Cover - 5/5
Theory - 19/20
Implementation - 9/10
Results - 49/50
Code - 5/5
Organization - 6/10

Overall - 93/100
Make sure to look over the entire report together after combining your reports.

CS 474 - Image Processing and Interpretation

Fall 2021 - Dr. George Bebis

Programming Assignment #2 Report

Written by Joseph Trierweiler and Logan Leavitt

Joseph was responsible for Part One, Noise Generation, and Part Four

Logan was responsible for Part Two, Median Filtering, and Part Five

Part One - Correlation

Theory

Image correlation is a method of spatial filtering. Spatial filtering is defined by a neighborhood and an operation. A neighborhood is an area of pixels of “K” size, usually in the shape of a square. An operation is a user-defined process that produces a new value for the image. That new value is then stored in the corresponding center location of the window in the output image. Correlation is an example of one of those operations. The new weights generated through correlation represent the amount a window matches the neighborhood of an image. Correlation is a common example of a linear operator, since its output is a linear combination of its inputs.

Implementation

The implementation for correlation can be defined as follows:

-1

$$g(i,j) = w(i,j) \bullet f(i,j) = \sum_{s=-K/2}^{K/2} \sum_{t=-K/2}^{K/2} w(s,t) f(i+s, j+t)$$

This is less an implementation and more the definition of correlation. This better fits within the theory section.

This algorithm is used for creating the array of products used in our implementation

In this algorithm, matrix “g” is the output array of products, matrix “w” is an array of weights called a mask, and matrix “f” is the input image. As the algorithm iterates through the input image, it gets the sum of the input image’s pixel value multiplied by the center of the mask as defined by variables “s” and “t”. This linear combination of weights and original pixel values is then saved into an array of products. To output the image back into its own format, the array of products is then scaled by a constant “C” defined by the original image’s quantization level. After being scaled down, the output image shows what pixel values were most highly correlated with the input mask/array of weights.

A good thing to talk about in the implementation section would be how you dealt with applying the mask at edges pixels.

Results and Discussion

To demonstrate the implementation of correlation, we were given a pattern and an image to be used as our mask and image inputs respectively. “Pattern.pgm” (Figure 1) was used to find K, although in this case, the neighborhood was not a square as defined in the theory portion. Therefore, we split K into the mask’s x and y components in order to find the mask’s center. Figure 2 represents our results.

When we label figures, we typically just label it like "Figure 1." Then, if we want to add captions, we do so as an additional sentence like "Figure 1. The given mask named..."

Figure 1 is the given mask named “Pattern.pgm”

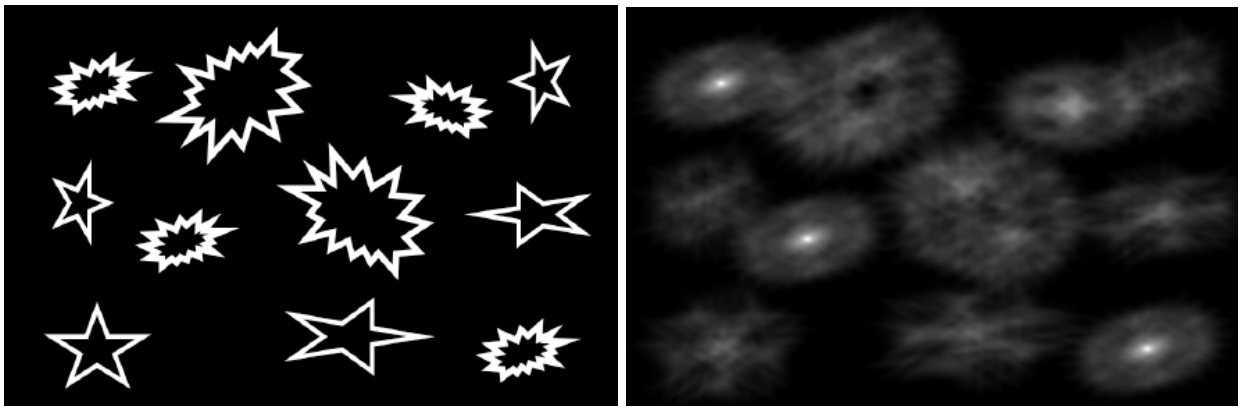


Figure 2 contains the input image (left) and the output image (right)

