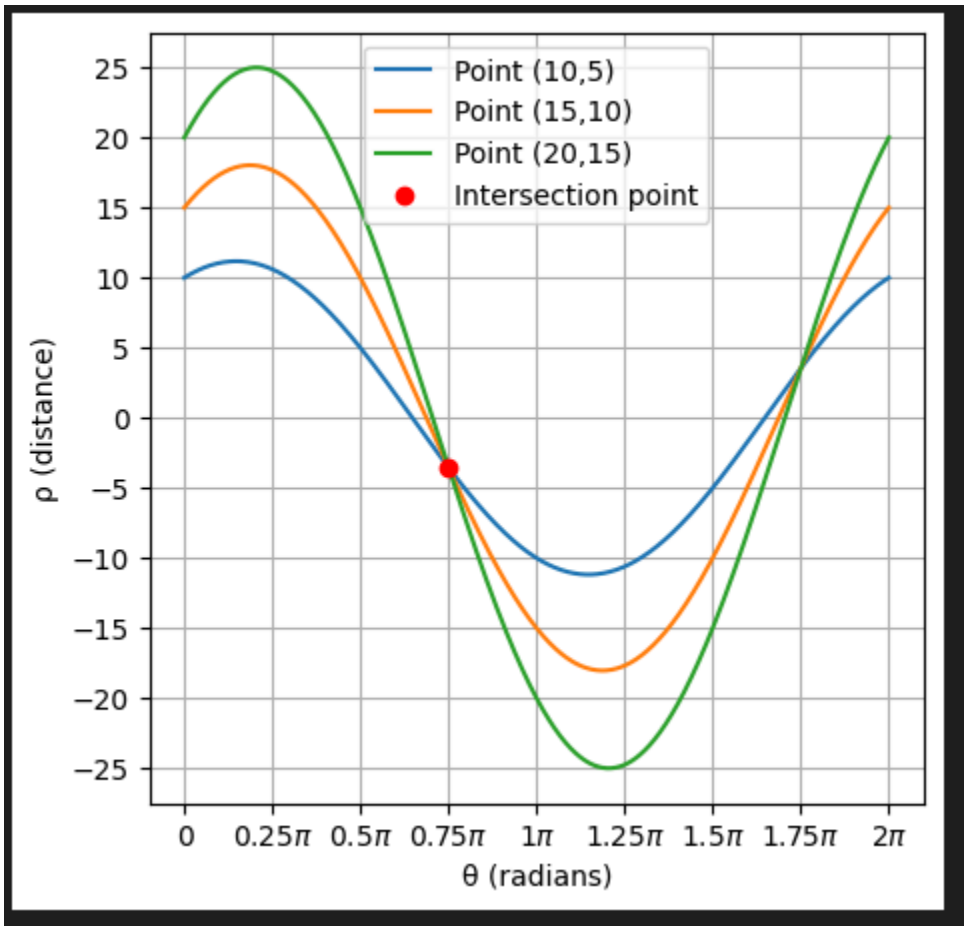
Q2.1
Image Plot:



Q2.2

(m,c) is (1,-5)
Normal form:
normal form equation:
rho = xcos(theta) + ysin(theta)
First, we isolate the y term:
ysin(theta) = -xcos(theta) + rho

Next, we divide both sides by sin(theta):
y = (-cos(theta) / sin(theta)) * x + (rho / sin(theta))

Using the trigonometric identity cot(theta) = cos(theta) / sin(theta), the equation simplifies to:
y = (-cot(theta)) * x + (rho / sin(theta))

From this, the slope and intercept are defined as:
Slope (m): m = -cot(theta)

Intercept (c): c = rho / sin(theta)

Using the intersection point identified in the Hough space plot where theta = 3pi / 4 and rho = -5 / sqrt(2):

m: m = -cot(3pi / 4) = -(-1) = 1

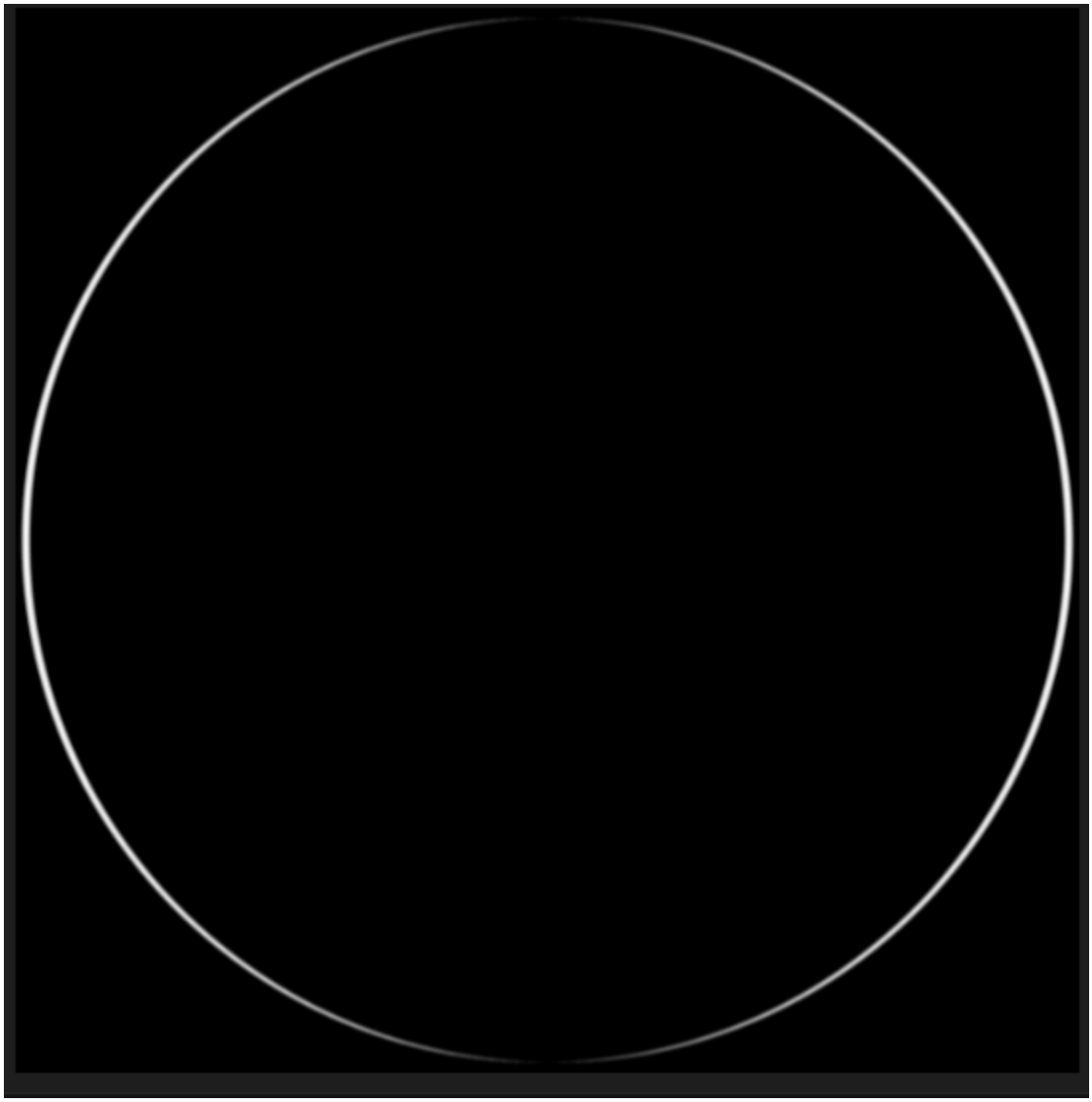
c: c = (-5 / sqrt(2)) / sin(3pi / 4) = (-5 / sqrt(2)) / (sqrt(2) / 2) = -5

Q3.1
My myImageFilter function uses correlation to apply filters to the images. To handle the edges where the filter hangs off the side, I used np.pad with 'edge' mode. This basically just copies the pixel at the very edge so you don't get any weird black borders. To make the code run fast enough, I vectorized it by looping over the filter size instead of every single pixel. This lets the code multiply a filter weight by a shifted version of the whole image at once. It’s also set up to work with different filter sizes automatically by calculating how much padding is needed based on the filter's dimensions.

