This is an example of a simple temple matching. If we want to find objects' locations in an image, we could apply a mask of the desired object to the image and then look for bright spots in the correlated image. This template matching was significant because the input image contains little noise, and the desired objects are not oriented differently.

*Original Image*



*Mask Used*

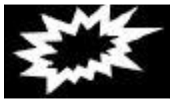
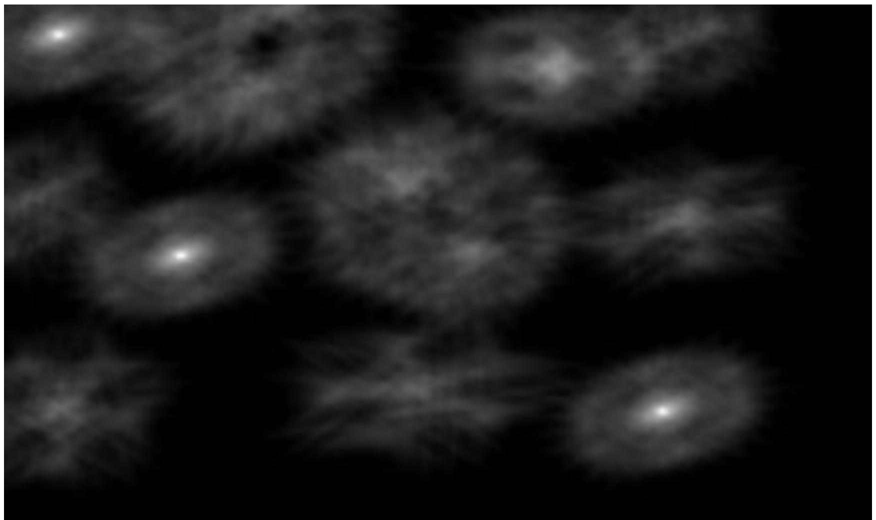


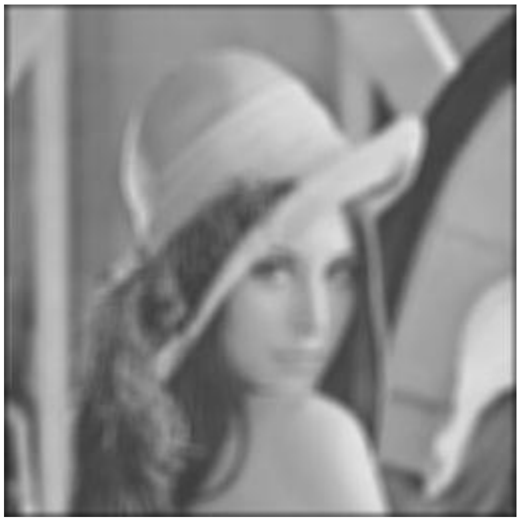
*Image with Filter Applied*



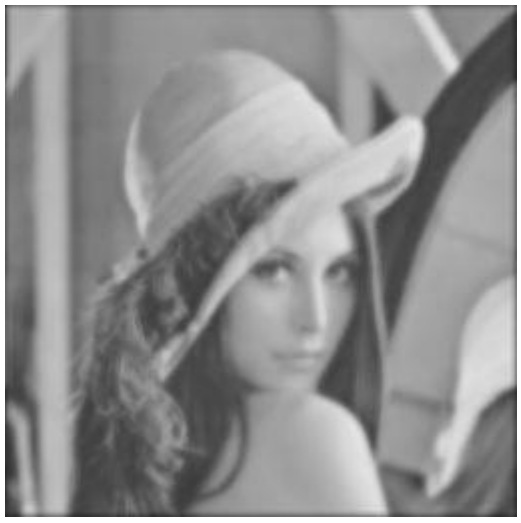
**Averaging and Gaussian Smoothing-** The procedure we used to compute averaging and Gaussian smoothing both worked as expected. It looks like Gaussian smoothing is more effective at preserving edges, which can be seen most clearly when comparing the 15x15 average smoothing vs. the 15x15 gaussian smoothing. The image is more recognizable after the gaussian procedure. Additionally, as the mask's size gets larger, the dark edges around the image become larger. This is because when the mask is applied at the image edge, the image is padded with 0s, which will skew the weighted sums in both procedures towards 0.

