

## INTRODUCTION

We implemented corner detection algorithms Kanade-Tomasi and Harris for the greyscale images. Rest of the report will demonstrate the implementations of these algorithms and their explanations. Then, their results will be shown. At the end of each part, algorithms will be discussed.

**Corner** in the image corresponds to two different edges intersection for different directions. In other words, the magnitude of the eigenvalues of 'H Matrix' should be large for the pixel to be considered as a corner.

### 1. Kanade-Tomasi Algorithm

Kanade-Tomasi Algorithm, one of the corner detection algorithms, is implemented during the lab. Kanade-Tomasi algorithm is following the Minimum Eigen Value finding algorithm.

According to Kanade-Tomasi algorithm, a pixel can be considered as corner when its minimum value of eigenvalues for the 'H matrix' is higher than an input parameter threshold.

**This algorithm follows these steps:**

- 1) Image converted to greyscale
- 2) Noise reduction with Gaussian Filter (Smoothing Operation)
- 3) Computing Gradients of the image (Gx and Gy)
- 4) Creating an H matrix for each pixel
- 5) Compute Eigen values of this H matrix
- 6) if minimum of the eigen values is larger than threshold consider this pixel as a corner

$$H = \begin{bmatrix} \sum I_x^2 & \sum I_x I_y \\ \sum I_x I_y & \sum I_y^2 \end{bmatrix}$$

(H matrix)

## 1.2 My Kanade-Tomasi Algorithm Explanation

In my implementation, I started as always by ensuring that the image is grey scale.

```
% Ensuring grey scale image
[row,col,ch] = size(img);
if(ch == 3)
    img = rgb2gray(img);
end
```

Then, I used Gaussian filter for smoothing operation. Smoothing is an important operation as the noise of the image may cause problematic results for the gradients of the image.

After, filtering our image with Gaussian Filter, I obtained gradients (Gx, Gy) of the image.

Then, I repeated the algorithm steps 4, 5, 6 with the following algorithm:

```
13 - [Gx, Gy] = imgradientxy(data);
14 - cornerlist = [];
15 - for i = k+1:row-k
16 -     for j = k+1:col-k
17 -         my_window = Gx((i-k):(i+k), (j-k):(j+k));
18 -         my_window2 = Gy((i-k):(i+k), (j-k):(j+k));
19 -         IX = sum(sum(my_window.*my_window));
20 -         IY = sum(sum(my_window2.*my_window2));
21 -         IXY = sum(sum(my_window.*my_window2));
22 -
23 -         H = [IX, IXY; IXY, IY];
24 -         values = eig(H);
25 -
26 -         if min(values) > threshold
27 -             cornerlist = [cornerlist; [i,j]];
28 -         end
29 -     end
30 - end
```

For each pixel, I took a window with size  $(2k+1) \times (2k+1)$ . I computed  $I_x$  and  $I_y$  image gradients of window along x and y directions to create H matrix. Then, I used built in 'eig' function to calculate eigenvalues of the H matrix. If minimum of these eigenvalues is higher than threshold, I added this pixel location to my corner list. Then, I marked these points on the image as following way (row and column values are interchanged):

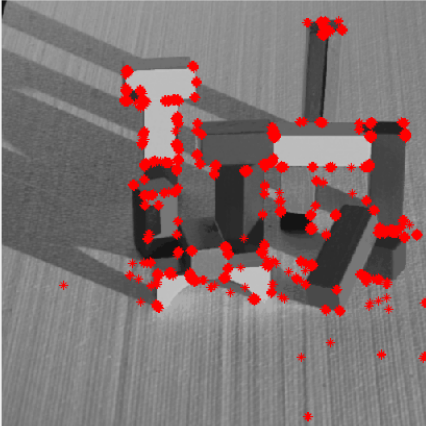
```
imshow(new_img);
% title("Kanade-Tomasi")
hold on
plot(cornerlist(:,2), cornerlist(:,1), 'r*')
```

## 1.3 Kanade-Tomasi Algorithm Results

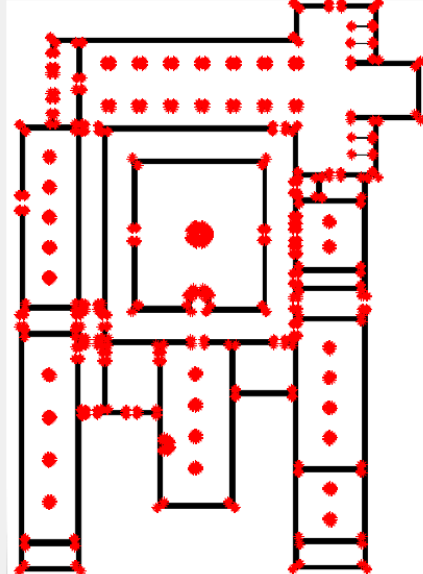
Given Images with different threshold values and window size:

Kanade-Tomasi Algorithm for Corner Detection

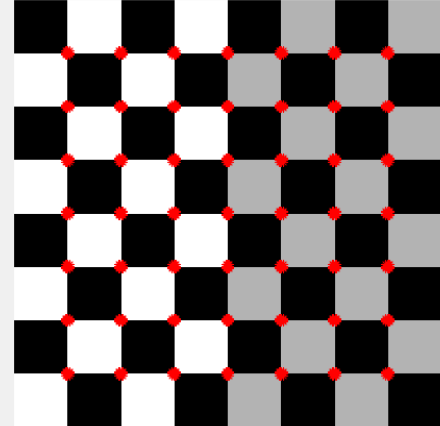
Threshold: 20000 Window Size: 3x3



Threshold: 20000 Window Size: 3x3

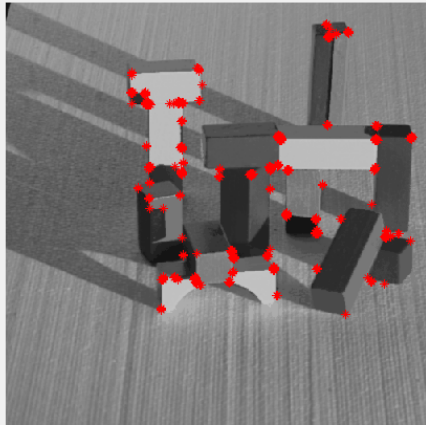


Threshold: 2 Window Size: 3x3

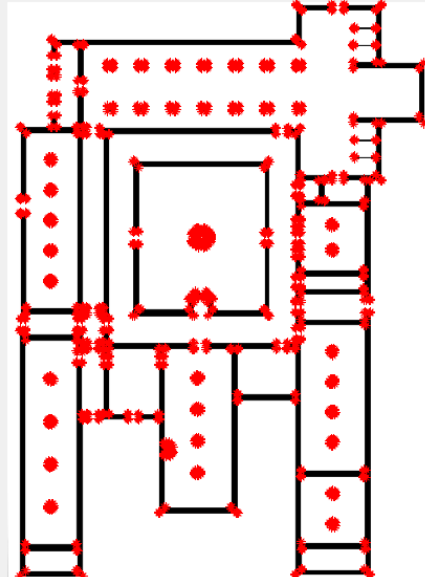


Kanade-Tomasi Algorithm for Corner Detection

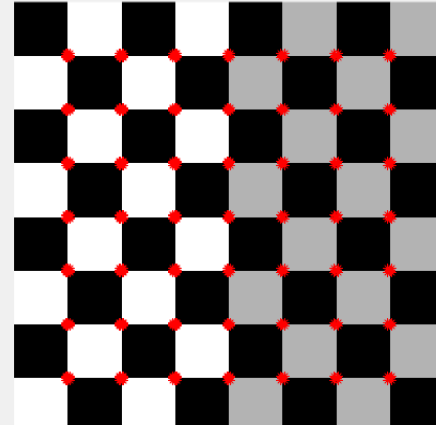
Threshold: 50000 Window Size: 3x3



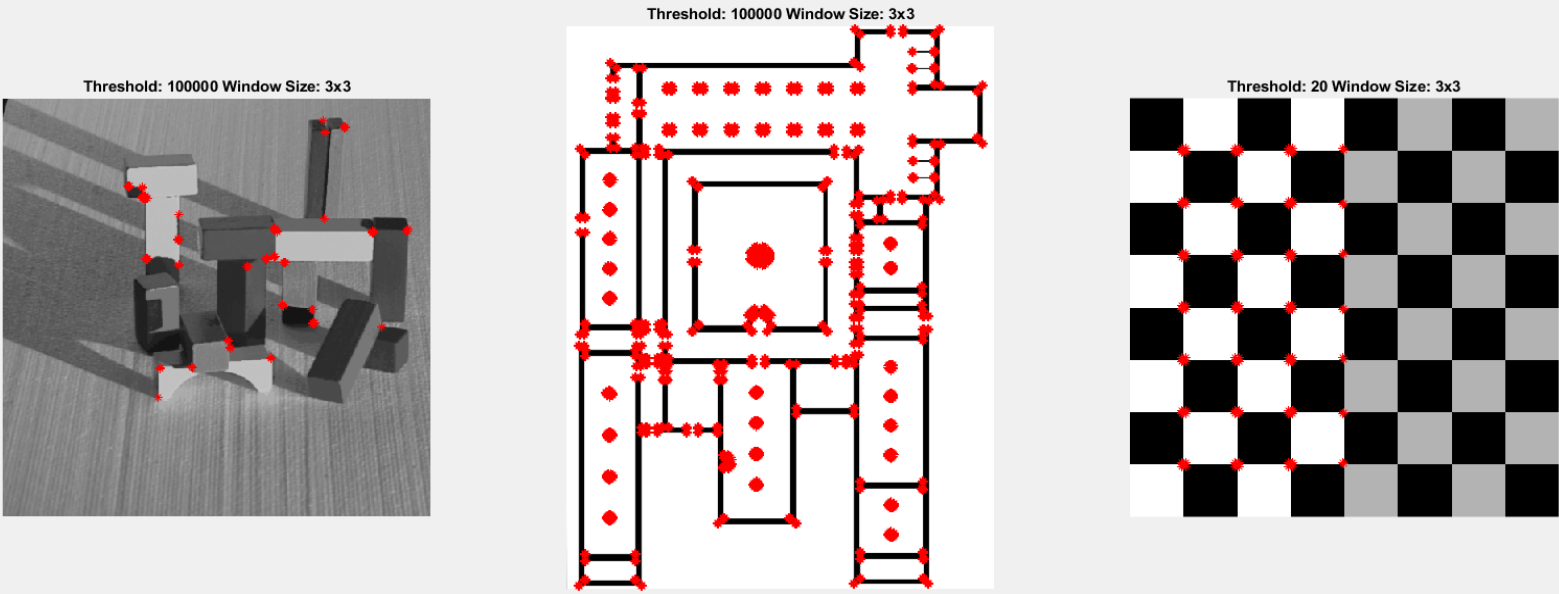
Threshold: 50000 Window Size: 3x3



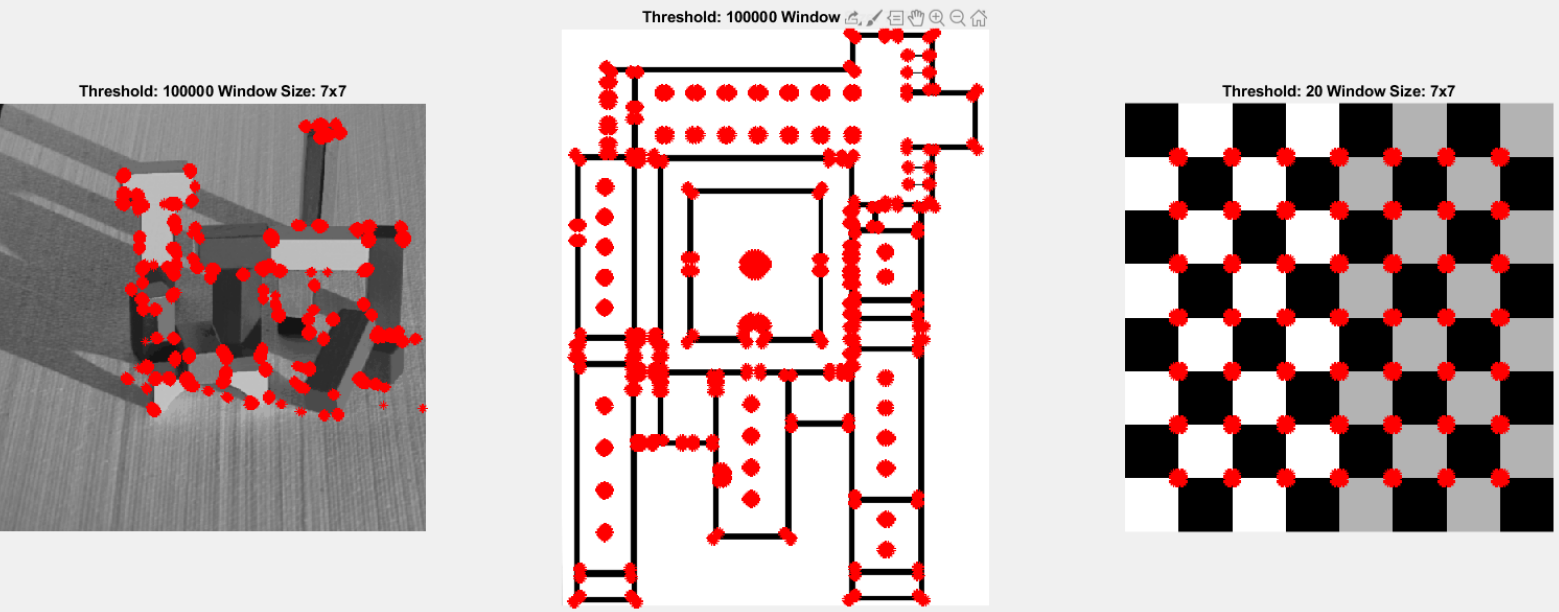
Threshold: 10 Window Size: 3x3



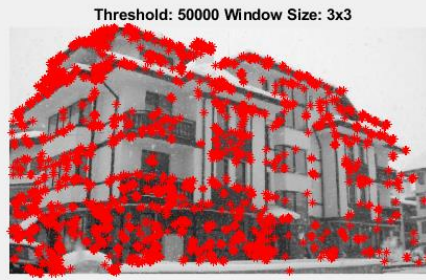
Kanade-Tomasi Algorithm for Corner Detection



Kanade-Tomasi Algorithm for Corner Detection



### New Image (House.png) with different threshold values:



## 1.4 Kanade-Tomasi Algorithm Discussion

We can see that corners in the images can be detected by Kanade-Tomasi Algorithm.

However, the threshold value has a crucial role for this algorithm. We observed that if we choose threshold relatively lower, we detect extra pixel points which are not corners as well. On the other hand, if we choose threshold higher, we may miss the corner pixels. Therefore, we need to find best threshold value for each image. This situation increases the cost of the algorithm as we need to run algorithm more than 1 time to find optimal threshold value.

We can also observe that change of threshold does not affect the output images dramatically at each time. For instance, when we play with threshold value for Blocks image, the number of markers in the images change rapidly. However, Monastery image is not affected too much from the change of threshold. I think that since Monastery image pixel values are binary (0 and 255), it is easier to detect corners.

We also observed that when we increased the window size, we can detect more corner pixels with same threshold. For instance, Checkerboard image with threshold 20 and 3x3 ( $k=1$ ) window could not detect corners in the right part of the image. However, when we increased our window size to the 7x7 ( $k=3$ ), Kanade-Tomasi managed to catch all corner pixels.