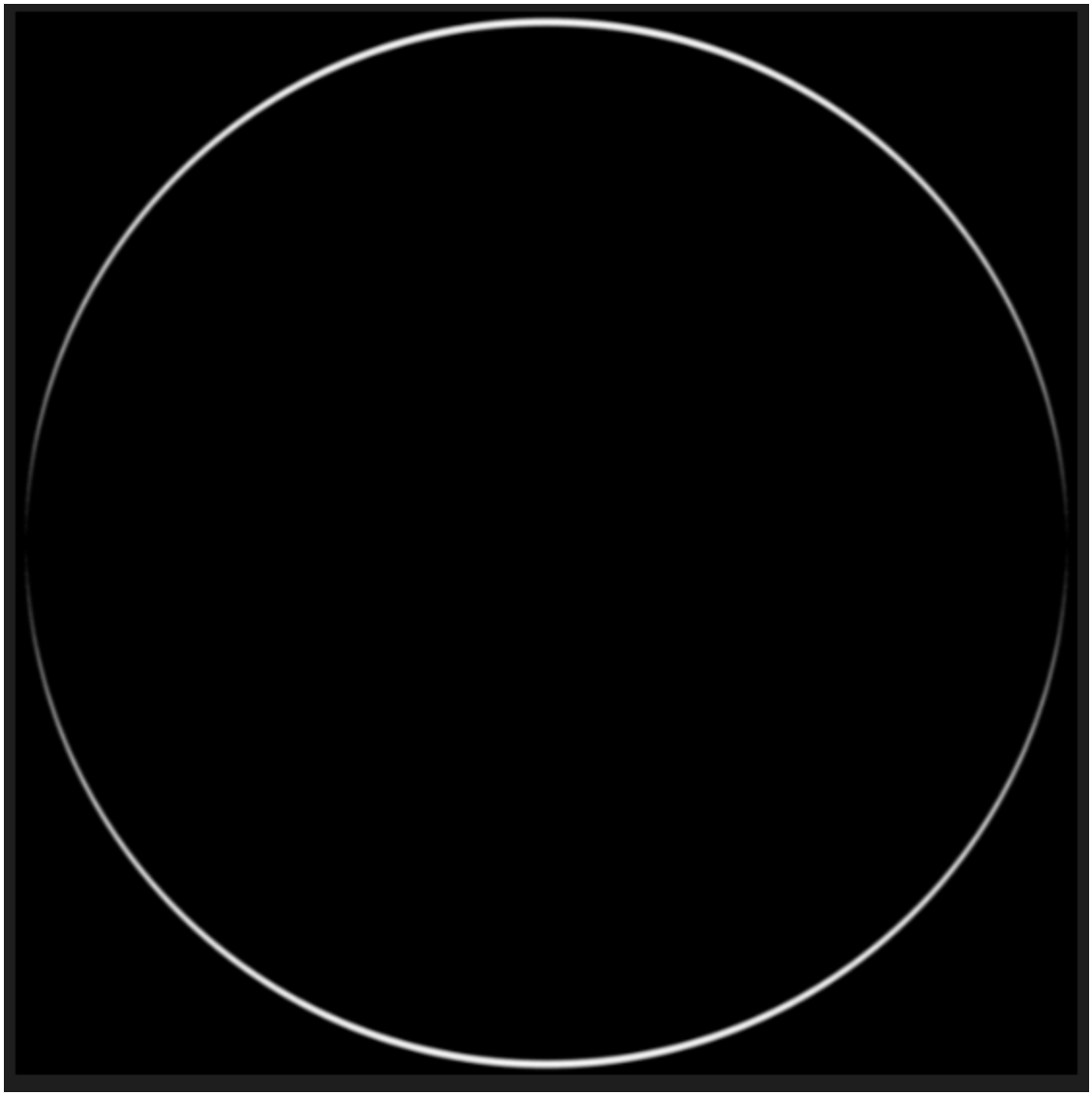
Q3.2
For the edge detection part, I created Sobel X and Y filters by combining a Gaussian kernel with a derivative filter to get the horizontal and vertical gradients. To get sharp results, I implemented Non-Maximum Suppression (NMS), which checks the gradient direction at each pixel and keeps only the strongest ones, turning thick edges into thin, 1-pixel wide lines.

When testing different filter sizes, I noticed that small filters were way too noisy, while large filters made the grid lines too blurry to detect. I decided to stick with the formula hsize = 2 * ceil(3 * sigma) + 1. This size was the best because it smoothed out the noise while keeping the edges sharp enough for the Hough Transform to work accurately.

Sobel X:

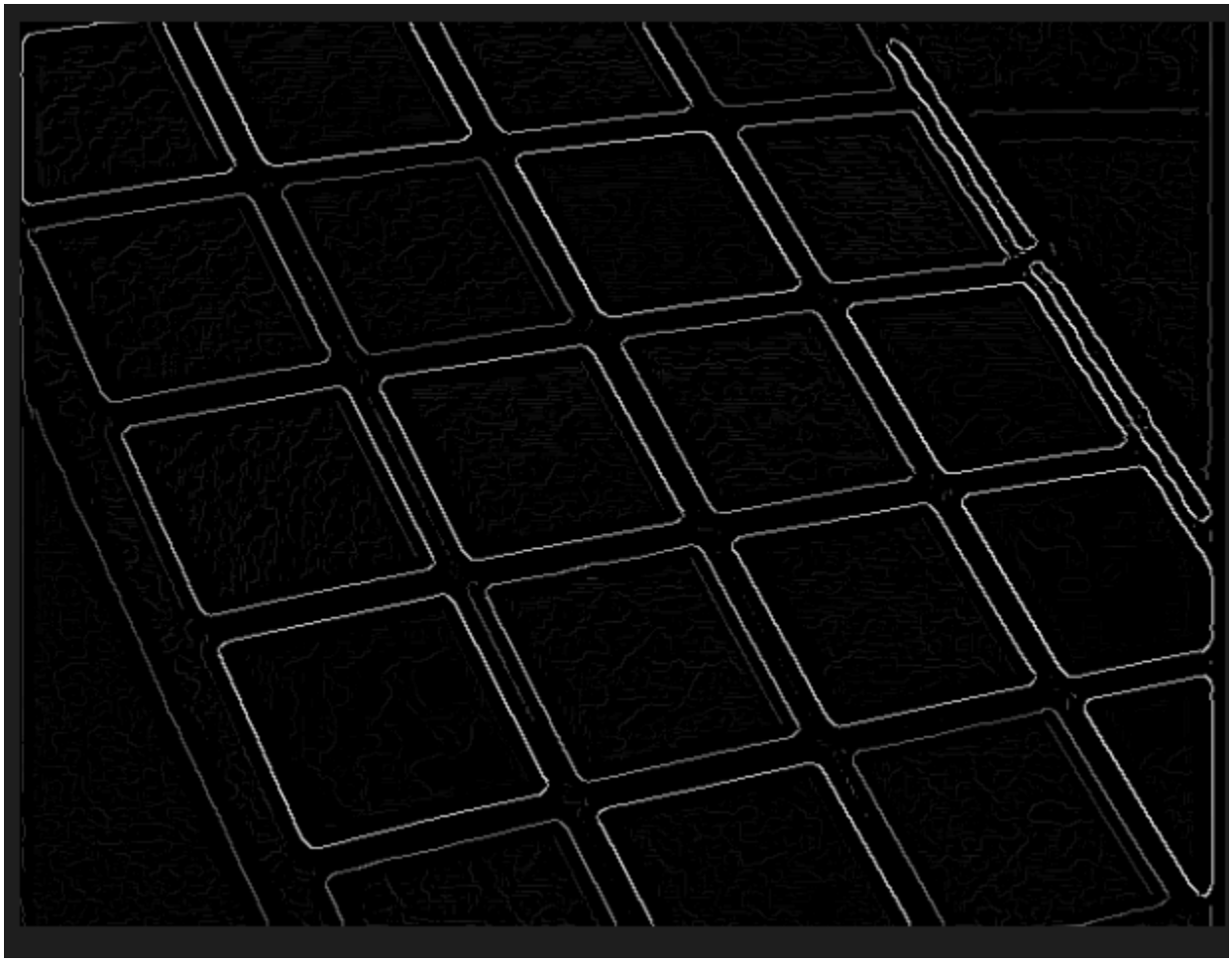


Sobel Y:

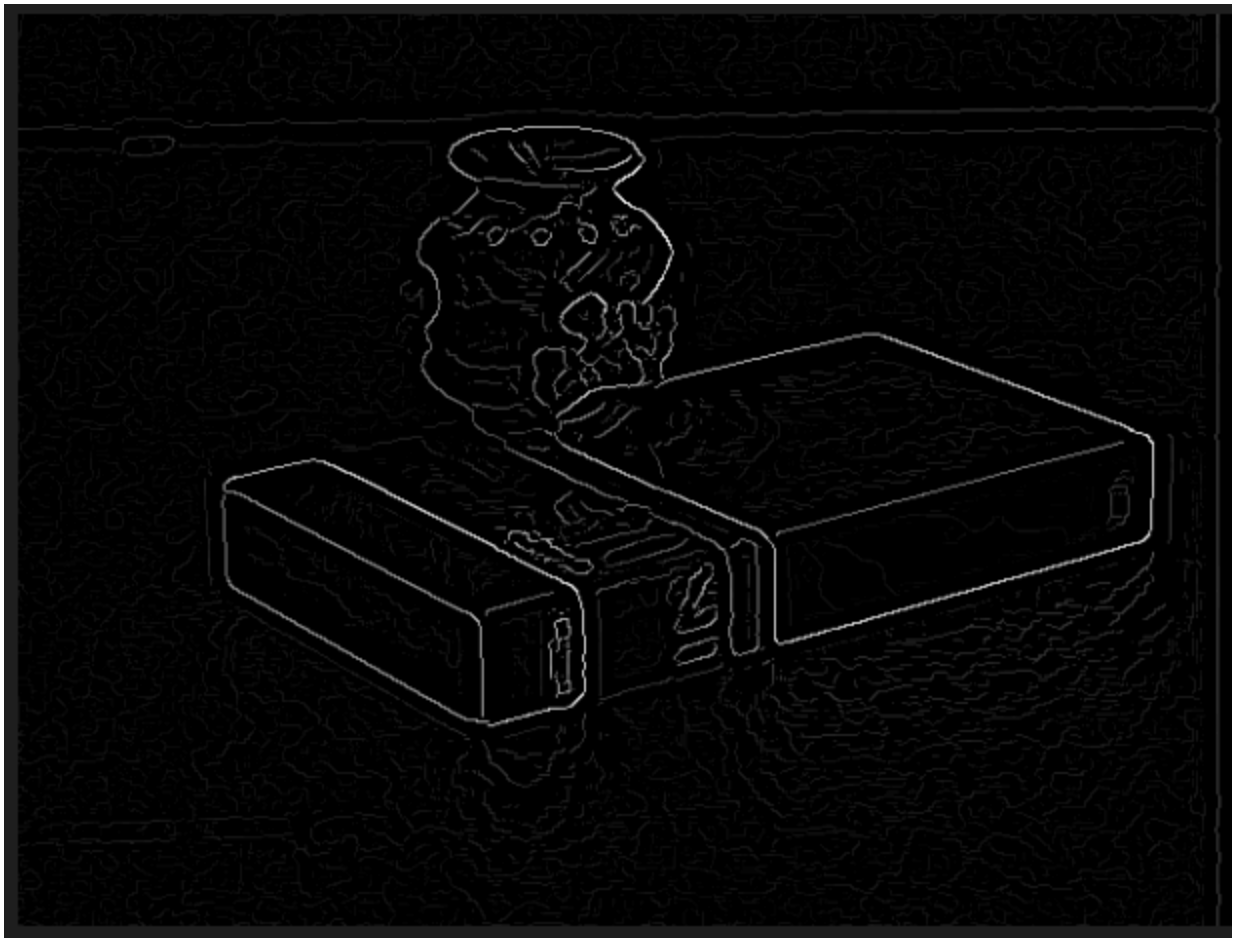


Final Output:

