EE 417 – COMPUTER VISION
LABORATORY #3
POST-LAB REPORT

GOKTUG KORKULU
27026

1) KANADE-TOMASI ALGORITHM

Implementation and Description:

```
1  function [C] = lab3ktcorners (I, k, T)
2
3
4  [row,col,ch] = size(I);
5  if (ch == 3)
6      I = rgb2gray(I);
7  end
8  I = double(I);
9
10
11 %Step 1: smoothen the greyscaled image with Gaussian filter.
12 I_smoothed = imgaussfilt(I);
13
14 %Step 2: compute image gradients
15 [Ix, Iy] = imgradientxy(I_smoothed);
16
17 %initialize the array which keeps the corner values inside.
18 C=[];
19
20
21 for i = (k+1):(2*k+1):(row-k)
22     for j = (k+1):(2*k+1):(col-k)
23
24         sub_img_x = Ix(i-k:i+k, j-k:j+k);
25         sub_img_y = Iy(i-k:i+k, j-k:j+k);
26
27         %Step 3: corner matrix H
28         H = [sum(sum(sub_img_x.*sub_img_x)) sum(sum(sub_img_x.*sub_img_y)); sum(sum(sub_img_x.*sub_img_y)) sum(sum(sub_img_y.*sub_img_y))];
29
30         %Step 4: eigenvalues
31         eigs = eig(H);
32         lamda1 = eigs(1,1);
33         lamda2 = eigs(2,1);
34
35         %if minimum of eigenvalues is greater than the threshold
36         if (min(lamda1,lamda2) > T)
37             C = [C; i j];
38         end
39     end
40 end
41
42 end
```

Figure 1: kanade-tomasi algorithm function

- This Kanade-Tomasi function that called ***lab3ktcorners*** is a function takes an ***image(I)***, ***window size(k)***, and ***threshold value(T)*** as inputs and returns the detected corner points of the input image as ***an array([C])***. (Shown in 1,2,3).
- Then, assigns the row, column, and channel values of input image to variables called ***row, col, ch*** respectively. It detects if the image is RGB or gray-scaled by checking the ***ch*** value. If it is RGB, then it converts the image to gray-scaled version; if already gray-scaled, leaves as it is. Finally, it converts the pixel values of gray-scaled converted image from integer to double so that when we perform mathematical operations to pixel values, we will not lose decimal values. (Shown in 4).
- After all these steps, the gray-scaled double valued image is smoothed by Gaussian Smoothing Filter. In order to achieve this goal, we utilize ***imgaussfilt()*** MATLAB built-in function whose input is gray-scaled double image I and output I_smoothed. (Shown in 5).
- Consecutively, we compute image gradients Ix and Iy by utilizing MATLAB built-in function called ***imgradientxy()*** whose input is gray-scaled doubled smoothed image I_smoothed and outputs are image gradients with respect to x and y; Ix and Iy computed with Sobel gradient operator. (Shown in 6).
- Before starting integral processing on the input image, we create empty vector ***C*** which will have the corner list at the end will be returned from the function. (Shown in 7).
- In order to detect the corners, we have to analyze each pixel by applying some operations to their neighbors. For that reason, we need to go and visit each of pixels of the image one by one. Hence, we use 2 for loops nested to each other. However, since we detect the corners and need only one single pixel within the neighbors, we increase the index values of pixels by window size rather than one on each step. This protects us to have several corner points in the given window size, so we have neater output (Shown in 8). In order to understand more, let's look below example.

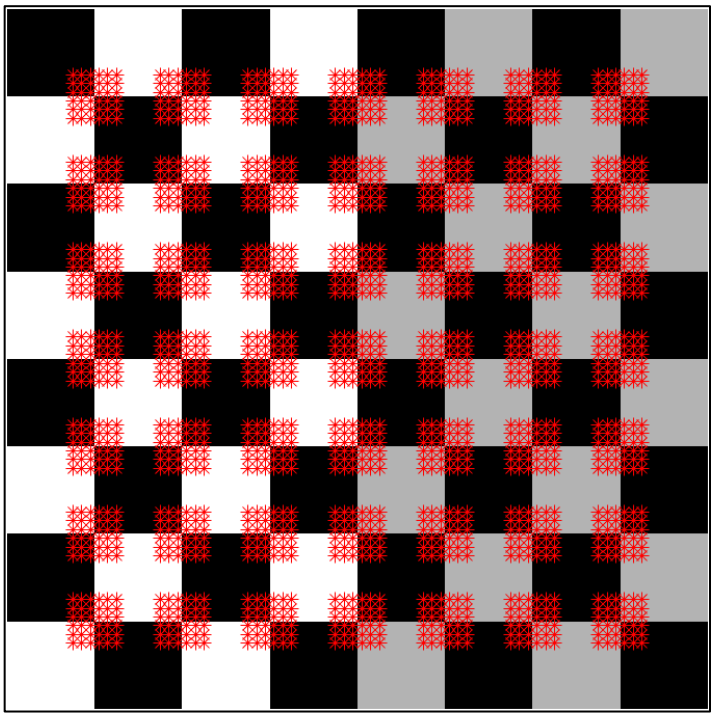


Figure 2: index values increased by 1

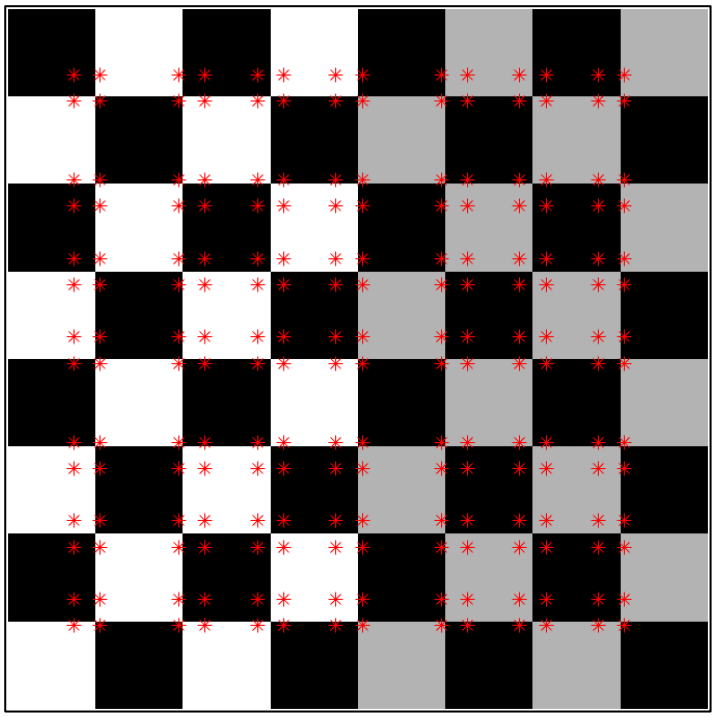


*Figure 3: index values increased by window size (2*k+1)*

Since we increase the pixel indexes one by one on the left-hand side example, algorithm marks each pixel within the window size as corner points. Thus, we end up with having redundant corner points than what we need. In order to avoid this situation to occur, we increase the pixel indexes by window size (2*k+1). You can see this implementation in white shaded area in above code labeled as 8.

- Inside the for loop and each iteration of for loop, we create two sub images. This time, sub images are gradient values with respect to x and y which are cropped by window size centered at (i,j). (Shown in 9).
- Right after calculating X and Y gradient values, we compute the corner matrix H whose equation has been given us in the lab documentation as;

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

Figure 4: corner matrix equation

  In this line of code, we utilize dot product in order to multiply X and Y gradient values by their selves or each other. (Shown in 10).
- After we obtain the corner matrix H, we calculate its eigenvalues respectively. Thankfully, there is a MATLAB built-in function called ***eig()*** which takes corner matrix as input and returns eigenvalues to variable called eigs. Since eigs is 2x1 matrix which hold eigenvalues of corner matrix H, for obtaining the eigenvalues separately, I perform 2 lines of codes as seen. (Shown in 11).
- As the final step, we use threshold value on eigenvalues to detect corners such that “if minimum of eigenvalues is greater than the threshold value”, the pixel coordinates are added to the corner list C. (Shown in 12).

