CMU Computer Vision HW1

Jiaqi Geng

September 22, 2020

Theory Question

2.1

The resulting sinusoid function is in the form of $A\sin(B\theta + t) + C$.
Then, $A\sin(B\theta + t) + C = x\cos\theta + y\sin\theta$
$A(\sin(B\theta)\cos t + \cos(B\theta)\sin t) + C = x\cos\theta + y\sin\theta$
$A\sin(B\theta)\cos t + A\cos(B\theta)\sin t + C = x\cos\theta + y\sin\theta$
In order for this equation to hold:
we must have $B = 1, C = 0$
so $x = A\sin t, y = A\cos t$
$\sin t^2 + \cos t^2 = (\frac{x}{A})^2 + (\frac{y}{A})^2 = 1$
so $A^2 = x^2 + y^2, A = \sqrt{x^2 + y^2}$
$\tan t = \frac{x}{y}$, so $t = \arctan\frac{x}{y}$

2.2

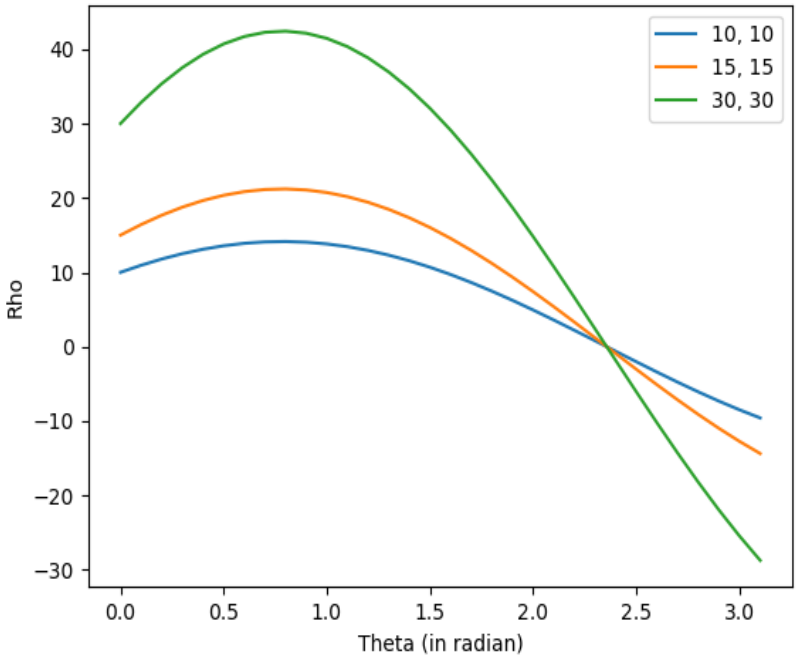
Slope m ranges from $-\infty$ to ∞ because of vertical lines,
so we cannot create an accumulator with a finite size.
We know that $x\cos\theta + y\sin\theta = \rho$,
we let $x = 0$, then $y_0 = \frac{\rho}{\sin\theta}$
we let $y = 0$, then $x_0 = \frac{\rho}{\cos\theta}$
so $m = \frac{-y_0}{x_0} = \frac{-\cos\theta}{\sin\theta}, b = \frac{\rho}{\sin\theta}$

2.3

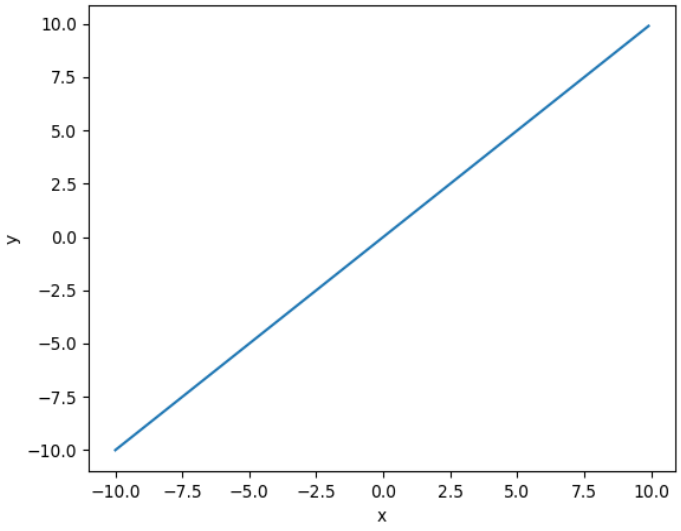
According to what we have done in 2.1, for a single point (x, y),
the maximum absolute value of ρ is $\sqrt{x^2 + y^2}$.
Therefore, considering all points in this image,
the maximum absolute value of ρ would be $\sqrt{W^2 + H^2}$.