*Original Image*

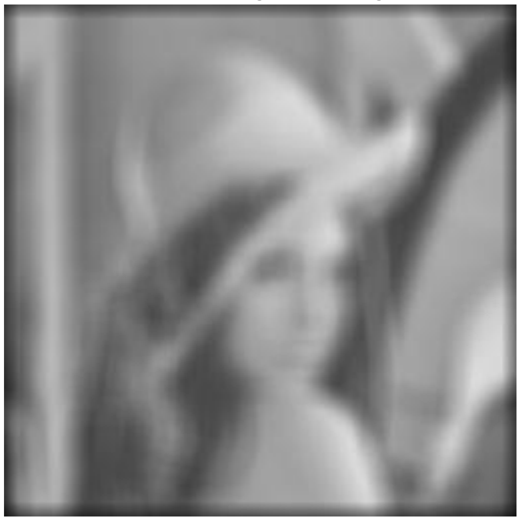
*7x7 Average Smoothing*



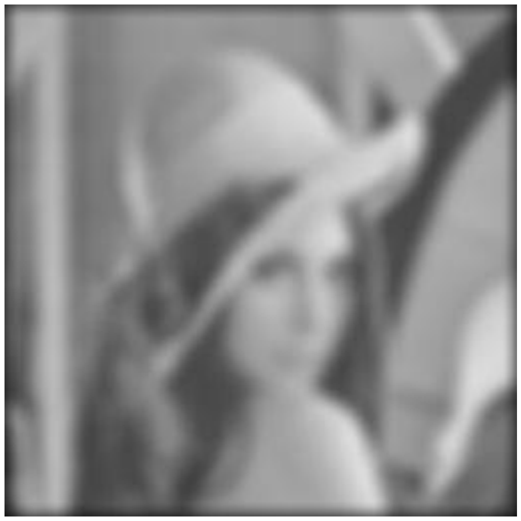
*7x7 Gaussian Smoothing*



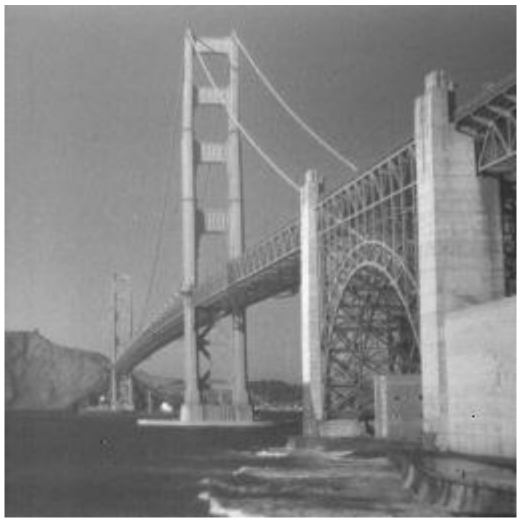
*15x15 Average Smoothing*



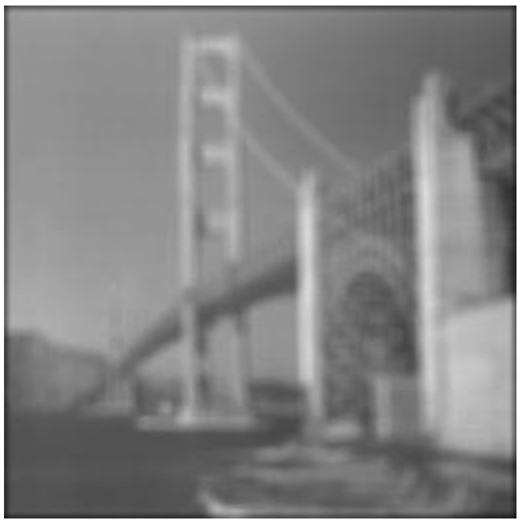
*15x15 Gaussian Smoothing*



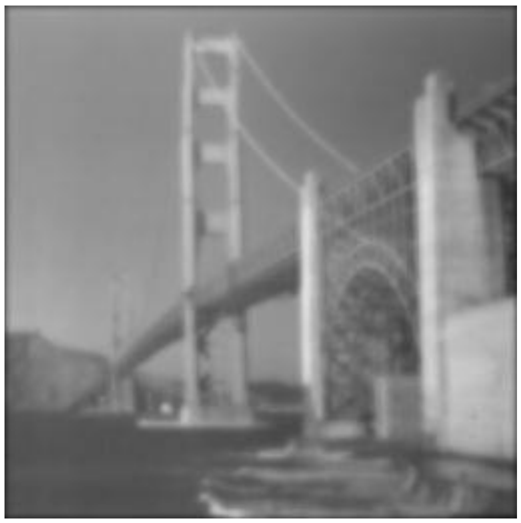
*Original Image*



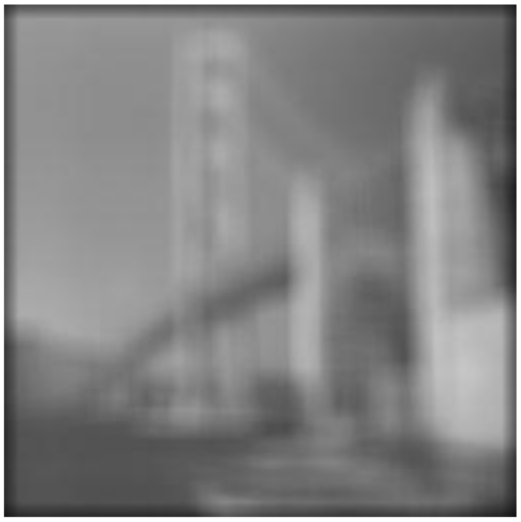
*7x7 Average Smoothing*



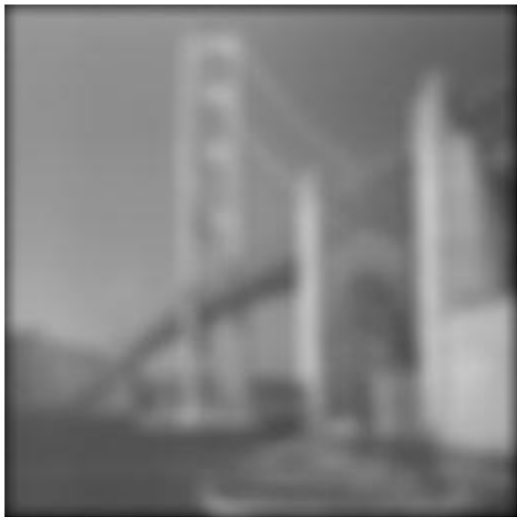
*7x7 Gaussian Smoothing*



*15x15 Average Smoothing*



*15x15 Gaussian Smoothing*



**Median Filtering-** The procedure for medial filtering worked as expected. Median filtering was much more effective at removing the salt and pepper noise than the averaging. After applying averaging filters to the corrupted image there was still the existence of extreme valued pixels. As the mask size and corruption levels increased the image quality took a dramatic dive. However, when the median filtering was applied the images looked blurred, but the salt and pepper noise was removed. Even when increasing the amount of corruption from 30% to 50%, median filtering still performed well. As the size of the filter increases, the amount of blurring in the output image increased as well.