OUTPUTS OF IMPLEMENTATION:

```
clear; close all; clc;
img_blocks = imread("Lab3 - Images/blocks.png");
img_lab = imread("Lab3 - Images/lab.png");
img_notredame = imread("Lab3 - Images/notredame.jpg");

```

Figure 5: reading the images in main function

- First of all, in the main function, I start reading the images that were given in the lab3 document folder and assign those to new variables called ***img_blocks, img_lab*** and ***img_notredame.*** (Figure 5)

Investigation on blocks.png image:

```
%% Kanade-Tomasi Algorithm
th = 1000;
Corners = lab3ktcorners(img_blocks, 1, th);

figure;
imshow(img_blocks);
hold on;
plot(Corners(:,2), Corners(:,1), 'r*');
xlabel("Threshold value is " + th);
```

```
%% Kanade-Tomasi Algorithm
th = 6000;
Corners = lab3ktcorners(img_blocks, 1, th);

figure;
imshow(img_blocks);
hold on;
plot(Corners(:,2), Corners(:,1), 'r*');
xlabel("Threshold value is " + th);
```

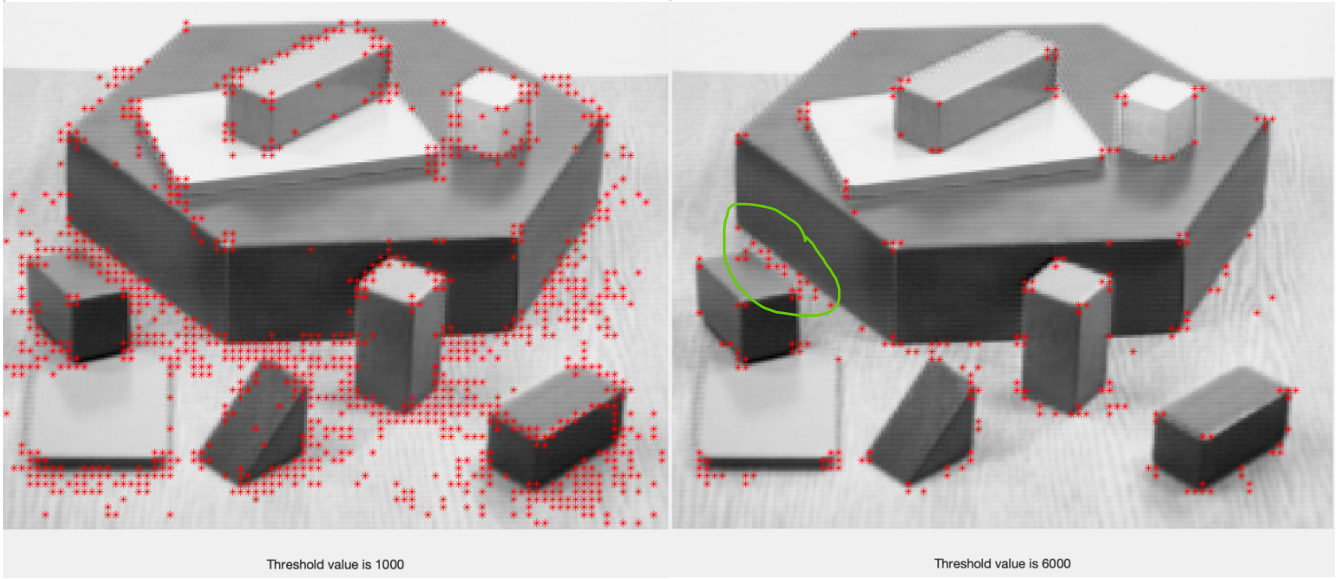


Figure 6: The comparison of threshold values on kanade-tomasi algorithm.

- Here, I have altered the threshold value on ***img_blocks*** and tried th=1000 and th=6000. As it is seen, on the left-hand side, the threshold value seems like too low so there are a lot of corner points indicated by little red stars which are not a real corner points actually. On the other hand, although there are still some minor problems such as finding not corner points as corners as in circled area, the threshold value seems like enough high so that we can utilize that value as threshold. (Figure 6)

```
%% Kanade-Tomasi Algorithm
k=1;
Corners = lab3ktcorners(img_blocks, k, 6000);

figure;
imshow(img_blocks);
hold on;
plot(Corners(:,2), Corners(:,1), 'r*');
xlabel("Window size is " + uint8((k*2)+1));
```

```
%% Kanade-Tomasi Algorithm
k=3;
Corners = lab3ktcorners(img_blocks, k, 6000);

figure;
imshow(img_blocks);
hold on;
plot(Corners(:,2), Corners(:,1), 'r*');
xlabel("Window size is " + uint8((k*2)+1));
```

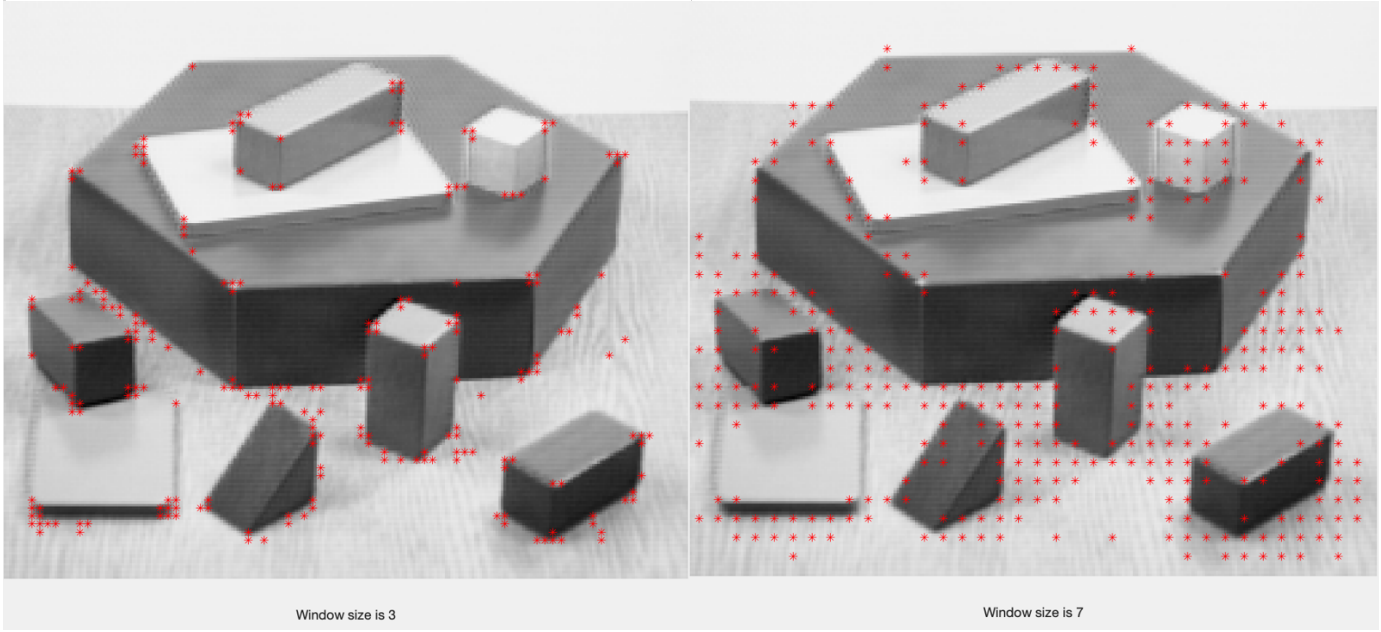


Figure 7: The comparison of window sizes on kanade-tomasi algorithm

- Here, I have used different window sizes so that we can choose the best window size fits our algorithm for this specific input. Here on the right-hand side, we see the window size is too large, so the corner detection creates a lot of mistakes. However, on the left-hand side we have small window size so that we obtain relatively better corner detection. Notice that also the distance between corner points (indicated by tiny red stars) increases when the window size increases. This means also we're increasing the ratio of mistake on detecting the corners' exact location. (Figure 7)
- As a result, I found threshold value ***6000*** and window size ***3*** (which implies k = 1) is the best values for blocks.png image on kanade-tomasi algorithm.

Investigation on lab.png image:

```
%% Kanade-Tomasi Algorithm
th = 1000;
Corners = lab3ktcorners(img_lab, 1, th);

figure;
imshow(img_lab);
hold on;
plot(Corners(:,2), Corners(:,1), 'r*');
xlabel("Threshold value is " + th);
```

```
%% Kanade-Tomasi Algorithm
th = 8000;
Corners = lab3ktcorners(img_lab, 1, th);

figure;
imshow(img_lab);
hold on;
plot(Corners(:,2), Corners(:,1), 'r*');
xlabel("Threshold value is " + th);
```

