CS 674
Assignment 2

Bryson Lingenfelter Nate Thom

Submitted: October 18, 2019
Due: October 18, 2019

0.1 Division of Work

0.1.1 Coding

Bryson: Convolution, Gaussian Function, Highboost
Nate: Correlation, Normalization, Median Filtering

0.1.2 Writing

Bryson: Sharpening, Unsharp Mask and High Boost
Nate: Correlation, Smoothing, Median Filtering

1 Technical Discussion

1.1 Correlation

Correlation is the building block for many traditional methods of image processing. The theory behind correlation is based on the idea of looking at a neighborhood of pixels, also known as spatial filtering. Spatial filtering defines a neighborhood (mask size) and an operation (values within the mask). This operation is then applied across all combinations of neighborhoods in our input image. Correlation is a weighted sum of the pixels in the current neighborhood. This weighted sum becomes the output for the pixel at the center of the neighborhood. Around the edges of the image, where there is no input data, we use the default value 0.

One application of correlation is to measure the similarity between images and parts of images. Here it should be noted that the output of correlation should be normalized because values are likely to be above the maximum pixel value of 255. Normalization of any set of integers to the range 0-255 can be achieved by using the following equation:

$$Normalized\_Value = (Original\_Value - Minimum\_Value) * \frac{255}{Maximum\_Value - Minimum\_Value}$$

Our implementation of correlation reads in an image and filter. The filter can also be manually specified. These objects are stored into lists (2 Dimensional Arrays). Next, we iterate across the input image and compute the dot product of the input pixels and the filter values. Each sum is stored in the appropriate location of a new output image. After all dot products have been computed we normalize all values in the output image. Finally the image is stored in memory.

1.2 Smoothing

Smoothing is an operation that blurs an image. In more technical terms, this is called a low-pass filter. Low-pass filters remove high-frequencies from some input signal. Our smoothing operation uses the two dimensional Gaussian function and averaging (mean) to generate filters. The Gaussian function in two dimensions is the distribution for uncorrelated variables X and Y with equal standard deviation. The mean filter consists of all ones, and the output of the sum of products is divided by the number of values in the filter. We create a filter of values with dimensions 7x7 and 15x15. Correlating these filters across an input image reduces noise and eliminates finer details. Two dimensional Gaussian is shown below. Note that small σ equates to more limited smoothing and large σ provides stronger smoothing.

$$G(X,Y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

1.3 Median Filtering

Median filtering is a technique that is very effective at removing "salt and pepper" noise from an image. This noise is a random distribution of black and white pixels throughout an image. This is commonly found in old photographs or videos. Median filtering is a spatial filtering technique in which a neighborhood of pixels is sorted by value and the median value is output. This is done across all neighborhoods in an image.

Since black and white are extreme values it is normally simple to filter out the noise. However, in images that are very dark or very light, this method can be ineffective.

1.4 Sharpening

Sharpening emphasizes details in an image by applying a highpass filter, which removes lower frequencies from an image. The sharpening result can be added back to the original image to obtain a sharped image. To detect high frequencies, derivatives can be used to track the rate of intensity change across the image. We try two different masks for estimating first-order intensity derivatives:

Prewitt: $\frac{\partial f}{\partial y} =$

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

$\frac{\partial f}{\partial x} =$

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

Sobel: $\frac{\partial f}{\partial y} =$

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

$\frac{\partial f}{\partial x} =$

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

Both of these masks estimate the change in intensity with respect to change in horizontal and vertical location. Areas with little change in intensity with have a small derivative because the intensities on either side of the mask will cancel out because of the negatives, whereas areas with large changes in intensity will not cancel out and therefore have large negative or positive derivatives. The Prewitt mask evenly considers horizontal or vertical change over the whole 3x3 filter, whereas Sobel weights the area immediately surrounding the pixel higher by using 2s instead of 1s. To obtain a single sharpened image we compute gradient magnitude as $\sqrt{\frac{\partial f}{\partial y}^2 + \frac{\partial f}{\partial y}^2}$ for each pixel.

In addition to computing first-order derivatives, we compute a sharpened image using second-order derivatives using the Laplacian filter. The second derivative is approximated as $\frac{\partial^2 f}{\partial x^2} = f(i, j+1) - 2f(i, j) + f(i, j-1)$ and $\frac{\partial^2 f}{\partial y^2} = f(i+1, j) - 2f(i, j) + f(i-1, j)$, which are summed to create the following filter:

$\nabla^2 = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} =$

0	1	0
1	-4	1
0	1	0

1.5 Unsharp Mask and High Boost

Unsharp masking is based on the observation that a highpass filter can be achieved by subtracting a lowpass filter from the original image. We use a 7x7 gaussian filter as the lowpass filter, and define unsharp mask as $Highpass = Original - Lowpass$. That is, we blur the image using a gaussian filter then subtract the result from the original image. This results in a very dark image, so a "high boost" filter can be achieved by amplifying the original image by some multiplier A before subtracting the lowpass filter. This results in the following: $Highboost = A \times Original - Lowpass$. We determine the best value for A by looking at the results for different values of A and qualitatively determining the best result.

