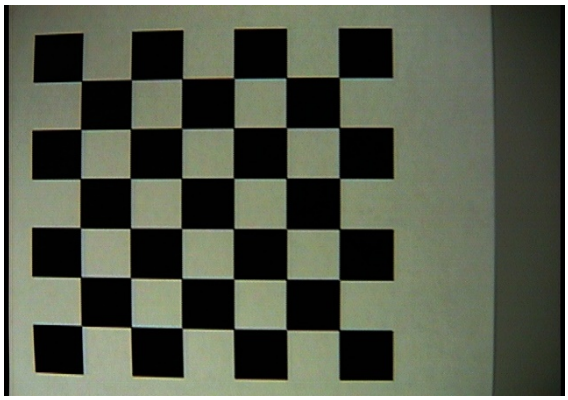
The Problem

The goal of this assignment was to detect features of an image – specifically, edge features. In order to accurately evaluate the results of this assignment later in this report, the two images directly below are the initial images in the problem set:

Building



Checkerboard

The Process

Gaussian Filtering

In order to minimize the probability of detecting a noise pixel as a corner feature of the image, the first step in the assignment is to perform Gaussian smoothing on the input image. The MATLAB code for this portion of the assignment was copied from Assignment 1. If you would like to revisit my process for developing the Gaussian filtering code, please refer to Assignment 1 for more information (I'll include Assignment 1 in the zip file).

CORNERS Algorithm

The CORNERS algorithm begins in the same fashion as Assignment 2. The first task of this algorithm is to find the image gradients in both the X and Y directions. To take the gradient in the X

direction, you simply convolve the image with the vector [1 0 -1]. Convolving the image with this vector looks at the difference between the pixel to the left and the pixel to the right if your immediate pixel value in the image. This way we can see how the pixel values shift as you progress throughout the image. Similarly, to get the gradient in the Y direction, you convolve the image with the transpose of [-1 0 1]. We take the transpose of the vector because we want the vector in the correct dimensions in order to properly perform convolution in the Y direction.

Once the image gradients in both the X and Y directions have been computed, the next step is to find each 2*N+1 neighborhood's minimum Eigen value (where N is a positive integer). The first step to computing each neighborhood's minimum Eigen value is to find the neighborhoods themselves. I decided to make use of MATLAB's im2col function, which creates a column in a matrix for each neighborhood and the number of rows corresponds to the number of elements in each neighborhood. Once I found each 2*N+1 neighborhood for both the X and Y gradient images, I could easily iterate over each column and reconstruct the neighborhood. The reason I am deconstructing the original image just to reconstruct the neighborhoods is because I need to compute the Eigen values for each neighborhood, which cannot be done with a vector. But before computing the Eigen values, there's still one more step – computing the 2x2 matrix used to compute the Eigen values. Computing this 2x2 matrix (let's name it C), is relatively simple. All we have to do is sum the square of each pixel in the X gradient neighborhood (let this be called: Xsq), sum the pixel-wise product of the X and Y gradient neighborhoods (let this be called: combo), and sum the square of each pixel in the Y gradient neighborhood (let this be called: Ysq). To form C, we simply create the new 2x2 matrix: [Xsq, combo; combo, Ysq]. Now that we have matrix, we pass it as an argument to MATLAB's Eigen function and use the min function to retrieve the smallest Eigen value for C. Since the Eigen values can be very large, I ran the process of finding the Eigen values again, but this time used the values we just computed to normalize the Eigen values I am currently computing. Now that I have the normalized Eigen values, all that's left to do for this step is to then check if the minimum normalized Eigen value is larger than the threshold we chose. Since the values are normalized,

$$C = \begin{bmatrix} \sum E_x^2 & \sum E_x E_y \\ \sum E_x E_y & \sum E_y^2 \end{bmatrix}$$

we can choose a number between 0 and 1 to be the threshold, rather than 0 and the maximum integer number. If the Eigen values is greater than the threshold, we store this Eigen value along with the X and Y coordinates of the pixel into a matrix.

