EECS490_hw2_tkg11_Guen

1-a-1. First Order Derivative Edge Detection

Motivation

Edge detection is an incredibly useful image processing technique that is commonly used as a preprocessing step in computer and machine vision. In general, object localization and object identification in images is determined by the silhouette and the defining features of the object in the image. Edge detection can effectively identify these features, segmenting the image, and making it easier for an algorithm or a person to a identify an object in an image. First order derivative edge detection is a simple and understandable way of performing this procesdure.

Approach

First Order Derivative Computation

The first order edge detection procedure works by taking two predefined 3×3 masks, a row mask M_R and a column mask M_C that, for the center pixel, approximate the discrete first order derivatives in their respective directions, $\frac{\delta f}{\delta dx}, \frac{\delta f}{\delta dy}$. For a 3×3 subimage S, the final first order derivative for the center pixel s is the magnitude of the gradient, calculated according to the formula:

$$||\nabla s|| = \sqrt{\frac{\delta f}{\delta dx} + \frac{\delta f}{\delta dy}}$$

This first order derivative represents the rate of change of gray values surrounding the pixel. A high rate of change implyies an edge. The chosen masks are then convolved over the image, and the above formula is computed for each pixel in the original image, returning a matrix D containing the approximated first order derivative value at each pixel.

Threshold Calculation

A threshold value t is then calculated such that a fraction, p, of pixel derivative values fall below that threshold. This done by taking the matrix D, computing the cumulative distribution C, and then finding the smallest gray value i for which

$$C(i) > n * p$$
$$n = \# \text{ pixels in image}$$

Threshold Application

The final edge detected image G is produced using a per pixel thresholding operation following the formula:

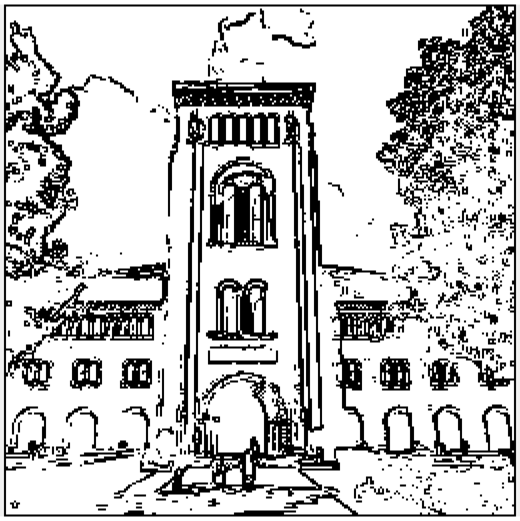
$$G(i,j) = \begin{cases} 255 & \text{if } D(i,j) > t \\ 0 & \text{otherwise} \end{cases}$$

Results

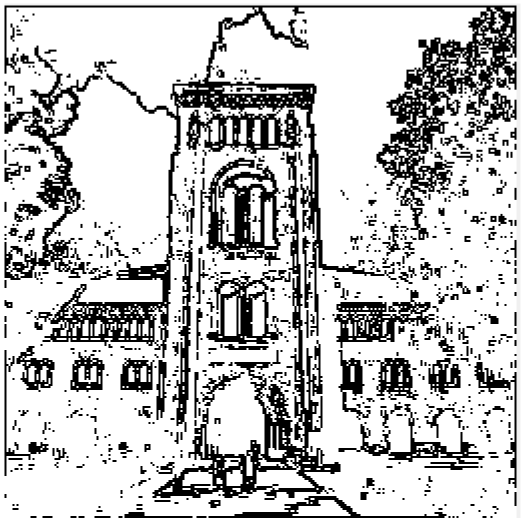
$p = 0.75$

operator=prewitt

building.raw



building_noise.raw



Discussion

Something to note before discussion is that the value of p greatly determines the appearance of the result, a greater value giving fewer edges, a smaller value giving more edges. It appears that the first order edge detection works well for the original image, although some distinct edges are lost, particularly on the left edge of the building where the contrast between the building background in the sky is quite small. There are not too many false edges though, and it manages to correctly outline most of the windows and defining features of the building. It has noticeably missed some smaller details though, which is particularly noticeable in the top part of the main tower, where there are small features that have been poorly identified. The wide edge that is expected with the first order derivative does not work well for that case. Another downside to the first order derivative is a lack of sensitivity to changes over a larger area of pixels (smooth edges), which is particularly apparent in the missed edge on the left side of the main archway.

The noisy image is quite poor, though, since there are lots and lots of false edges detected where there are larger local changes due to noise. This result can be improved by reducing p, however, other details and edges that should be marked are also removed in this process.

1-a-2. Second Order Derivative Edge Detection

Motivation

The edge detection motivation is the same as that of *1-a-1. First Order Derivative Edge Detection*. The motivation to use the second order derivative as opposed to the first order derivative is important though. In general, the first order derivative lacks in identification of smooth, wide edges, resitance to noise, as well representation of fine detail. These are all traits of an edge detector that could be a hindrance in future use of the image. The second order derivative is then proposed in hopes that the property of the second order derivative to accurately detect maxima and minima across pixel values will provide finer, more tightly localized edge detection, as well as improve detection over wider edges.

Approach

Second Order Derivative Computation

Similar to the first order derivative, the second order derivative computation is done using a discrete approximation over a 3×3 window with a 3×3 mask. The mask is convolved over the image to produce an output matrix D containing the second order derivative value per pixel.

Threshold Calculation

The threshold calculation itself is the same as in *1-a-1. First Order Derivative Edge Detection: Approach: Threshold Calculation*, however the thresholding operation is not performed directly on the convolution output matrix D. Instead, because the classification of an edge in second order edge detection is based on zero crossing locations, a matrix Z is created that quantifies the "degree" of zero crossing in a 3×3 subimage S centered at pixel location $D(i, j)$ by the following formula:

$$Z(i, j) = |max\_positive(S) - min\_negative(S)|$$

From this matrix Z, then, the same threshold calculation from *1-a-1* is performed, preducing a threshold value t.

Threshold Application

The threshold application works conceptually by assuming that a larger difference between the maximum positive value and the minimum negative value indicates a more likely zero crossing in the area. The thresholding operation then follows the formula

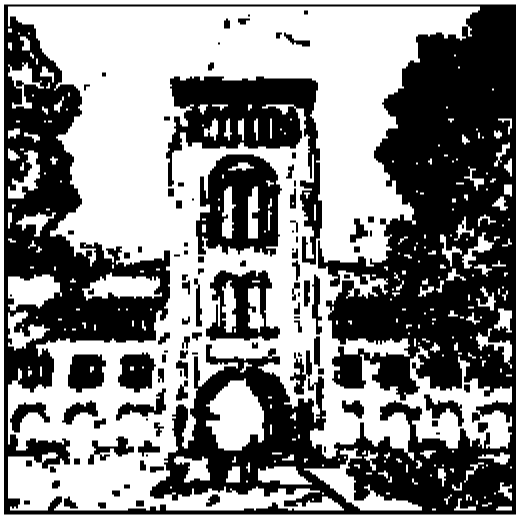
$$G(i,j) = \begin{cases} 0 & \text{if } Z(i,j) > t \\ 255 & \text{otherwise} \end{cases}$$