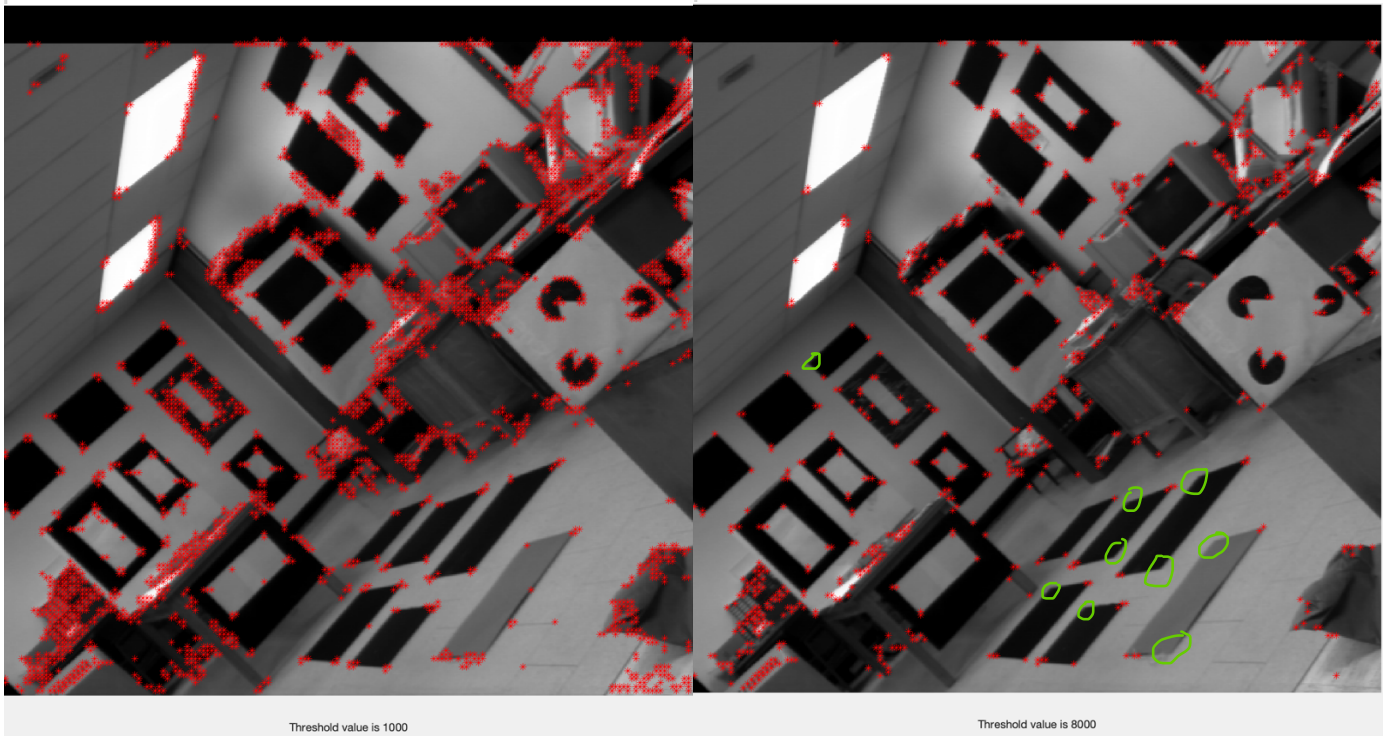


Figure 8: The comparison of threshold values on kanade-tomasi algorithm.

- Here, I have altered the threshold value on ***img_lab*** and tried th=1000 and th=8000. As it is seen, on the left-hand side, the threshold value seems like too low so there are a lot of corner points indicated by little red stars which are not a real corner point actually. On the other hand, although there are still some minor problems such as not detecting some corners as in circled areas, the threshold value seems like enough high so that we can utilize that value as threshold. (Figure 8)

```
%% Kanade-Tomasi Algorithm
k=2;
Corners = lab3ktcorners(img_lab, k, 8000);

figure;
imshow(img_lab);
hold on;
plot(Corners(:,2), Corners(:,1), 'r*');
xlabel("Window size is " + uint8((k*2)+1));
```

```
%% Kanade-Tomasi Algorithm
k=1;
Corners = lab3ktcorners(img_lab, k, 8000);

figure;
imshow(img_lab);
hold on;
plot(Corners(:,2), Corners(:,1), 'r*');
xlabel("Window size is " + uint8((k*2)+1));
```

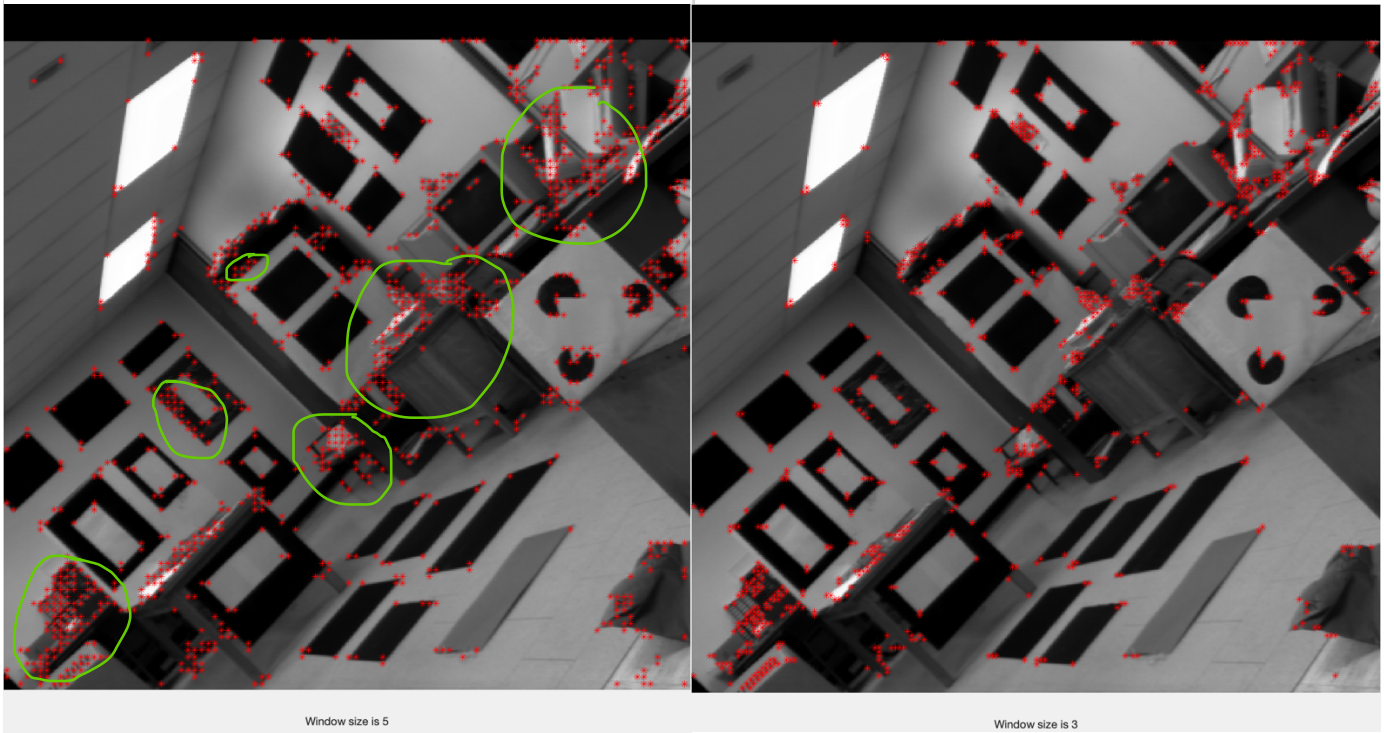


Figure 9: The comparison of window sizes on kanade-tomasi algorithm.

- Here, I have used different window sizes so that we can choose the best window size fits our algorithm for this specific input. Here on the left-hand side, we see the window size is too large, so the corner detection creates a lot of mistakes as seen in circled areas. However, on the right-hand side we have small window size so that we obtain relatively better corner detection. Notice that also the distance between corner points (indicated by tiny red stars) increases when the window size increases. This means also we're increasing the ratio of mistake on detecting the corners' exact location. (Figure 9)
- As a result, I found threshold value ***8000*** and window size ***3*** (which implies k = 1) is the best values for lab.png image on kanade-tomasi algorithm.

Investigation on notredame.jpg image:

```
%% Kanade-Tomasi Algorithm
th=10000;
Corners = lab3ktcorners(img_notredame, 1, th);

figure;
imshow(img_notredame);
hold on;
plot(Corners(:,2), Corners(:,1), 'r*');
xlabel("Threshold value is " + th);
```

```
%% Kanade-Tomasi Algorithm
th=75000;
Corners = lab3ktcorners(img_notredame, 1, th);

figure;
imshow(img_notredame);
hold on;
plot(Corners(:,2), Corners(:,1), 'r*');
xlabel("Threshold value is " + th);
```

Figure 10: The comparison of threshold values on kanade-tomasi algorithm.

- Here, I have altered the threshold value on ***img_notredame*** and tried th=1000 and th=8000. As it is seen, on the left-hand side, the threshold value seems like too low so there are a lot of corner points indicated by little red stars which are not a real corner point actually. On the other hand, although it seems like it still detects a lot of points which are not corners, you can see the zoomed in version on the appendix A of the report that they are corners actually. Therefore, the threshold value seems like enough high so that we can utilize that value as threshold. (Figure 10)

```
%% Kanade-Tomasi Algorithm
k=3;
Corners = lab3ktcorners(img_notredame, k, 75000);

figure;
imshow(img_notredame);
hold on;
plot(Corners(:,2), Corners(:,1), 'r*');
xlabel("Window size is " + uint8((k*2)+1));
```

```
%% Kanade-Tomasi Algorithm
k=1;
Corners = lab3ktcorners(img_notredame, k, 75000);

figure;
imshow(img_notredame);
hold on;
plot(Corners(:,2), Corners(:,1), 'r*');
xlabel("Window size is " + uint8((k*2)+1));
```