Analysis of figure 2 demonstrates how this implementation of correlation works. The white regions of the image represent the amount of correlation found between the center of the mask and the input image. Notice how in the center of the regions highlighted, three have high values. Those areas demonstrate a near-perfect match, which makes sense since they match Figure 1. Also notice how there is one high correlation spot in the upper right region of the image. It is not a perfect match as the image is flipped, but it is still correlated with Figure 1. It’s worth mentioning that, had we used an algorithm for convolution, it too would have been a perfect match.

Part Two - Averaging and Gaussian Smoothing

Theory

Smoothing filters, as they pertain to images, are a useful tool for removing noise in a image. In this section, we specifically consider averaging filters and gaussian smoothing filters. Both filters aim at making the features of an image appear smoother. In averaging, each pixel value is assigned the average value of the pixels in its $k\ x\ k$ neighbourhood. The result is that pixel values that are close to one another, become closer in value. Gaussian smoothing is very similar to averaging, except that each pixel value is now assigned a weighted average of the pixels in its neighbourhood according to a Gaussian distribution. This means that closer pixels are weighted more heavily, and that the resulting image will still appear smoother, but with less impact on the details of the image.

Implementation

Averaging is implemented in a straightforward manner; by summing up the values in each $k\ x\ k$ neighbourhood and dividing the sum by k^2. For Guassian smoothing, the filters are coded into the .cpp file as constants, and then applied to the image with a more general function “apply_filter”. Applying the filter is also calculated in the expected way, by manually calculating the weighted sum at each neighbourhood. The image is treated as if it is padded with zeroes on all sides to handle the edge cases. If an index falls out of bounds while applying the filter, it is treated as a zero.

This is what I was looking for in the previous implementation section.

Results and Discussion

The performance of averaging and gaussian smoothing is evaluated below by experimentally applying them to “lenna.pgm” and “sf.pgm”.

Figure 1: lenna.pgm and sf.pgm

-4

Make sure that when you stitch together your report that you go back over it together to make sure it is consistent. The captioning style is inconsistent with above and this isn't figure 1.

Also, the figure labels were never inserted into the text.

The results of averaging are shown in Figure ? and Figure ?. As can be seen, averaging effectively smooths the image. With a larger filter, the resulting image becomes even smoother, and more details are lost. It can also be noted that the edges of the images appear darker. This is a result of padding the input images with zeroes.

Figure 2: Results of averaging with 7x7 filter

Figure 3: Results of averaging with 15x15 filter

The performance of gaussian smoothing is shown in Figure ? and Figure ?. As predicted, gaussian smoothing seems to preserve the quality of the image with the same size filter, while still smoothing the image. As the mask size increases, the details of the image become difficult to make out.

Figure 4: Results of gaussian smoothing with a 7x7 filter

Figure 5: Results of gaussian smoothing with a 15x15 filter

In conclusion, both averaging and gaussian smoothing are effective at reducing noise in an image, but gaussian smoothing preserves more details for both the 7x7 and the 15x15 case.

Part Three - Median Filtering

Theory

An alternative to averaging and gaussian smoothing is median filtering. Since the average and median are both conceptually used to describe the "middle" of a dataset, it makes sense to instead pick the median of a $k \; x \; k$ neighbourhood in order to smooth an image. While similar on the surface, median filtering has a distinct advantage at reduced "salt and pepper" noise, as shown below.

-1

But why? What is salt and pepper noise?

