Jonah Fullen

Scientific Computing

03-14-2022

Assignment 3: Harris Corners

Finding "Cornerness"

The Harris Corners algorithm is a way of analyzing an image and distinguishing key points within that image. These key points – or corners – are calculated by comparing the level of color contrast between one pixel and it's neighboring pixels. To do this, you import the image in a grayscale form as well as a copy of that image. You then create empty images of equal size to the original. The use of these empty images is to store different calculated values for each pixel. These empty images are Ix, Iy, IxIx, IyIy, and IxIy and their formulas are as follows for each given pixel:

Ix = difference between the pixel to the left and right of the given pixel

Iy = difference between the pixel above and below the given pixel

IxIx = Ix squared IyIy = Iy squared IxIy = Ix times Iy

Using these values, you then go through the image and determine a "cornerness" value for each pixel. This is done by first creating a new empty image to store the values in and then analyzing the pixels in the surrounding "neighborhood" of the focus pixel and calculating the following:

IxxSum = summation of IxIx values in the neighborhood

IyySum = summation of IyIy values in the neighborhood

IxySum = summation of IxIy values in the neighborhood

These values are then used to create the following matrix for the individual pixel:

$$M = \begin{bmatrix} IxxSum & IxySum \\ IxySum & IyySum \end{bmatrix}$$

From this matrix the cornerness value (c-value) for the pixel is calculated using the following formula:

c-value = $\det(M) - k(\text{trace}(M))^2$; where k = 0.05

This c-value is then stored into the new blank image at the location of the appropriate pixel.

Determining Corners

Once the c-value has been calculated for each pixel using the method above, there are then two different methods for determining which pixels should be marked as corners. For each of these methods corners are determined by relatively large c-values.

Method 1

For the first corner finding method, you look at all c-values in the image and determine the max. From there you again look through all c-values and mark any pixel with a c-value within a certain percentage of the max to be a corner.

> Ex: If Cmax = 10000 and you want corners to be the top 10%, you select any pixels with a c-value >= 9000.

A flaw with this method of selecting corners is that images with large uniform areas will often have no corners marked in large portions of the image.

Method 2

For the second corner finding method, rather than looking for the max c-value in the entire image, you split the image into separate quadrants (I divided my images into a 10x10 grid) and then find the local max c-value within that section. Once the local max has been found, you then continue like Method 1 and select any pixels in that quadrant within five percent (5%) of the local max.

> Ex: If you want corner values to be within 10% of the local max. If Q1max = 10000, then any pixels in Q1 with a c-value >= 9000 are selected. Then, if Q2max = 1000, then any pixels in Q2 with a c-value >= 900 are selected.

A flaw with this method of selecting corners is that there is a possibility for quadrants that are highly uniform within an image can select a large percentage of their pixels to be corners due to the lack in variation in each pixel's c-value.

Marking Corners

Once the corners have been selected using either method above, the pixels marked as corners are given a high value in a new B&W image and the pixels that are not corners are given a zero value. Using this image, you iterate through the pixels. Once a pixel value that is not zero is found the corresponding pixel in the original image is changed to red. In large images, these pixels are not easily visible, so they need to be marked with circles of various sizes depending on the size of the original image.

Images/Figures

Figure 1: Small Checkerboard

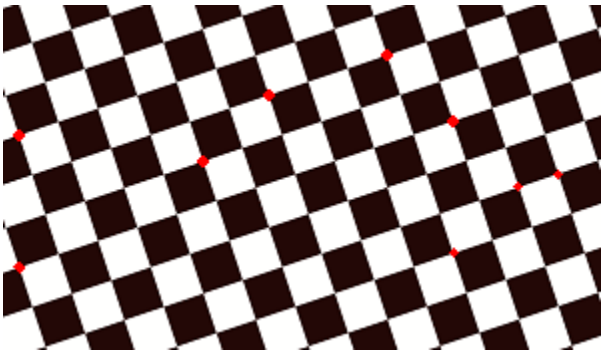

Figure 1A: Small checkerboard w/ corners determined by Method 1 in the top 10%.