2 Implementation Details

Most of the project was based on a function titled `convolution`, which takes an image, a filter, and a padding value as input and returns a new image generated by the filter. The function is implemented using a 4x nested for loop which works as follows:

for i from 0 to $image\_width$ **do**
  for j from 0 to $image\_height$ **do**
    $weighted\_sum \leftarrow 0$
    for n from 0 to $filter\_width$ **do**

```
            for k from 0 to filter_height do
                h_offset ← ⌊(n − filter_width)/2⌋
                v_offset ← ⌊(k − filter_height)/2⌋
                weighted_sum ← weighted_sum + image[i + h_offset, j + v_offset] × filter[n, k]
            end for
        end for
        new_image[i, j] ← weighted_sum
    end for
end for
```

If the vertical or horizontal offset goes off the image, then the padding value is added to the weighted sum rather than a pixel value multiplied by a filter value.

Most of the spatial filtering is done simply by defining filters to pass to this function, with some other helper functions defined for various tasks. There is a `normalize` function which normalizes the values of a filter to sum up to 1 and a `highboost` function which takes in an original image, a lowpass image, and a value for A then performs multiplication and subtraction to compute the highboost. These functions are implemented using a 2x nested for loop which computes new values over every pixel in the image.

3 Results and Discussion

3.1 Correlation

The correlation algorithm was tested with the provided input image and pattern (filter) image. We found that the output image had to be normalized before saving it. When taking the dot product it is very likely that the sum of products is greater than the max pixel value of 255. When this occurs the image defaults to 255.

We experimented with two techniques of normalization. The first was normalizing the input filter's values between 0 and 1. The second was normalizing the output image to be between 0 and 255. Although both methods are effective, normalizing the filter first provides a much darker and less clear result. This makes sense because the latter technique is more likely to achieve an even distribution of values.

There are many activated regions where the filter is similar to the image. It is interesting to note that thresholding might show us where the filter is most similar to the input image. I believe that this would yield a more representative output.

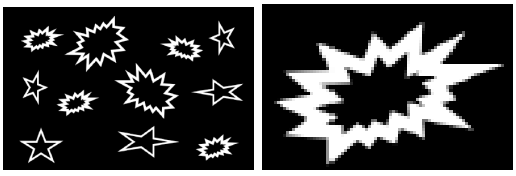

Figure 1: The input image and the input filter used to test the correlation function.

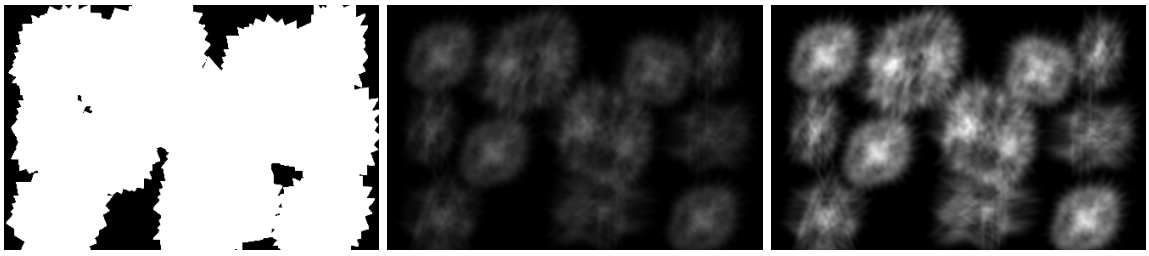

Figure 2: Outputs of the filter correlated with the input image. The left image is not normalized, the middle image is a normalization of the input filter, and the right is a normalization of the image after filtering.

Run time of the correlation operation is not noticeable when using filters with dimensions less than or equal to 9. Computation of correlation with very large input images or filters leads to significant delay. Explicitly, when running the operation on the provided image and filter the algorithm takes about 2 minutes. Most of this computation time appears to be happening in the correlation portion of the program and not the normalization.

3.2 Smoothing

Smoothing was sampled on the SF and Lenna images. As mentioned in the technical discussion section, we use both Gaussian filtering and simple average filtering. Although Gaussian filtering is a more complex operation, it appears to be a more effective low-pass filter. Mean filtering seems to be filtering most, but not all, high frequencies.

As the size of both filters increases, so does the strength of the blurring effect. This is expected. It can be seen however that the Gaussian filter results in smoother outputs when using a larger filter than average filtering does. Already with a filter size of 15 it can be seen that groups of similar pixel values are starting to merge. This makes sense because with a large enough averaging filter the image would converge to a single color. Thus, Gaussian filtering appears to also be superior when stronger blurring effect is desired.

Figure 3: Results on Lenna and SF images. Top to Bottom: Original images, smoothing Lenna, and smoothing SF. Left to Right: 7x7 Gaussian, 7x7 Averaging, 15x15 Gaussian, 15x15 Averaging.

3.3 Median Filtering

We test our implementation of median filtering on the lenna and boat images. The primary use of median filtering is the removal of salt and pepper noise. To this end, it appears to be very effective. One negative

attribute of the output images is the border that appears around the outside of the output. This border is expected because of the zero padding, but in some of the samples it extends beyond the the immediate edge of the image. This occurs when more zero values exist in an area than other values, thus a zero is the median value. It is different in each image because the distribution of noise is random on each run of our program.

Another observation is that the median filtering causes a toon-shading effect on the image. This effect is intensified as the size of the filter increases. Similar to a note we made about average filtering, a large enough median filter will cause the entire image to converge to the same value. Before convergence, objects in similar scenes as the boat may disappear entirely. This is especially likely for objects that appear in low contrast backgrounds. Objects that appear on higher contrast backgrounds, such as lenna's profile, will reduce to distinct shapes.