The range of θ would be from 0 degree to 180 degree.
When θ is greater than 180 degree,
we can write $\theta = \theta^* + \pi$, and θ^* is from 0 to 180 degree.
we can represent the line as $x\cos(\theta^* + \pi) + y\sin(\theta^* + \pi) = -\rho$,
and this is the same line as $x\cos\theta^* + y\sin\theta^* = \rho$

2.4



The intersecting point is $(\frac{3}{4}\pi, 0)$
$m = \frac{-\cos\theta}{\sin\theta} = 1, b = \frac{\rho}{\sin\theta} = 0$
Therefore, the line is: $y = x$



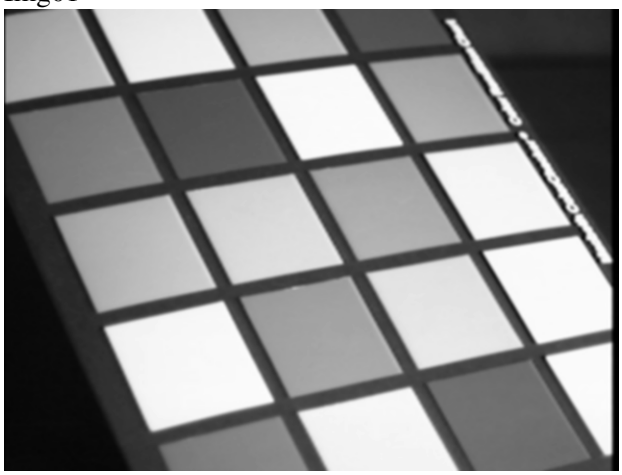
2.5

If we have 3D (or 4D) parameter space,
we could change our accumulator to 3D (or 4D) array.
Then, we follow the same procedure as before:
Set every cell in the accumulator to be 0.
For each edge point (x, y),
we loop through possible parameters,
if they satisfy our parametric equations,
then increase the value of the corresponding cell by 1.
Then, we find the local maximums using non maximum suppression.
Instead of looking at the neighbors in 2D,
we will compare each cell with its neighbors in 3D (or 4D).
We consider a cell as local maximum only when it is greater than all of its neighbor in 3D (or 4D).
We get the detected lines by plugging in parameters of local maximums to our parametric equations.

Resulting Images

3.1

Img01



Img02

Img03

Img04

Img05

Img06

Img07

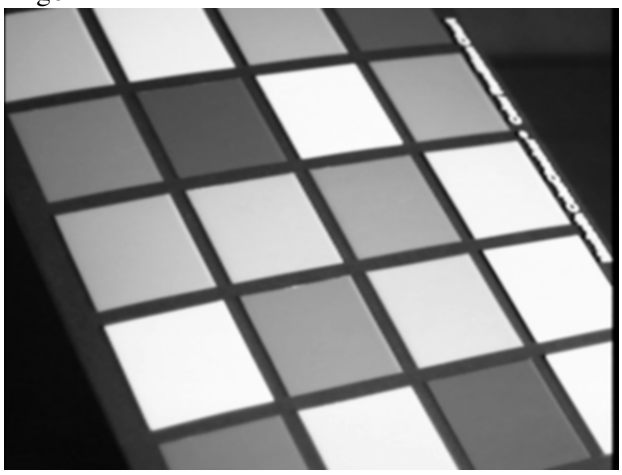
Img08

Img09

3.2

How I implemented it:
First assume that we have a filter at the top left corner of the image,
then calculate the index (if we flatten the image) of each pixel covered by this filter
and put them in a numpy array.
Then we calculate all possible starting index and make it a numpy array.
We can add these two arrays with broadcasting to get a matrix
with each row contains the pixel indices covered by one filter.
We get a matrix with values indexed by the above matrix.
We do dot product between this matrix and the flattened kernel (after flipping).