Results

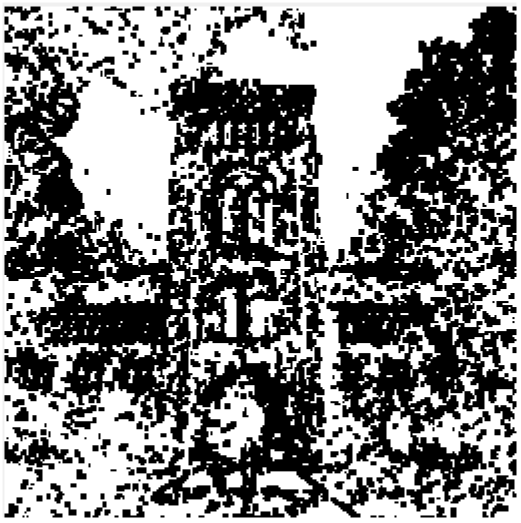
$p = 0.5$

$\text{operator}{=}\text{laplacian}$

building.raw



building_noise.raw



Discussion

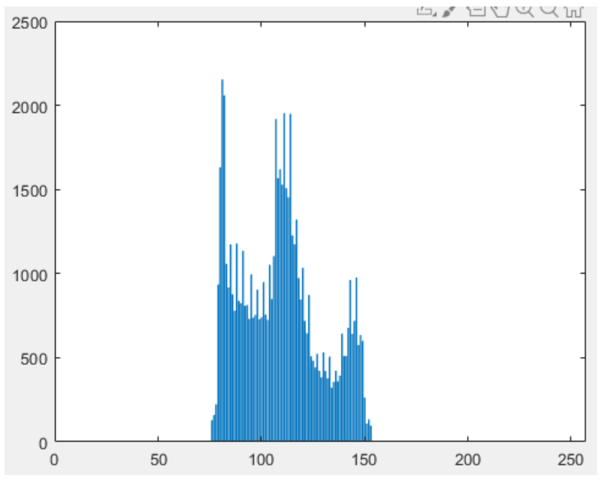
The results of the edge detection look generally inferior to that of the first order edge detection. However, this is to be expected considering the nature of the images before being preprocessed to remove noise and/or improve contrast. This is because second order edge detection is generally more sensitive to very small fluctuations in gradient value, which is why the noisy image results have so many incorrectly identified edges. The edge detection itself looks noisy. And for the low contrast image, the edges look large and blocky, because the low contrast image creates wide ramps that are better detected by the second order edge detection than the first order edge detection.

1-b . Preprocessing for Improved Edge Detection

Image Analysis

building.raw

This image clearly suffers from poor contrast, although not much noise.



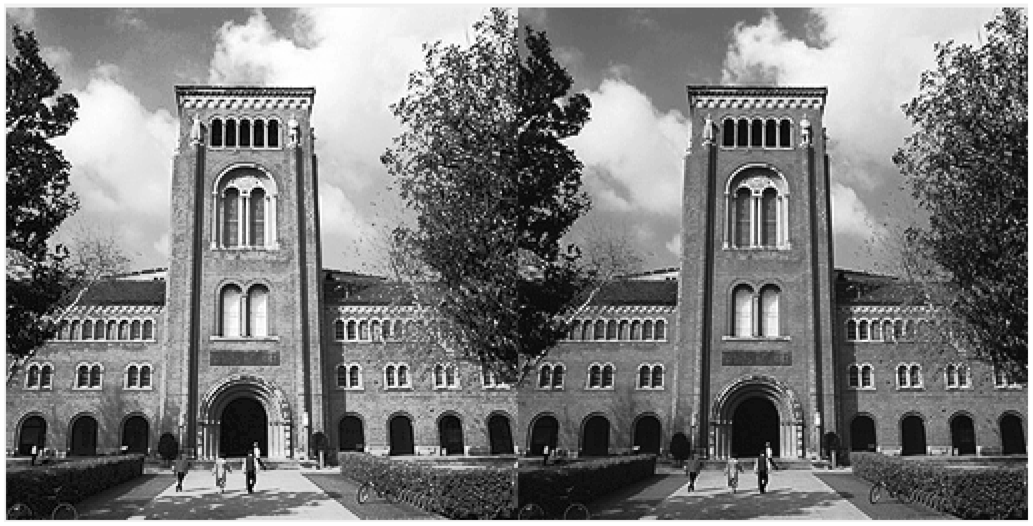
Looking at the histogram, it's clear that the image values are distributed across a small range. For this image, we want to use either histogram equalization or linear scaling in order to improve the contrast. In terms of aiding edge detection, we want a method that provides as high contrast as possible, since edges are determined by the rate of change of gray value relative to their neighbors. The two methods we are familiar with are histogram equalization and linear scaling.

Conceptually, we would expect histogram equalization to be the better method to use for edge detection since it will distribute the gray values across a wider range, making the difference in gray value across any set of pixels greater. This is really the definition of contrast.

Visually, we can also see that histogram equalization appears to provide better contrast:

histogram *full range linear scaling*

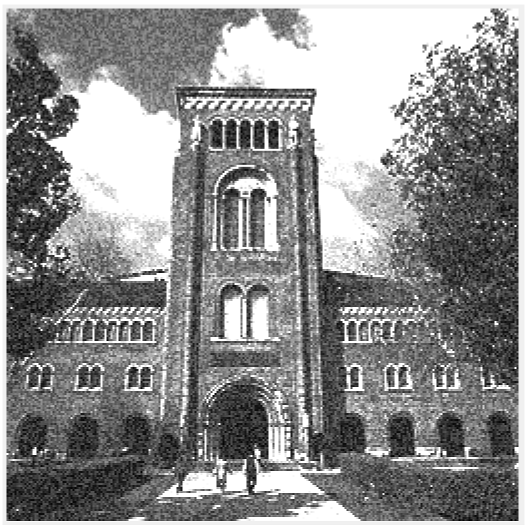
equalization



The improved contrast is most apparent in the windows of the building, where the darkness inside the windows is fairly well maintained and consistent across the two images. However, the building is far lighter in the histogram equalized image vs the linearly scaled image. In edge detection, this should provide a better environment for determining edges.

building_noise.raw

Looking at the image, it appears that it suffers from uniform noise. Although there appear to be some small impulses, after looking at the contrast enhanced versions of the original image above, it appears these impulses already existed in the image from sun highlights.

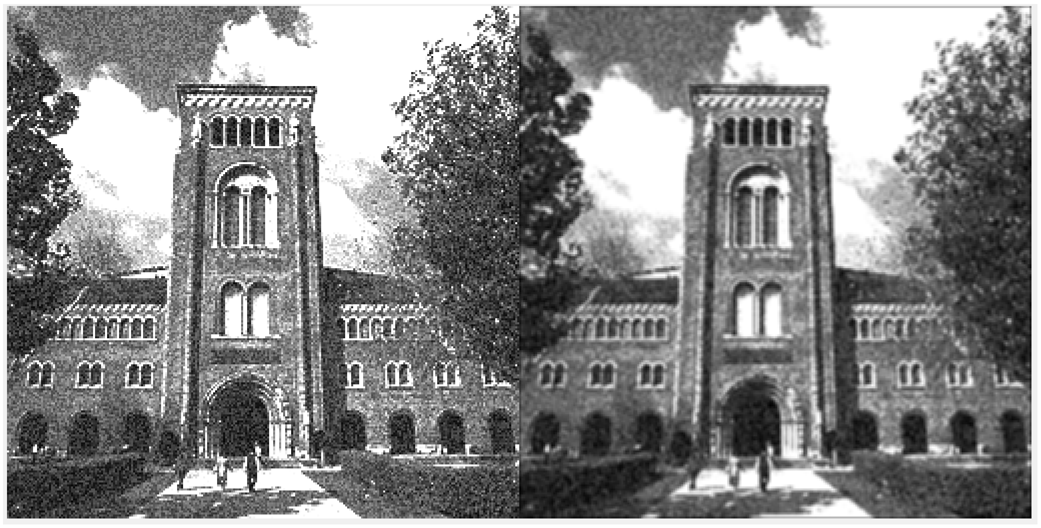


In the previous assignment, it appeared that *Mask 10.3-2c* performed the best for both Gaussian and uniform noise removal.

$$\mathbf{H} = \frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} \qquad (10.3\text{-}2c)$$

Looking at the noisy and denoised image side by side:

Original *Denoised*



Clearly the denoised image looks less noisy, but it also looks more blurry. The blurriness may be a hindrance during edge detection since the algorithm may have a hard time localizing edges to a single pixel. However, the blurriness should also reduce the false positives that occur during edge detection due to rapid changes in gray value as a result of the noise.