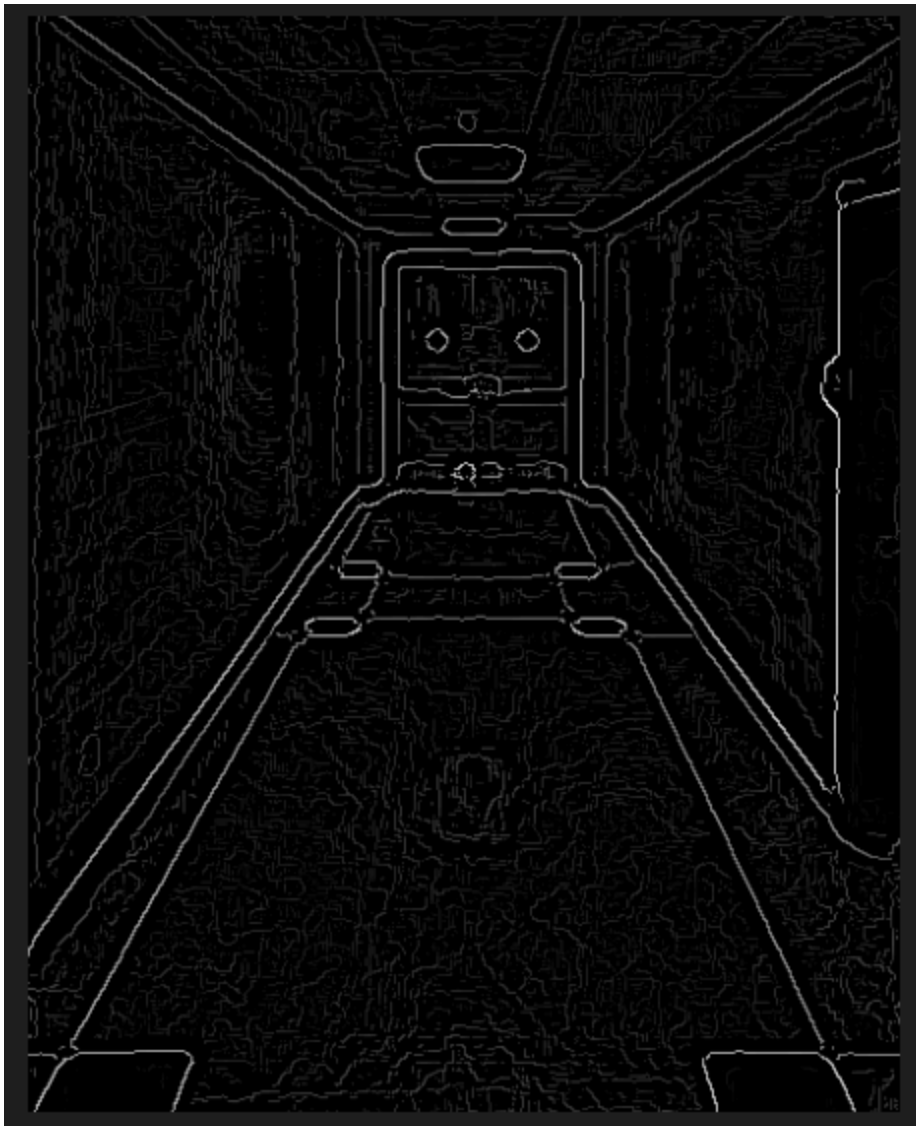
1



2



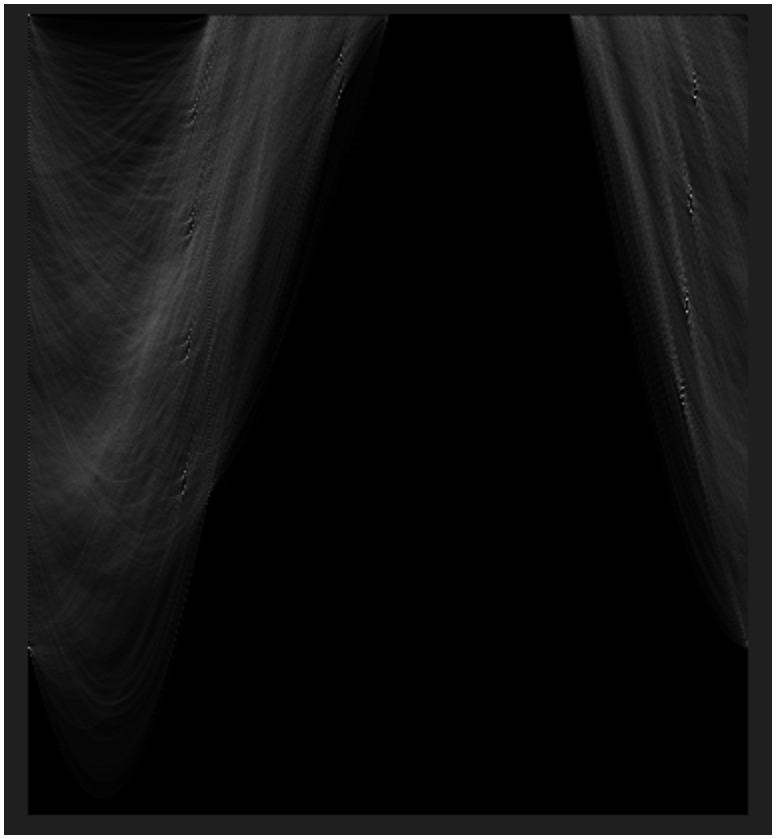
3



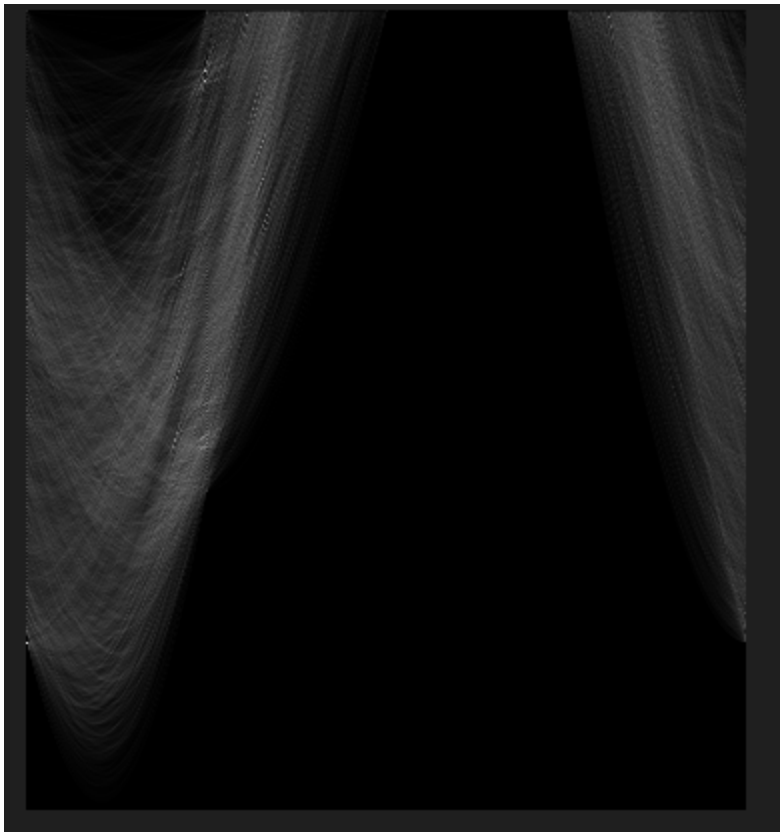
Q3.3

For the Hough Transform, I created an accumulator array where the rows represent rho and the columns represent theta. The size of this array depends on the resolution I chose for rhoRes and thetaRes. For every white pixel in my thresholded edge image, I calculated its potential rho values for every possible angle using the formula rho = xcos(theta) + ysin(theta). Then, I incremented the corresponding "bin" in the accumulator. I used np.add.at for the voting part to make sure every single vote was counted correctly even if multiple pixels wanted to vote for the same bin at once.I also noticed that the resolution parameters really matter. If I made the resolution too low, the peaks in the accumulator got really blurry and it was hard to find the exact lines later. But if the resolution was too high, the votes were too spread out and it was harder to get a strong peak for the actual lines.