*Image with 30% Corruption*

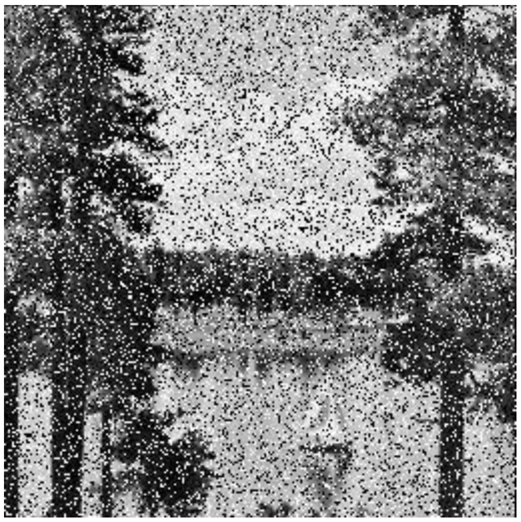


*Image with 7x7 Median Filter Applied*

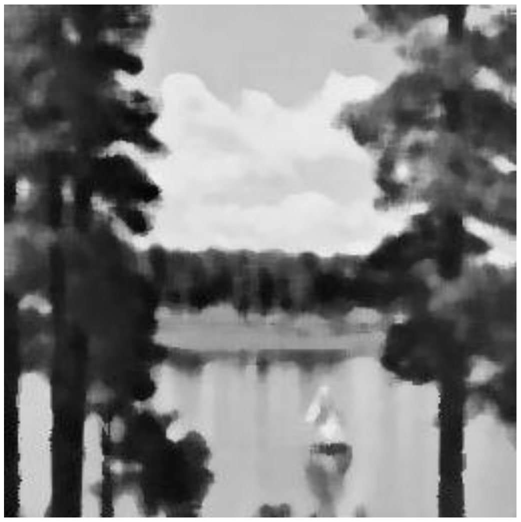


*Image with 7x7 Averaging Filter Applied*

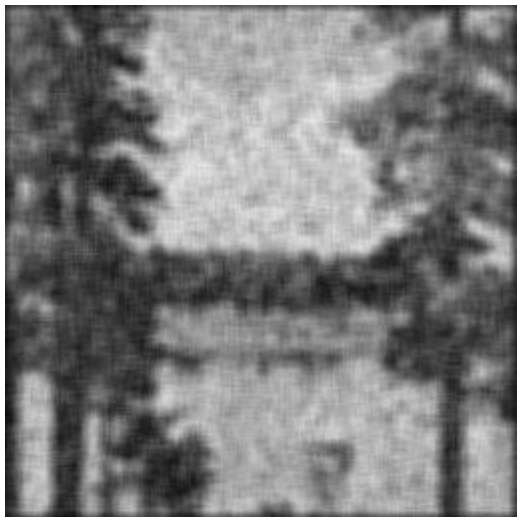


*Image with 15x15 Median Filter Applied*

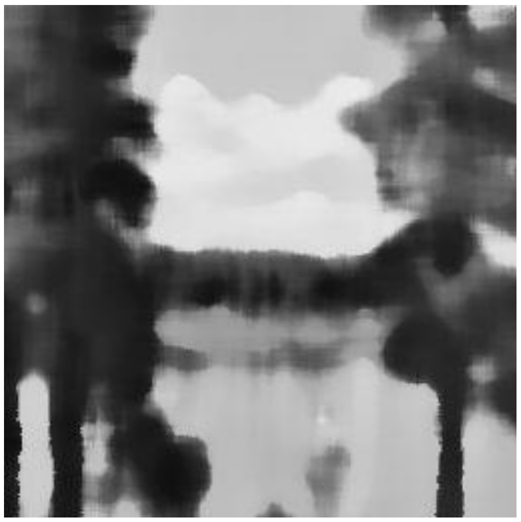


*Image with 15x15 Averaging Filter Applied*

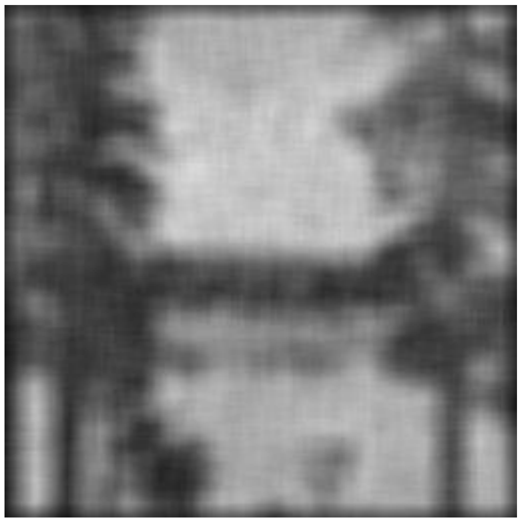


*Image with 50% Corruption*

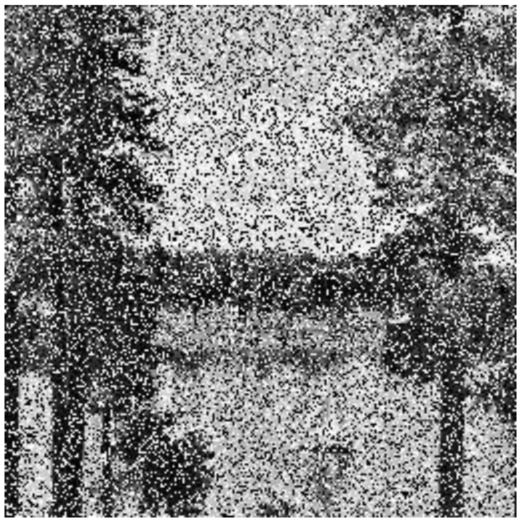


*Image with 7x7 Median Filter Applied*

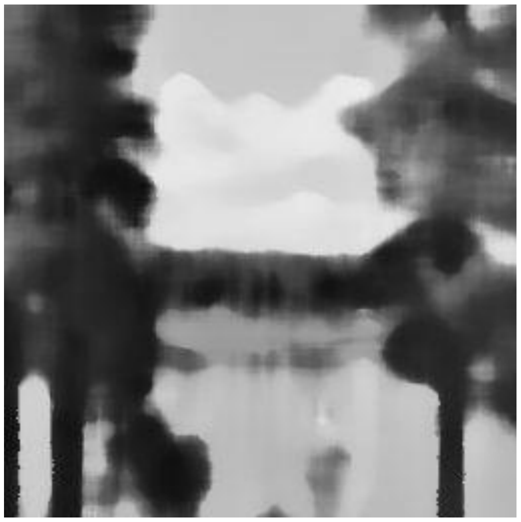


*Image with 7x7 Averaging Filter Applied*