1-b-1. First Order Edge Detection With Preprocessing

Motivation

The motivation of preprocessing is apparent in the poor performance of the first order edge detection for *building_noise.raw*, as we hope we can reduce the noise in the image in order to reduce the number of dots marked as edges throughout the image. For the low contrast image, the edge detection appears to work well, but it should work better with higher contrast, hopefully

Approach

*For details on the preprocessing used for both images, see **1-b. Preprocessing for Improved Edge Detection***

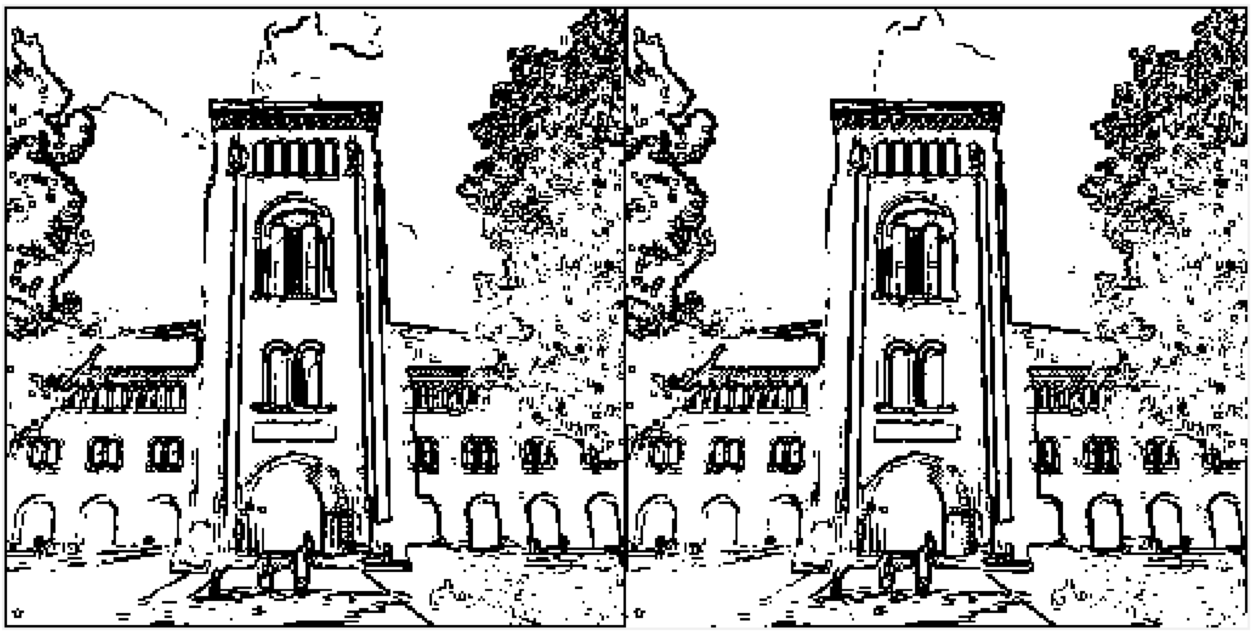
*For details on the first order derivative edge detection method, see **1-a-1. First Order Derivative Edge Detection: Approach***

Results

p=0.8

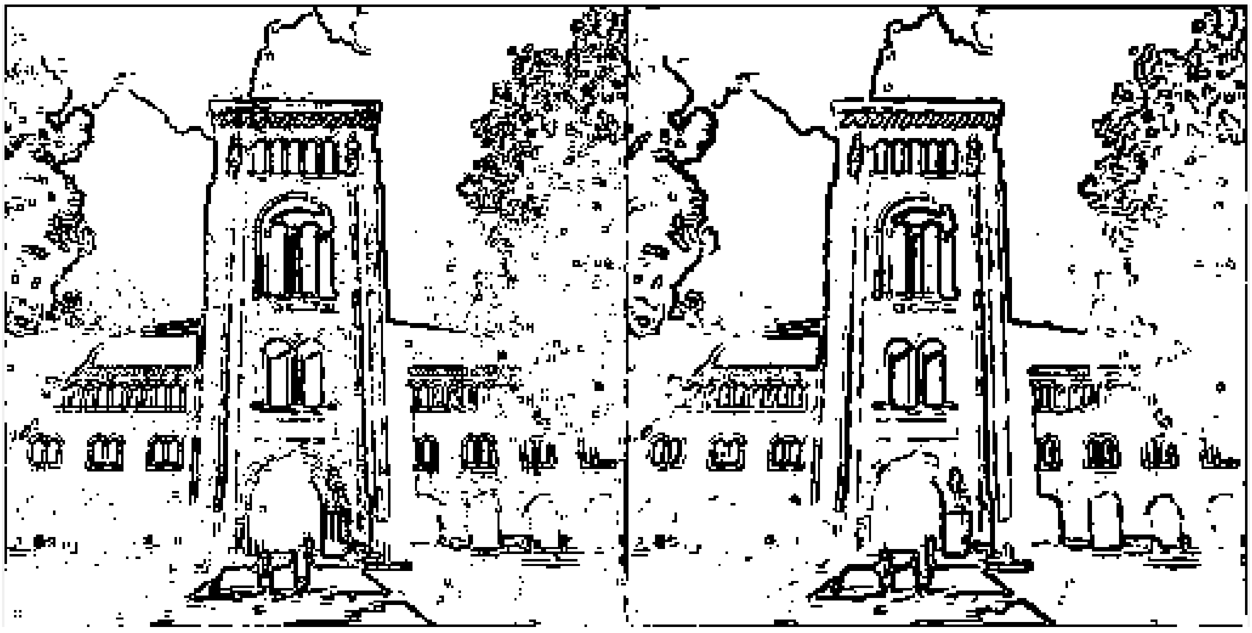
building.raw

Original *Histogram Equalized*



building_noise.raw

Original *Noise Removed*



Discussion

The result for *building.raw,* the original with first order edge detection compared to the histogram equalized image with first order edge detection, is quite poor. There is actually very little change between the two images and, in some cases like the left side of the tower and the left-most cloud, the non-preprocessed image appears to localize the edges better than the histogram equalized version. My hypothesis for why there is such little change is that the important part of first order edge detection is the relative rate of change between pixels in the image. If the original image had a sufficient contrast for the algorithm to still compute high and low derivatives where necessary, then increasing the contrast (and in turn the magnitude of the derivative values on average) would have little effect. The only case where we would see a large change between the original and the contrast-enhanced images is when the original had such poor contrast that the edge detection algorithm was unable to appropriately label derivative values. Actually, this result demonstrates the robustness of the algorithm against low contrast images, probably due to the use of the dynamically calculated threshold value. Because the threshold simply selects the top $1 - p$ fraction of derivative values to be edges, even if the difference in maximum and minimum derivative values changes, the actual number of derivative values selected to be edges does not. This causes a similar output for different contrast images, in spite of their differences in derivative magnitude.

The result for *building_noise.raw*, on the other hand, is much better and for obvious reasons. It is clear that many of the falsely labeled edges in the original are removed in the de-noised version. This was the goal of performing a noise reduction on the image. There are also more clearly defined lines that don't wiggle as much, as can be seen most clearly in the top of the tower. However, the blurring of the image means that the edges and derivative values are more widely distributed, which causes the edges to be generally localized to larger areas. Thin edges in the original are now thicker in the de-noised version, which is to be expected.