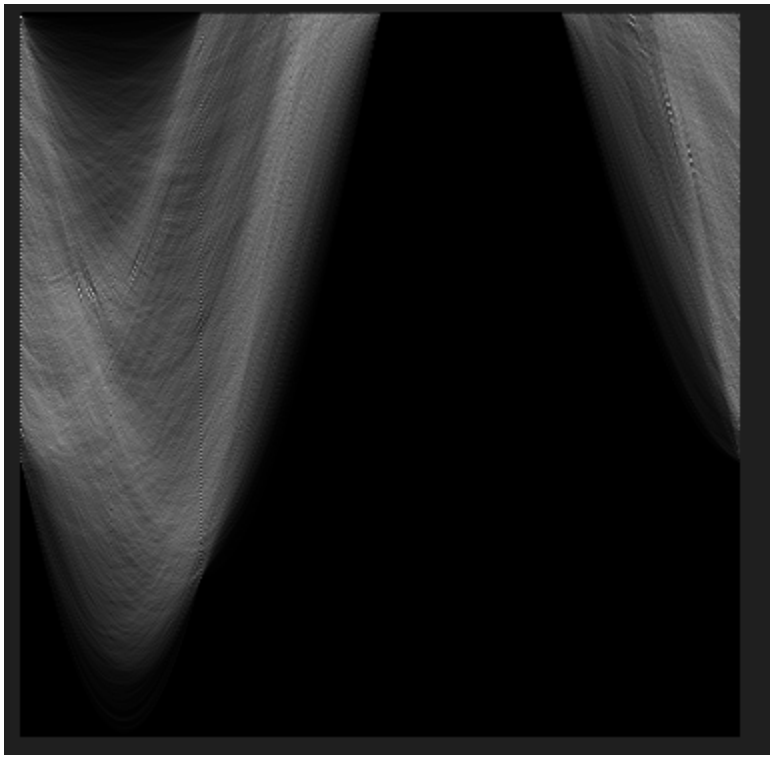
1



2



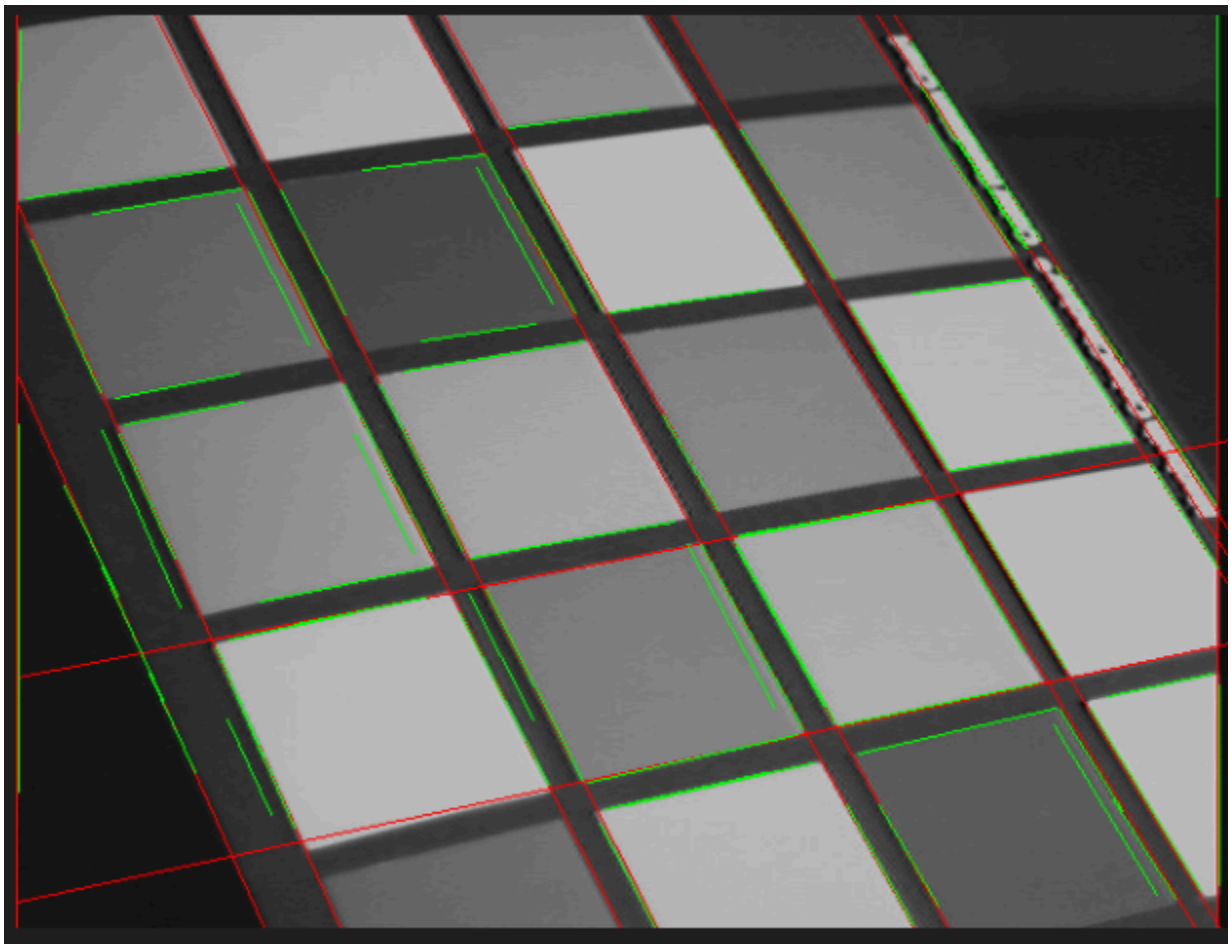
3



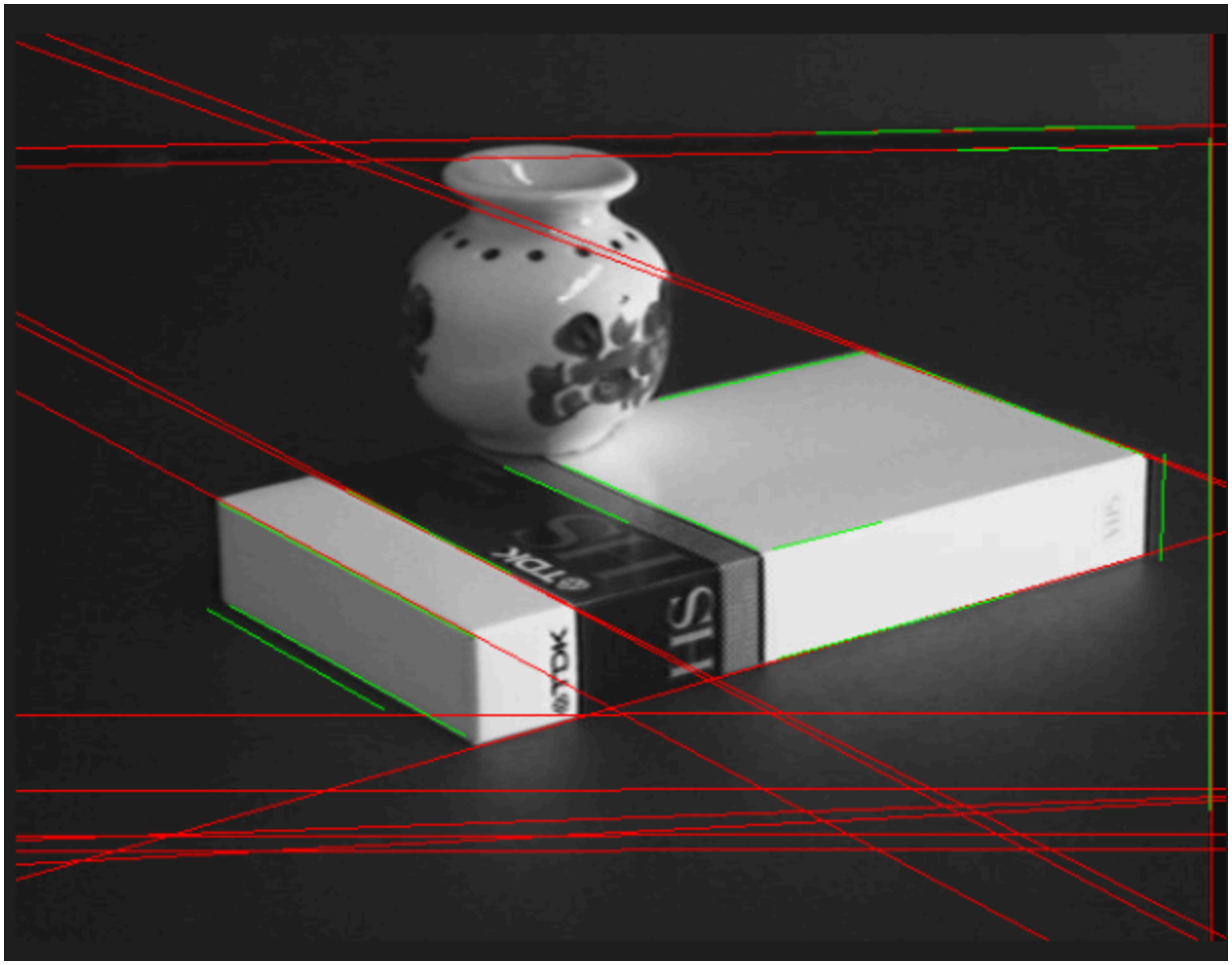
Q3.4
To find the actual lines from the accumulator, I first had to look for local peaks so I didn't get a bunch of lines that were basically in the same spot. I used a method to suppress any values that weren't the highest in their neighborhood, which really helped clean up the potential candidates. After thinning those out, I used np.argpartition to grab the top nLines with the highest number of votes. Then I just took those rho and theta values and turned them back into coordinates to draw the red lines on the image.