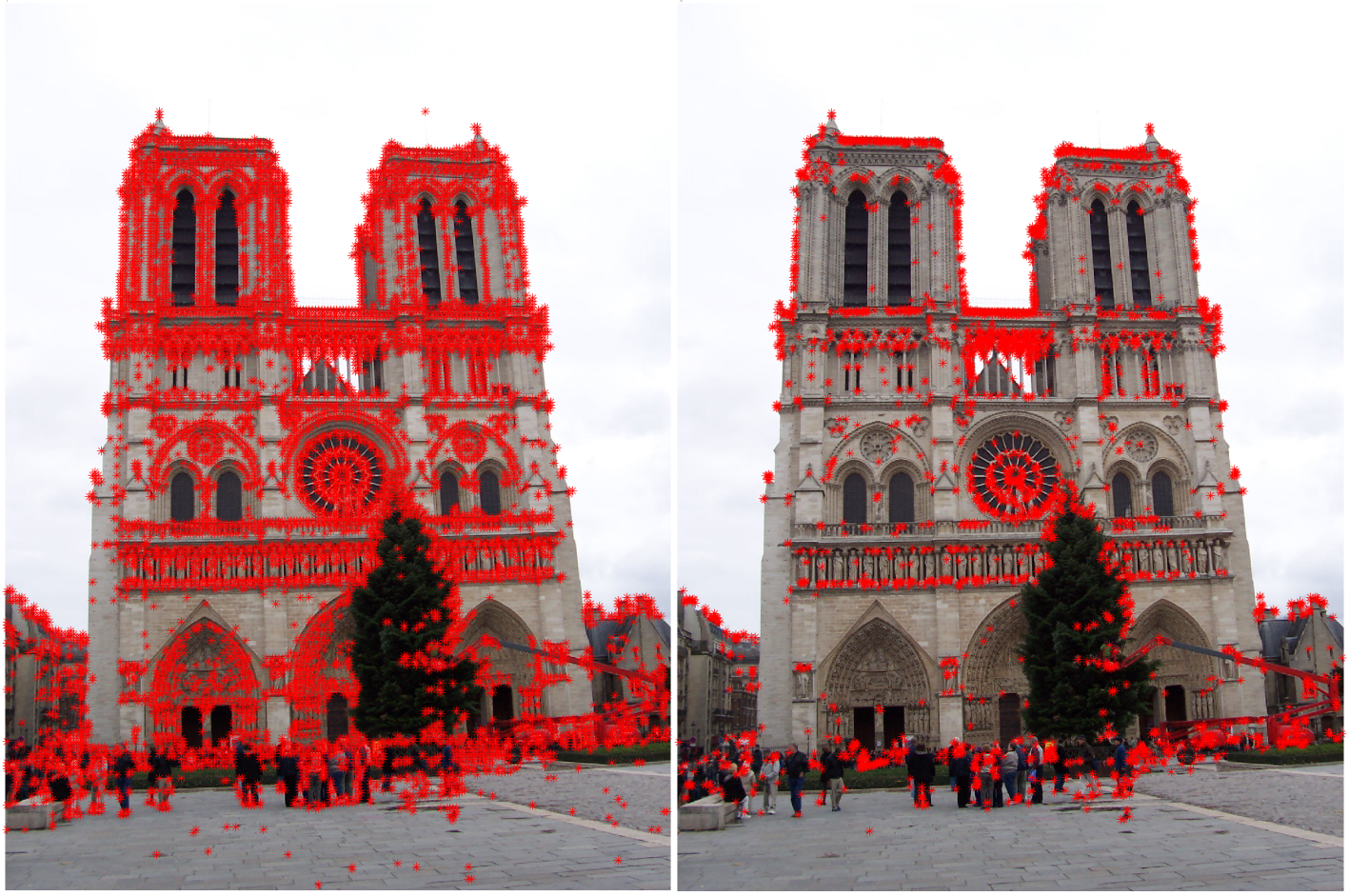


Figure 11: The comparison of window sizes on kanade-tomasi algorithm.

- Here, I have used different window sizes so that we can choose the best window size fits our algorithm for this specific input. Here on the left-hand side, we see the window size is too large, so the corner detection creates a lot of mistakes as seen everywhere on the image. However, on the right-hand side, we have small window size so that we obtain relatively better corner detection. Notice that also the distance between corner points (indicated by tiny red stars) increases when the window size increases. This means also we're increasing the ratio of mistake on detecting the corners' exact location. (Figure 11)
- As a result, I found threshold value ***75000*** and window size ***3*** (which implies k = 1) is the best values for notredame.png image on kanade-tomasi algorithm.

2) HARRIS ALGORITHM USING DETERMINANT AND TRACE

Implementation and Description:

```
function [C] = lab3Harriscorners (I, k, T)

[row,col,ch] = size(I);
if (ch == 3)
    I = rgb2gray(I);
end
I = double(I);


%Step 1: smoothen the greyscaled image with Gaussian filter.
I_smoothed = imgaussfilt(I);

%Step 2: compute image gradients
[Ix, Iy] = imgradientxy(I_smoothed);

%initialize the array which keeps the corner values inside.
C=[];


for i = (k+1):(2*k+1):(row-k)
    for j = (k+1):(2*k+1):(col-k)

        sub_img_x = Ix(i-k:i+k, j-k:j+k);
        sub_img_y = Iy(i-k:i+k, j-k:j+k);

        %Step 3: corner matrix H
        H = [sum(sum(sub_img_x.*sub_img_x)) sum(sum(sub_img_x.*sub_img_y)); sum(sum(sub_img_x.*sub_img_y)) sum(sum(sub_img_y.*sub_img_y))];

        %Step 4: compute f
        f = det(H)/trace(H);

        %if f is greater than the threshold
        if (f > T)
            C = [C; i j];
        end
    end
end

end
```

Figure 12: Harris Algorithm using determinant and trace on detecting corners.

- This Harris Algorithm function that called ***lab3Harriscorners*** is a function takes an ***image(I)***, ***window size(k)***, and ***threshold value(T)*** as inputs and returns the detected corner points of the input image as ***an array([C])***. (Shown in 1,2,3).
- Then, assigns the row, column, and channel values of input image to variables called ***row, col, ch*** respectively. It detects if the image is RGB or gray-scaled by checking the ***ch*** value. If it is RGB, then it converts the image to gray-scaled version; if already gray-scaled, leaves as it is. Finally, it converts the pixel values of gray-scaled converted image from integer to double so that when we perform mathematical operations to pixel values, we will not lose decimal values. (Shown in 4).
- After all these steps, the gray-scaled double valued image is smoothed by Gaussian Smoothing Filter. In order to achieve this goal, we utilize ***imgaussfilt()*** MATLAB built-in function whose input is gray-scaled double image I and output I_smoothed. (Shown in 5).
- Consecutively, we compute image gradients Ix and Iy by utilizing MATLAB built-in function called ***imgradientxy()*** whose input is gray-scaled doubled smoothed image I_smoothed and outputs are image gradients with respect to x and y; Ix and Iy computed with Sobel gradient operator. (Shown in 6).
- Before starting integral processing on the input image, we create empty vector ***C*** which will have the corner list at the end will be returned from the function. (Shown in 7).
- In order to detect the corners, we have to analyze each pixel by applying some operations to their neighbors. For that reason, we need to go and visit each of pixels of the image one by one. Hence, we use 2 for loops nested to each other. However, since we detect the corners and need only one single pixel within the neighbors, we increase the index values of pixels by window size rather than one on each step. This protects us to have several corner points in the given window size, so we have neater output (Shown in 8). In order to understand more, let's look figure2 and figure3 examples and the explanation.

- Inside the for loop and each iteration of for loop, we create two sub images. This time, sub images are gradient values with respect to x and y which are cropped by window size centered at (i,j). (Shown in 9).
- Right after calculating X and Y gradient values, we compute the corner matrix H whose equation has been given us in the lab documentation as;

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

Figure 2: corner matrix equation

In this line of code, we utilize dot product in order to multiply X and Y gradient values by their selves or each other. (Shown in 10).

- After we calculate the corner matrix H, we calculate f by utilizing given equation;

$$f = det(H)/trace(H)$$

Figure 13: calculating f

Where det() is determinant and trace() is the sum of diagonals of the given matrix. (Shown in 11).

- As the final step, we use threshold value on f to detect corners such that "if f is greater than the threshold value", the pixel coordinates are added to the corner list C. (Shown in 12).