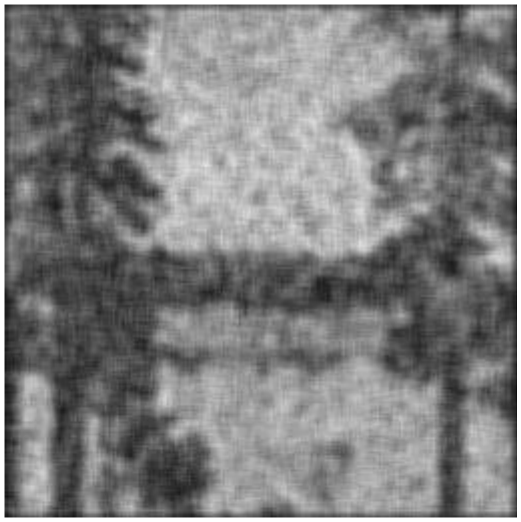


*Image with 15x15 Median Filter Applied*

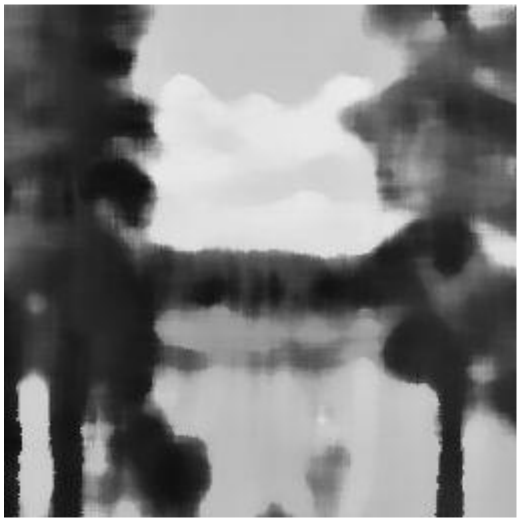


*Image with 15x15 Averaging Filter Applied*

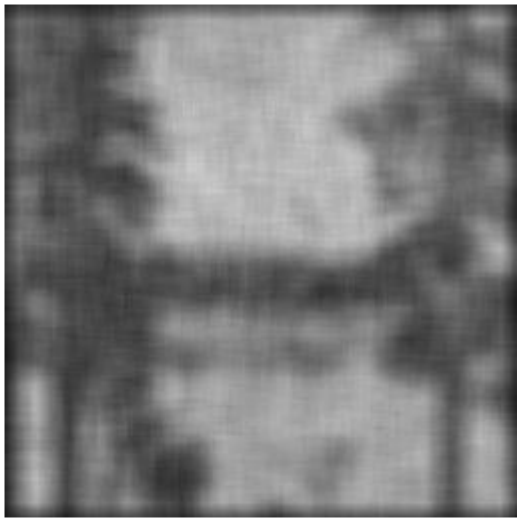


*Image with 30% Corruption*

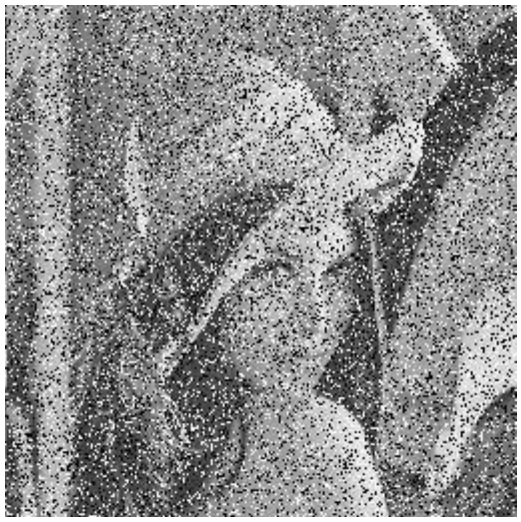


*Image with 7x7 Median Filter Applied*

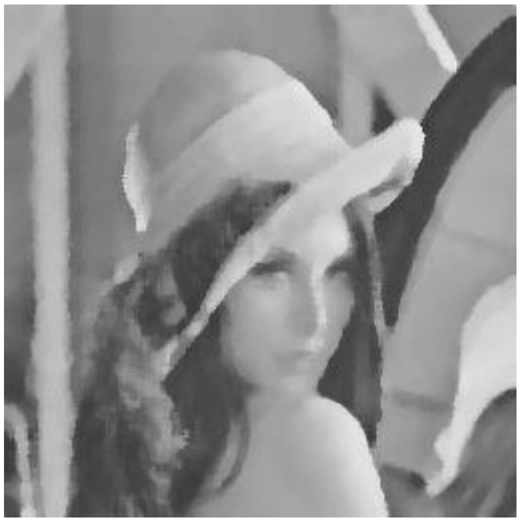


*Image with 7x7 Averaging Filter Applied*

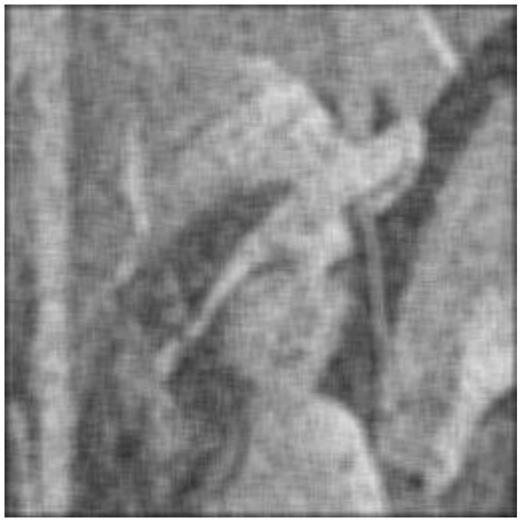


*Image with 15x15 Median Filter Applied*

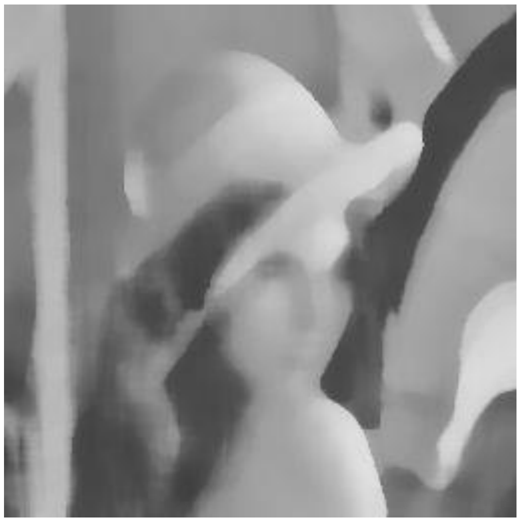


*Image with 15x15 Averaging Filter Applied*