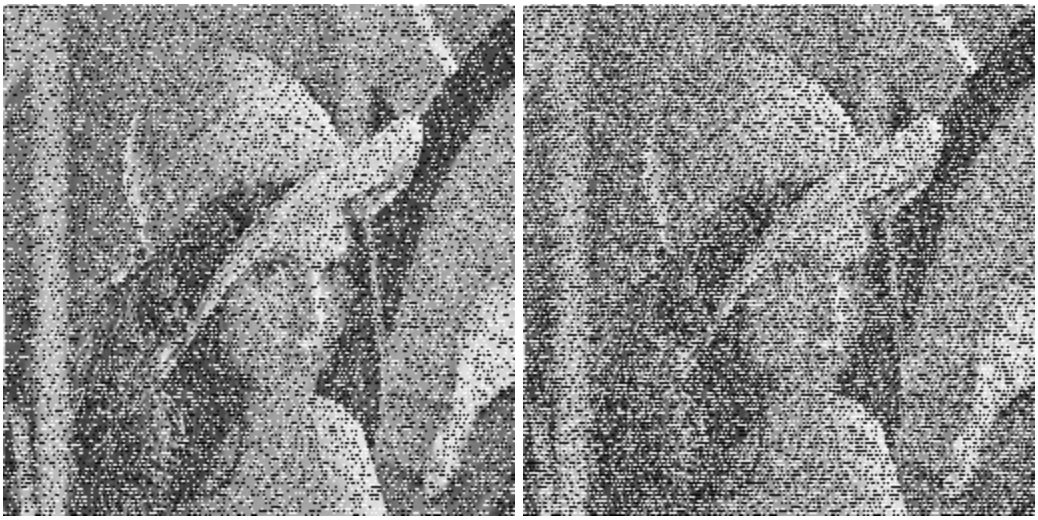
Implementation

To demonstrate the effects of median filtering, a separate program for generating a noisy image had to be made. The implementation of that program simply takes an input image's area and multiplies it by "X" representing the percentage of pixels the user wants to corrupt. It then iterates that product's amount until X% of the pixel values found in the image are set to 0 or 255. To see the results of median filtering, an algorithm to implement median filtering was also necessary. The implementation iterates over each $k \; x \; k$ neighbourhood, and stores all the neighbouring values in an array. The array is sorted using std::sort, after which the median can be easily found in the middle of the array. The pixel is assigned the median value, and the process repeats with the next neighbourhood of values.

Results and Discussion

Figures 1 and 2 illustrate the effects of simulating "salt and pepper". Both "lenna.pgm" and "boat.pgm" are corrupted with X = 30% and X = 50% for each image.

-1



This is almost, but not quite, salt and pepper noise. You need to be choosing the pixel color randomly, and independent of the row.

Figure 1 shows "lenna.pgm" when X = 30 (left), and when X = 50 (right)

Figure 2 shows "boat.pgm" when X = 30 (left), and when X = 50 (right)

Median filtering with a 7x7 mask is then applied to all corrupted images, as seen in Figures 3 and 4. While some details from the non corrupted images are not recovered, the filter is effective overall at removing the noise from the corrupted images. One can also note that the images with 50% corruption are slightly grainier than the images with 30% corruption as one might expect. In summary, this example showcases that 7x7 median filtering can be effective at making images more legible for the human eye.

Note the clear horizontal lines here - this is due to how you generated the noise

Figure 3 shows "lenna.pgm" with 7x7 median filtering applied, when X = 30 (left), and when X = 50 (right)

Figure 4 shows "boat.pgm" with 7x7 median filtering applied, when X = 30 (left), and when X = 50 (right)

The same experiment is repeated, but now with 15x15 median filtering. Results are shown in Figures 5 and 6. Again, it can be noted that the noise from the corrupted images is entirely gone. However, much more detail is lost in the images as a result of the larger mask. For this scenario, a larger mask was unnecessary. The correct size mask will likely depend on the size of the image, but these results suggest that a 7x7 mask or lower is optimal for removing "salt and pepper" noise from an image.

Figure 5 shows "lenna.pgm" with 15x15 median filtering applied, when X = 30 (left), and when X = 50 (right)

Figure 6 shows "boat.pgm" with 15x15 median filtering applied, when X = 30 (left), and when X = 50 (right)

In conclusion, median filtering is an alternative smoothing technique that performs especially well at removing "salt and pepper" noise from an image.