OUTPUTS OF IMPLEMENTATION:

```
1   clear; close all; clc;
2   img_blocks = imread("Lab3 - Images/blocks.png");
3   img_lab = imread("Lab3 - Images/lab.png");
4   img_notredame = imread("Lab3 - Images/notredame.jpg");
5
```

Figure 14: reading the images in main function

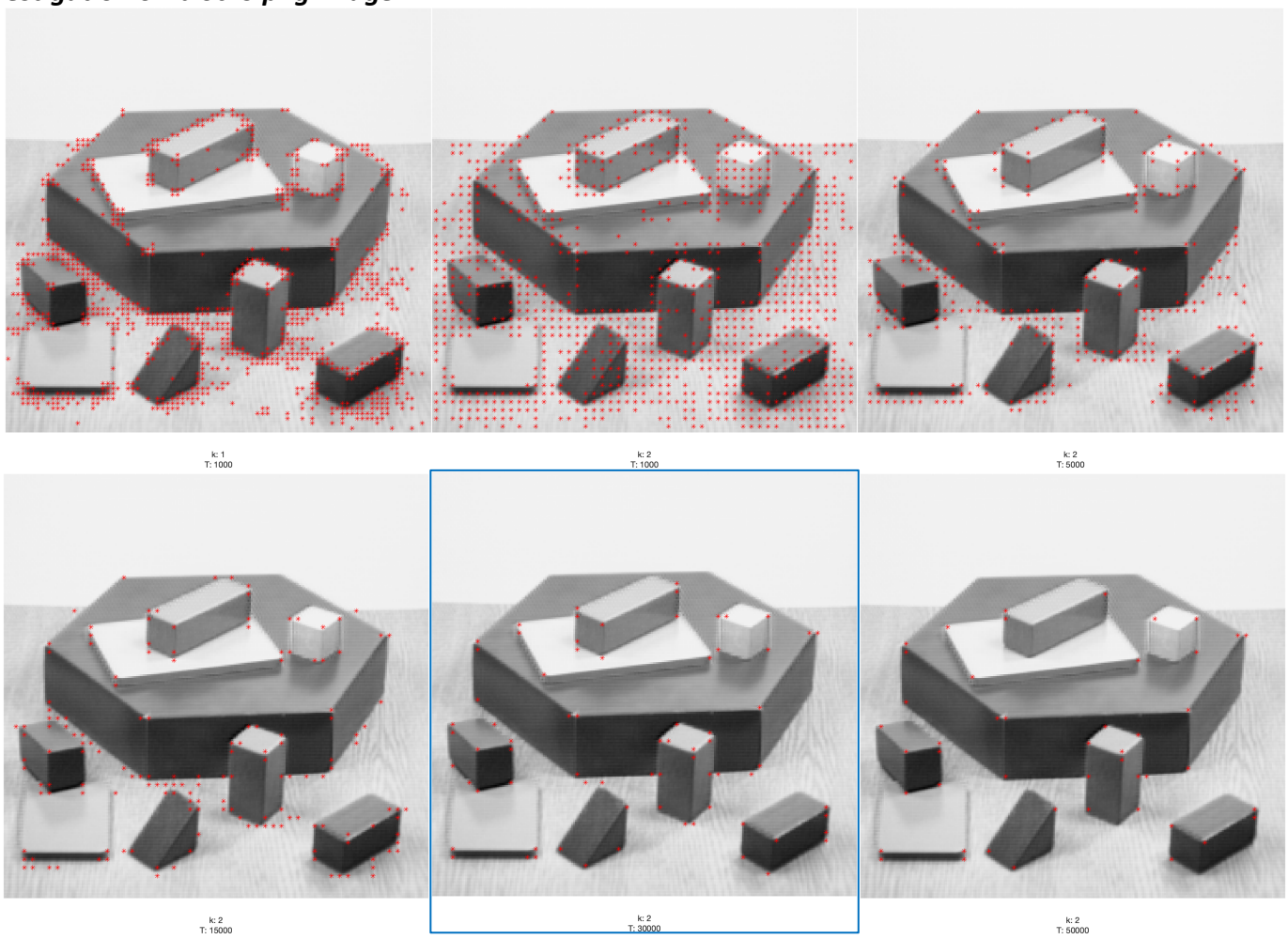
- First of all, in the main function, I start reading the images that were given in the lab3 document folder and assign those to new variables called ***img_blocks, img_lab*** and ***img_notredame.***

```
%% Harris Algorithm
th2=1000;
k2=1;
Corners2 = lab3Harriscorners(img_blocks, k2, th2);

figure;
imshow(img_blocks);
hold on;
plot(Corners2(:,2), Corners2(:,1), 'r*');
xlabel( {'k: ' + string(k2), 'T: '+ string(th2)} );
```

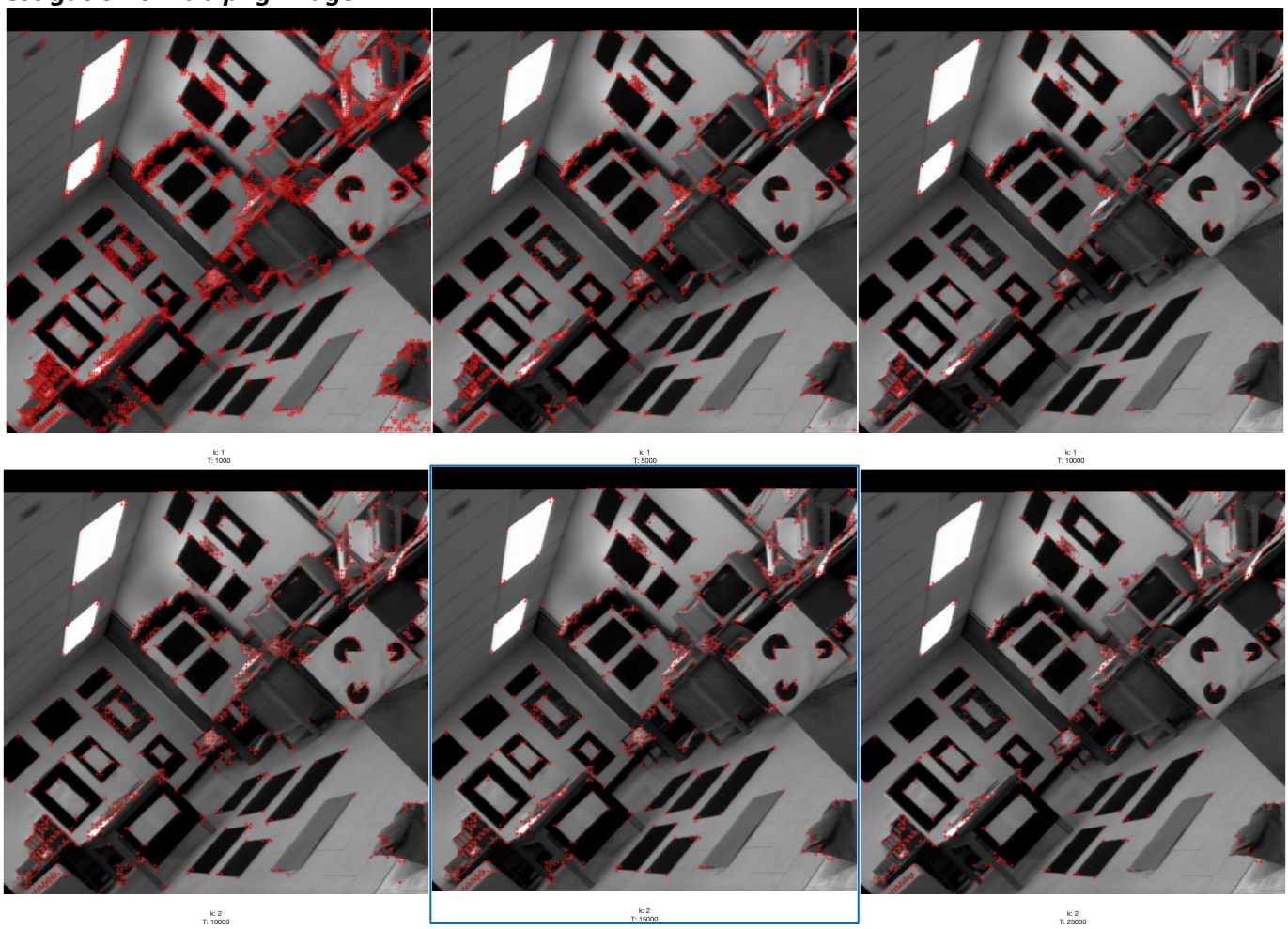
- This is generic Harris algorithm call from main function. Th2, k2 and images are variable so that when comparing some values, these variables will change.

Investigation on blocks.png image:



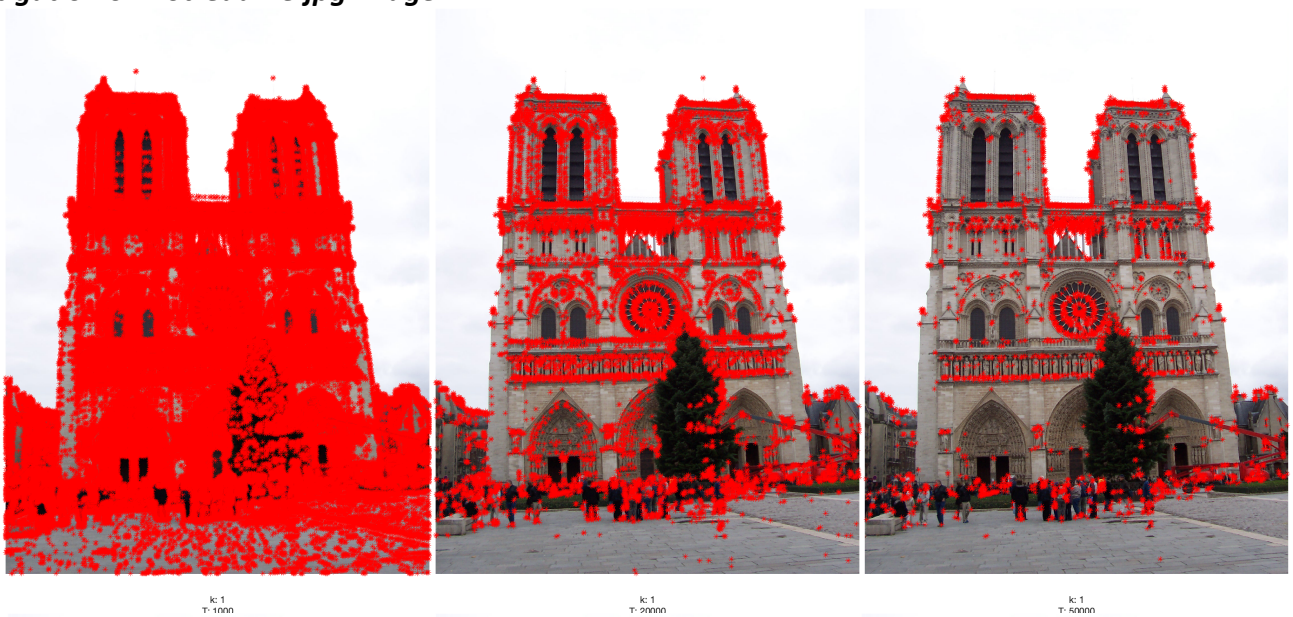
- First, I started with giving k:1, T:1000. Then I increased k by 1 and saw that it is getting worse. So, then I needed either decrease k again or increase T instead. I chose increasing T with k:2 fixed and obtained 5 different results according to their T values. Apparently, when k:2, T:50000 is too high and T:5000 is still too low. According to one's decision, T:15000, T:30000 or tome T value in between can be chosen regarding one's choice because T:15000 is still having non-corner points detected and T:30000 is missing some corner points. It is kind of a tradeoff and choose depending on which is more valuable to you. My own preference would be k:2 and T:30000.