We compare the effectiveness of median filtering on this task with average filtering. It is clear that median filtering is the superior technique. The smaller averaging filter maintains more clarity in the image because more low-frequencies are able to come through the filter.

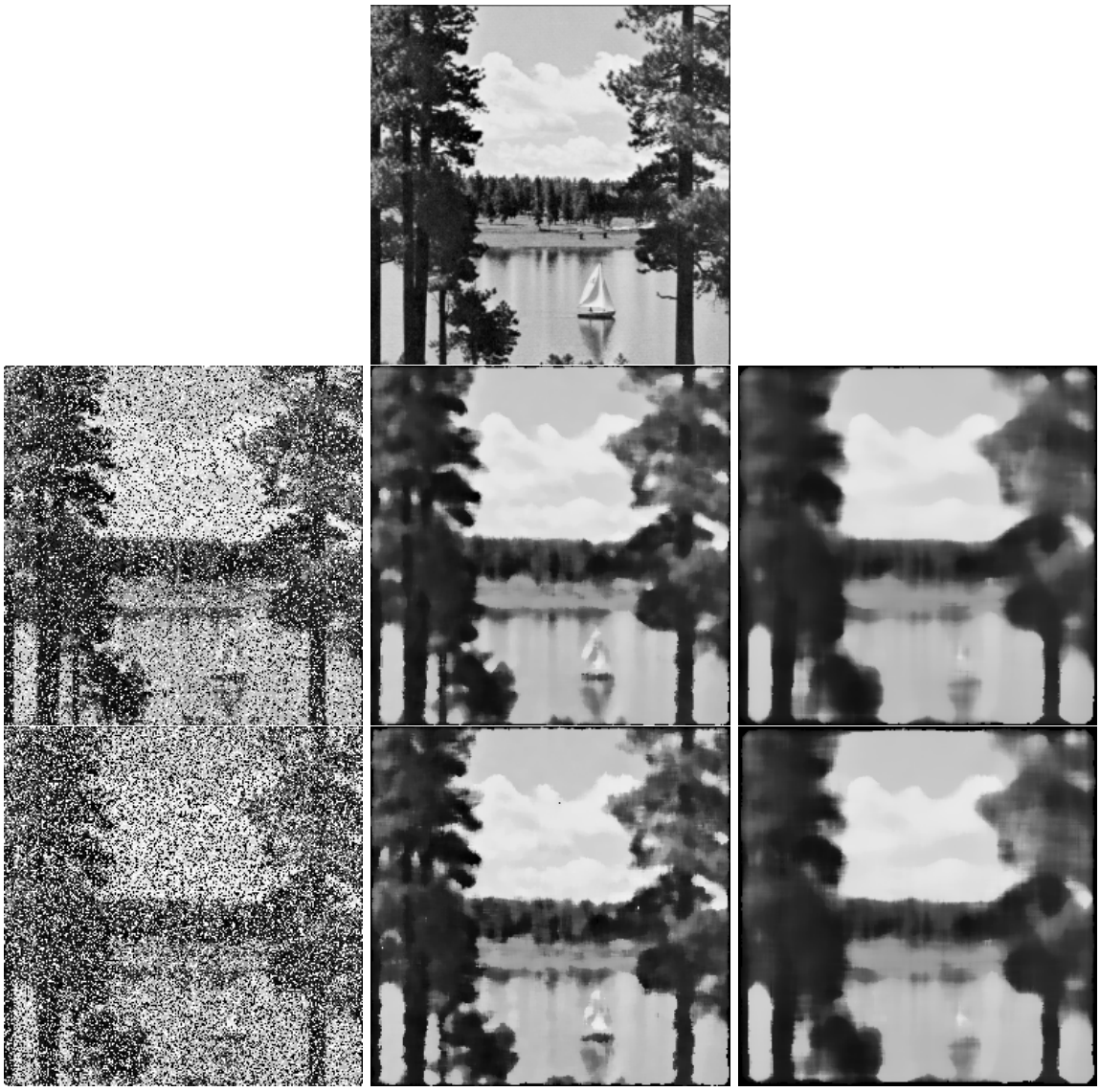

Figure 4: Filtering salt and pepper corruption with median filtering.Top to Bottom: Original image, 30 percent noise, and 50 percent noise. Left to Right: Unfiltered image, 7x7 median filter, and 15x15 median filter.