1-b-2. Second Order Edge Detection With Preprocessing

Motivation

The motivation is essentially the same as *1-b-1. First Order Edge Detection With Preprocessing: Motivation*. There is an additional desire in the case of second order derivative edge detection, however, to reduce noise since in general the second order derivative is more sensitive to gray level fluctuations.

Approach

*For details on the preprocessing used for both images, see **1-b. Preprocessing for Improved Edge Detection***

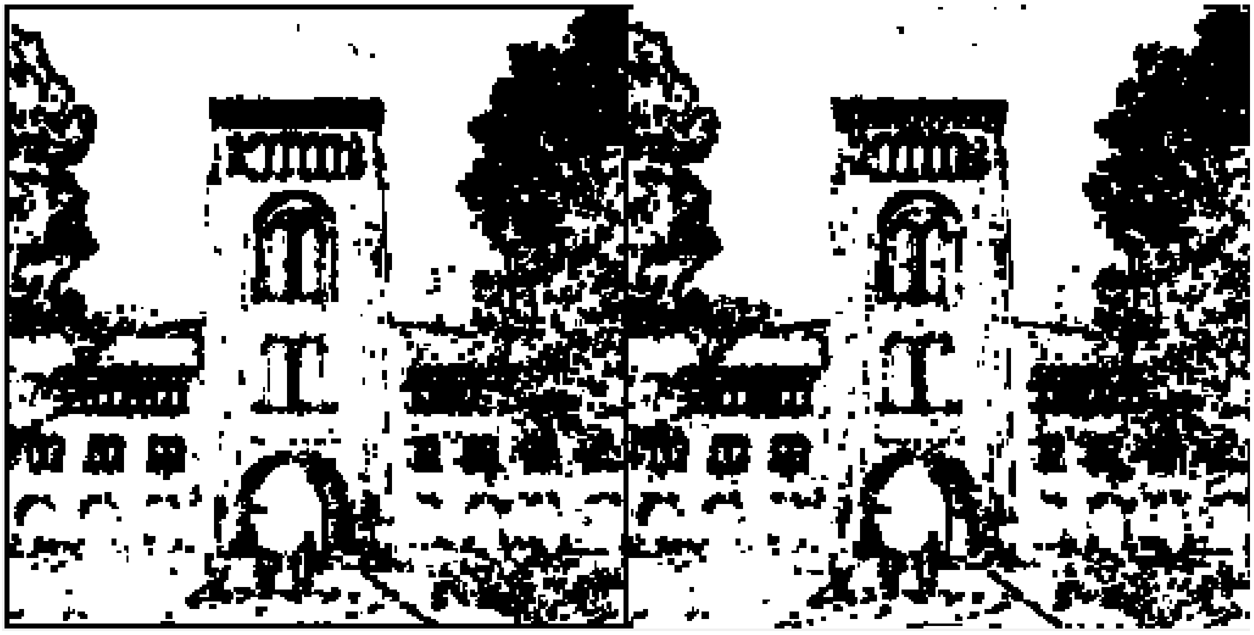
Something to note is that there are other options for denoising images that are used for second order derivative edge detection, such as the Laplacian of Gaussian, which can allow the smoothing and gradient calculations to be applied at the same time. The effect should be the same as applying the smoothing first and then applying the Laplacian. Because the code for applying the smoothing was already written in the first assignment, the smooth first, detect second method was used.

*For details on the first order derivative edge detection method, see **1-a-2. First Second Derivative Edge Detection: Approach***

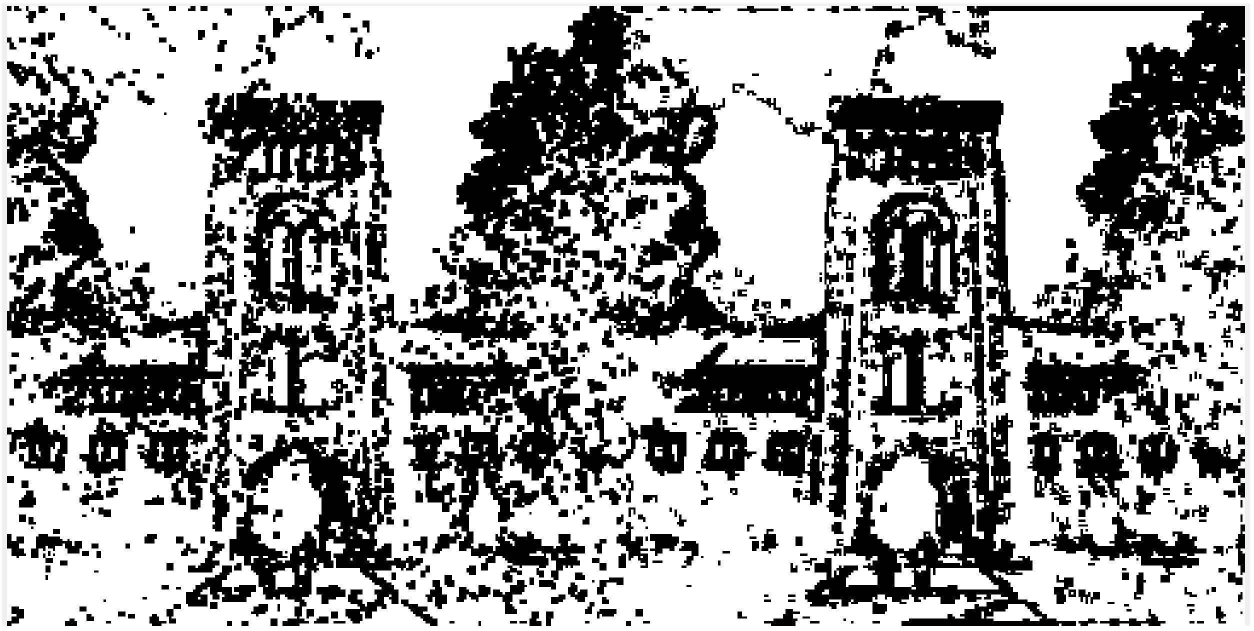
Results

$p = 0.6$

building.raw



building_noise.raw



Discussion

We see similar problems with the second order edge detection after preprocessing as we did without preprocessing. Lots of the edges are being localized to more than one pixel. Similar to the first order edge detection with contrast enhancement, the *building.raw* enhanced image did not show much improvement, although there are more edges detected many are false positives as well. For *building_noise.raw*, the reduction in noise is pretty significant, as expected, but the overall results are not as good as in first order edge detection.

2-a. Shrinking

Motivation

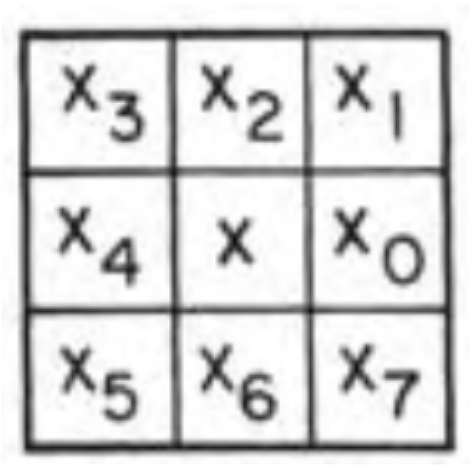
Shrinking has the goal of reducing the size of objects while roughly maintaining their proportions. This manifests as hole-less objects get small towards the center, and objects with holes shrink from the outside while the hole also gets bigger. Shrinking is useful usually as a preprocessing step for computer vision or image processing applications where one wants to be able to, more easily, individually identify objects in an image. A number of objects that are too close together will be harder to differentiate, so shrinking them will make them easier to detect. The shrinking still maintains their general shapes though, so the actual detection of the objects should not have to change to accomodate the size.