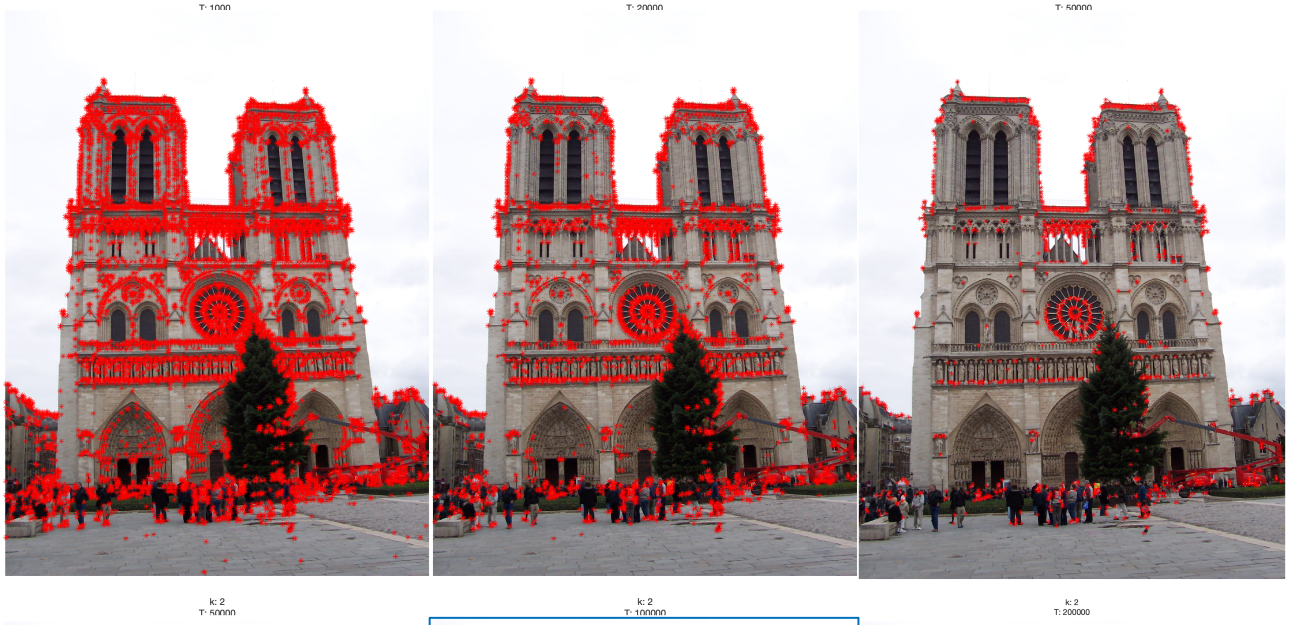
Investigation on lab.png image:

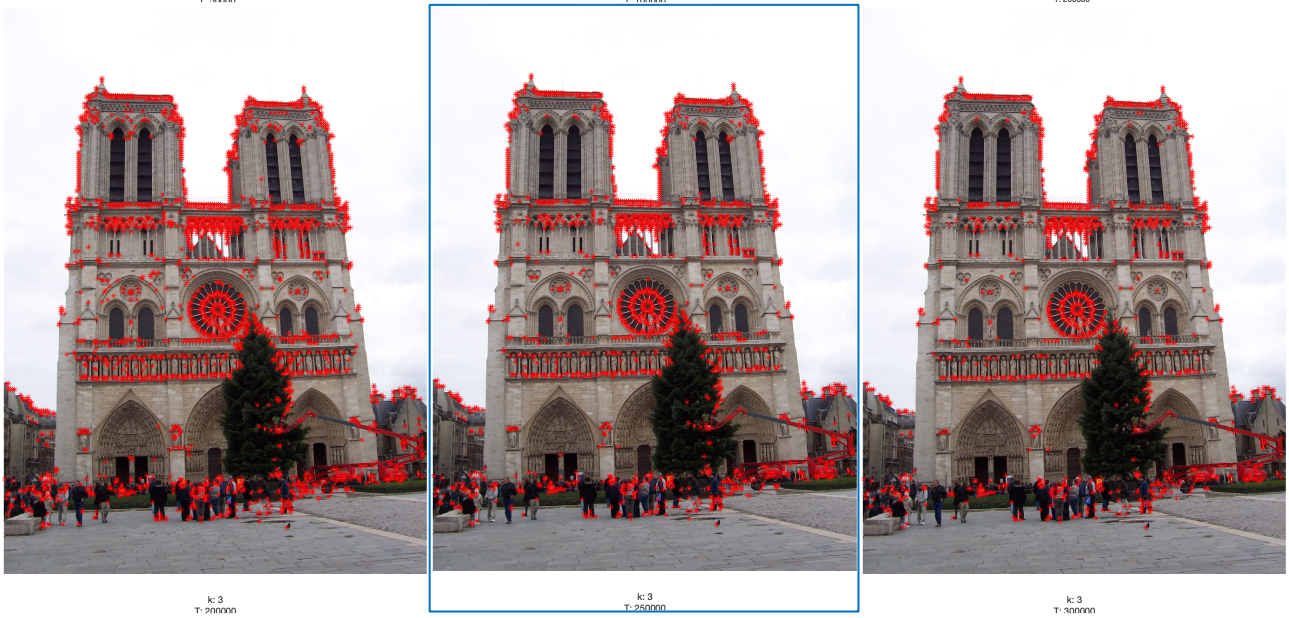


- Here, I again started with k:1 and T:1000. I increased T to 5000 by not changing k and obtained relatively good result but needed more. So increased T to 1000 and it got worse since I lost a lot of corner points this time. So, I tried to increase k value to 2 this time and again it gave me false positive corner points. I increased the T value again with k:2 and got those results above. Preferably, I would choose k:2 and T:15000 since comparing to others it is the best but not so good result still.

Investigation on notredame.jpg image:







- I don’t even talk about k:1, T:1000 case. When I increase T by keeping k:1, I realized that there will be a lot of false positive corner detection since we investigate so detailly by k:1. So, I increased and tried to alter T with respect to k:2 and again I came up with the idea that k:2 is still too small. Then, I increased the k:3 and played with T value and came up with the idea that k:3 and T:250000 is quite acceptable by considering the image so detailed and having a lot of different items. Again, depending on one’s preference, k and T value is chosen, and my preference is on k:3 T:250000.

3) HARRIS ALGORITHM USING CORNER RESPONSE

Implementation and Description:

```
function [C] = lab3Harriscorners2 (I, k, T)

[row,col,ch] = size(I);
if (ch == 3)
    I = rgb2gray(I);
end
I = double(I);


%Step 1: smoothen the greyscaled image with Gaussian filter.
I_smoothed = imgaussfilt(I);

%Step 2: compute image gradients
[Ix, Iy] = imgradientxy(I_smoothed);

%initialize the array which keeps the corner values inside.
C=[];


for i = (k+1):(2*k+1):(row-k-1)
    for j = (k+1):(2*k+1):(col-k-1)

        sub_img_x = Ix(i-k:i+k, j-k:j+k);
        sub_img_y = Iy(i-k:i+k, j-k:j+k);

        %Step 3: corner matrix H
        H = [sum(sum(sub_img_x.*sub_img_x)) sum(sum(sub_img_x.*sub_img_y)); sum(sum(sub_img_x.*sub_img_y)) sum(sum(sub_img_y.*sub_img_y))];

        %Step 4: compute R
        kk = 0.06;
        R = det(H) - kk*((trace(H))^2);

        %if R is greater than the threshold
        if (R > T)
            C = [C; i j];
        end
    end
end

end
```

Figure 15: Harris Algorithm using corner response on detecting corners.