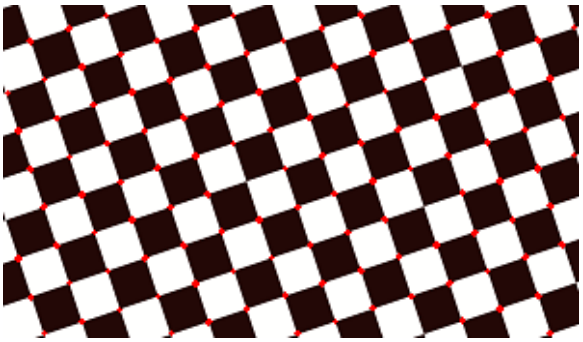

Figure 1B: Small checkerboard w/ corners determined by Method 1 in the top 25%.

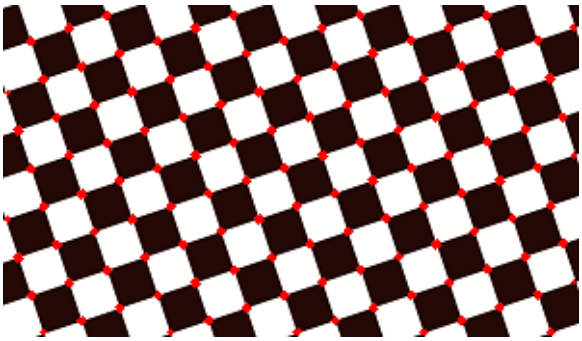

Figure 1C: Small checkerboard w/ corners determined by Method 1 in the top 50%.

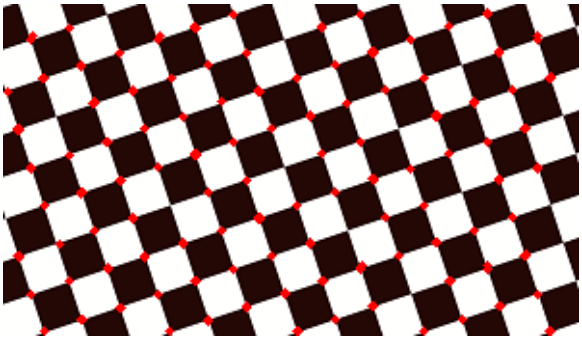

Figure 1D: Small checkerboard w/ corners determined by Method 2.