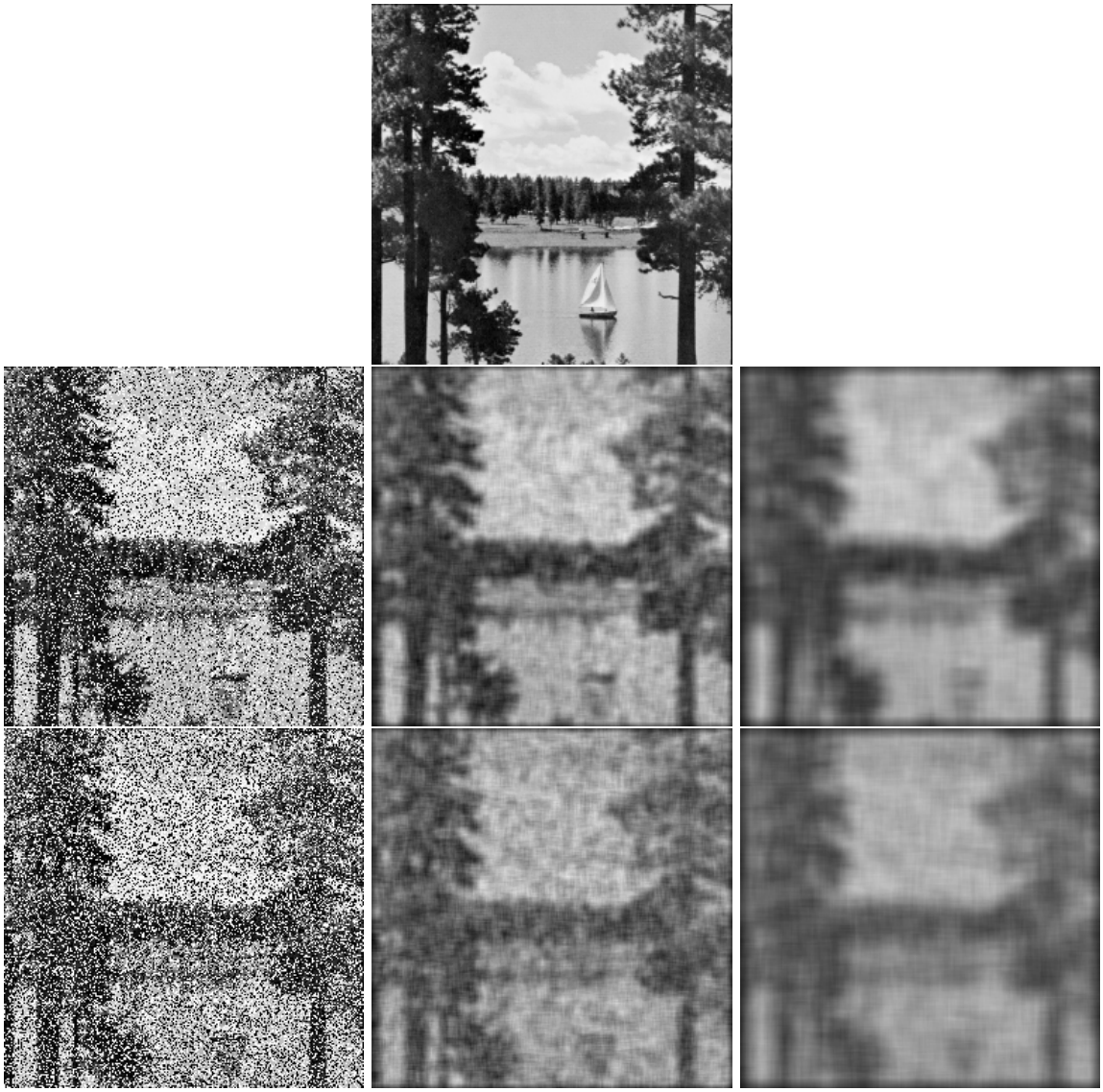

Figure 5: Filtering salt and pepper corruption with average filtering.Top to Bottom: Original image, 30 percent noise, and 50 percent noise. Left to Right: Unfiltered image, 7x7 averaging filter, and 15x15 averaging filter.