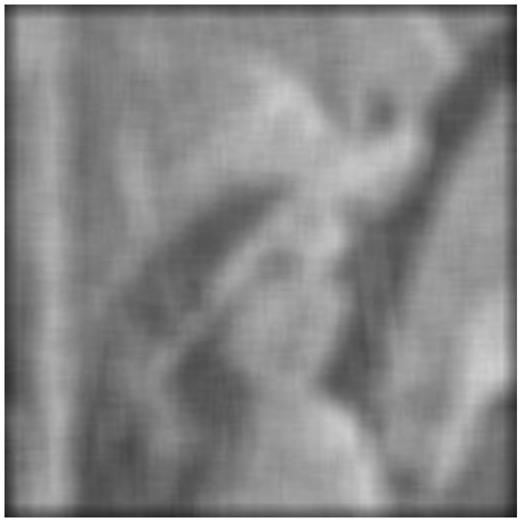


*Image with 50% Corruption*

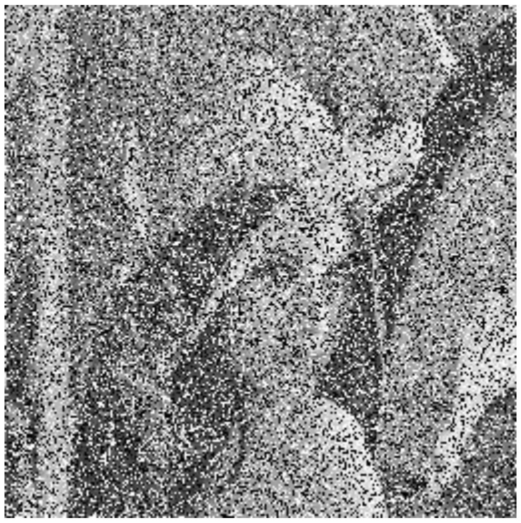


*Image with 7x7 Median Filter Applied*

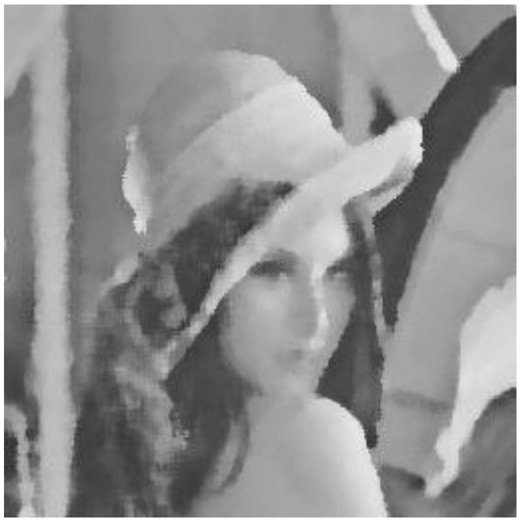


*Image with 7x7 Averaging Filter Applied*

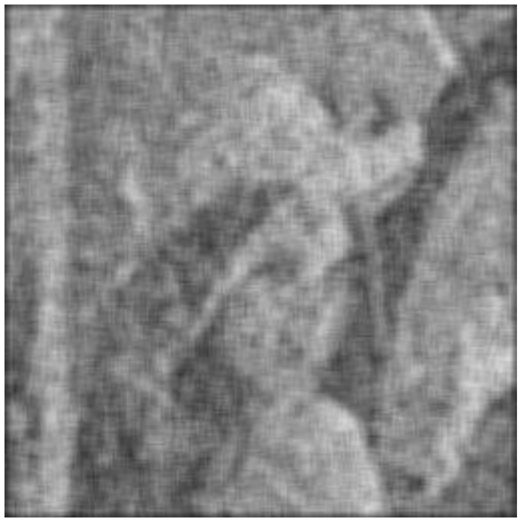


*Image with 15x15 Median Filter Applied*

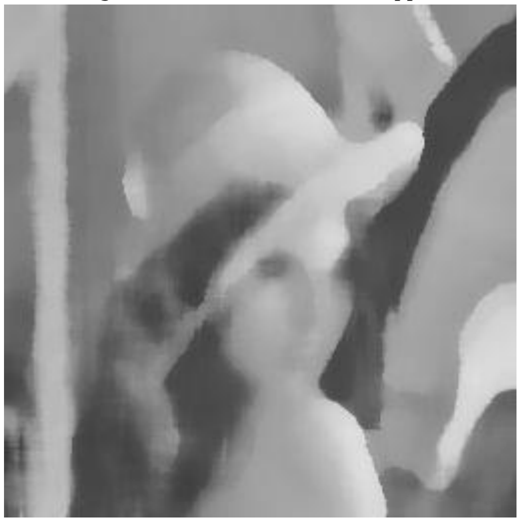
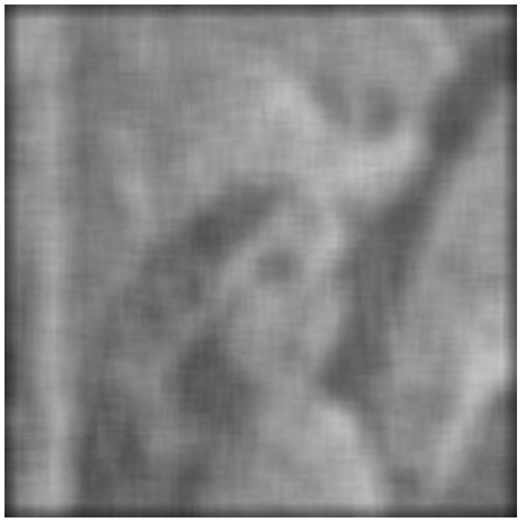


*Image with 15x15 Averaging Filter Applied*

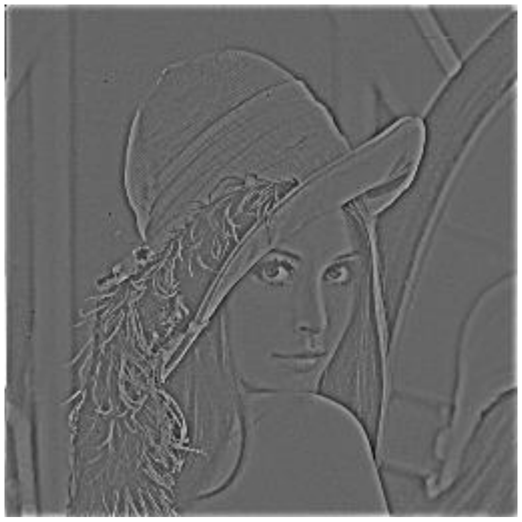


**Unsharp Masking & High Boost Filtering** – The procedure for Unsharp Masking and High Boost filtering worked as