Part Four - Unsharp Masking and High Boost Filtering

Theory

Unsharp masking and High Boost Filtering are both methods of spatial filtering. The theory behind High Boost Filtering is to first obtain a sharp image by subtracting an unsharp (low-passed) image from the original image. This sharpened image can then be added to the original image, transforming it in two ways depending on the variable “K”. Variable K multiplies the sharp filter's pixel values. If K = 1, then the transformation is called Unsharp Masking. If K > 1, then the transformation is called High Boost Filtering. Both transformations result in an image with emphasized edges, but some details are lost.

You talk about K here in theory, but introduce the formula in implementation. The theory section is the best place to put the formulae you use.

Implementation

The following components were used for our implementation:

$$g_{mask}(x,y) = f(x,y) - f_{LP}(x,y)$$

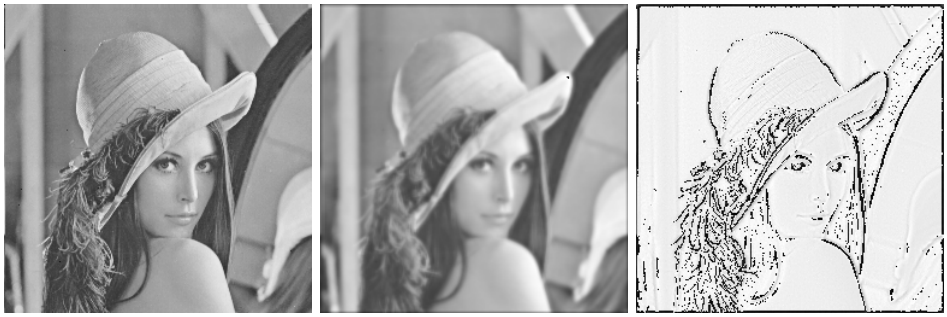
$$g(x,y) = f(x,y) + kg_{mask}(x,y), \quad k \geq 0$$

The figure on the left is a 7x7 Gaussian Mask, used for finding “f_LP” referenced in the figure on the right

As mentioned in the theory section, first we must obtain a sharp filter (g_mask) of the input image by subtracting a smooth filter (f_LP) of the input image from the input (f) itself. To create this smooth filter, a 7 x 7 Gaussian mask was used in the same way it was implemented in Part Two. After generating the low-pass filter it was subtracted from the image, leaving us with the sharp filter. We then use the sharp filter to generate our Unsharp Masked or High Boost Filtered Image. For our implementation, K was set to 2 for our High Boost Filtered Image.

Results and Discussion

Figure 1 demonstrates the effect of subtracting the image “Lenna.pgm” by a low-pass filter of itself.



Make sure you're remapping to [0,255] for visualization.

Figure 1 - Lenna.pgm (left), Lenna.pgm after a low pass filter is applied (middle), and Lenna.pgm after subtracting the low pass filter from the original image (right)

By removing all detail from the image, all that’s left are the original image’s edges. Also note, that the image in the figure is not completely accurate. That is because the white space in the sharpened mask is actually composed of negative values. These negative values help transform the image for the Unsharp Masked Transformation and the High Boost Filter Transformation shown in figure 2.

Figure 2 - Lenna.pgm (left), Unsharp Masked Transformation (middle), High Boost Filter Transformation (right)

Notice that the edges in both the Unsharp Masked Transformation and the High Boost Filter are more pronounced than the original image. Also notice that as the K value increases, some details such as the shadows and the feathers on the women's cap become less distinguishable.

This is because as K increases, the image will more resemble our sharpened mask. This is an important factor to consider when using a High Boost Filter.

Part Five - Gradient and Laplacian

Theory

Other methods which are useful for sharpening images include the gradient and the laplacian. Both methods take similar approaches, but use different mathematical derivatives. In essence, the two methods calculate the rate of change in pixel values at every location in an image. A low rate of change implies a low amount of detail in that area, while a high rate of change implies a high amount of detail. Thus, we can use the rate of change to sharpen the details in an image. The gradient is given by $\nabla f = (\partial f/\partial x,\ \partial f/\partial y)$. To calculate the magnitude, we use two separate masks to individually calculate $\partial f/\partial x$ and $\partial f/\partial y$, and combine the results. On the other hand, the laplacian is given by $\nabla^2 f = \partial^2 f/\partial x^2 + \partial^2 f/\partial y^2$, and can be calculated with a single mask. In both cases, we can use the results to sharpen the input image.