Once we have the matrix with the minimum Eigen values and their corresponding X and Y coordinates, we can just use MATLAB's sortrows function to sort the rows of the matrix in either ascending or descending order. The algorithm specifies that we sort the matrix in descending order and traverse it top-down.

The final step in the CORNERS algorithm is to then traverse the output, removing all entries appearing further down the list that are in the same neighborhood as an entry further up the list. We do this by traversing the matrix, and if there is a pixel in the +/- N range (since our neighborhood is 2*N+1) of a given Eigen value's X and Y coordinate, we simply replace that entry with a vector of negative ones in order to flag the entry for removal. We can then continue traversing, and if we come across any flagged entries, we simply skip them. Now that we have all the flagged entries, we can then traverse over this output and create a new matrix of X and Y coordinates, saving only the non-flagged entries. Therefore, the output of this final operation is a list of non-overlapping corner feature points.

Although this is not part of the CORNERS algorithm, we can modify these corner feature pixels in the original image to point out where the corners are. There is a nice way of doing this using insertMarker, where you can insert shapes at specified locations. However, I did not have access to the Computer Vision Tool Box, so I could not use this function. Instead, I placed a white dot at each corner pixel.

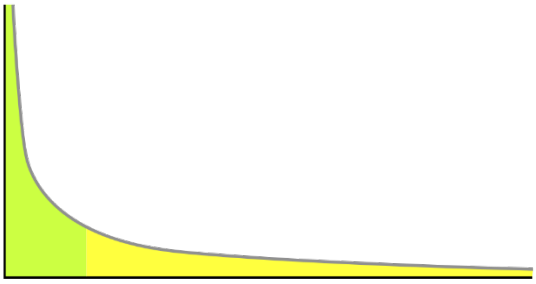
The Results

Important Note: Since I couldn't use insertMarker and only recolored the pixels that denote the corners, you can't see the corner pixels when the image is imported into Microsoft Word. Instead of importing them into this document, please look in the zip file for the images. Each image is labeled with the function call and parameters passed.

Building1

Finding the balance between the right amount of smoothing, the right neighborhood size, and the right threshold value is tricky. This image is a little bit trickier than the checkerboard image, since there is much more variation in the results from this image than the results from the checkerboard image. I presume the reason there is more variation than the checkerboard image is because there is much more activity in the building image and the corners are not as well defined. Please look in the zip file to see the output images.

After analyzing the Eigen value histogram, I found that the values took on a long-tail distribution (image to the right). This told me that if I set my threshold too high, there would be no corners detected. Therefore, I chose smaller threshold for the Eigen values. Since thresholding was very difficult with non-normalized values, I normalized the Eigen values and found that anything above .1 would either remove all corners or not have very many remaining. The threshold seemed to be more correlated with the smoothing since the smoothing directly effects the image gradients, and thus effects the Eigen values. When I had larger sigmas (for smoothing) the output wasn't as good as the smaller sigmas. Therefore, I decided to stick to smaller sigma values (I chose sigma to be 3 in most cases). Lastly, I found that having a greater neighborhood size seemed to work better for this image than a smaller neighborhood size. Therefore, I chose the neighborhood to be larger (10 seemed to work the best). This may be because more of the surrounding is taken into consideration, hence, a lower chance of a false positive occurring.



https://en.wikipedia.org/wiki/Long_tail#/media/File:Long_tail.svg

Checkerboard

As I stated above, the checkerboard image was easier to work with since there is not a lot of activity in this image and the corners on the checkerboard are relatively distinct. Because of these features, the outputs from this image were much more consistent – when varying parameters – across tests than the building image.

My observations in the building image about the sigma, threshold, and neighborhood values are all still true in the checkerboard image. However, I found that I had much more freedom with the variations in the numbers and I would still get a relatively consistent output. This is most likely because the checkerboard image was nicer and cleaner (reasons stated above) than the building image. For this image I generally found that the larger neighborhood values (~10) and that larger thresholds (below, but approaching .1) performed best. I used both 3 and 5 as sigma values and found that the difference between the two wasn't too extreme. The outputs seemed a little cleaner with sigma 3, but when the neighborhood value was raised, the greater sigma value performed just as well. To see these examples, please look in the zip file to see the output images.