Figure 2: Checkerboard

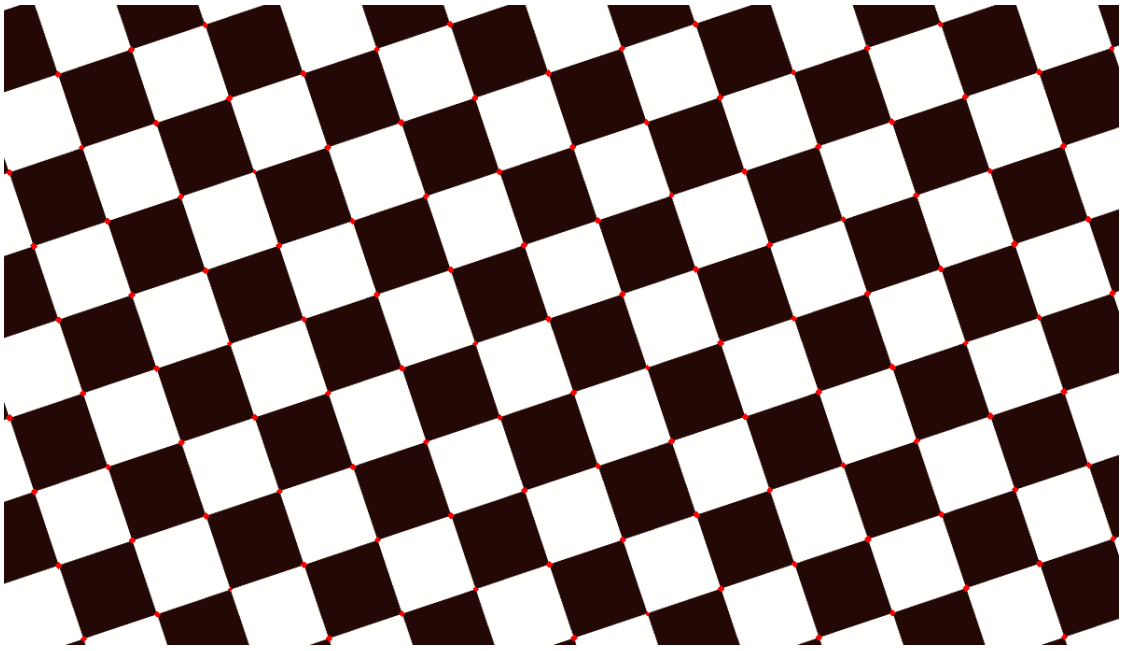

Figure 2A: Checkerboard w/ corners determined by Method 1 in the top 50%.

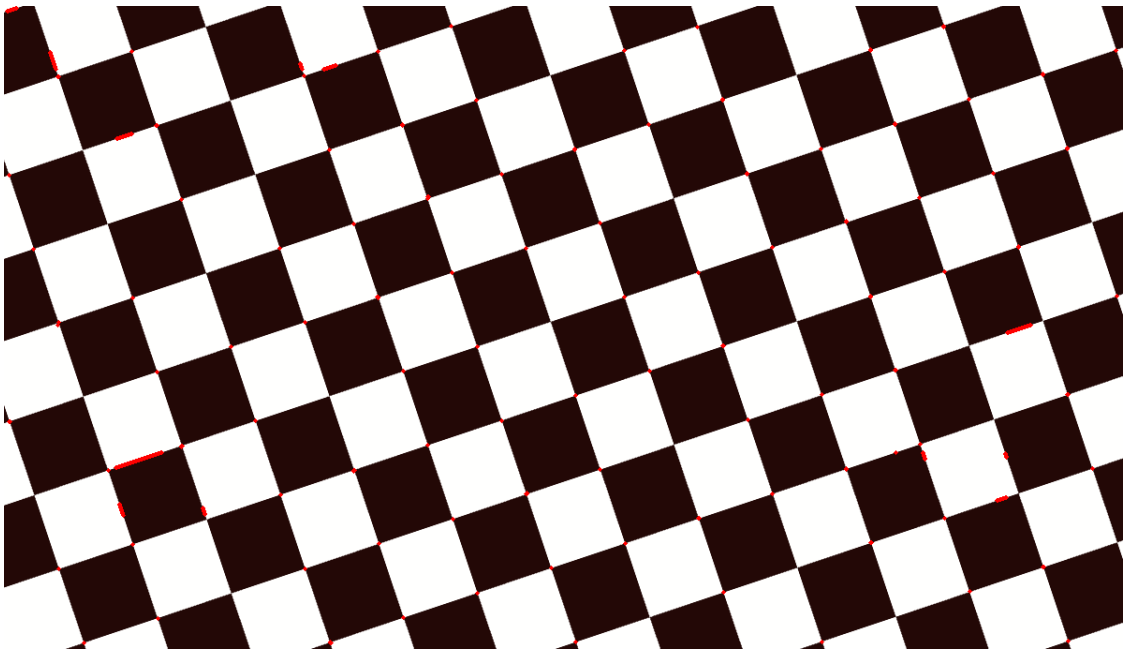

Figure 2B: Checkerboard w/ corners determined by Method 2.