When I was playing around with the nLines value, I noticed it’s pretty sensitive. If I set it too low, the code would skip over some of the actual grid lines and the output looked incomplete. But if I turned it up too high, the algorithm started picking up "ghost" lines from random noise or tiny textures that weren't really lines at all. Finding the right balance was key to making the red lines match up with the green OpenCV ones.

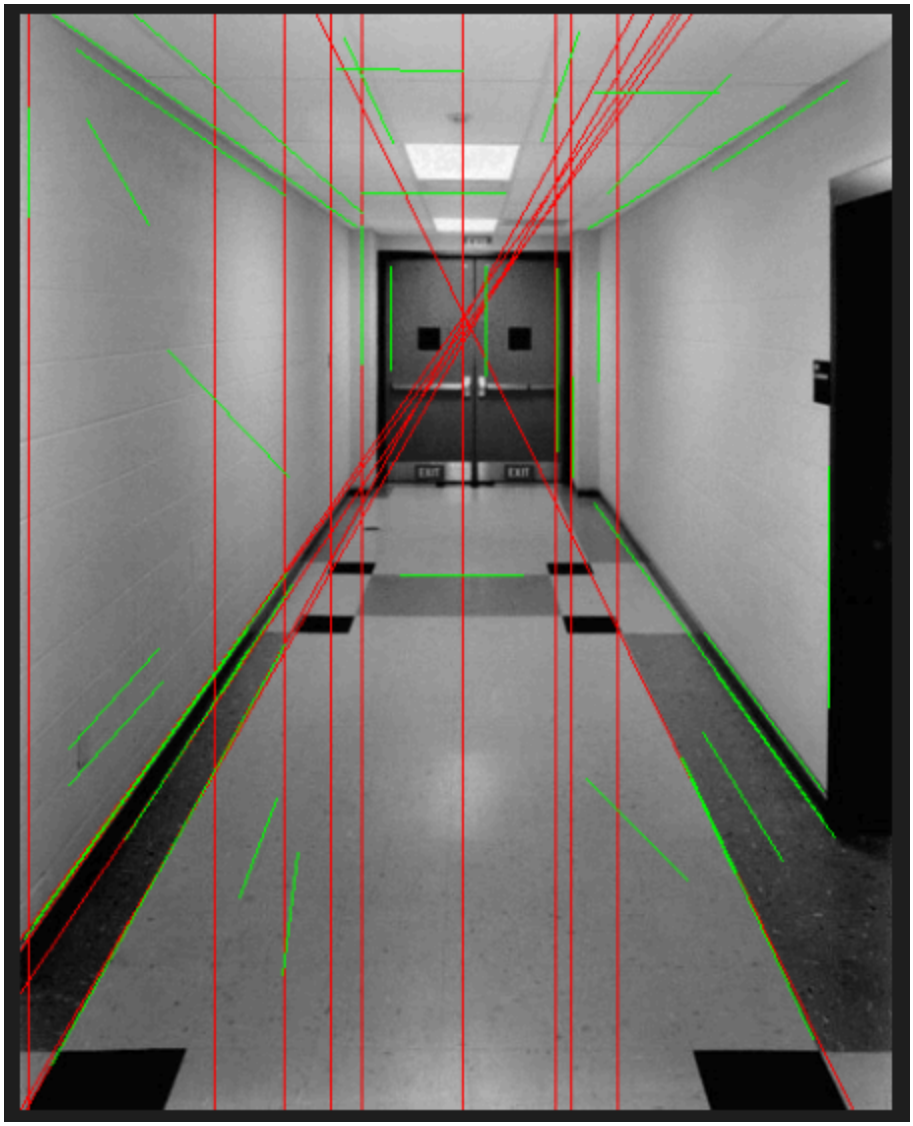
1



2

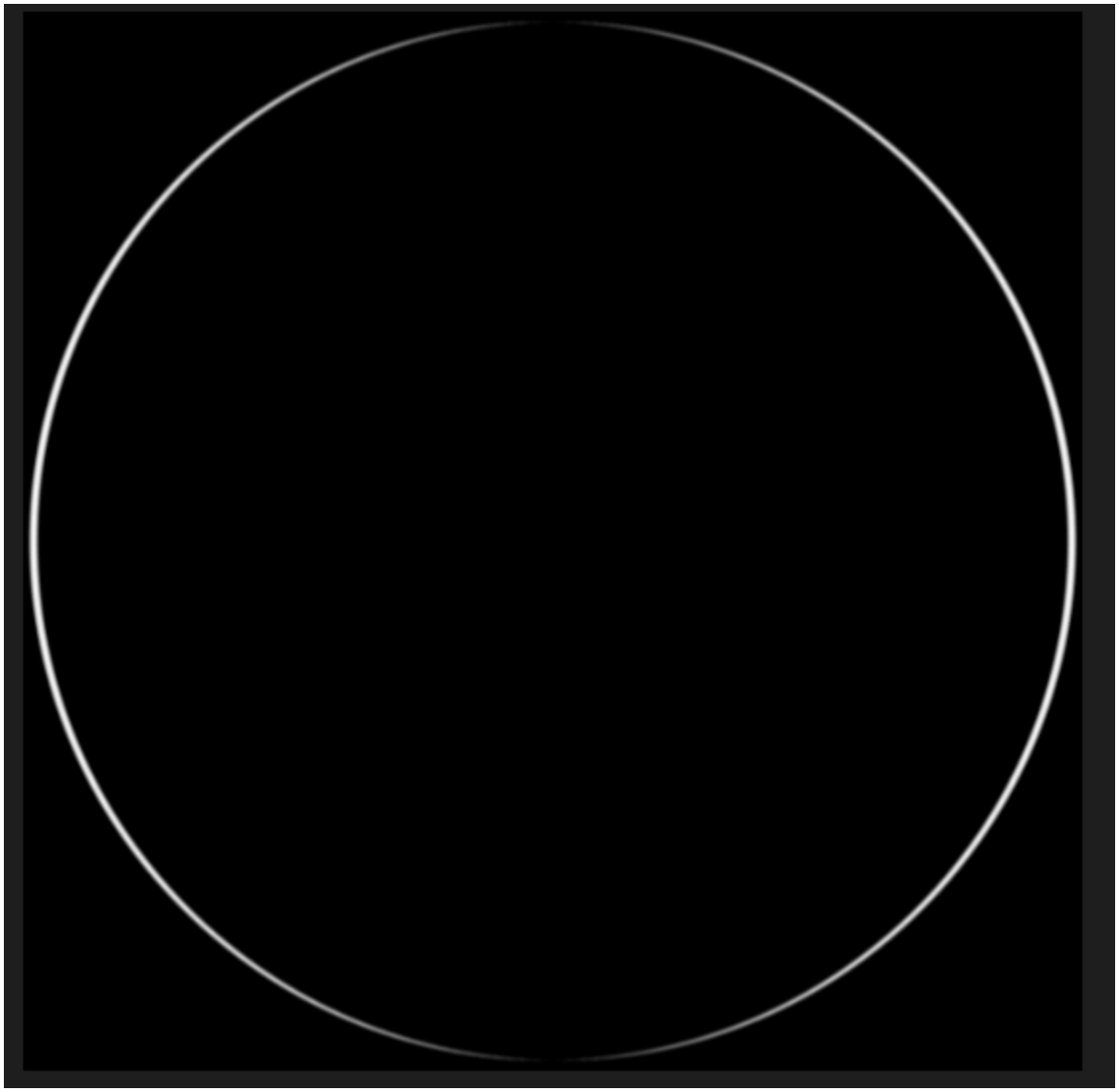


3

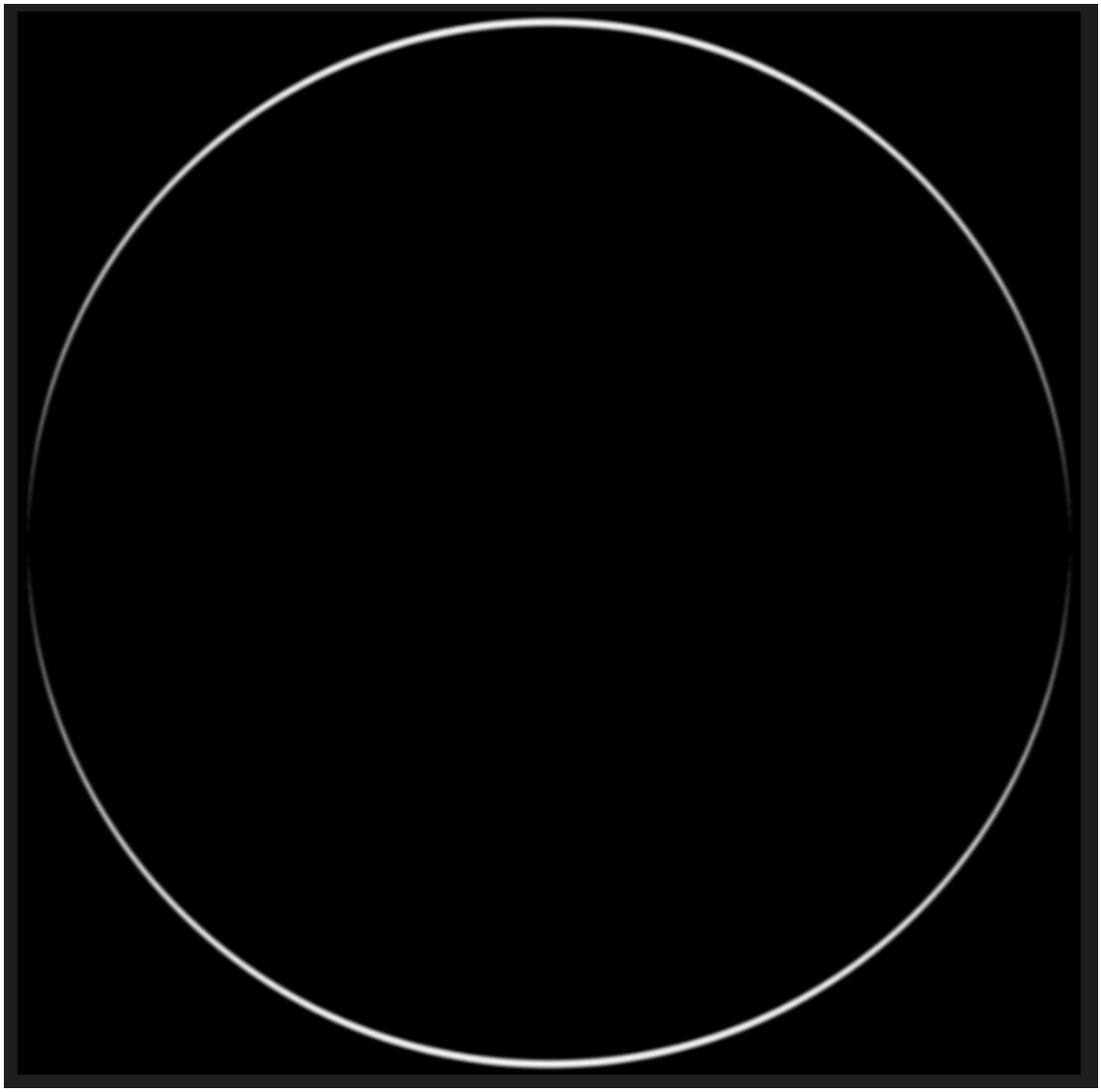


Q3.5

Sobel X



Sobel Y

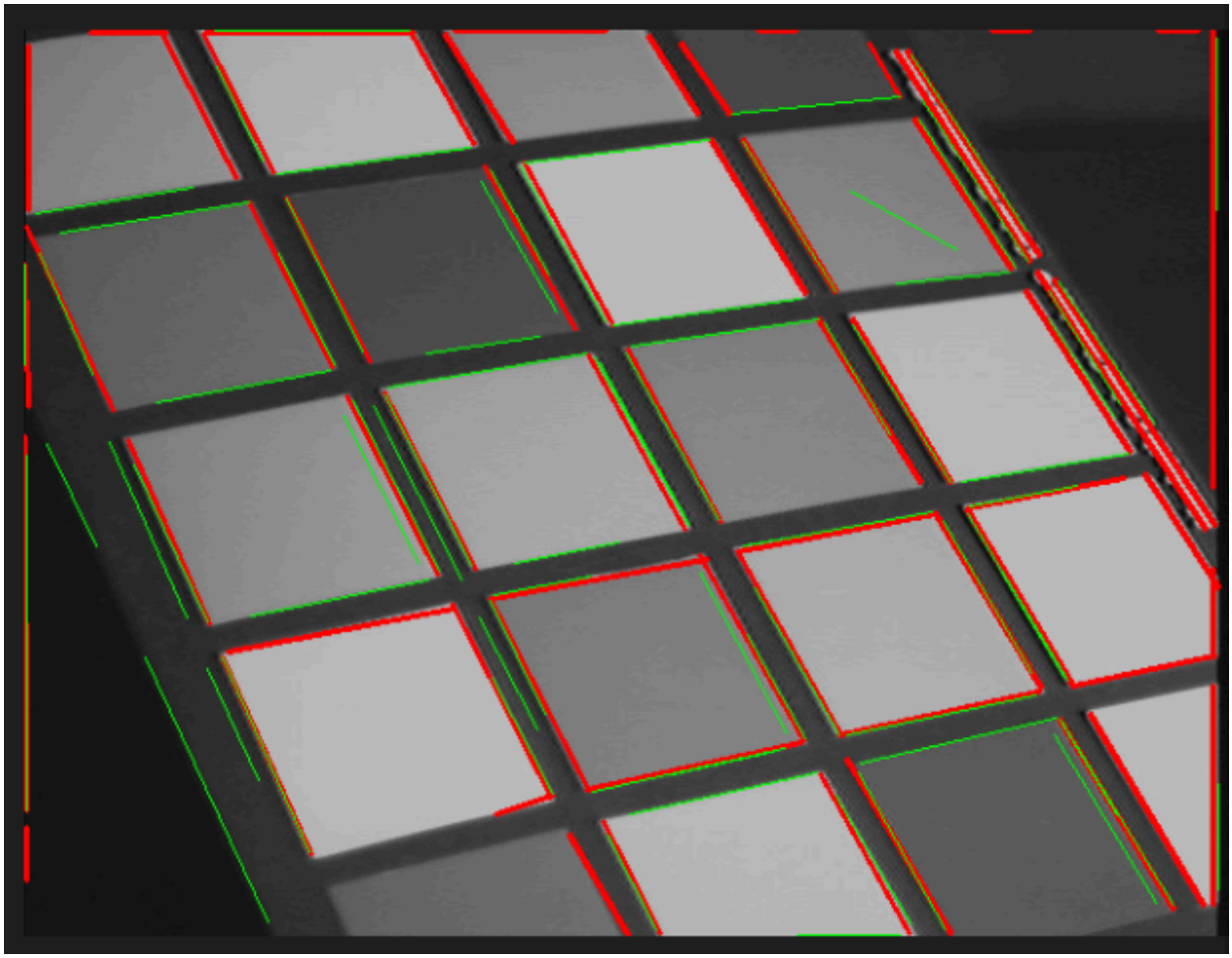


Q3.6