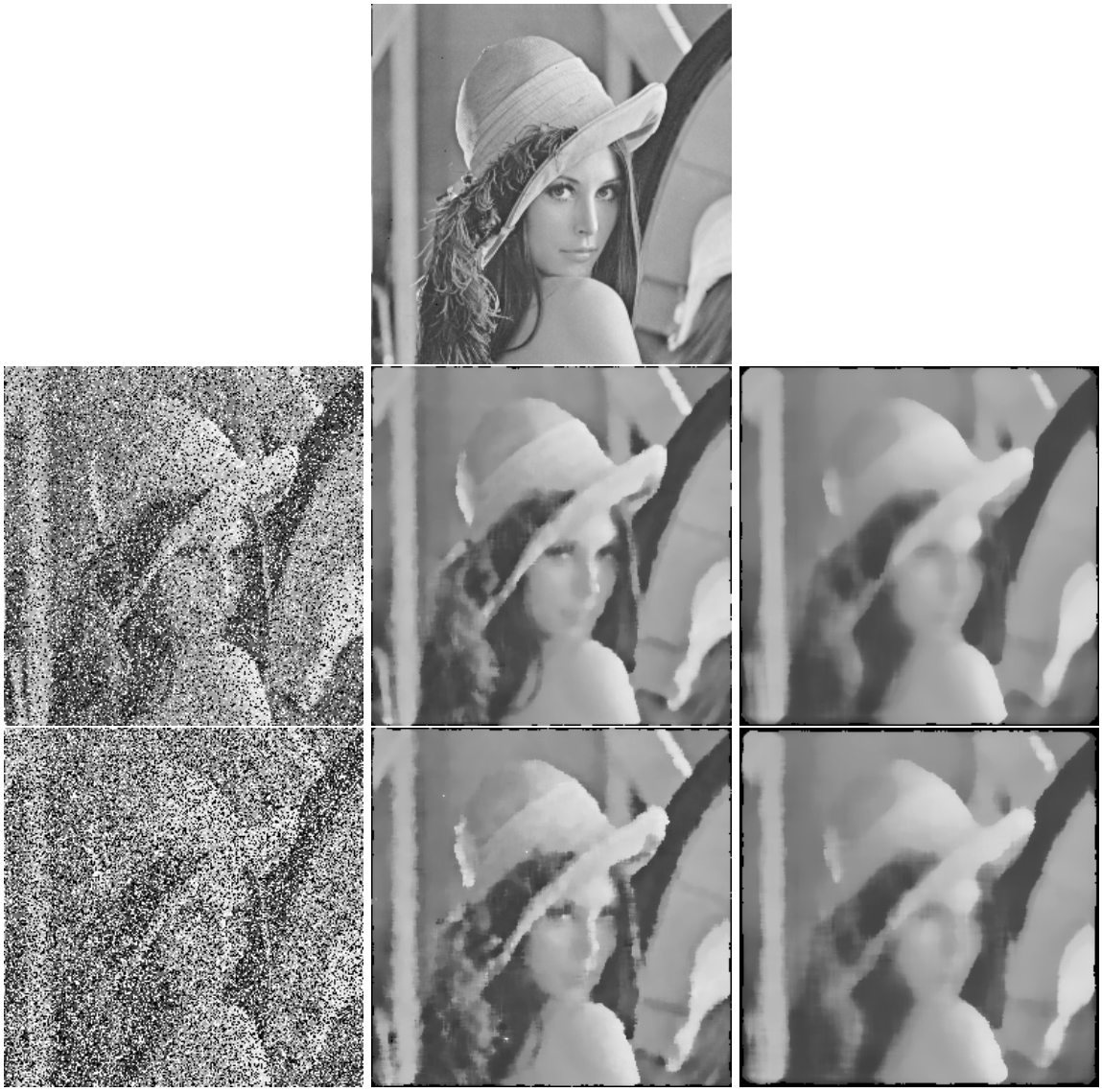

Figure 6: Filtering salt and pepper corruption with median filtering.Top to Bottom: Original image, 30 percent noise, and 50 percent noise. Left to Right: Unfiltered image, 7x7 median filter, and 15x15 median filter.