Approach

The shrinking process works by applying two morphing filters which tell the program what pixels to erase. Both morphing filters work on the basis of defining a set of 3×3 binary patterns. These patterns are entered manually into a $3 \times 3 \times N,\ N = \#patterns$, matrix and ten are, as a group, converted into an integer vector of length N by applying the following formula for each pattern:

$$x + x_0 + 2^2 * x_1 + 2^3 * x_2 + 2^4 * x_3 + 2^5 * x_4 + 2^6 * x_5 + 2^7 * x_6 + 2^8 * x_7$$

Where x_i in 3×3 window is given by:



This formula provides a unique integer value for every pattern. The patterns are defined with 1's being object pixels and 0's being background pixels. The provided images, however, (*patterns.raw* and *pcb.raw*) are given in the reverse. Thus the image is first inverted before applying the filters.

The first filter moves a 3×3 sliding window across the original image, calculates the integer id of the pattern, and compares that to the list of conditional patterns defined for the first filter. If that id exists in the list, it marks that pixel as a pixel to erase.

The second filter is then given the first filter, and does the same sliding window and id calculation procedure. It compares the window patterns to its own set of patterns (unconditional patterns). If there is a match, then it marks that pixel to not be erased. The second filter simply modifies the first, removing certain erasure suggestions.

Finally, the second filter is used as a guide to erase pixels from the original image. Any pixel in the filter marked 255 is erased (set to background color), and any pixel in the filter marked 0 is left as is in the original image.

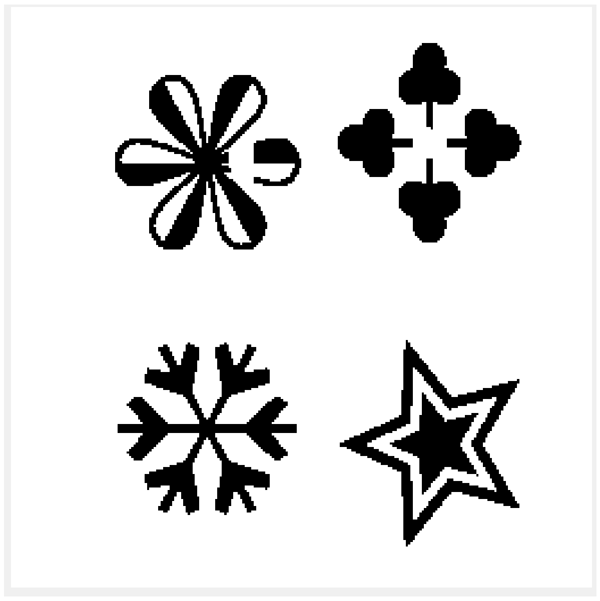
This process of applying the first and second filter is repeated for a number of iterations, where upon each iteration the original image passed into filter 1 is the output of the previous iteration.

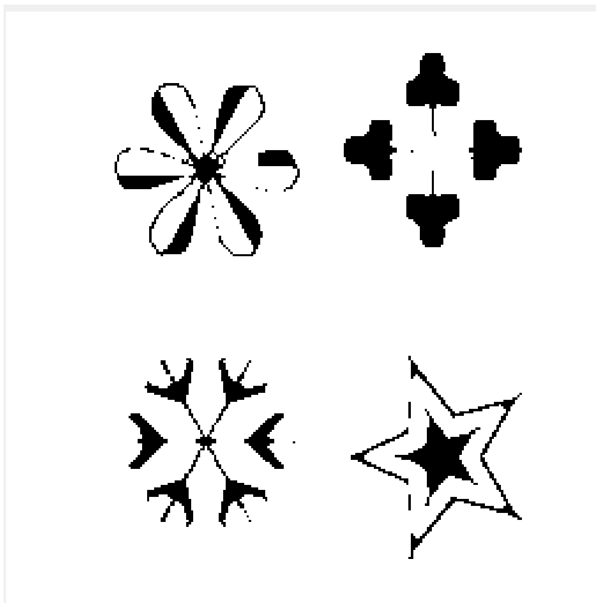
Finally, the image is inverted again.

Results

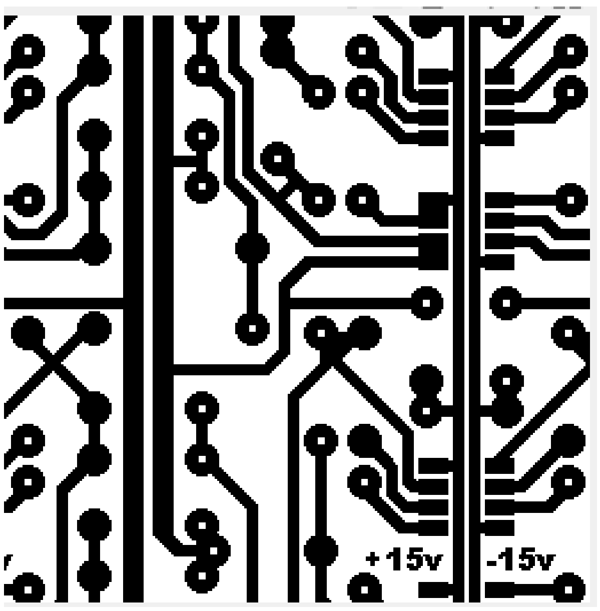
2 *iterations*

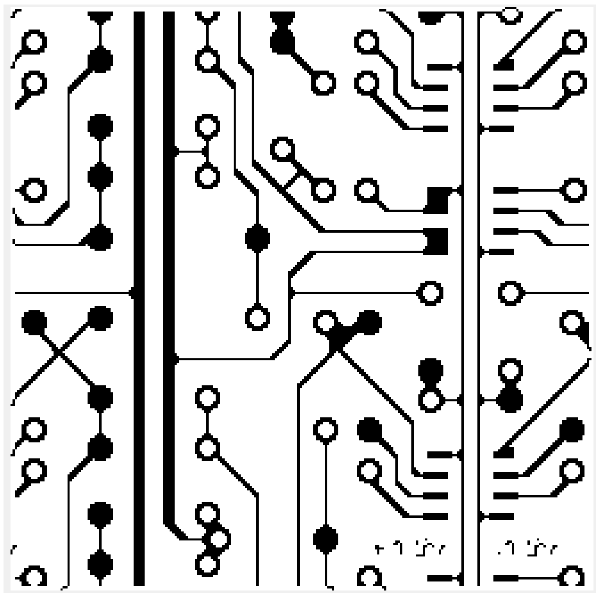
patterns.raw





pcb.raw

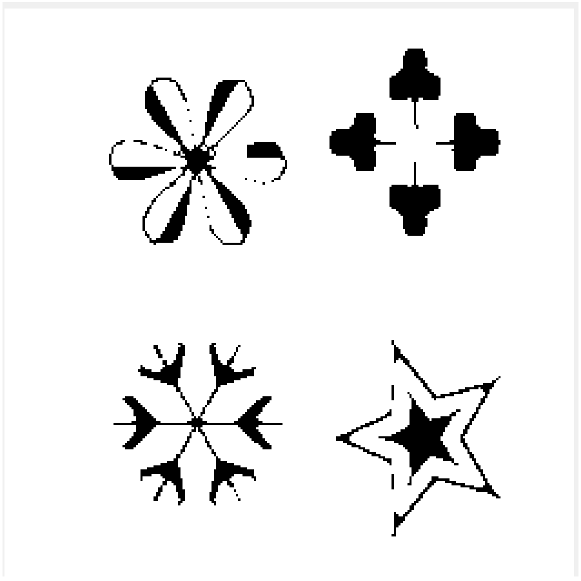




Discussion

2 iterations appeared to provide the most accurate representation of the defined function of shrinking. More iterations still appeared correct, however more edges and shapes were reduced to smaller points, making the effect less apparent and useful.