Img01



Img02

Img03

Img04

Img05

Img06

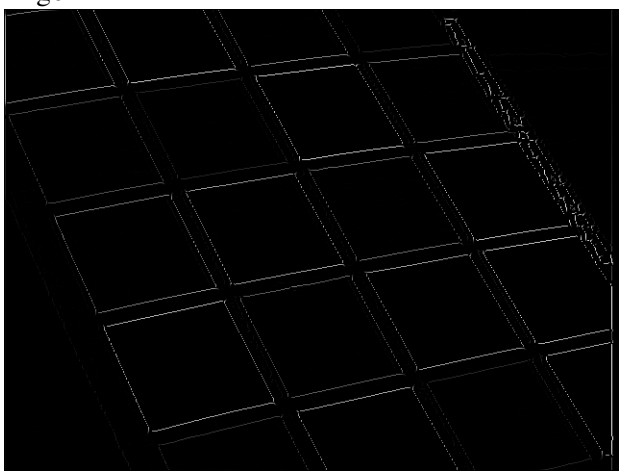
Img07

Img08

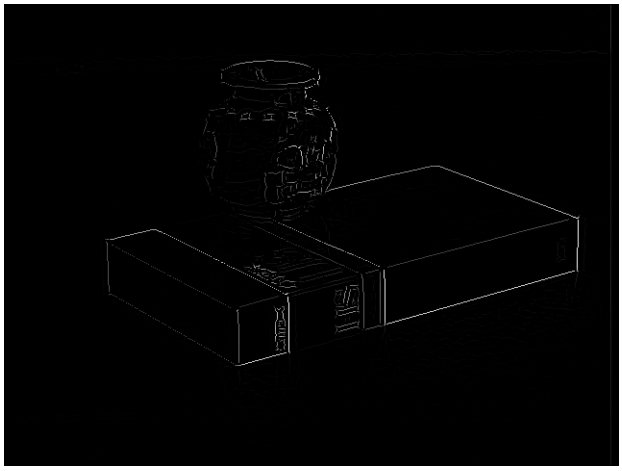
Img09

3.3

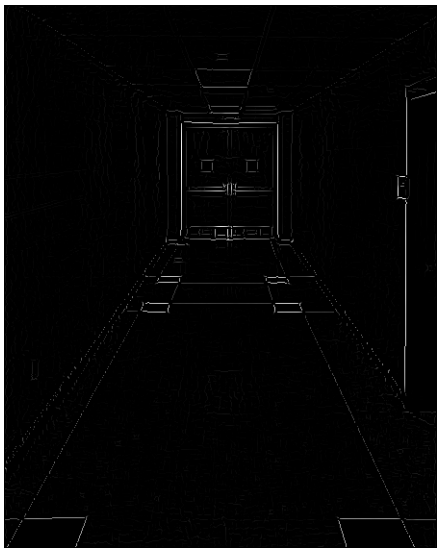
Img01



Img02



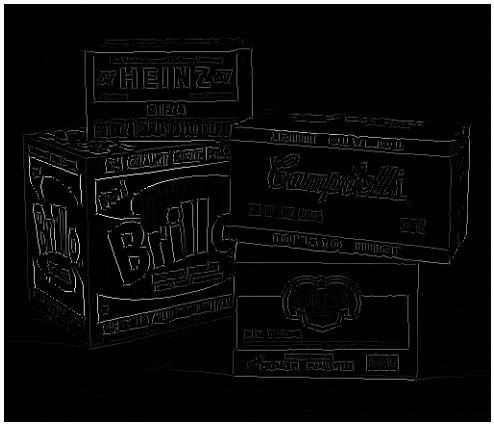
Img03



Img04

Img05

Img06



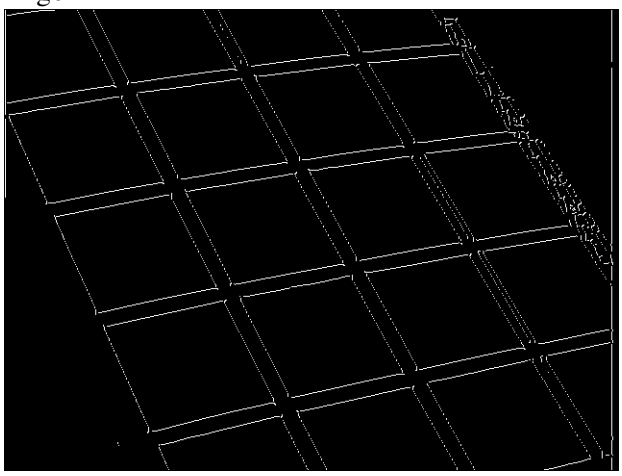
Img07

Img08

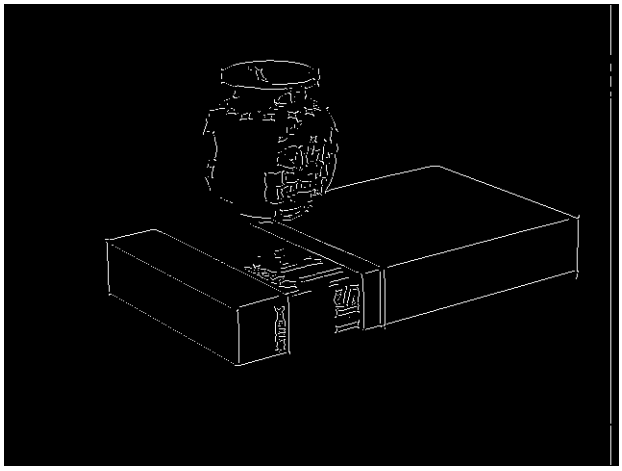
Img09

3.4

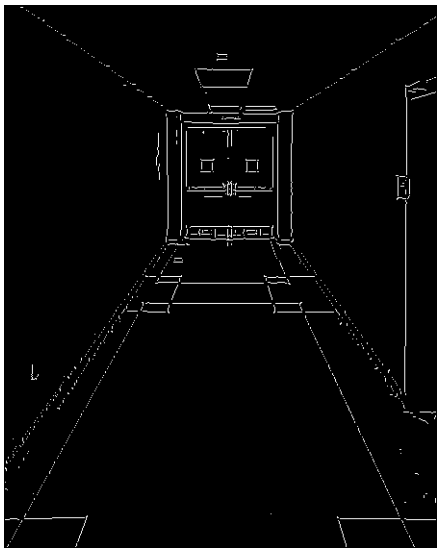
Img01



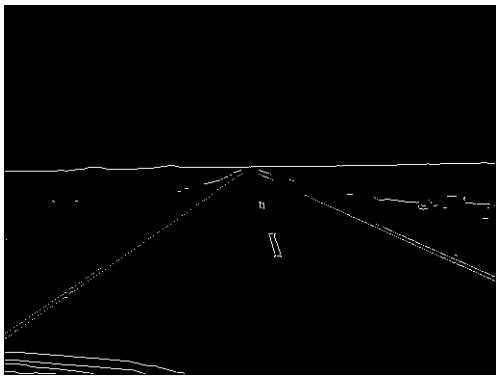
Img02



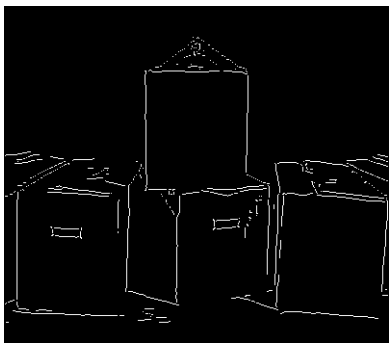