## 2. Harris Algorithm

Harris Algorithm, one of the corner detection algorithms, is implemented during the lab.

Harris algorithm uses first order derivatives of the smoothed image to detect corners. Harris algorithm uses measure of corner response (cornerness function). According to this function, Harris Algorithm can be implemented in two ways.

**Harris Algorithm follows these steps:**

- 1) Image converted to greyscale
- 2) Noise reduction with Gaussian Filter (Smoothing Operation)
- 3) Computing Gradients of the image (Gx and Gy)
- 4) Creating an H matrix for each pixel
- 5) Compute corner function f or R.
- 6) if function value is larger than threshold consider this pixel as a corner

$$f = \frac{\lambda_1 \lambda_2}{\lambda_1 + \lambda_2}$$

$$= \frac{\text{determinant}(H)}{\text{trace}(H)}$$

*(f function)*

$$R = \det H - k(\text{trace} H)^2$$

*(R function)*

## 2.2 My Harris Algorithms Functions Explanations

In my implementation, I started as always by ensuring that the image is grey scale.

```
% Ensuring grey scale image
[row,col,ch] = size(img);
if(ch == 3)
    img = rgb2gray(img);
end
```

Then, I used Gaussian filter for smoothing operation. Smoothing is an important operation as the noise of the image may cause problematic results for the gradients of the image.

After, filtering our image with Gaussian Filter, I obtained gradients (Gx, Gy) of the image.

Then, I repeated the algorithm steps 4, 5, 6 with the following algorithm:

### Harris Algorithm with f

```
12 - [Gx, Gy] = imgradientxy(data);
13 - cornerlist = [];
14 - for i = k+1:row-k %Edges are zero
15 -     for j = k+1:col-k
16 -         my_window = Gx((i-k):(i+k), (j-k):(j+k));
17 -         my_window2 = Gy((i-k):(i+k), (j-k):(j+k));
18 -         IX = sum(sum(my_window.*my_window));
19 -         IY = sum(sum(my_window2.*my_window2));
20 -         IXY = sum(sum(my_window.*my_window2));
21 -
22 -         H = [IX, IXY; IXY, IY];
23 -         my_det = (H(1,1)*H(2,2)) - (H(1,2) * H(2, 1));
24 -         trace = H(1,1) + H(2,2);
25 -         f = my_det / trace;
26 -         if f > threshold
27 -             cornerlist = [cornerlist; [i,j]];
28 -         end
29 -     end
30 - end
```

### Harris Algorithm with R (const = 0.05)

```
12 - [Gx, Gy] = imgradientxy(data);
13 - cornerlist = [];
14 - for i = k+1:row-k %Edges are zero
15 -     for j = k+1:col-k
16 -         my_window = Gx((i-k):(i+k), (j-k):(j+k));
17 -         my_window2 = Gy((i-k):(i+k), (j-k):(j+k));
18 -         IX = sum(sum(my_window.*my_window));
19 -         IY = sum(sum(my_window2.*my_window2));
20 -         IXY = sum(sum(my_window.*my_window2));
21 -
22 -         H = [IX, IXY; IXY, IY];
23 -         my_det = (H(1,1)*H(2,2)) - (H(1,2) * H(2, 1));
24 -         trace = H(1,1) + H(2,2);
25 -         R = my_det - const*(trace^2);
26 -         if R > threshold
27 -             cornerlist = [cornerlist; [i,j]];
28 -         end
29 -     end
30 - end
```

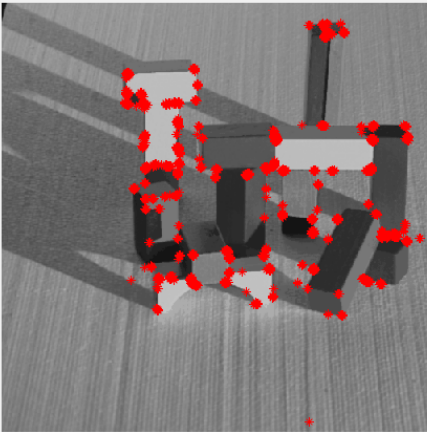
For each pixel, I took a window with size  $(2k+1)*(2k+1)$ . I computed  $I_x$  and  $I_y$  image gradients of window along x and y directions to create H matrix. Then, I computed corner response function values. If the corner response function value is higher than threshold, I added this pixel location to my corner list.



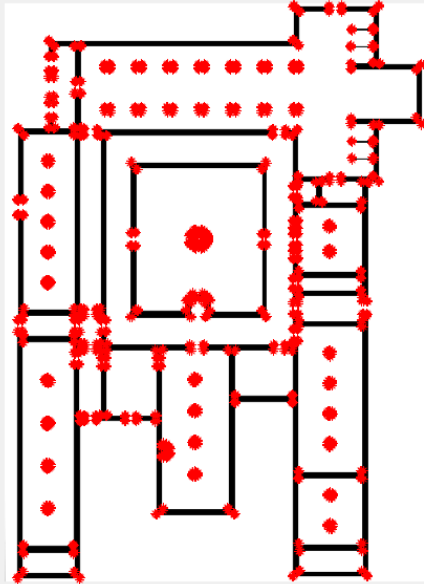
## 2.3 Harris Algorithms with F Results

Harris Algorithm with 'f' function with different threshold values and window size:

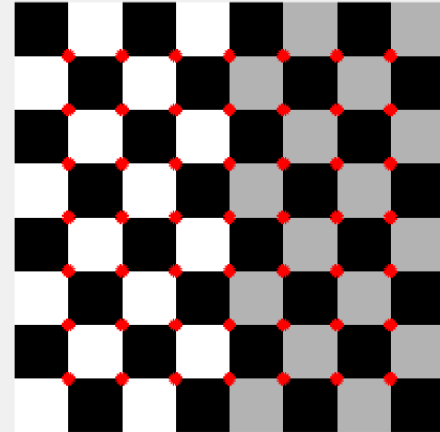
Threshold: 20000 Window Size: 3x3



Threshold: 20000 Window Size: 3x3



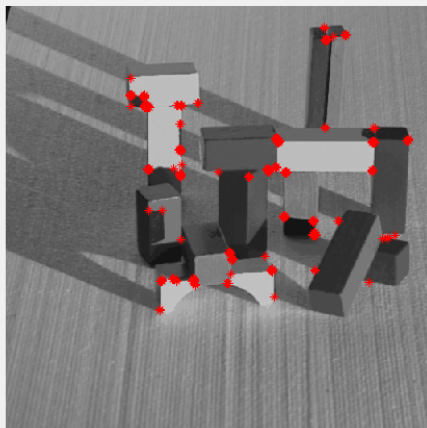
Threshold: 2 Window Size: 3x3



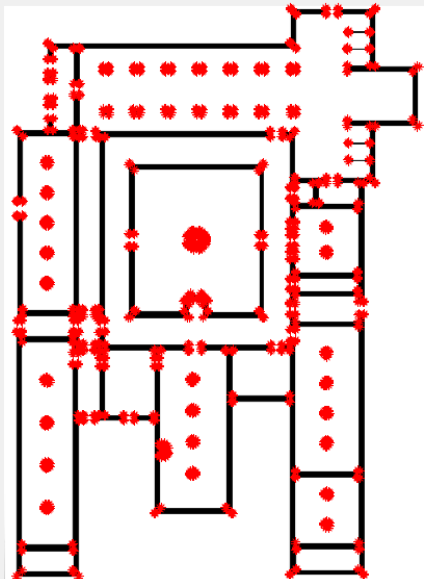
---

Harris with f Algorithm for Corner Detection

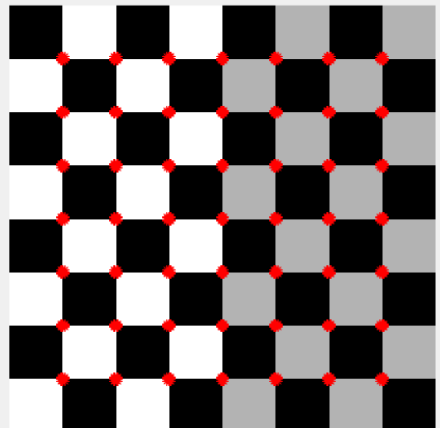
Threshold: 50000 Window Size: 3x3



Threshold: 50000 Window Size: 3x3



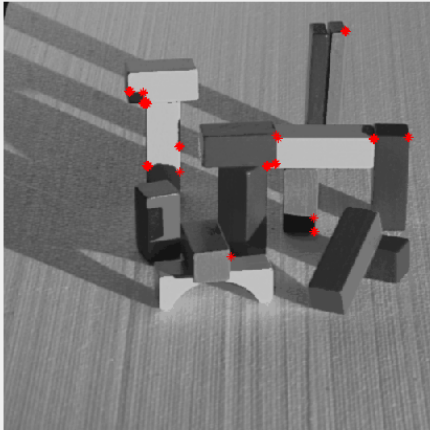
Threshold: 5 Window Size: 3x3



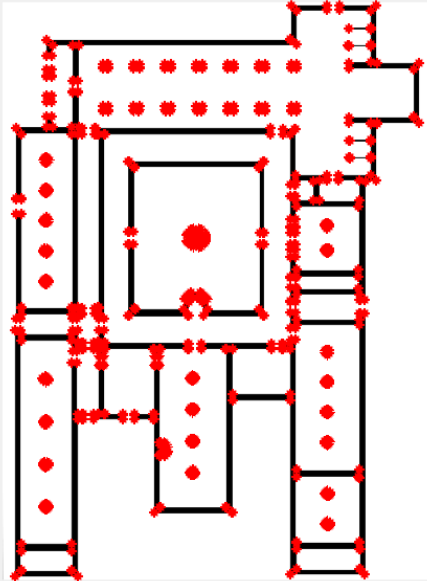


Harris with f Algorithm for Corner Detection

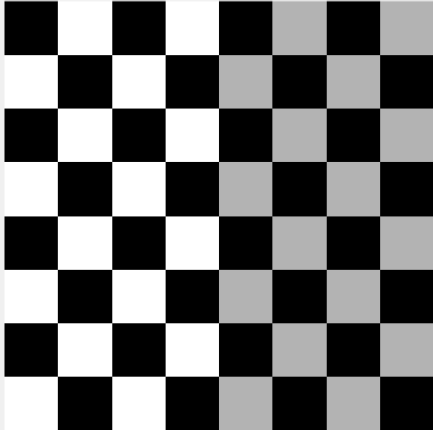
Threshold: 100000 Window Size: 3x3