For *patterns.raw*, the effect works quite well, wherein even difficult shapes to redefine at low resolution, like the star, still maintain their general shape. Shapes like the "flower" in the top left corner clearly have the whitespace (hole) grow, while also shrinking from the outside. It does appear, however, that both the snowflake and the two horizontal clovers suffered from disconnect or complete erasure on perfectly horizontal lines that were reduced after the first iteration to a $\begin{bmatrix}1 & 1 & 1\\1 & 1 & 1\\0 & 0 & 0\end{bmatrix}$ or $\begin{bmatrix}0 & 0 & 0\\1 & 1 & 1\\1 & 1 & 1\end{bmatrix}$ pattern. This behavior is apparent, though, when inspecting the shrink patterntable wherein these two patterns are marked for erase by the conditional patterns, producing the same pattern after the first filter, yet are not inhibited by the unconditional patterns after. To prove so, this connection can be corrected by simply including one of these patterns in the unconditional table, producing this output for *patterns.raw*:



For *pcb.raw,* The effect is the same and we see a good representation of what shrinking is supposed to do. The circles with holes are reduced in perimeter size, yet also have larger holes, and the larger circles without holes become smaller while maintaining their solidity. Lines are preserved yet reduced in thickness.

2-b. Thinning

Motivation

Thinning is a process that seeks to remove pixels from objects such that lines are reduced to single pixels while still maintaining connectivity. Thinning can be useful for cleaning up results from edge detection, in order to take edges that were localized to several pixels down to one.

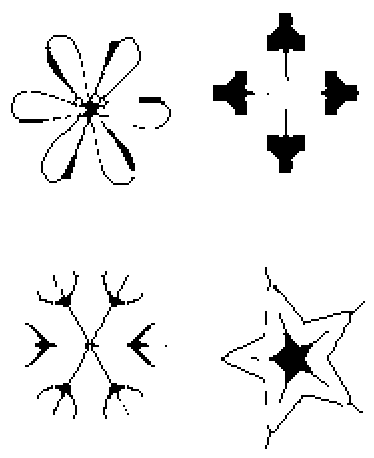
Approach

The approach for thinning is identical to that of ***2-a: Shrinking*** with the exception that the conditional and unconditional patterns are different, as defined by tables *14.3-1* and *14.3-2* from the textbook.

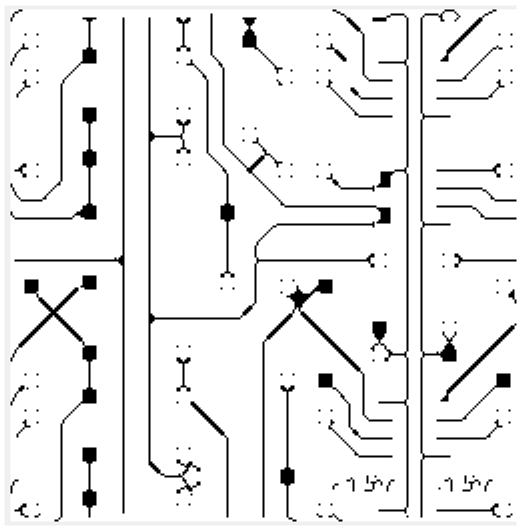
Results

4 Iterations

patterns.raw



pcb.raw



Discussion

There is not much to say here except that the result is as expected. In almost all cases, large shapes and thick lines are reduced to smaller shapes and single pixel lines respectively. There is little disconnect though. The quality of the result is especially apparent in the *pcb.raw* image where previously very thick lines have been reduced in all cases (except for junctures) to single pixels. The algorithm does a good job maintaining continuity in non-straight lines as well, as can be seen in many places in *pcb.raw* where there are more curved lines. There are, however, reductions of the circles with holes to squares with 4 black corners. Although this is to be expected, the effect is not necessarily in line with the maintenance of continuity that is expected with thinning.

An interesting note is that because it only takes a couple of iterations in order to reduce most of these lines to their minimum sizes for thinning, further iterations have effect. In fact, 2 iterations alone appears to be enough to thin *pcb.raw* sufficiently.

2-c. Skeletonizing

Motivation

Skeletonization seeks to reduce an object such that it represents the region originally contained by the object with only thin lines. Skeletonizing is generally useful because it provides a simplification of an object, while also maintaining its general size and shape. This can be used for more easily determining sizes and shapes of objects, as only the central, essential points need to be considered in order to characterize these features of the object.

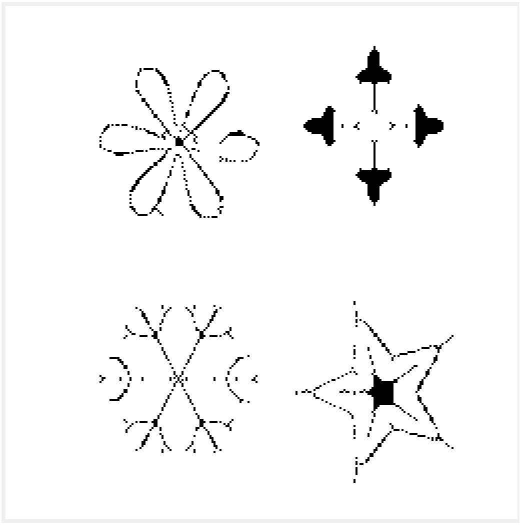
Approach

The approach for thinning is identical to that of ***2-a: Shrinking*** with the exception that the conditional and unconditional patterns are different, as defined by tables *14.3-1* and *14.3-2* from the textbook.

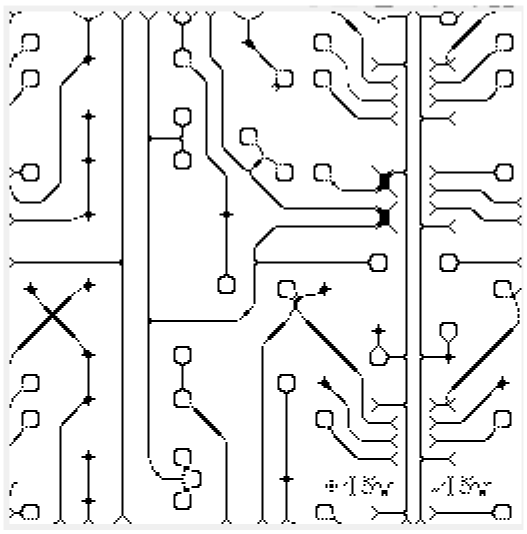
Results

8 *Iterations*

patterns.raw



pcb.raw



Discussion

In general, the skeletonizing results look good. In *patterns.raw*, there are some clear disconnections in horizontal lines (similar to what was seen in thinning), however the general effect is in line with the goal of skeletonizing. 8 Iterations was picked because it seemed sufficient to demonstrate the ability of the skeletonizing operation to reduce the objects to their more basic structures. Although the two patterns on the right side are not completely skeletonized, their substructures for the parts that are fully skeletonized match the expectation of the operation; he skeleton gives a spatial description of the object in a much smaller form.

In *pcb.raw*, the results are again as expected, and differ from thinning as it properly should. Although lines are similarly localized to single pixel lines, the larger circles in the image are reduced to small, skeletonized points, and the circles with holes maintain their outlines.