Img03



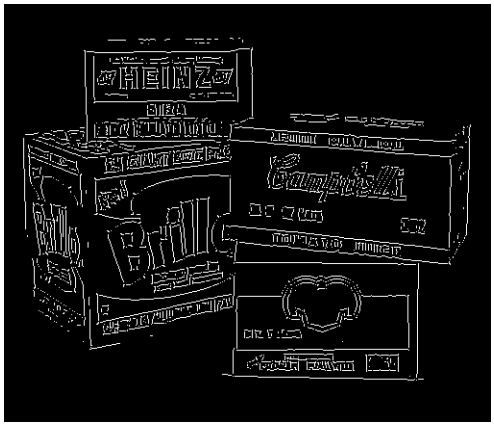
Img04



Img05



Img06



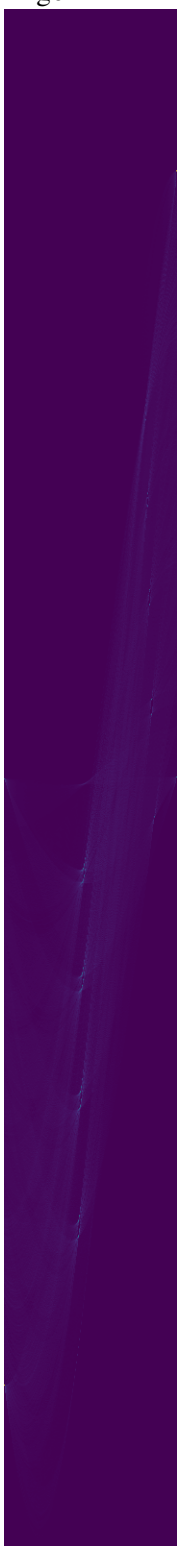
Img07

Img08

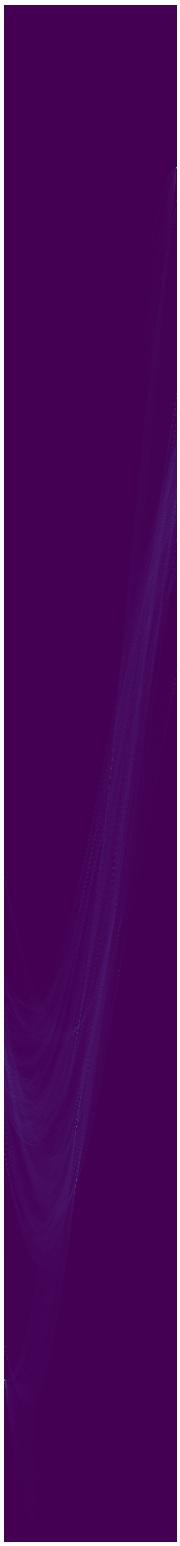
Img09

3.5

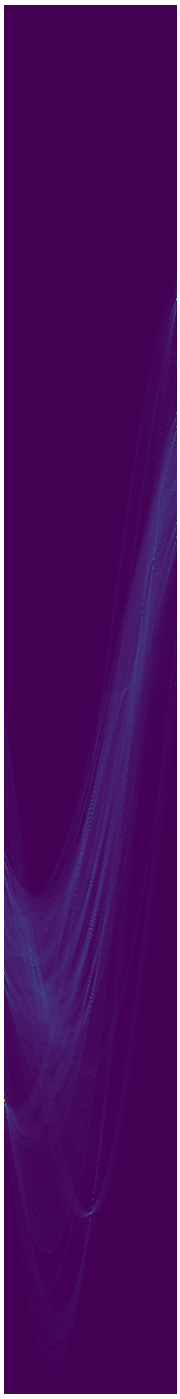
Img01



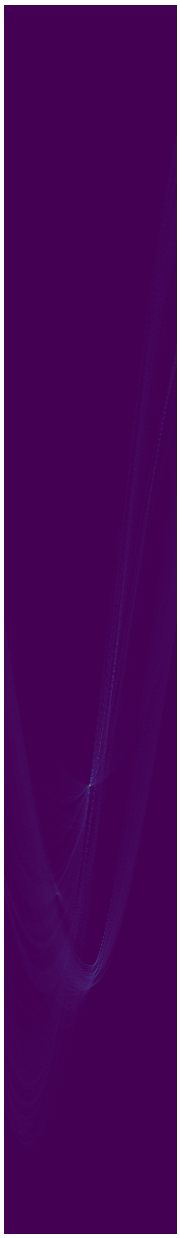
Img02



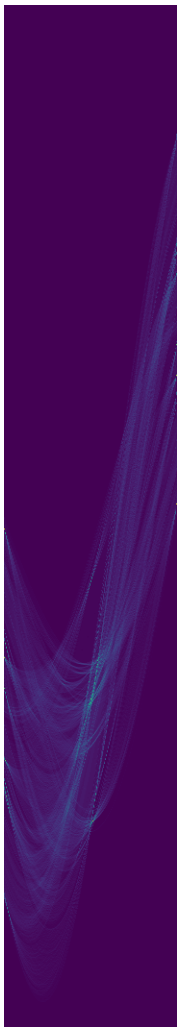
Img03



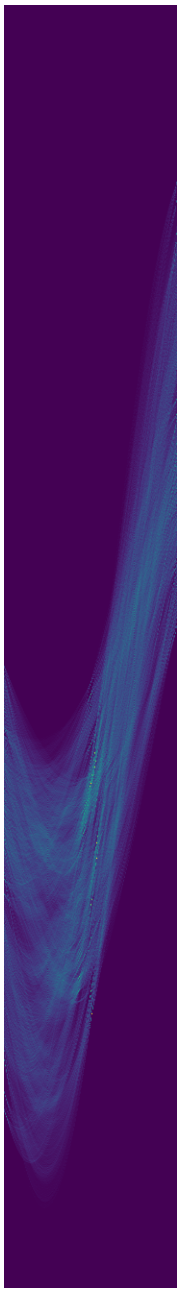
Img04



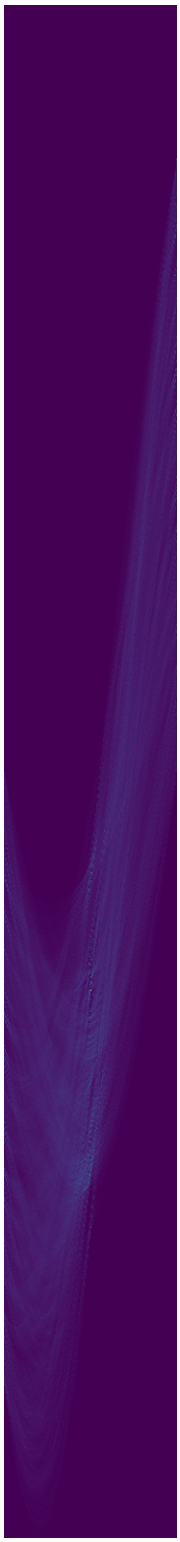
Img05



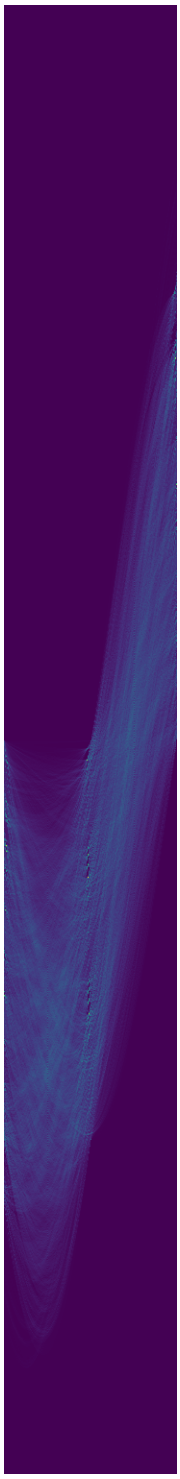
Img06



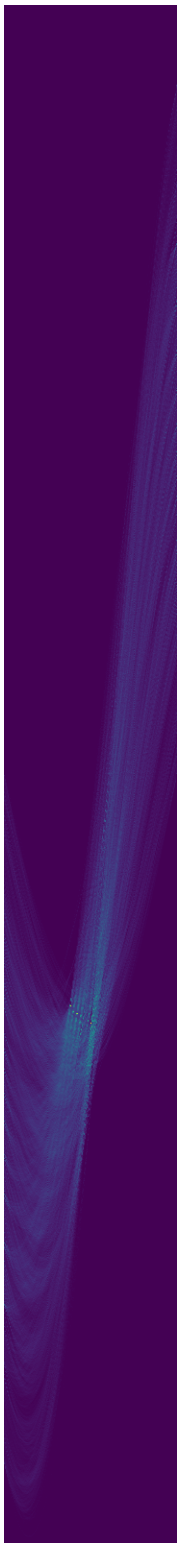