Threshold: 100000 Window Size: 3x3

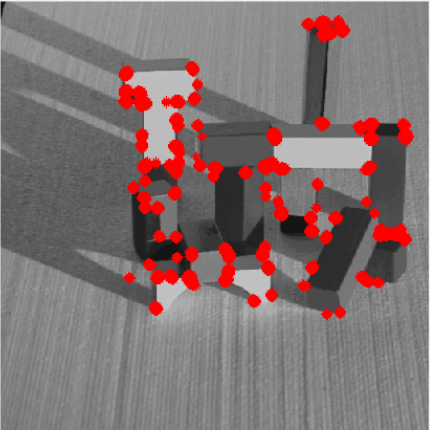


Threshold: 20 Window Size: 3x3

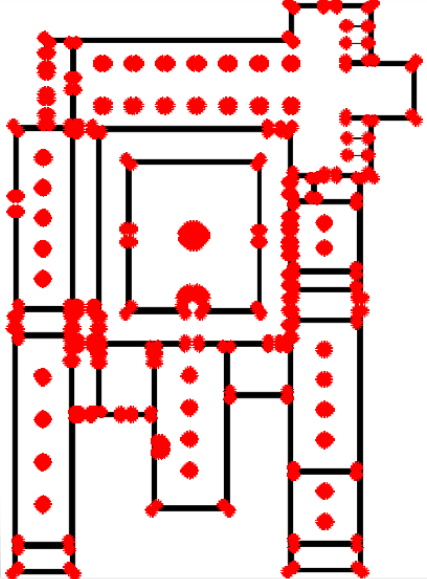


Harris with f Algorithm for Corner Detection

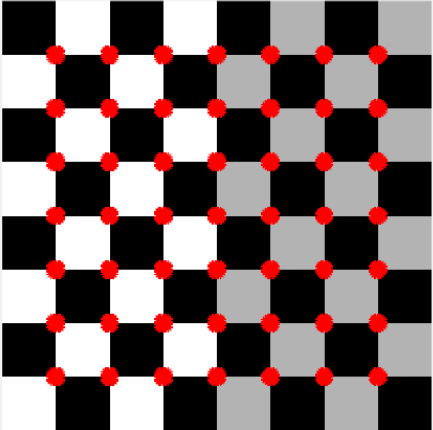
Threshold: 100000 Window Size: 7x7



Threshold: 100000 Window Size: 7x7



Threshold: 20 Window Size: 7x7

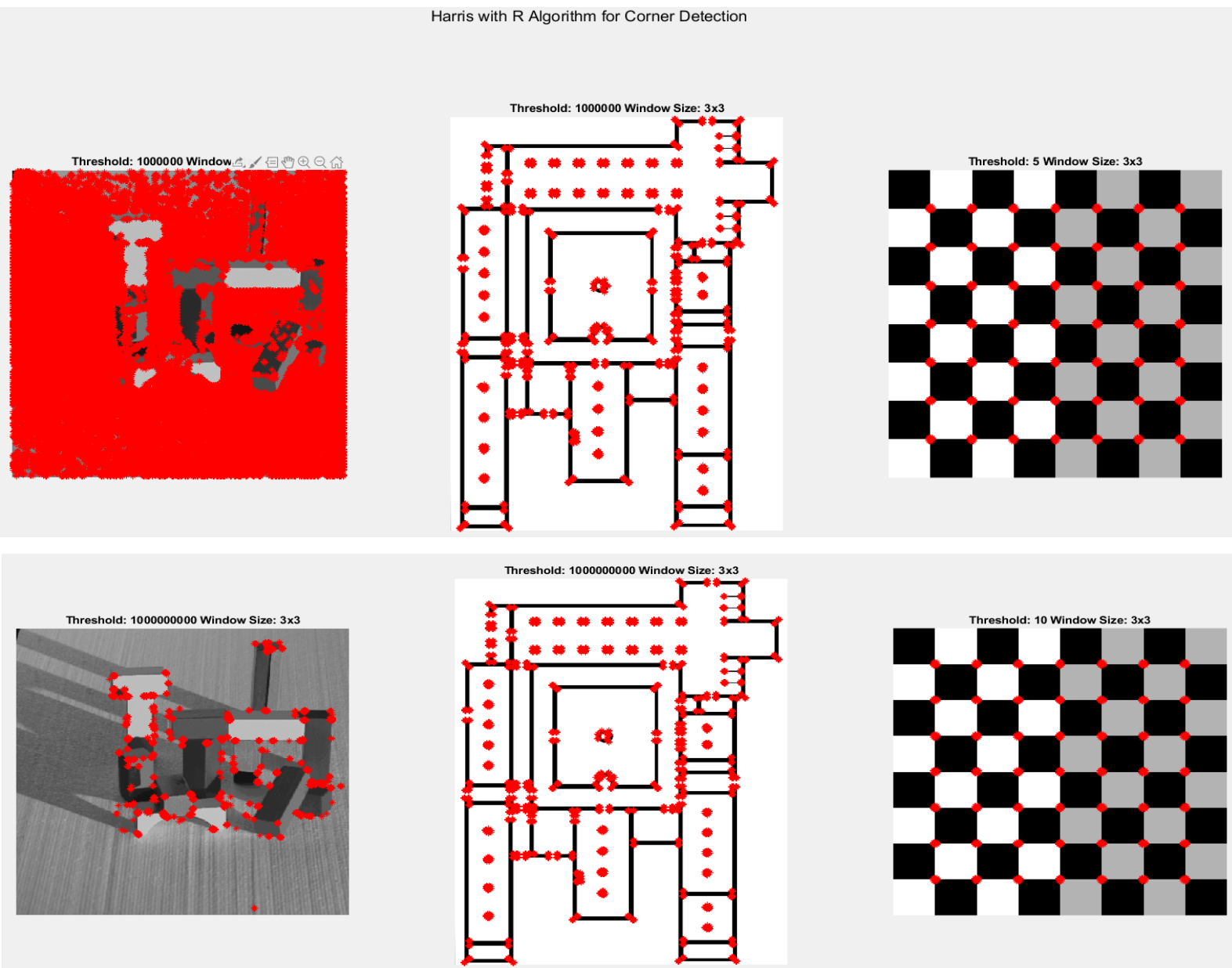


New Image (House.png) with different threshold values for Harris with f function:



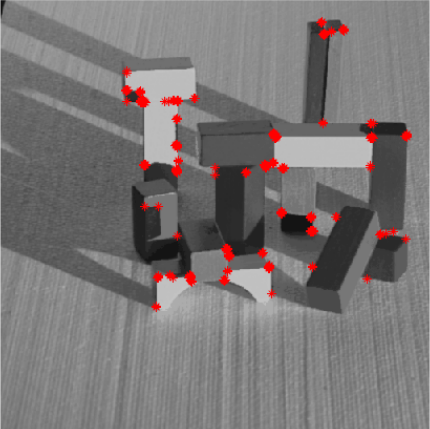
2.4 Harris Algorithms with R Function Results

Harris Algorithm with ‘R’ function with different threshold values and window size:

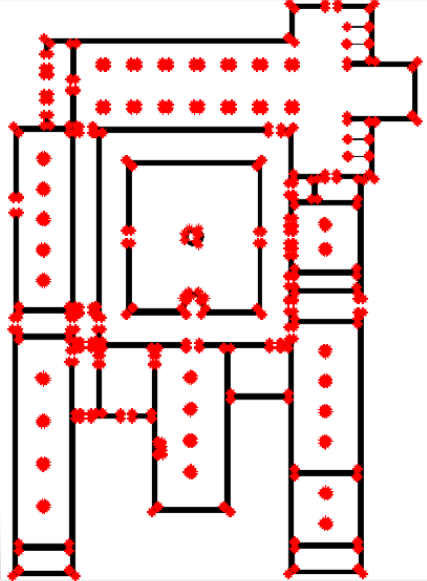


Harris with R Algorithm for Corner Detection

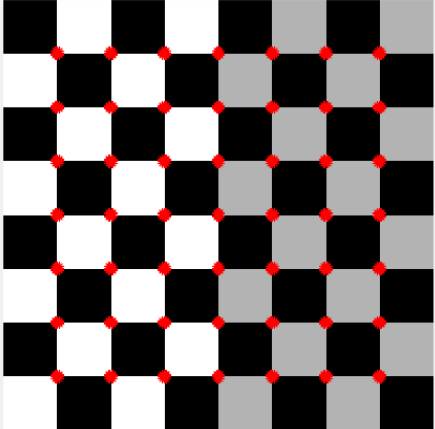
Threshold: 10000000000 Window Size: 3x3



Threshold: 10000000000 Window Size: 3x3

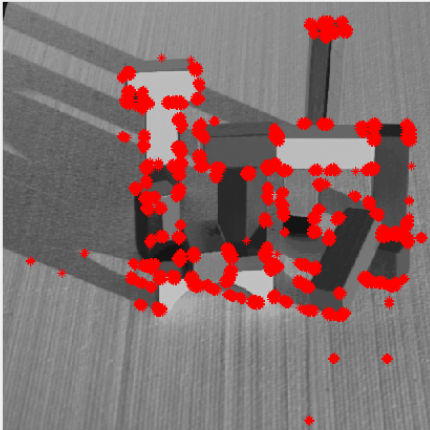


Threshold: 20 Window Size: 3x3

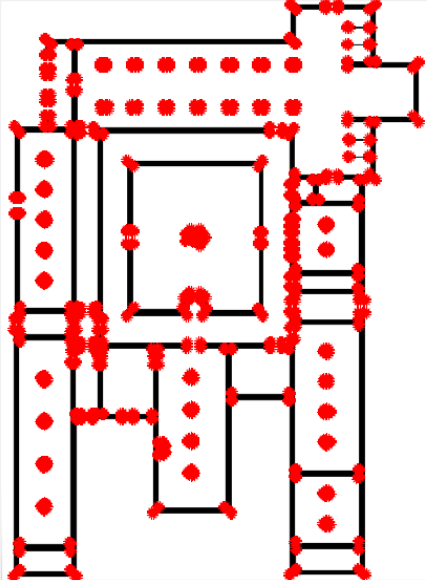


Harris with R Algorithm for Corner Detection

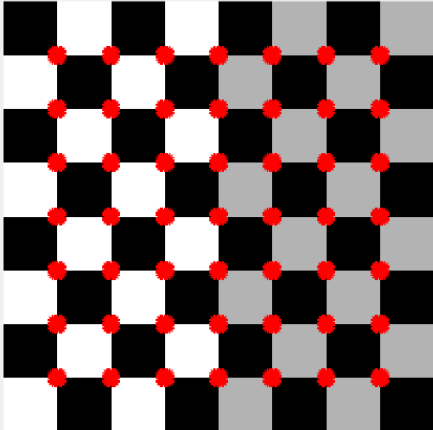
Threshold: 10000000000 Window Size: 7x7



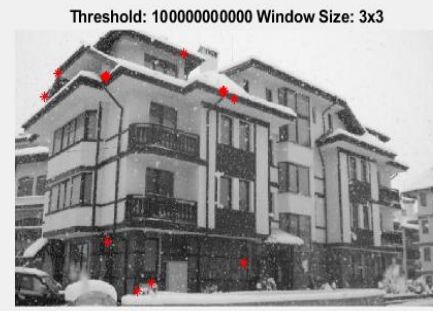
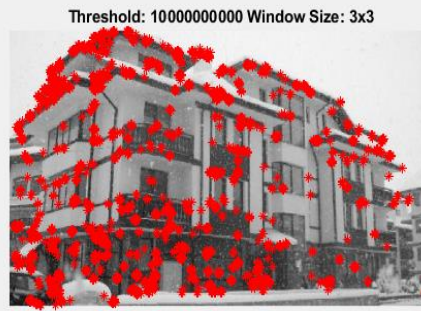
Threshold: 10000000000 Window Size: 7x7



Threshold: 20 Window Size: 7x7



### New Image (House.png) with different threshold values for Harris with R function:



## 2.5 Harris Algorithms Discussion

We can see that corners in the images can be detected by Harris Algorithm. Threshold value is crucial for this algorithm too. I will not repeat myself as I explained the effect of threshold in the Kanade-Tomasi Discussion part.

We also observed that when we increased the window size, we can detect more corner pixels with same threshold. For instance, Checkerboard image with threshold 20 and 3x3 ( $k=1$ ) window could not detect any corners in the image. However, when we increased our window size to the 7x7 ( $k=3$ ), Harris Algorithm managed to catch all corner pixels.

## 2.6 Comparison of Harris Algorithm and Kanade-Tomasi Algorithm

We observed that both algorithms can find the corners in the image. Both of them is sensitive to threshold value. However, they have differences with each other.

Firstly, we do not need to compute eigen values in the Harris Algorithm which reduces the cost of algorithm because we need more mathematical operations to calculate eigen values such as square root.

Secondly, their outputs can be different for same threshold and window size. For instance for the CheckerBoard image, Kanade-Tomasi with threshold 20 and  $k = 1$  is able to catch corners

in the left part of the image. However, Harris Algorithm with  $f$  function could not catch any corners in the image with the same parameters.