Figure 3: Architecture

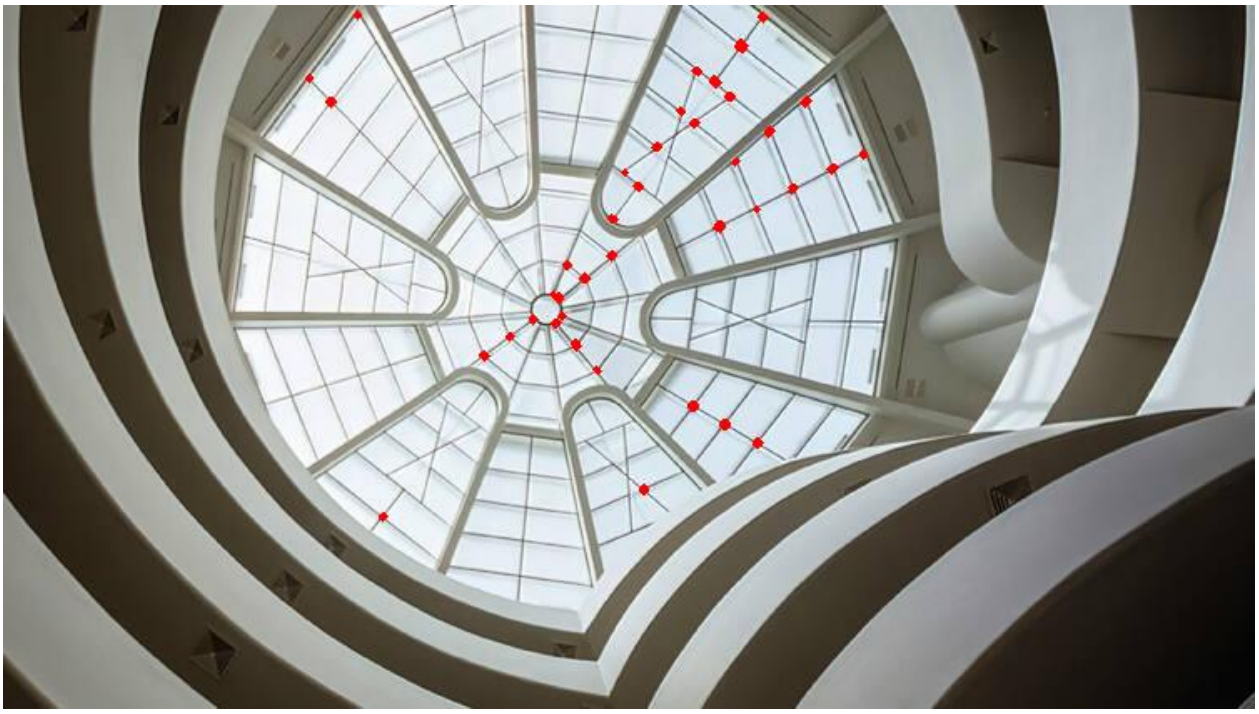

Figure 3A: Architecture w/ corners determined by Method 1 in the top 75%.

Figure 3B: Architecture w/ corners determined by Method 2.

Figure 4: Still Life

Figure 4A: Still life w/ corners determined by Method 1 in the top 85%.

Figure 4B: Still life w/ corners determined by Method 2.

Figure 5: Painting

Figure 5A: Painting w/ corners determined by Method 1 in the top 75%.

Figure 5B: Painting w/ corners determined by Method 2.

Figure 6: City

Figure 6A: City w/ corners determined by Method 1 in the top 50%.

Figure 6B: City w/ corners determined by Method 2.

Figure 7: Portrait

Figure 7A: Portrait w/ corners determined by Method 1 in the top 85%.

Figure 7C: Portrait w/ corners determined by Method 2.

Figure 7B: Cropped portrait w/ corners determined by Method 1 in the top 75%.

Figure 7D: Cropped portrait w/ corners determined by Method 2.

Analysis of Figures

Figure 1: Small Checkerboard

The small checkerboard image is the first image I worked with in the program in order to get the code functioning properly. I began with Method 1 of determining corners and found that the range of c-values that select corners needed to be increased in order for all corners of the checkerboard to be selected (compare **Figures 1A, 1B,** and **1C**). Only selecting pixels with a c-value within 10% of the max left the majority of corners unselected. I had to go all the way to selecting c-values within 50% of the max for all corners to be selected. Using Method 2 – which divided the image into a 10x10 grid and uses local max values and a set comparison to c-values within 5% of the local max – the majority of the checkerboard's corners are selected, but not all of them are (**Figure 1D**).