- This another Harris Algorithm function that called ***lab3Harriscorners2*** is a function takes an ***image(I)***, ***window size(k)***, and ***threshold value(T)*** as inputs and returns the detected corner points of the input image as ***an array([C])***. (Shown in 1,2,3).
- Then, assigns the row, column, and channel values of input image to variables called ***row, col, ch*** respectively. It detects if the image is RGB or gray-scaled by checking the ***ch*** value. If it is RGB, then it converts the image to gray-scaled version; if already gray-scaled, leaves as it is. Finally, it converts the pixel values of gray-scaled converted image from integer to double so that when we perform mathematical operations to pixel values, we will not lose decimal values. (Shown in 4).
- After all these steps, the gray-scaled double valued image is smoothed by Gaussian Smoothing Filter. In order to achieve this goal, we utilize ***imgaussfilt()*** MATLAB built-in function whose input is gray-scaled double image I and output I_smoothed. (Shown in 5).
- Consecutively, we compute image gradients Ix and Iy by utilizing MATLAB built-in function called ***imgradientxy()*** whose input is gray-scaled doubled smoothed image I_smoothed and outputs are image gradients with respect to x and y; Ix and Iy computed with Sobel gradient operator. (Shown in 6).
- Before starting integral processing on the input image, we create empty vector ***C*** which will have the corner list at the end will be returned from the function. (Shown in 7).
- In order to detect the corners, we have to analyze each pixel by applying some operations to their neighbors. For that reason, we need to go and visit each of pixels of the image one by one. Hence, we use 2 for loops nested to each other. However, since we detect the corners and need only one single pixel within the neighbors, we increase the index values of pixels by window size rather than one on each step. This protects us to have several corner points in the given window size, so we have neater

output (Shown in 8). In order to understand more, let's look figure2 and figure3 examples and the explanation.

- Inside the for loop and each iteration of for loop, we create two sub images. This time, sub images are gradient values with respect to x and y which are cropped by window size centered at (i,j). (Shown in 9).
- Right after calculating X and Y gradient values, we compute the corner matrix H whose equation has been given us in the lab documentation as;

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

Figure 4: corner matrix equation

In this line of code, we utilize dot product in order to multiply X and Y gradient values by their selves or each other. (Shown in 10).

- After we calculate the corner matrix H, we calculate R by utilizing given equation;

$$R = det(H) - \kappa(trace(H))^2$$

Figure 5: calculating R

Where det() is determinant and trace() is the sum of diagonals of the given matrix and $\kappa \, \epsilon \, [0.04, 0.06]$ is an empirical constant. (Shown in 11).

- As the final step, we use threshold value on f to detect corners such that "if f is greater than the threshold value", the pixel coordinates are added to the corner list C. (Shown in 12).

OUTPUTS OF IMPLEMENTATION:

```
1   clear; close all; clc;
2   img_blocks = imread("Lab3 - Images/blocks.png");
3   img_lab = imread("Lab3 - Images/lab.png");
4   img_notredame = imread("Lab3 - Images/notredame.jpg");
5
```

Figure 16: reading the images in main function

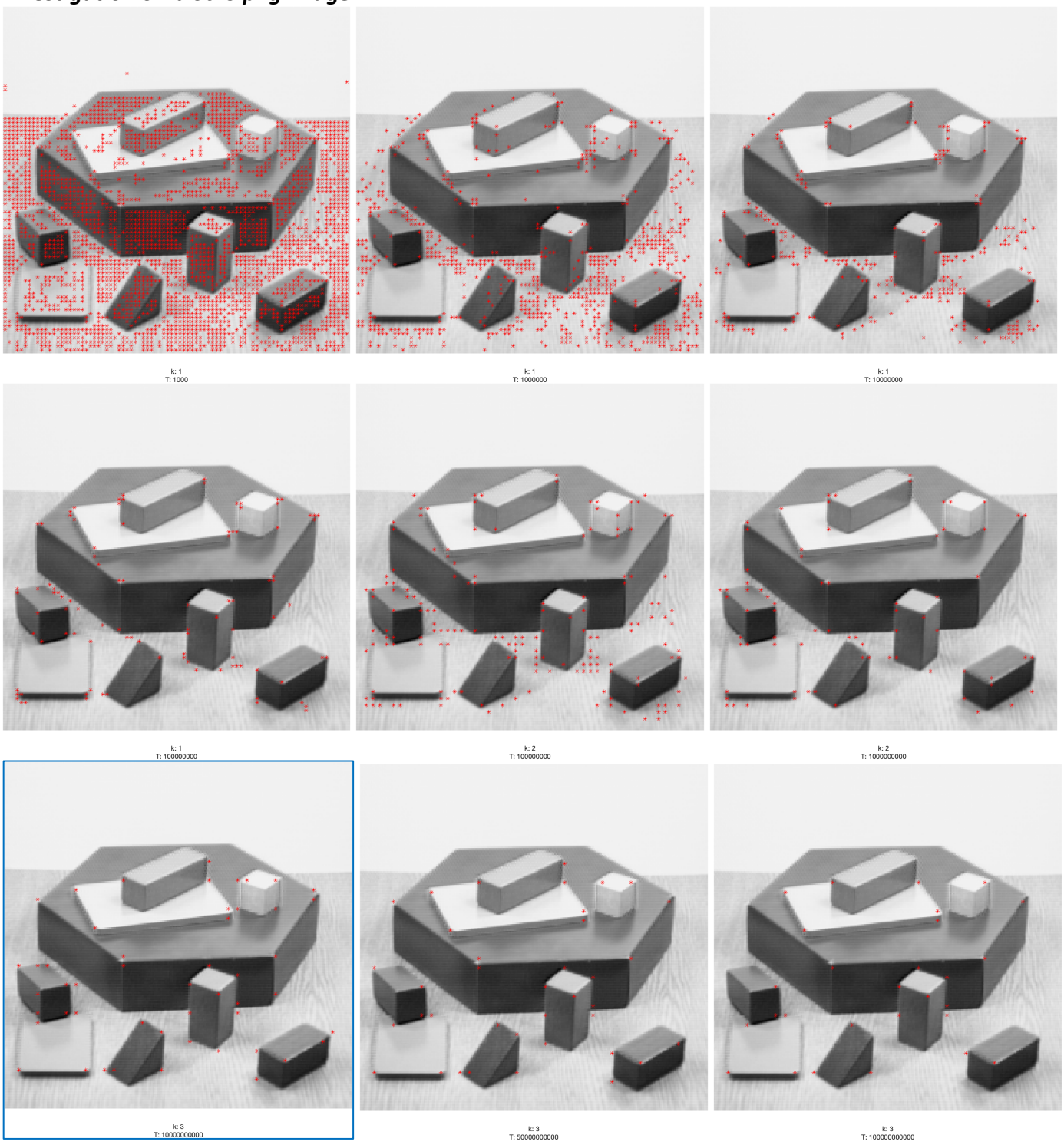
- First of all, in the main function, I start reading the images that were given in the lab3 document folder and assign those to new variables called ***img_blocks, img_lab*** and ***img_notredame.***

```
%% Harris Algorithm v.2
th3=1000;
k3=1;
Coreners3 = lab3Harriscorners2(img_blocks, k3, th3);

figure;
imshow(img_blocks);
hold on;
plot(Coreners3(:,2), Coreners3(:,1), 'r*');
xlabel( {'k: ' + string(k3), 'T: '+ string(th3)} );
```