Img07



Img08

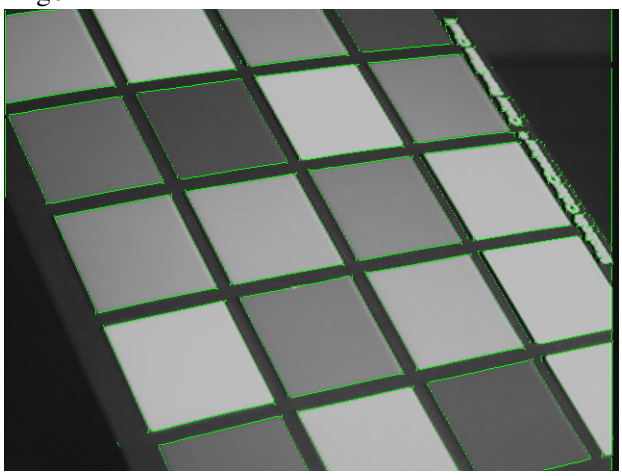


Img09

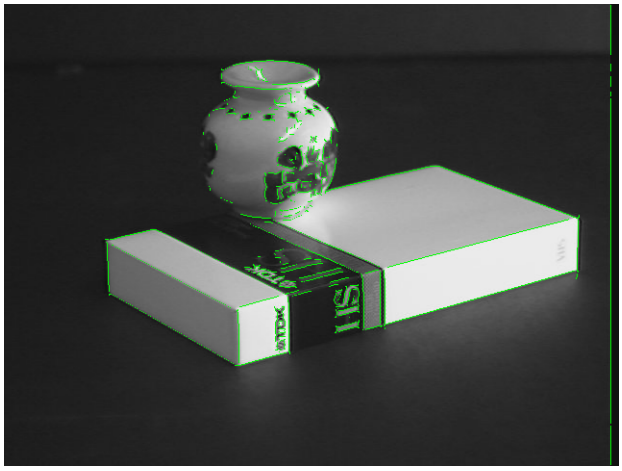


3.6

Img01



Img02



Img03

Img04

Img05

Img06

Img07

Img08

Img09


www.dexigner.com

Experiment

I focus on changing three parameters:
threshold (for ImEdge), nLines, and sigma (for gaussian filter)

Result images are on the next few pages.

threshold

I first make sure I normalize my ImEdge so that each pixel value is within 0 and 1.
Then I choose 4 different threshold: 0.01, 0.1, 0.3 and 0.5