Figure 2: Checkerboard

The checkerboard is similar to the small checkerboard image except that it is much larger and so the program has to operate over many more pixels to do the same operations that were done on the small checkerboard image. Like the small checkerboard image, it took selecting c-values within 50% of the max to select every corner of the checkerboard (**Figure 2A**). When using Method 2 to select corners, very few of the checkerboard's corners were selected and it tended to select values that were consecutively along different edges in seemingly random locations throughout the image (**Figure 2B**).

Figure 3: Architecture

The architecture image was the first image I ran the program on after I got it working on the checkerboard images. I selected this image because there a lot of distinguished angles and corners in it and I wanted to see how the program would do at identifying these. The first time I ran it I used Method 1 for determining corners and had it set to select values within 10% of the max. This led to there only being three or four corners selected in the whole image. After messing with this percentage value a bit I landed on selecting c-values within 75% of the max (**Figure 3A**). This, in my opinion, did a fairly good job of selecting sharp corners in the image. Most of the corners are within the center of the image, but that is where the majority of sharp corners are as the surrounding regions of the image consist of more smooth edges and softer transitions. In comparison, using Method 2 on this image spread out the corner selection so that there are corners in every region of the image (**Figure 3B**). This led to corners being selected in locations that are simply edges rather than corners or in places that don't have any corners at all.

Figure 4: Still Life

The still life image was selected due to the high contrast between the background and table as well as the vast regions without any noticeable change in color. This interested me after seeing how the program handled the more uniform regions in the architecture image. When using Method 1 for selecting corners the selection value had to be increased all the way to within 85% of the max. Even after doing this, because of the uniformity and smoothness of the image, only 8 corners were located in the entire image (**Figure 4A**). In comparison, when using Method 2, the program had difficulty in the regions of the image where there was little to no contrast. In some instances where a quadrant was extremely uniform seemingly every pixel in the section was selected as a corner (**Figure 4B**).

Figure 5: Painting

The painting image was selected because it has very little drastic contrast throughout the image and also has very few sharp corners as most of the transitions are softer. I was curious to see how the program would identify corners in this type of an image after seeing what happened in the low contrast areas in the still life image. Initially, using a small percentage range in Method 1 led to very few corners being selected. After increasing this range, the program selected corners within the more detailed and darker regions of the image (**Figure 5A**). Using Method 2 of determining corners dispersed the corner selections so that there were now corners in every region of the image selected from key features in that given quadrant (**Figure 5B**).

Figure 6: City

The city image was selected because I began to notice that the program would tend to select more corners in darker regions than lighter regions. This city image puts that theory to the test, and it appears that my hypothesis was confirmed. When using Method 1 for finding corners there were hardly any selected in the daytime side of the image, but the nighttime side had plenty of corners selected (**Figure 6A**). Using Method 2 evened out the distribution of corners – as it is designed to do – but these corners rarely matched from one side of the image to the other (**Figure 6B**).

Figure 7: Portrait

The portrait image was the last image I looked at to attempt to use what I had learned from the trial and error of the other images. This image was selected because the background is very uniform, and the subject is the only area of the image that has any details. Because of this, I was hoping that the program would mainly focus on the subject of the portrait rather than the background. When I found corners on the original image using Method 1, I had to increase the range all the way to 85% of the max. The program still wouldn't select any corners on the subjects face but focused on the details of her necklace and along the edge of her shirt (**Figure 7A**). Using Method 2 on the original image caused the same issues that seen previously in