Implementation

To calculate the partial derivatives and the laplacian, the code to apply a mask to an image is reused here. The masks to apply are the Prewitt, Sobel, and Laplacian masks. To complete the procedure, there is code to calculate the gradient's magnitude from the partial derivatives, and code to add two images together to obtain the sharpened image. In addition, any of the images representing the partial derivatives, magnitude of the gradient, or the laplacian are normalized to have pixel values in the range [0, 255].

Results and Discussion

Figure 1 shows "lenna.pgm" and "sf.pgm"

The results of the Prewitt masks on lenna.pgm and sf.pgm are shown in Figures 2 and 3. As can be seen, the Prewitt masks effectively captures the edges in the input image, and enhances them in the output image. One observation is that the highlighted edges in the resulting image are too bright to look natural.

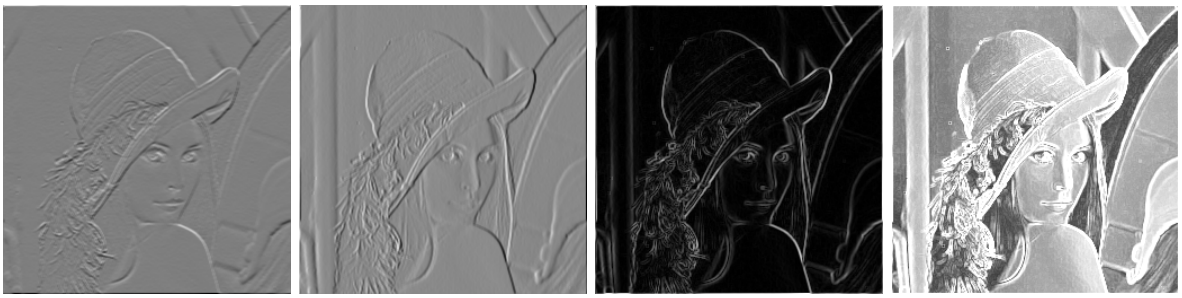


Figure 2 shows results on "lenna.pgm" with Prewitt masks. From left to right: partial derivative of y, partial derivative of x, magnitude of the gradient, and the sharpened image.

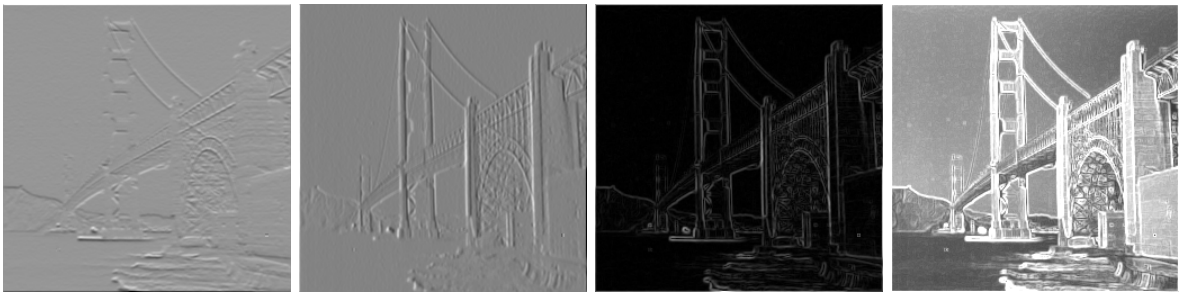


Figure 3 shows results on "sf.pgm" with Prewitt masks. From left to right: partial derivative of y, partial derivative of x, magnitude of the gradient, and the sharpened image.

The same experiment is repeated with the Sobel masks in Figures 4 and 5. The results are similar, without any discernible differences.