For smaller threshold, we would include more noises in our final result.
For larger threshold, we might not include all the lines.
We get better results for Img1 using smaller threshold,
and we get better results for Img4 using bigger threshold.
It is clear that the code does not work equally well on all images with the same set of parameters.

nLines

The effect of the value of nLines seems to be more consistent across images.
When nLines is small, then it is very common that the final result won't have many detected edges.
When the value of nLines increases, we can detect more lines, but also include more noises.

sigma

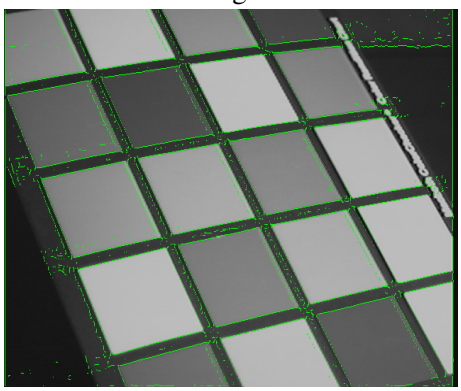
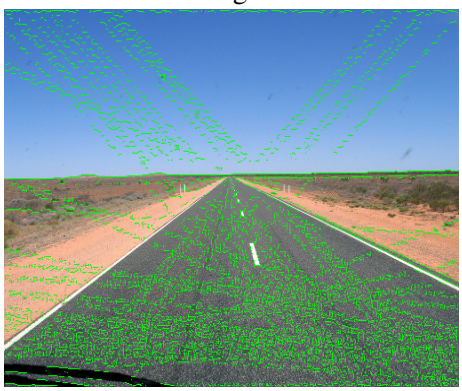
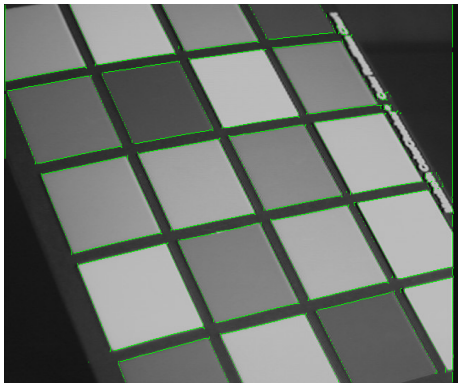
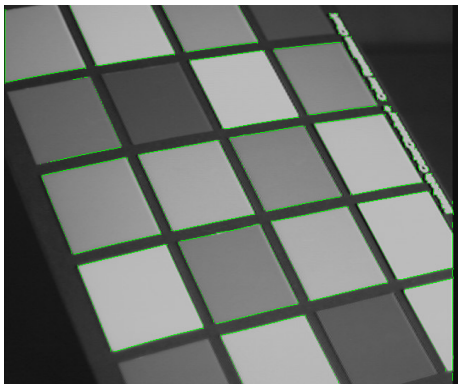
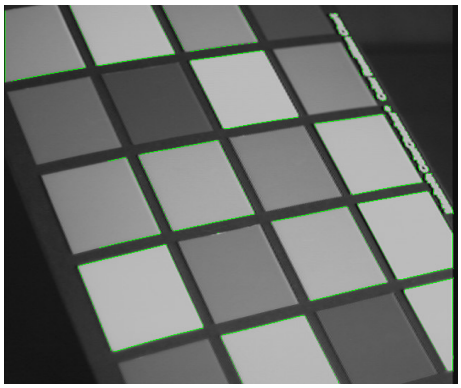
When sigma is 0.5,
Img1 does not seem to be affected too much,
but we can see a ton of noises from Img7.
While the sigma increases,
we see less and less noise from Img7,
while we can not see too much changes from Img1.