quadrants that were largely uniform and dark. Within these regions the program would select nearly every pixel as a corner. When a quadrant in Method 2 was on the subjects face, however, the program would appropriately select corners based on the subject's key features such as her eyes, nose, mouth, chin, and freckles (**Figure 7C**). From this point I started to wonder how I could improve these corner selections and thought about Apple's FaceID. In the FaceID setup it requires the user to take pictures close to their face so that there is little to no background to worry about. Using this concept, I decided to crop the image so that it focused solely on the subject's face and took out the aspects of the background and clothing. Doing this with Method 1 greatly improved the results. The program now selected corners on the subject's key features, mainly focusing on her eyes and nose with some being selected form the waves in her hair (**Figure 7B**). Using Method 2 on the cropped image saw an even greater improvement. The program still selected corners in featureless regions, but it no longer selected entire dark quadrants. Along with this, it also selected corners on even more of the subject's key features in quadrants that overlapped onto both the subject and the background (**Figure 7D**). Cropping the image greatly assisted the program in distinguishing key features in the subjects face rather than it getting "confused" by the background or clothing.

Conclusion

After running the program on all of these images there are a few general observations that can be made about each of the two methods for determining cornerness. When using Method 1, corners tend to only be selected in darker regions of the image where there is a sudden change in contrast. Regions of uniformity are generally ignored, and no corners are selected there. In comparison, lighter areas that also have a contrast change are less likely to be selected as corners (see **Figure 6**). When using Method 2, corners are always more evenly distributed around the image, but regions that consist of more uniformity struggle form over-selection because the c-values in the quadrant are all very close to each other. If I only had to rely on one cornerness method, I would simply use Method 1 and play with the percentage value to find the appropriate number of corners. I believe that an ideal solution for determining corners would be to create a function that combines the two methods. It would divide the image into different quadrants and analyze the overall c-value differentiation in that region. If that differentiation was low, it would be very selective of values selected as corners. Alternatively, as the differentiation increased, it would widen the range for corner selection. This, in theory, would remedy the over-selection of corners in uniform regions, but would allow for corners to be selected easier in key regions of the image.

Bonus: Gradient

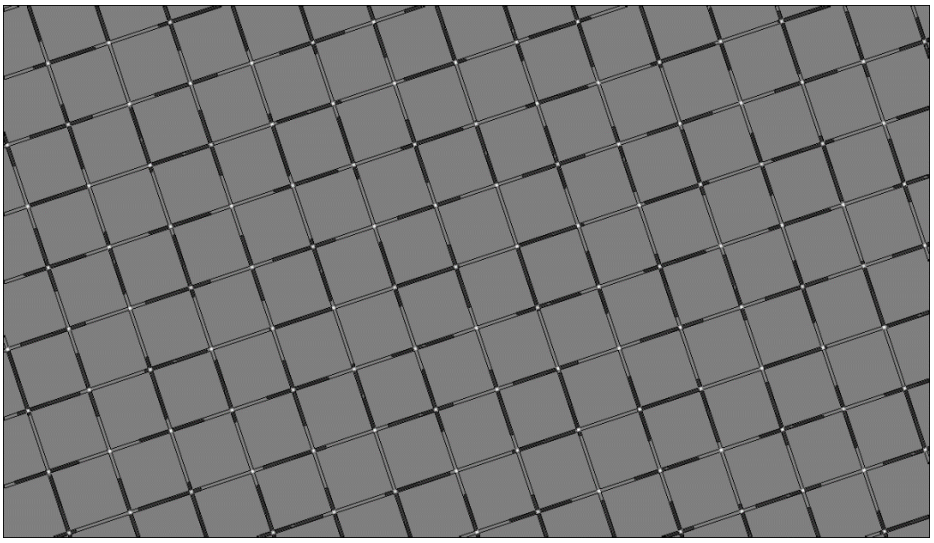

Figure 8A: Checkerboard image displayed using a gradient generated by c-values.

Figure 8B: Portrait image displayed using a gradient generated by c-values.

Bonus Analysis

The challenge of this portion of the assignment was to take the image containing the c-values for each pixel and map that to a new grayscale image that produced a gradient based on the range of c-values. The max color value for a pixel in grayscale is 255. Because of this, I mapped any negative c-values to color in range 0 – 127 while mapping positive c-values to the second half of range values. This creates an output where pixels with high c-values appear lighter and pixels with low c-values appear darker.