3-a. Fixed Threshold Dithering

Motivation

Dithering is a useful image processing technique that attempts to convert an image with many gray levels (or colors) to a binary image, while maintaining the appearance of the image. In general all techniques seek to give the same sense of color scale that the original image had but using only black and white. This technique is useful in any situation where the hardware that is displaying an image can only do so with a limited range of colors, yet still wants to maintain as much of the original image appearance as possible.

Approach

The fixed threshold dithering algorithm performs a per-pixel thresholding operation on the input image F producing an output image G as defined by:

$$G(i,j) = \begin{cases} 255 & \text{if } F(i,j) > t \\ 0 & \text{otherwise} \end{cases}$$
$$t = \text{0-255 threshold value}$$

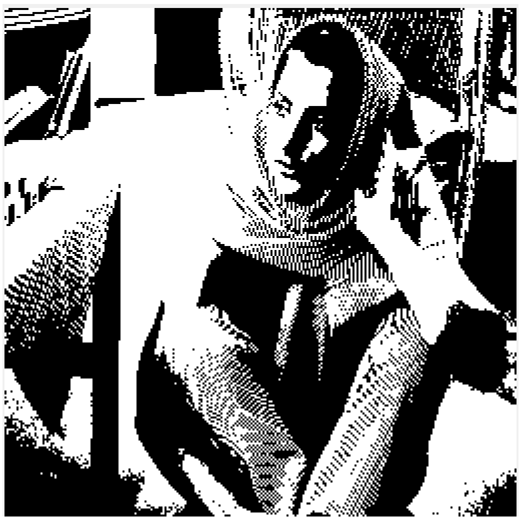
The threshold can be picked manually, however the threshold for the results below was selected by using the cumulative distribution to find a value such that roughly half are less than and half are greater than the value. This value was 96 for *barbara.raw*.

Note: For details regarding cumulative distribution threshold calculation, see section ***1-a-1: Approach: Threshold Calculation***

Results

Theshold = 96

barbara.raw



Discussion

The results of the fixed threshold dithering are good, however areas that were not already patterned do not do a good job of mixing white and black together to give apparent gray. Although the pants and side of the tablecloth look good, the head has a really stark line between the hair and the face, as a result of the thresholding operation sending all face value pixels to white and all hair value pixels to black. the outcome is pretty good for this image, but loses a lot of gradient detail and would probably not look as good for images with no small, repeating patterns like this image has.

3-b. Random Dithering

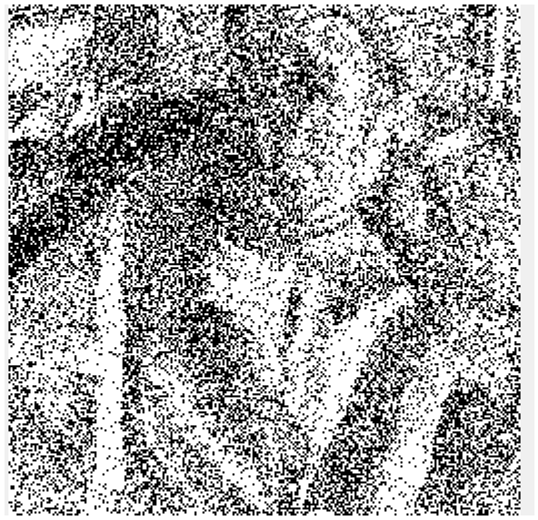
Motivation

The motivation for random dithering in use is the same as *3-a: Fixed Threshold Dithering*. The use of a random threshold as opposed to a fixed threshold is probably to fix the issue discussed above with fixed threshold dithering wherein there is a really stark contrast between areas that don't have large, low spatially distributed variations in color. The random threshold selection should make it so that not all pixels in a region below $255/2$ are mapped to 0 and vice versa for those mapped to 255.

Approach

The approach is the same as *3-a Fixed Threshold Dithering* except that the threshold is not constant, rather it is selected randomly for each pixel in the image.

Results



Discussion

The result of this method is terrible, and understanbly so, since there is no guide for the random threshold selection. There appears to be a really rough outline of objects in the scene, but their colors vary pretty greatly compared to the original image. This is to be expected, though, since the output value for any pixel is more or less random, the probability that it is mapped to 0 or 255 only being determined by how close or far away its original value is to $255/2$.

Perhaps a better method would be to apply a random threshold to only a certain percentage, maybe 10% in order to give variation but still have a well defined image

3-c. Dithering Matrix (Pattern)

Motivation

The motivation for matrix dithering is the same as *3-a: Fixed Threshold Dithering*. The goal of the dithering matrix, however, is to improve the smoothness of the variation of apparent gray levels across the image. This is an effect that you don't see in fixed or random threshold dithering.

Approach

2×2 Dither:

The 2×2 matrix dither works by using the index matrix given by:

$I_2(i,j) = \begin{bmatrix} 1 & 2 \\ 3 & 0 \end{bmatrix}$

This index matrix is then converted to a 2×2 threshold matrix via the formula:

$T(x,y) = \frac{I(x,y)+0.5}{N^2},\ N = 2$

```
N = size(I, 1);
for i=1:N
    for j=1:N
      T(i,j) = (I(i,j)+0.5) / (N*N);
    end
end
```

Using the matrix T, the algorithm then iterates over all pixels in the image, gets what would be the threshold value and if that value is less than the pixel value, the pixel value is mapped to 255 and otherwise it is mapped to 0.

Note: Because the matrix T is 2×2 and the image is greater than 2×2, T is effectively tiled by using a simple modulo operation on the image indices given by:

$$(i_T, j_T) = (i_F \mod n, j_F \mod n)$$

```
thresh = T(mod(j,N)+1, mod(i,N)+1) * 255;
```

The value from T is multiplied by 255 since the formula for T is a normalized value.

4×4 Dither:

The 4×4 dither is is similar to the 2×2 level dither in that there is a defined index matrix computed by the following formula:

$$I_{2N} = \begin{bmatrix} 4 * I_N + 1 & 4 * I_N + 2 \\ 4 * I_N + 3 & 4 * I_N \end{bmatrix}$$

For $N = 4$, the resulting matrix is:

$$I_4 = \begin{bmatrix} 5 & 9 & 6 & 10 \\ 13 & 1 & 14 & 2 \\ 7 & 11 & 4 & 8 \\ 15 & 3 & 12 & 0 \end{bmatrix}$$

The conversion to a threshold table is is the same as in the two level dither.

2 Gray level Thresholding:

The two gray level thresholding simply takes the threshold value t from T and applies a basic thresholding operation:

$$G(i,j) = \begin{cases} 255 & \text{if } F(i,j) > t \\ 0 & \text{otherwise} \end{cases}$$

4 Gray Level Thresholding:

The 4 level dither maps the image values across 4 gray levels, the range of each determined by the original threshold value according to the function:

$$G(i,j) = \begin{cases} 255 & \text{if } F(i,j) > t_h \\ 170 & \text{if } t \leq F(i,j) < t_h \\ 85 & \text{if } t_l \leq F(i,j) < t \\ 0 & \text{otherwise} \end{cases}$$
$$t_h = t + \frac{255 - t}{2}$$
$$t_l = \frac{t}{2}$$

This function takes the threshold to be the middle threshold value, then picks a low threshold t_l and a high threshold t_h such that both are evenly split between the threshold and 0, and the threshold and 255 respectively. Given each of the four ranges within $0 - 255$ has id $i = 0 : 4$, the value for each section is calculated by: $v = i * (255/3)$.