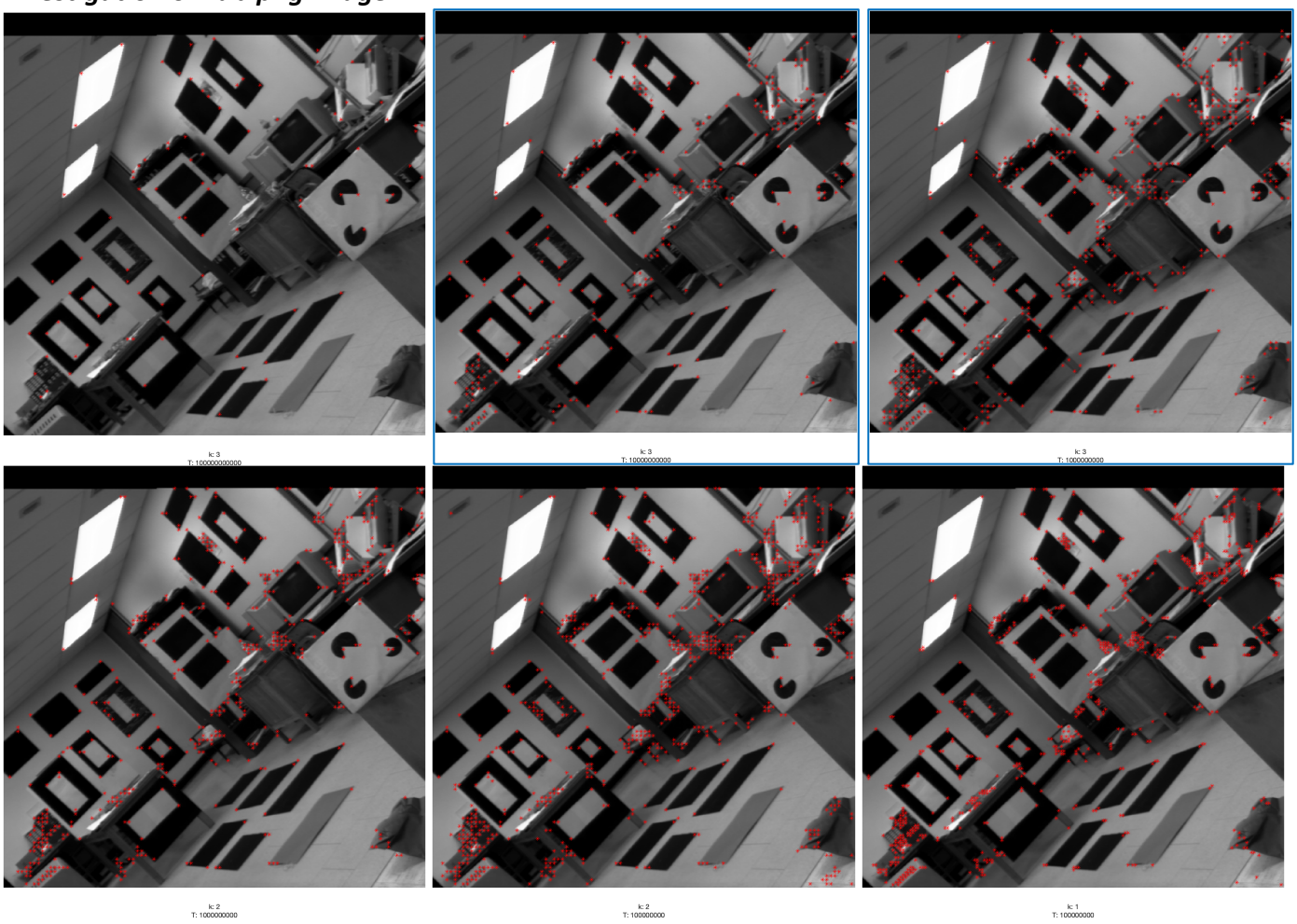
- This is generic Harris algorithm vol.2 call from main function. Th3, k3 and images are variable so that when comparing some values, these variables will change.

Investigation on blocks.png image:



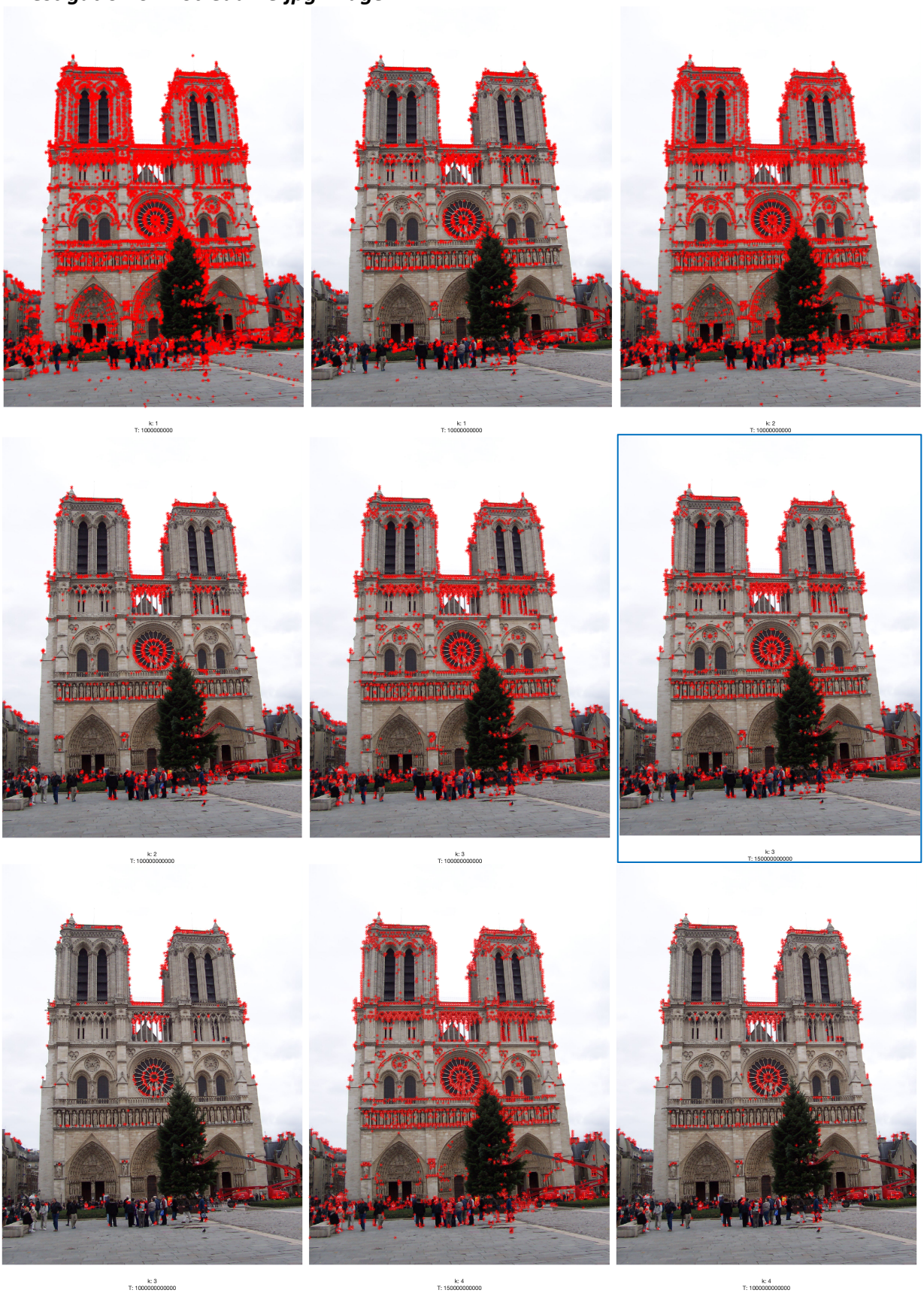
- By playing T and k values, I have obtained several results. Again, depending on one's aim and tradeoff preference, different outcomes could be chosen. I chose k:3 and T:10,000,000,000 since it is the most neat one within them. However, it fails to detect some corner points which are detected in other values but the others have more non-corner values. So, k:3 and T:10,000,000,000.

Investigation on lab.png image:



- I would choose either k:3 T:1,000,000,000 or T:10,000,000,000 or something in between. As seen above there are a lot of false positive corner points on last 3 results and too few corner points on the first result. That's why I'w either choose k:3 T:1,000,000,000 or T:10,000,000,000 depending on my preference.

Investigation on notredame.jpg image:

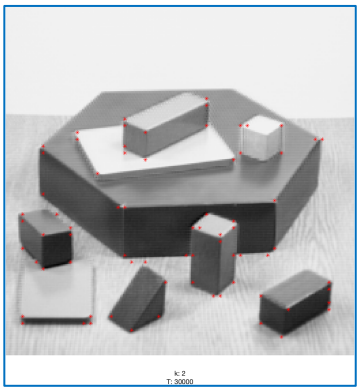


- I would say k:3, T: 150,000,000,000 is the best within the above results since it doesn't lose too much corner points and doesn't detect that much false positive corners.

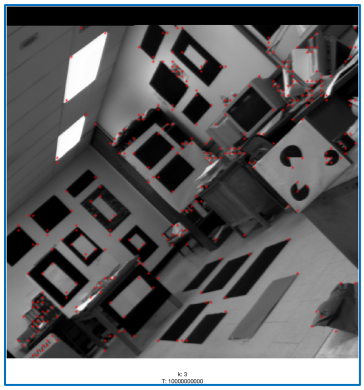
GENERAL DISCUSSION AND CONCLUSION

In this lab, we have used three different algorithms for corner detection. They are Kanade-Tomasi, Harris with f, and Harris with corner response. I have explained each of them by investigating the algorithms line by line and after that implied all those algorithms on three different input images: namely, blocks.png, lab.png, and notredame.jpg. As a result, there are 9 different results that was chosen by my preference on the tradeoff of false-positive corner detection and failure of detecting some corner points. All of them have been investigated one by one by altering the values of window size(k) and threshold(T) and the most optimal values have been chosen one by one. As a result of all those steps, I can say that for;

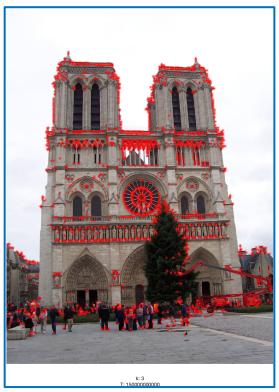
1) Blocks.png: I have chosen Harris algorithm with f with k:2, T:30,000 as shown above. I believe Harris with f gives the best result on this particular image since it is the neatest one and there are less failure on detecting corners.



2) Lab.png: I have chosen Harris algorithm with corner response by having k:3, T: 10,000,000,000. For same reasons, this algorithm with these values is the best for this particular image I believe.



3) Notredame.jpg: I have chosen Harris algorithm with corner response by having k:3, T: 10,000,000,000 for the same reason above.



APPENDIX A

Threshold value is 75000

Threshold value is 75000

Threshold value is 75000

Threshold value is 75000