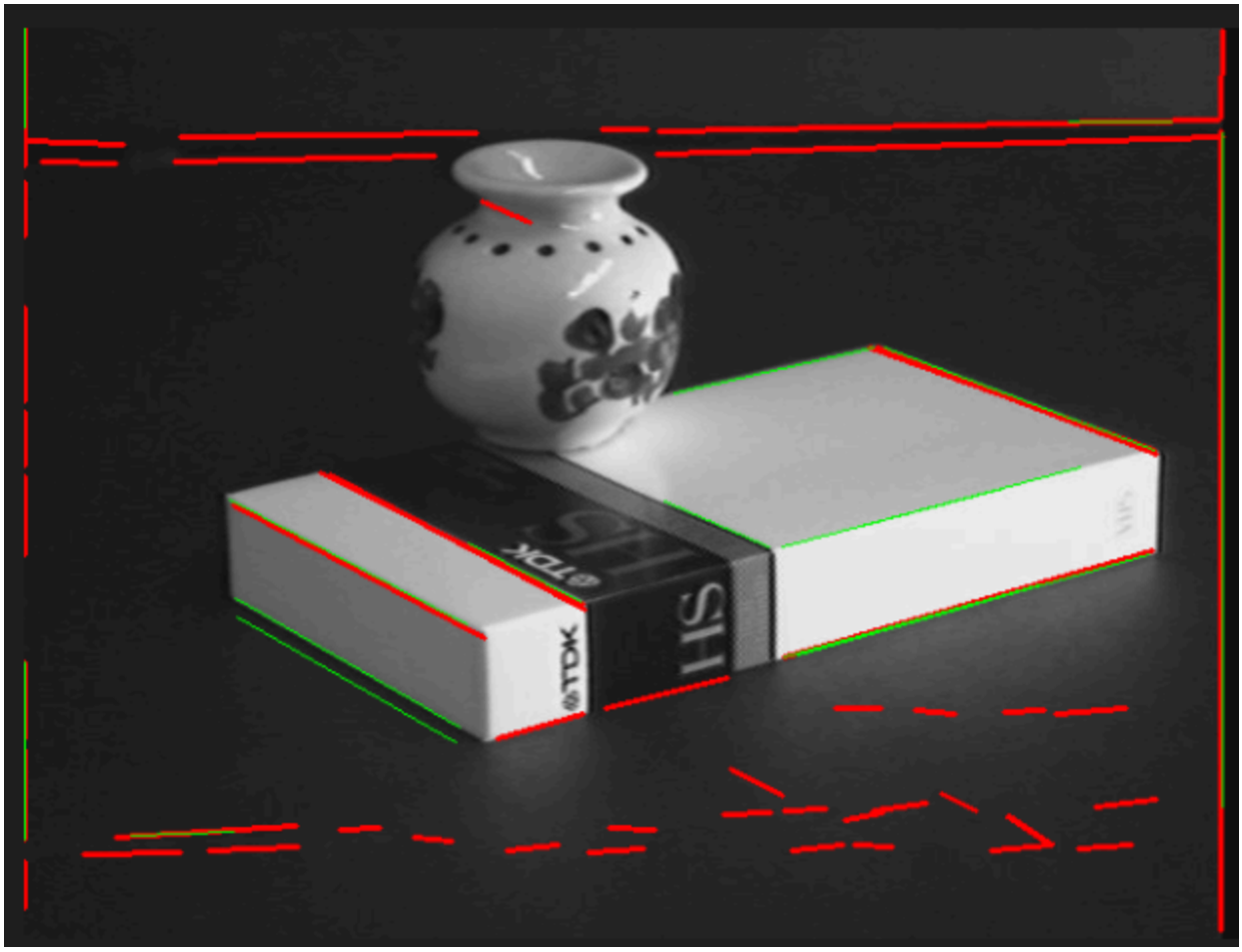
To implement the HoughLineSegments function, I had to figure out where the lines actually start and stop instead of just letting them run across the whole image. I took the (rho, theta) pairs from my earlier Hough function and used the line equation to find the coordinates of every pixel that could be part of that line. Then, I checked those spots in my binary edge image to see which ones were actually "white." I walked along the line and grouped these pixels into segments as long as the gaps between them weren't too big. If I hit a gap that was wider than my fillGap setting, I'd end that segment and start a new one if I found more edge pixels further down. This way, the red lines actually sit on top of the real edges in the photo.

When I compared my results to OpenCV's HoughLinesP, they were actually really close. In the final images, my segments are red and the OpenCV ones are green. They overlap almost perfectly on the main grid lines of the board, which shows my grouping logic is working. The biggest difference I noticed was that OpenCV's segments are sometimes a bit cleaner because it's probably better at filtering out tiny, random lines that aren't important, but for the main task of finding the board structure, both methods pretty much did the same thing.

1



2



3

Q3.7 DID NOT DO