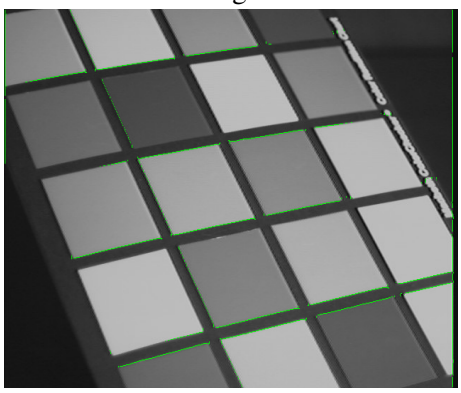
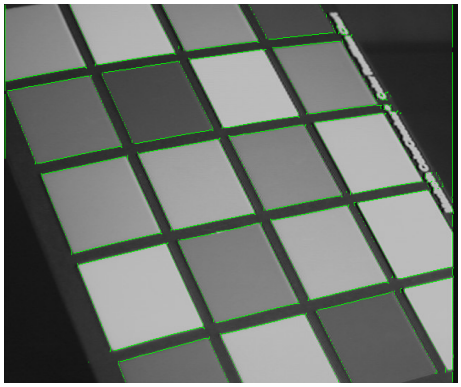
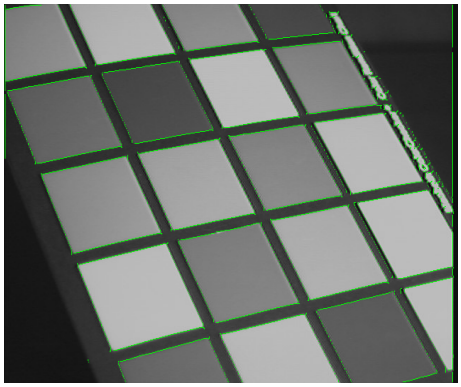
The step of the algorithm causes the most problems

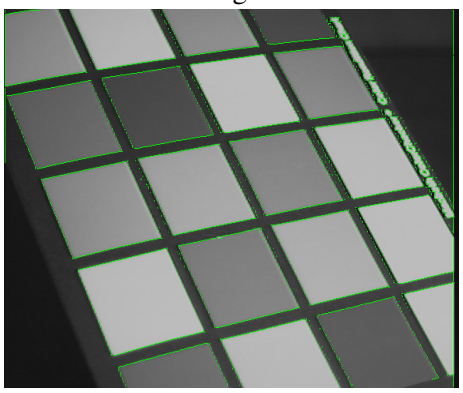
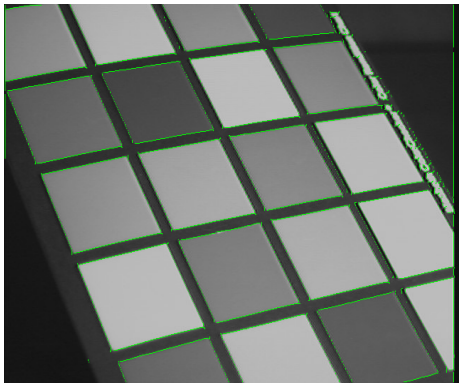
Personally I think it is the last step where we draw hough line segments.
For this part, technically we should find the exact end points for each line.
However, we are told that it is okay to only visualize the line segments
using the edge detection result we got from 3.3
Since not all edge points are connected,
we can see that some points on a line are not connected in the final result.

Improvement

Besides from tuning the parameters for different images,
I found that vectorization for the calcuation of hough transform increases the running speed.

Thres	Img1	Img4
0.01		
0.1		
0.3		
0.5		

nLines	Img1	Img7
10		
50		
250		

sigma	Img1	Img7
0.5		
1.5		
2.5