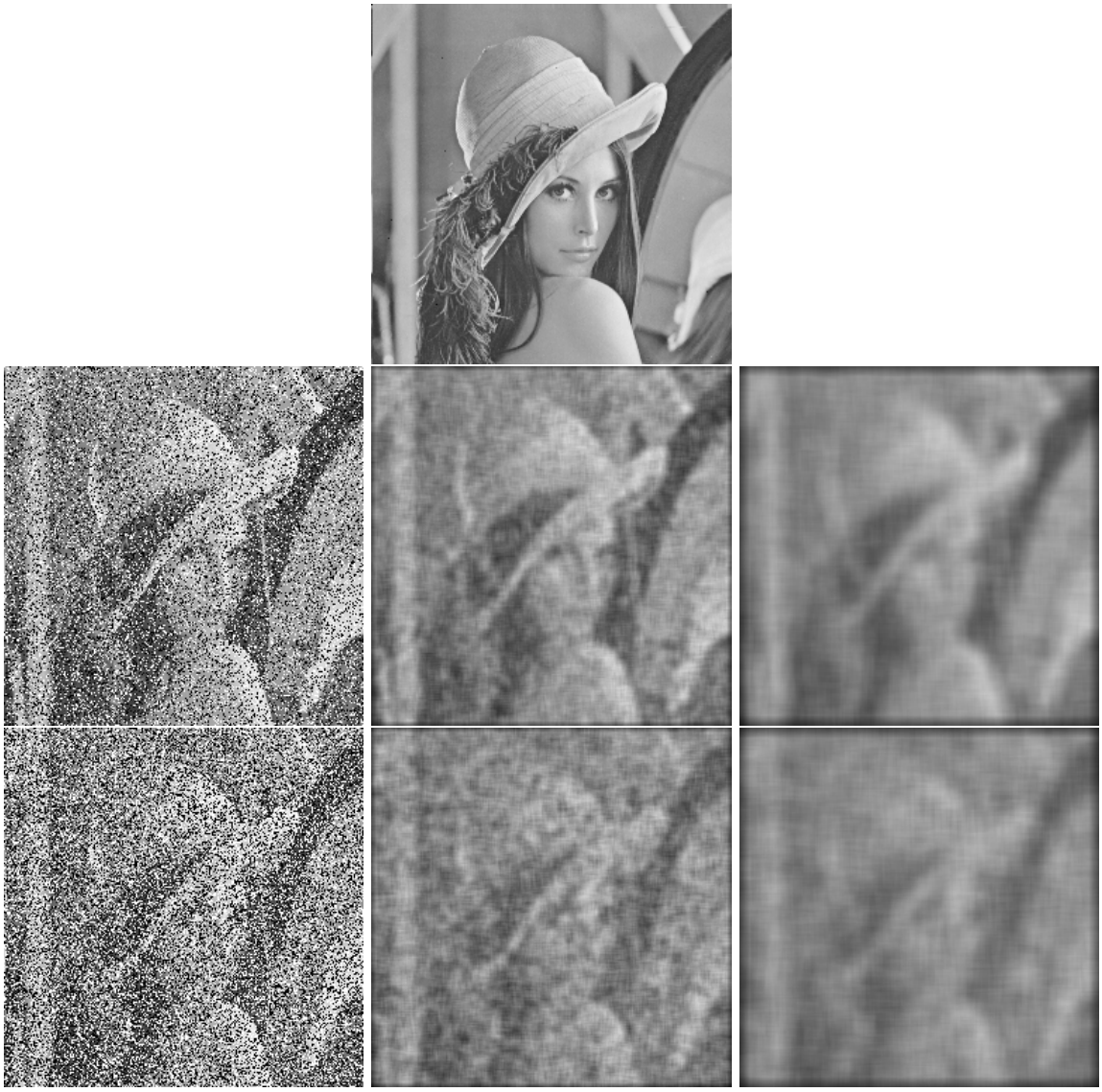

Figure 7: Filtering salt and pepper corruption with average filtering. Top to Bottom: Original image, 30 percent noise, and 50 percent noise. Left to Right: Unfiltered image, 7x7 averaging filter, and 15x15 averaging filter.

3.4 Sharpening

We evaluated the sharpening filters on the Lenna and SF images. We include the magnitude of the horizontal derivatives and vertical derivatives, along with the overall gradient magnitude. We first tried the Prewitt mask, which evenly considers change across a 3x3 area. To best represent both highly negative and highly positive changes in intensity, we use the absolute value of the convolution as the output intensity value for the vertical and horizontal derivative images. The magnitude is the square root of both components squared, so negative values don't need to be removed.

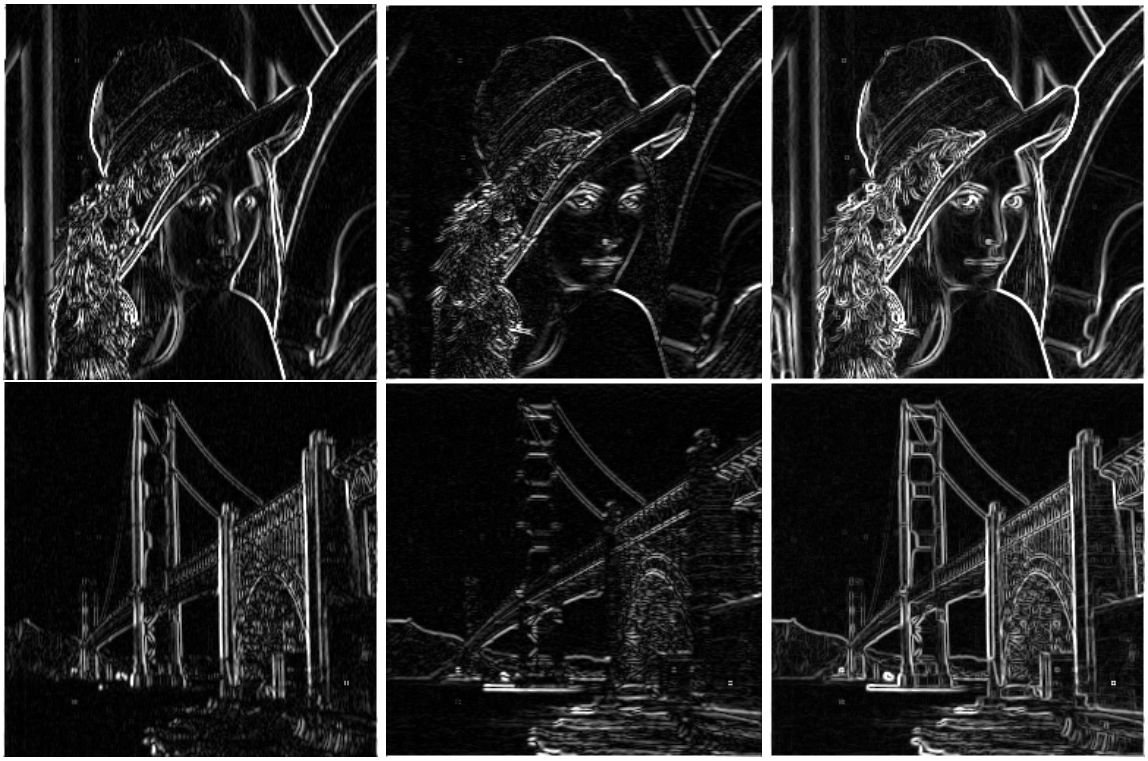

Figure 8: Results using Prewitt mask on Lenna and SF images. Left to right: horizontal derivative, vertical derivative, gradient magnitude.

As seen in Figure 8, the horizontal derivative captures most of the vertical edges in the image. This makes sense because vertical elements will result in a large horizontal change in intensity as the filter is passing across the image. Likewise, the horizontal edges are captured by the vertical derivative. When the two derivatives are combined to compute the gradient magnitude, both vertical and horizontal edges are captured. This produces an image which looks similar to an outline of the original image, with sharper transitions appearing more bright.

We next tested the Sobel mask, which is very similar to the Prewitt mask but pays more attention to change in the pixels immediately next to the current location of the filter. The sharpening results for this mask appear to capture a bit more noise in the image, with the background appearing bumpy and the detail under the bridge more clearly standing out. This result makes sense, becaues the Prewitt mask is evenly averaging over 3 pixels whereas the Sobel mask is more focused on horizontal or vertical change over a single pixel.