expected. In Unsharp Masking, we can see the edges, but the details are lost, as expected. For High Boost Filtering, we utilized the Gaussian Filter of size seven, which smoothened the input image and used the amplification factor A = 1.6. This value of A gave us the best results, where we can see the edges and the fine details of the input image. In the High Boost

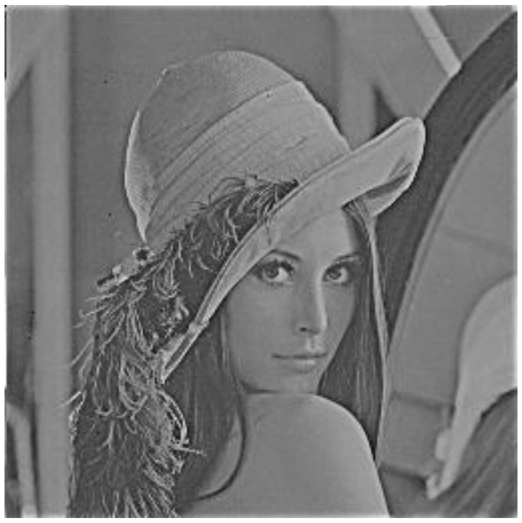
Lenna image, we can clearly see the fine details and the hair's edges. In the F16 image, the result of Unsharp masking is as expected, but the High Boost Image consists of more gray-colored pixels.