Thirdly, we can compare the running time of each algorithm with inbuilt tic-toc functions of MATLAB. I used same threshold value and input image for both functions and record their run times as following way:

```
% % Compare Harris and Kanade-Tomasi
k = 1;
tStart1 = tic;
[new_img, cornerlist1] = lab4ktcorners(house, 10^9, k);
tEnd1 = toc(tStart1)

tStart2 = tic;
[new_img, cornerlist2] = lab4HarrisCorners(house, 10^9, k);
tEnd2 = toc(tStart2)

tStart3 = tic;
[new_img, cornerlist3] = lab4HarrisCorners_R(house, 10^9, k);
tEnd3 = toc(tStart3)
```


## OUTPUT:

## Number of Found Corners for each algorithm:

```
tEnd1 =
    3.7499

tEnd2 =
    2.6582

tEnd3 =
    3.8704
```

	cornerlist1	<code>[]</code>
	cornerlist2	<code>[]</code>
	cornerlist3	<code>27806x2 double</code>

**OBSERVATION:** Even we could not any possible corner pixels with Kanade-Tomasi, its running time is approximately equal to Harris Algorithm which have found 27806 pixel points. This is because of computation of eigenvalues in Kanade-Tomasi. Therefore, we can say that Harris Algorithm has much less complexity compared to Kanade-Tomasi.

## 2.7 Comparison of Harris Algorithm with different corner functions

Firstly, we can see that Harris Algorithm with R function works with higher threshold values compared to Harris Algorithm which uses f function. For instance, threshold as around  $10^9$  needed for catching corners in the blocks image properly in the Harris Algorithm with R function. On the other hand, we were able to catch corners in the blocks image with threshold around 50000.

Secondly, we can compare the running time of each function with inbuilt tic-toc functions of MATLAB. I used same threshold value and input image for both functions and record their run times as following way:

CODE	OUTPUT
<pre> k = 1; tStart1 = tic; [new_img, cornerlist] = lab4Harrisorners_R(house, 10^9, k); tEnd1 = toc(tStart1)   tStart2 = tic; [new_img, cornerlist] = lab4Harrisorners(house, 10^9, k); tEnd2 = toc(tStart2) </pre>	<pre> tEnd1 =     3.2634  tEnd2 =     2.7468 </pre>

We can see that Harris with R is taking more time as it finds more possible corner points compared to Harris with f function.