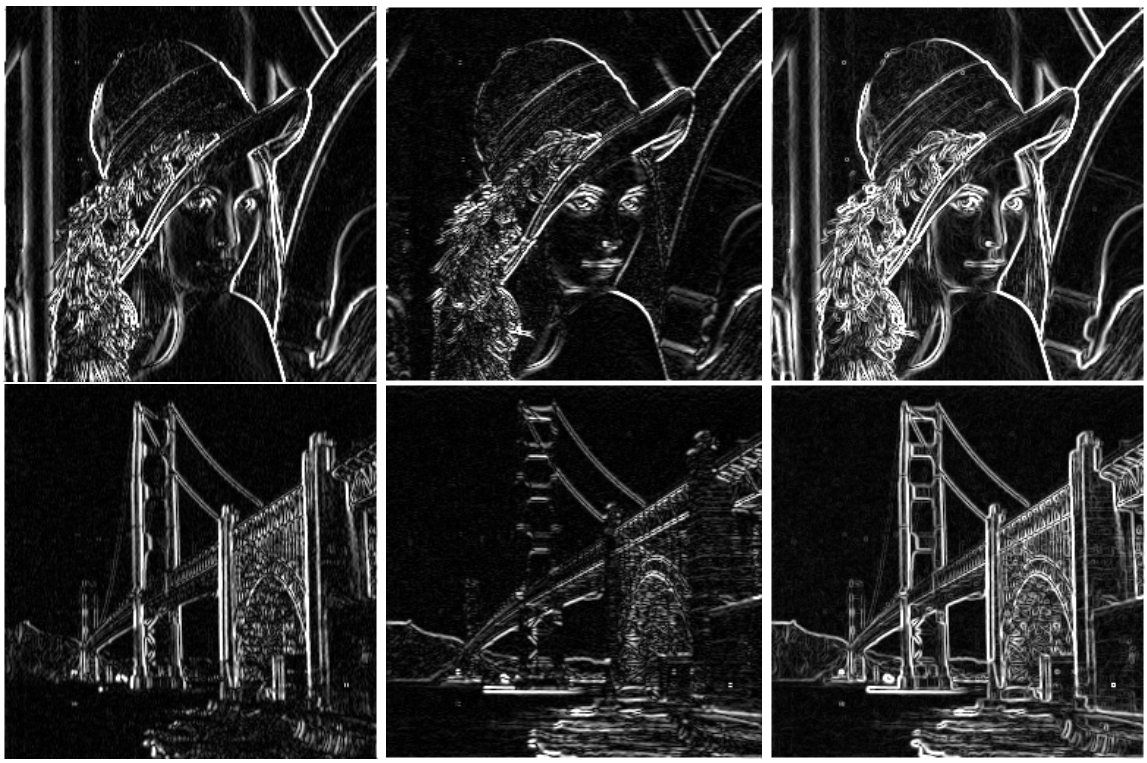

Figure 9: Results using Sobel mask on Lenna and SF images. Left to right: horizontal derivative, vertical derivative, gradient magnitude.

Finally, we tested the Laplacian filter. This filter captures the second-degree spatial derivative of the intensity rather than the first degree derivative. It appears to be much better at detecting edges than the first-order derivative masks, with the images looking like sketched versions of the original image. In the SF image, the frame under the bridge is far more detailed unlike in the first-order derivative filters where it looks like a solid material. However, the Laplacian appears to be more sensitive to noise. In the lenna image, there are two clear portions in the hat which are well distinguished by a bright line in the Sobel and Prewitt images. In the Laplacian image, however, every minor change in intensity in the hat is marked by a line. Additionally, a lot of uninteresting texture details are captured by the Laplacian filter. This makes sense because the Laplacian filter detects edges from zero-crossings, so it has thinner, more detailed edges whereas the other filters have thicker edges but ones that are brighter for larger changes in intensity.

Figure 10: Laplacian filtering applied to the Lenna and SF images.

3.5 Unsharp Mask and High Boost

We evaluated unsharp masking and high boost on the Lenna and F_16 images. For this we used a 7x7 gaussian filter with a σ of 1.4. Figure 11 shows the results of the unsharp mask as well as highboost with A values of 1.5 and 2. If not enough of the original image is added back, the image appears dark. We found that A values between 1.8 and 2 seem to match the average intensities of the original image best.

Figure 11: Results on Lenna and SF images. Left to right: Unsharp mask, highboost with A=1.5, and highboost with A=2.0. The original images are provided above for comparison

The unsharp mask, like other sharpening filters, traces and outline of the edges in the image. When added back to the original image in highboosting, the edges become more clear. In the Lenna image, the hat and shoulders more clearly stand out from the background. However, the feathers in the hat also become brighter because they vary in intensity, which may not be desired. The F16 also more clearly stands out from the background and the text on the plane is more readable, but the mountain in the background also stands out more.

To experiment with the effects of changing the gaussian filter, we tried highboost with $A = 1.9$ with varying sizes of gaussian filters with varying standard deviations.

Figure 12: Left to right: standard deviation varying from 0.6 to 1.4 to 2.2. Top to bottom: filter size varying from 3 to 7 to 11.