*Original Lenna Image*

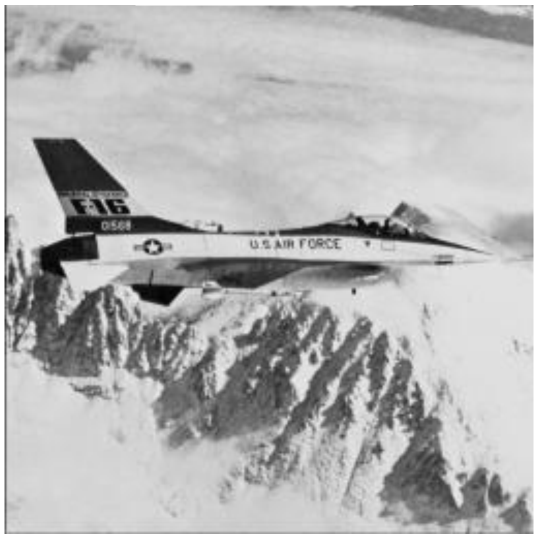
*Unsharp Masked Lenna Image*



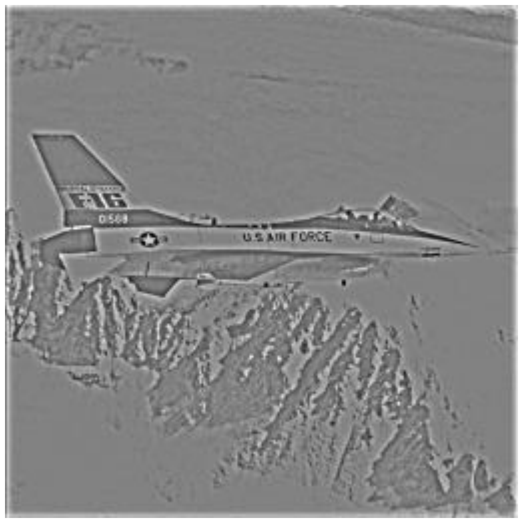
*High Boost Filtered Lenna Image*



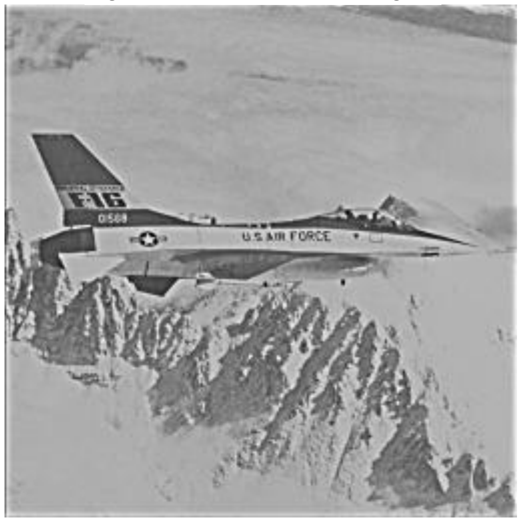
*Original F-16 Image*



*Unsharp Masked F-16 Image*

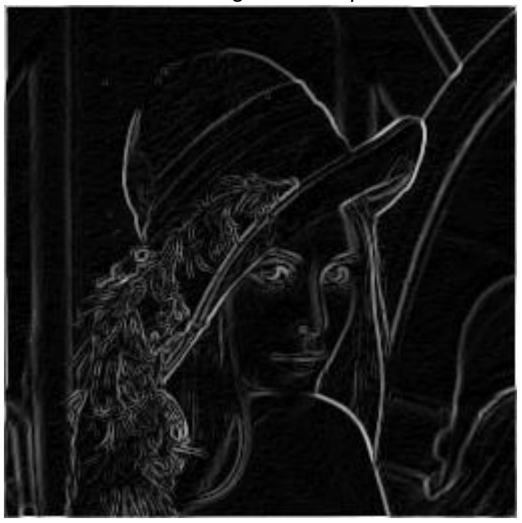


*High Boost Filtered F-16 Image*



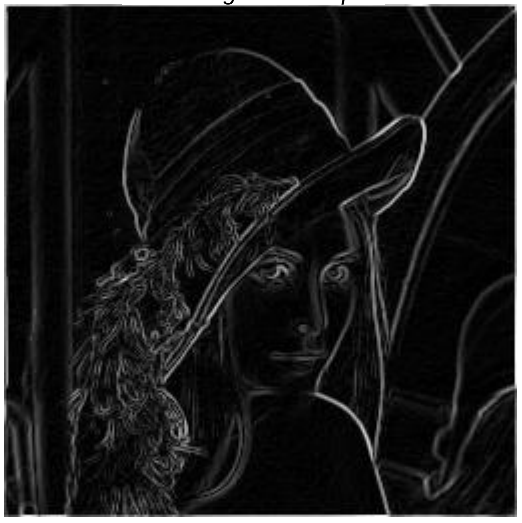
**Gradient & Laplacian –** The procedure for the Gradient and the Laplacian is expected. Compared to Unsharp Masking and High Boost Filtering, the results of Gradient and Laplacian are promising. All the filters have performed exceptionally well in the Lenna image and have detected the edges beautifully. The magnitude map in the case of Laplacian looks gray, but the edges are visible clearly. There seems to be no much difference between the Prewitt and Sobel magnitude maps. In the sf image, the Laplacian mask has captured more refined edges than the Sobel and Prewitt mask. These differences can be seen on the rectangular slabs of the pillars.