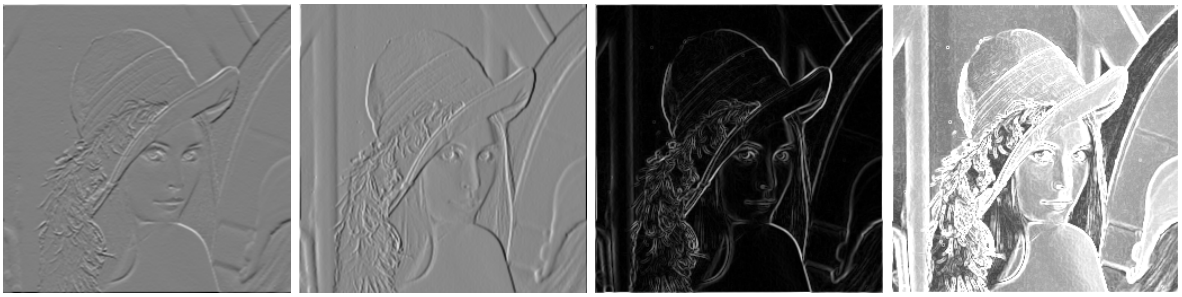


Figure 4 shows results on "lenna.pgm" with Sobel masks. From left to right: partial derivative of y, partial derivative of x, magnitude of the gradient, and the sharpened image.

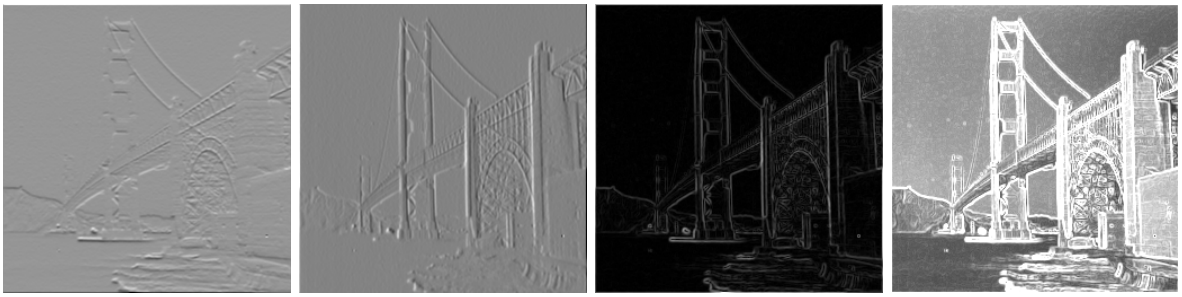


Figure 5 shows results on "sf.pgm" with Sobel masks. From left to right: partial derivative of y, partial derivative of x, magnitude of the gradient, and the sharpened image.

Finally, the experiment is also attempted with the Laplacian mask in Figures 6 and 7. The Laplacian is not only more efficient, but appears to produce a more natural image as well. Certain details (like the hair in “lenna.pgm”) are sharpened without looking excessively bright.

Figure 6 shows results on “lenna.pgm” with Laplacian mask. The Laplacian of the image (left), and the sharpened image (right).

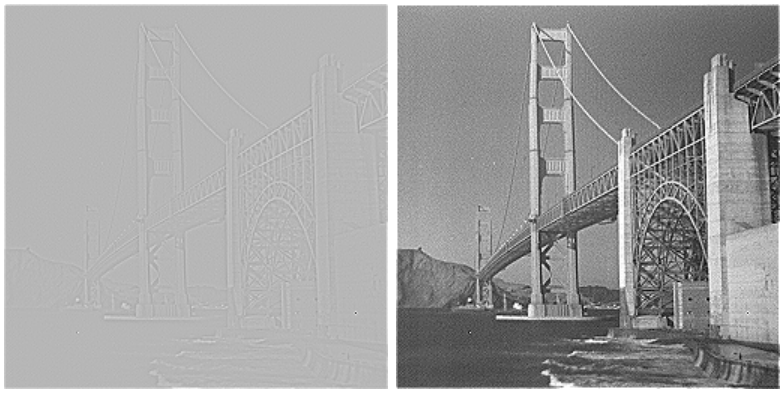


Figure 7 shows results on “sf.pgm” with Laplacian mask. The Laplacian of the image (left), and the sharpened image (right).

In conclusion, while the Prewitt and Sobel masks are effective for detecting edges in an image, the Laplacian mask resulted in the most natural sharpened image.