Figure 12 shows how increasing the size of the gaussian filter increases the sharpening amount. The images along the diagonal show increasing filter sizes with the standard deviation equal to the filter size divided by 5. As the gaussian becomes wider, more lower frequencies are removed so the image becomes sharper when this gaussian is subtracted from the image. If the standard deviation is not increased to account for the size of the filter, the sharpening does not improve. For example, with a standard deviation of 0.6 all filter sizes look the same because a 3x3 filter is enough to account for greater than 99% of the gaussian function so the larger filters have no effect. This can be seen along the left column. Likewise, a 3x3 filter only accounts for 1.36 standard deviations of a filter with a standard deviation of 2.2, so the sharpening results are also poor.

4 Program Listing

4.1 Convolution

```
def convolve(image, convolution, padding=0):
    pixels = image.load()
    width, height = image.size
    conv_size = len(convolution)

    new_img = Image.new(image.mode, image.size)
    pixels_new = new_img.load()

    for i in range(width):
        for j in range(height):
            weighted_sum = 0
            for n in range(conv_size):
                for k in range(conv_size):
                    v_offset = n - conv_size // 2
                    h_offset = k - conv_size // 2
                    if i + h_offset < 0 or i + h_offset >= width\
                                        or j + v_offset < 0\
                                        or j + v_offset >= height:
                        weighted_sum += padding
                    else:
                        weighted_sum += pixels[i+h_offset,j+v_offset]*convolution[k][n]
            pixels_new[i,j] = int(weighted_sum)

    return new_img
```

4.2 Gaussian

```
def gaussian_filter(size, var):
    return [[gauss(i - size//2, j - size//2,var) for i in range(size)]
                                                  for j in range(size)]

def gauss(x, y, var):
    return 1/(2*pi*var)*e**(-(x**2+y**2)/(2*var))
```

4.3 Normalizing

```
def normalize_1d(input_list):
    output_list = [float(input_list[i])/sum(input_list) for i in range(len(input_list))]

    return output_list

def normalize_2d(input_list):
    list_sum = 0
    for i in input_list:
        list_sum += sum(i)

    output_list = [[float(input_list[i][j])/list_sum
                    for j in range(len(input_list[i]))]
                   for i in range(len(input_list))]

    return output_list
```

```
def normalize_0_255(input_list):
    min = 999999
    max = 0
    for width in range(len(input_list)):
        for height in range(len(input_list[width])):
            if min > input_list[width][height]:
                min = input_list[width][height]
            if max < input_list[width][height]:
                max = input_list[width][height]

    for width in range(len(input_list)):
        for height in range(len(input_list[width])):
            input_list[width][height] = int((input_list[width][height] - min)
                                            * (255 / (max - min)))

    return input_list
```

4.4 Sharpening

```
def magnitude(h_prime, v_prime):
    h_pixels = h_prime.load()
    width, height = h_prime.size

    v_pixels = v_prime.load()
    width, height = v_prime.size

    new_img = Image.new(h_prime.mode, h_prime.size)
    pixels_new = new_img.load()

    for i in range(width):
        for j in range(height):
            pixels_new[i,j] = int((h_pixels[i,j]**2 + v_pixels[i,j]**2)**(1/2))

    return new_img

def highboost(image, lowpass, a):
    im_pixels = image.load()
    width, height = image.size

    lp_pixels = lowpass.load()
    width, height = image.size

    new_img = Image.new(image.mode, image.size)
    pixels_new = new_img.load()

    for i in range(width):
        for j in range(height):
            pixels_new[i,j] = int(a*im_pixels[i,j] - lp_pixels[i,j])

    return new_img
```

4.5 Correlation, Average Filtering, and Median Filtering

```
def correlate(image, correlation, padding=0, median_flag=0):
    image_list = image_to_list(image)

    width, height = image.size
    corr_width = len(correlation)
    corr_height = len(correlation[0])
    pixels = image_list
    width = len(image_list)
    height = len(image_list[0])

    new_image_list = []

    for i in range(width):
        print(i)
        new_image_list.append([])
        for j in range(height):

            if median_flag is 1:
                neighborhood_pixels = []

            weighted_sum = 0
            for n in range(corr_width):
                for k in range(corr_height):
                    v_offset = n - corr_width // 2
                    h_offset = k - corr_height // 2
                    if i + h_offset < 0 or i + h_offset >= width\
                                        or j + v_offset < 0\
                                        or j + v_offset >= height:
                        if median_flag is 1:
                            neighborhood_pixels.append(padding)
                        else:
                            weighted_sum += padding
                    else:
                        if median_flag is 1:
                            neighborhood_pixels.append(pixels[i+h_offset][j+v_offset])
                        else:
                            weighted_sum += pixels[i+h_offset][j+v_offset]
                                            * correlation[n][k]
            if median_flag is 1:
                new_image_list[i].append(sorted(neighborhood_pixels)[
                                            len(neighborhood_pixels)//2])
            else:
                new_image_list[i].append(int(weighted_sum))

    new_image = list_to_image(new_image_list, image)
    return new_image
```

4.6 Converting Between Images and Lists

```
def image_to_list(image):
    width, height = image.size
    output_list = []
    for i in range(width):
```

```
        output_list.append([])
        for j in range(height):
            output_list[i].append(image.getpixel((i,j)))
    return output_list

def list_to_image(image_list, image):
    width = len(image_list)
    height = len(image_list[0])
    output_image = Image.new(image.mode, image.size)
    output_image_pixels = output_image.load()
    for i in range(width):
        for j in range(height):
            output_image_pixels[i, j] = image_list[i][j]
    return output_image
```