*Prewitt Magnitude Map*



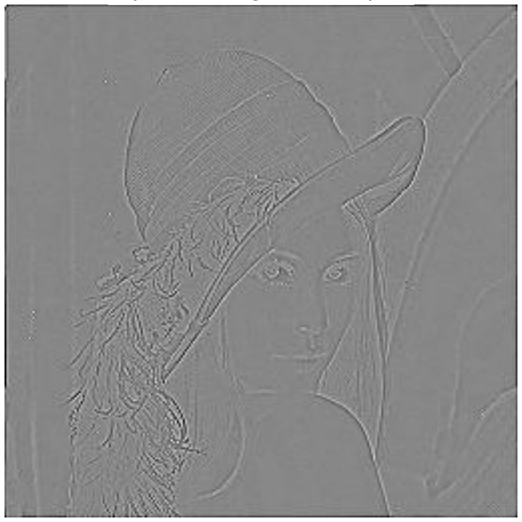
*Prewitt Added Map*

*Sobel Magnitude Map*

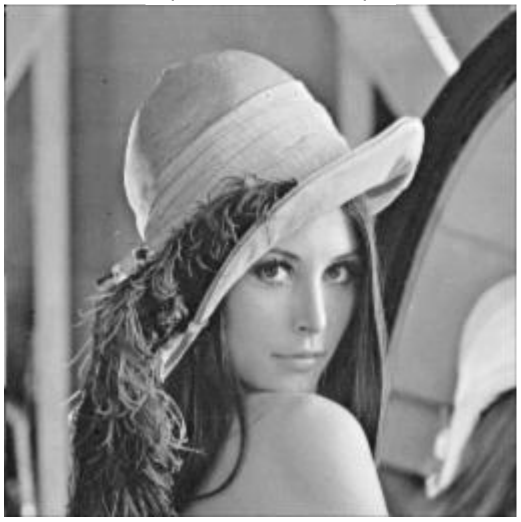


*Sobel Added Map*

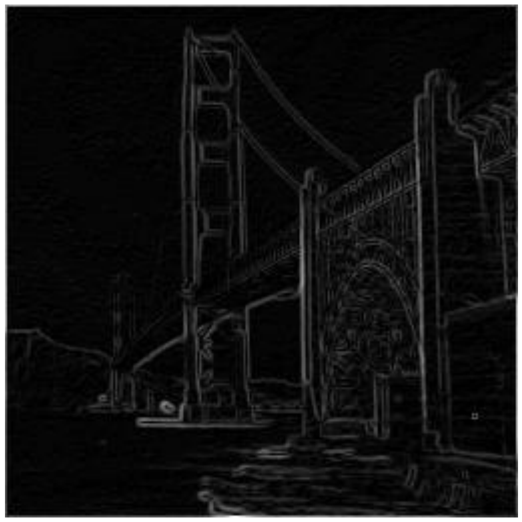
*Laplacian Magnitude Map*



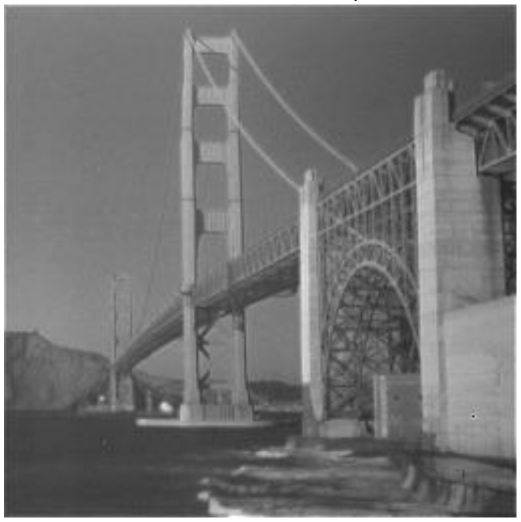
*Laplacian Added Map*



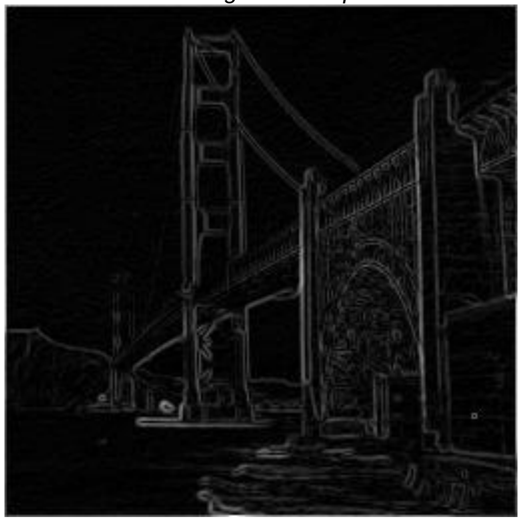
*Prewitt Magnitude Map*



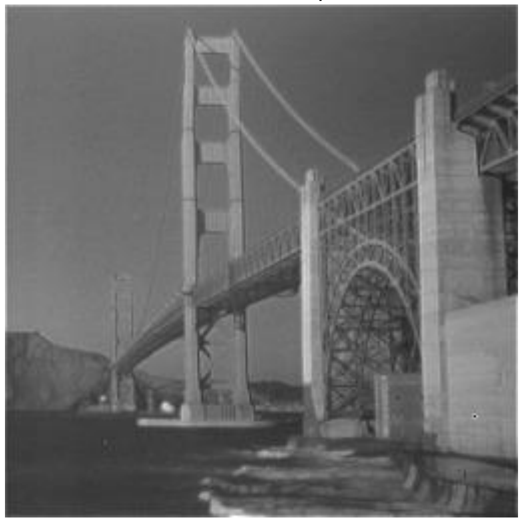
*Prewitt Added Map*



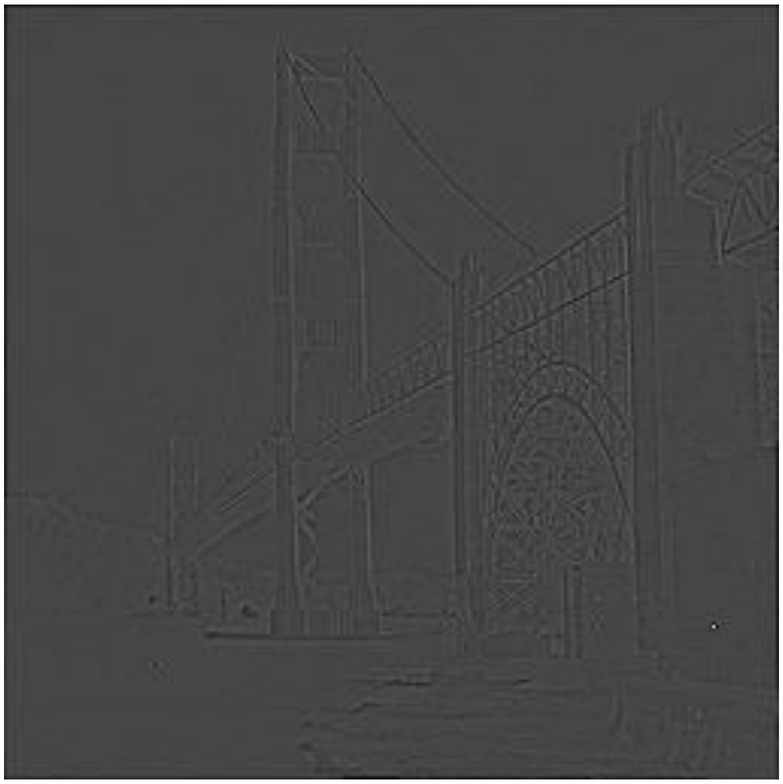
*Sobel Magnitude Map*



*Sobel Added Map*



*Laplacian Magnitude Map*



*Laplacian Added Map*