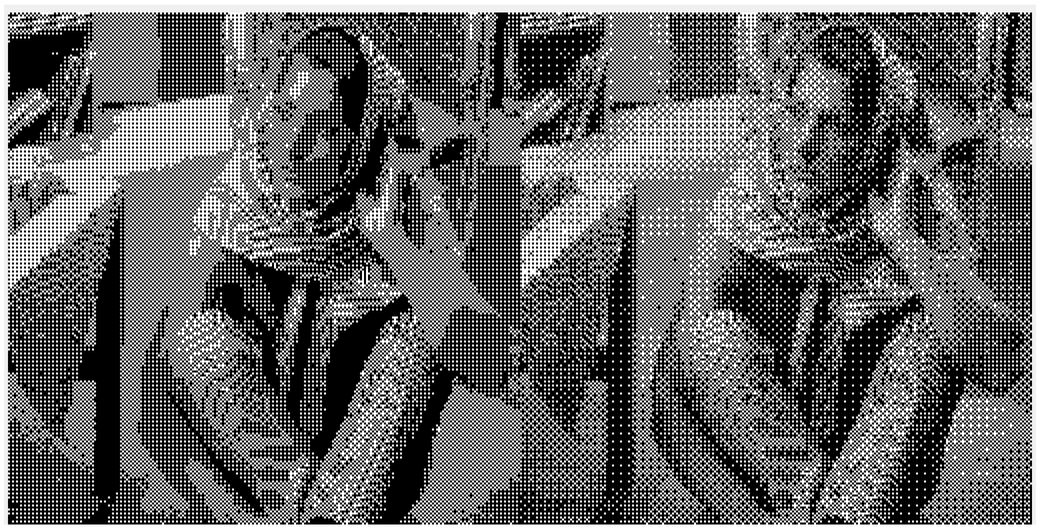
The goal of this procedure is to use the threshold to calculate the other thresholds so as to retain the merit of the matrix dithering, while still evenly distributing the values across the 4 gray level spectrum. This is achieved by evenly spacing the upper and lower thresholds evenly between the original and 0 and 255.

Result

barbara.raw

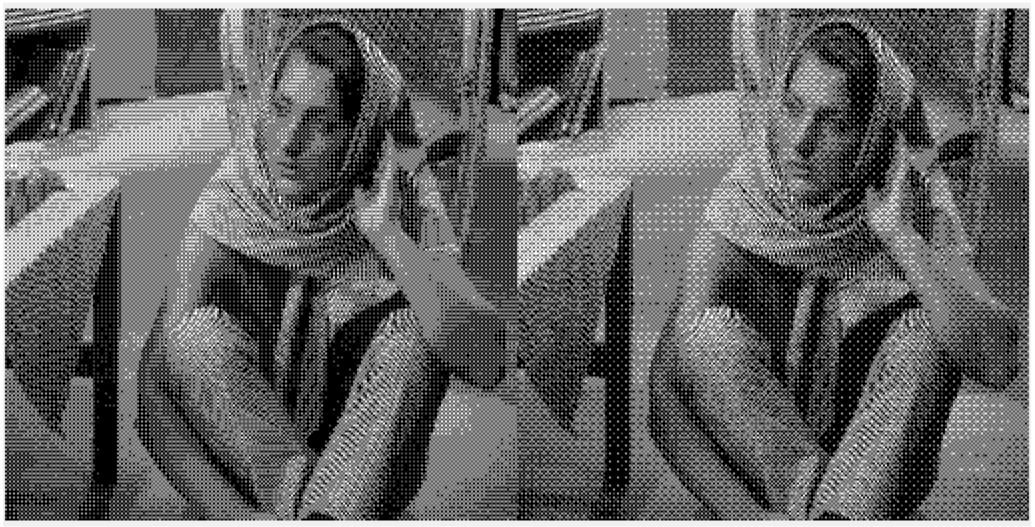
2 Gray Levels

2×2 4×4



4 Gray Levels

2×2 4×4



Discussion

The results of this method are quite good. In the two gray level version, the algorithm does a really good job creating apparent gray values using tight, alternating black and white pixels. The 2×2 version, as compared to the 4×4 provides less detail. Although the 4×4 appears to have more large fluctuations and clear patterns, the detail is much more pronounced, likely due to the larger spatial awareness of the algorithm.

The four gray level has all of the merits of the two gray level, as well as the same differences between the 2×2 and the 4×4 matrix versions. It does, as expected, provide more detail and variation in gray value, and appears to distribute it quite well. Some of the best examples of this improved variation are in the carpet in the background as well as the face of the individual in the photo. it does appear that the image is darker overall as compared to the 2 gray level images, however when compared to the original it is of a similar overall darkness.

An apparent downside to this algorithm however is that there are pretty strong distinctions between different gray levels. The image does not blend together too well over most of the image.

3-d. Floyd Steinberg's Error Diffusion w/ Serpentine Scaling

Motivation

The motivation for floyd steinberg dithering is the same as *3-a: Fixed Threshold Dithering*. Floyd steinberg dither, however, seeks to provide a better looking image than other methods, with more apparent gray values and variation in gradient.

Approach

The Floyd Steinberg Error Diffuson algorithm works in essentially the same way as the fixed thresholding dithering with $\text{threshold} = 127.5$, except that it does not simply throw away the error that occurs when mapping a value to another. Instead, it takes that error and pushes it off to the pixels that are yet to be processed, according to the matrices:

Right to left: $\begin{bmatrix} 0 & 0 & 0 \\ 0 & X & \frac{7}{16} \\ \frac{3}{16} & \frac{5}{16} & \frac{1}{16} \end{bmatrix}$

Left to right: $\begin{bmatrix} 0 & 0 & 0 \\ \frac{7}{16} & X & 0 \\ \frac{1}{16} & \frac{5}{16} & \frac{3}{16} \end{bmatrix}$

In practice, this means that the values in the original image surrounding the pixel being processed are increased or decreased by a percentage of the error, that percentage being given by the matrices above, depending on the direction of processing.

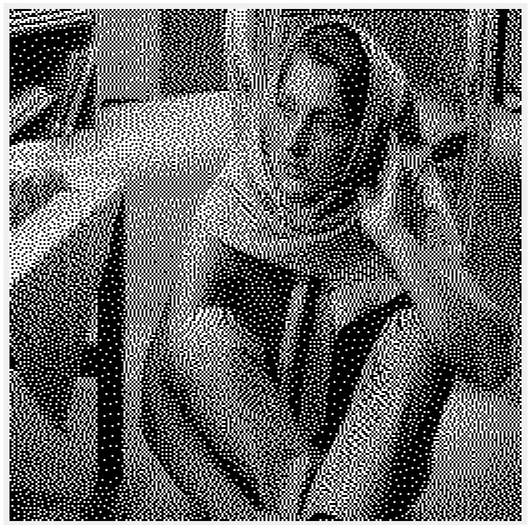
```
% Right to left
img(i,j+1)   = img(i,j+1)   + err * (7.0/16.0);
img(i+1,j-1) = img(i+1,j-1) + err * (3.0/16.0);
img(i+1,j)   = img(i+1,j)   + err * (5.0/16.0);
img(i+1,j+1) = img(i+1,j+1) + err * (1.0/16.0);
```

Where err is the difference between the original image value and the value that the original was mapped to.

This operation is repeated for every pixel in the image, and is also conducted using the serpentine scaling method, wherein the processing is done left to right for one row, then right to left for the next row, and so on. This is simple in implementation.

Results

barbara.raw



Discussion

The result of the floyd steinberg dithering is really good in many respects, in that it maintains quite a high level of detail while also producing a large variation of gray values but doing so with generally smooth transitions. The outcome makes sense in that it accounts for the information lost in the thresholding operation by pushing it off to other pixels. This spatial distribution of error then means that, when looked at from a distance, the pixels appear to be quite similar to the original. The definite drawback of this method, though, is its grain-iness, as well as speckled dark zones that are not handled as well as they were in the matrix dither. Overall the effect is pretty remarkable and is a significant improvement upon the first fixed threshold dithering method.