EE417 - Assignment 3 - Post-Lab Report

Baris Sevilmis
Lab Date: 23/10/2018
Due Date: 29/10/2018

Overview

Third lab assignment of EE417 focuses on three different edge detection techniques as well as a new corner detection technique. Although Sobel and Prewitt are giving almost the same result in terms of edge detection, they are both included in the given assignment. Consecutively, Sobel Edge Detection, Prewitt Edge Detection, LoG Edge Detection and lastly but most importantly, Kanade-Tomasi Corner Detection Algorithm will be explained. Explanations and results of these algorithms on various images will be in further sections of the report.

Part 1: Sobel Edge Detection

Sobel Filtering is a discrete 2D first order derivative operation, in which through taking derivative of images edges are detected. There are two different filters/kernels used for edge detection in sobel filtering. There are two 3x3 Sobel filters, one of them is for horizontal edge detection and other one is for vertical edge detection. Below, Figure 1 depicts both of these filters, in which Figure 1(a) filter is responsible of vertical edge detection and Figure 1(b) is responsible of horizontal edge detection. However, edge detection is not through yet. Next step would be to combine the vertical and horizontal gradients in order to find the gradient of the whole image both in horizontal and vertical direction. Formula for this operation is following:

$$G(p) = \sqrt{G_x(p)^2 + G_y(p)^2}$$

As the gradient of the image is calculated , last step would be to remove non-edge results from the result because gradient of image are not yet consisting of only the true edges. Therefore, gradient results need to pass a certain threshold in terms of detecting true edges. Threshold value will be user input, therefore for the sake of clarity threshold value will be chosen as 100 in terms of demonstrating a explicit result.

-1	0	+1
-2	0	+2
-1	0	+1

(a) Vertical Edge Detector

+1	+2	+1
0	0	0
-1	-2	-1

(b) Horizontal Edge Detector

Figure 1: Sobel Filters

In Listing 1, MatLab Code for Sobel Edge Detection is provided, as it is a MatLab function and plot of the images are provided also within the function.

Listing 1: MatLab Code for Sobel Edge Detection

```
function J = lab3sobel(img, threshold)
    [row, col, ch] = size(img);
    k = 1;
    x_filter = [-1 0 1; -2 0 2; -1 0 1];
    y_filter = [1 2 1; 0 0 0; -1 -2 -1];

    if(ch == 3)
       newimg = rgb2gray(img);
    end
    newimg = double(newimg);
    J = zeros(size(newimg));
    J_hor = zeros(size(newimg));
    J_ver = zeros(size(newimg));

    for i = k + 1: 1: row - k -1
         for j = k + 1: 1 : col - k - 1

             subimg = newimg(i-k: i+k, j-k:j+k) ;
             J_ver(i,j) =  sum(sum(subimg .* x_filter));
             J_hor(i,j) =  sum(sum(subimg .* y_filter));
         end
    end
    subplot(2,3,1);imshow(img);title('Original Image');
    subplot(2,3,2);imshow(uint8(newimg));title('Greyscale Image'
       );
    subplot(2,3,3);imshow(uint8(J_ver));title('Sobel Vertical
       Edge Detection');
    subplot(2,3,4);imshow(uint8(J_hor));title('Sobel Horizontal
       Edge Detection');
    J = sqrt(J_hor.^2 + J_ver.^2);
    subplot(2,3,5);imshow(uint8(J));title('Sobel Gradient');
    J = uint8(J);
    J = J > threshold;
    subplot(2,3,6);imshow(J);title('Sobel Image');
end
```

In Figure 2, it is edge detection process of Sobel is clearly shown, such that consecutive order of the operations are the following:

$$Original -> Greyscale -> Horizontal\&VerticalEdges -> Gradient -> Sobel$$

Figure 2: Sobel Edge Detection

Part 2: Prewitt Edge Detection

Prewitt Edge Detection is a discrete 2D operation on images as Sobel. Actually, Sobel and Prewitt can be considered as incredible similar operations, because only difference between these methods are the kernels that are used for detecting vertical and horizontal edges. For the rest of the both techniques, gradient calculation and threshold value follows the same computation. For the sake of clarity, gradient formula is the following:

$$G(p) = \sqrt{G_x(p)^2 + G_y(p)^2}$$

Prewitt kernels are depicted in Figure 3, such that Figure 3(a) depicts the horizontal edge detector and Figure 3(b) depicts the vertical edge detector. Lastly, for sake of simplicity, threshold value is chosen as 100 also for Prewitt Edge Detection.

-1	0	+1
-1	0	+1
-1	0	+1

(a) Vertical Edge Detector

+1	+1	+1
0	0	0
-1	-1	-1

(b) Horizontal Edge Detector

Figure 3: Prewitt Filters

In Listing 2, MatLab code of Prewitt Edge Detection is provided with additional plotting options.

Listing 2: MatLab Code for Prewitt Edge Detection

```
function J = lab3prewitt(img, threshold)
    [row, col, ch] = size(img);
    k = 1;
    x_filter = [-1 0 1; -1 0 1; -1 0 1];
    y_filter = [1 1 1; 0 0 0; -1 -1 -1];

    if(ch == 3)
       newimg = rgb2gray(img);
    end
    newimg = double(newimg);
    J = zeros(size(newimg));
    J_hor = zeros(size(newimg));
    J_ver = zeros(size(newimg));

    for i = k + 1: 1: row - k -1
         for j = k + 1: 1 : col - k - 1

              subimg = newimg(i-k: i+k, j-k:j+k) ;
              J_ver(i,j) =  sum(sum(subimg .* x_filter));
              J_hor(i,j) =  sum(sum(subimg .* y_filter));

         end
    end

    subplot(2,3,1);imshow(img);title('Original Image');
    subplot(2,3,2);imshow(uint8(newimg));title('Greyscale Image'
       );
    subplot(2,3,3);imshow(uint8(J_ver));title('Prewitt Vertical
       Edge Detection');
    subplot(2,3,4);imshow(uint8(J_hor));title('Prewitt
       Horizontal Edge Detection');
    J = sqrt(J_hor.^2 + J_ver.^2);
    subplot(2,3,5);imshow(uint8(J));title('Prewitt Gradient');
    J = uint8(J);
    J = J > threshold;
    subplot(2,3,6);imshow(J);title('Prewitt Image');
end
```

As for the results of Prewitt Edge Detection, Figure 4 demonstrates the explicit results in consecutive order of operations:

$$Original -> Greyscale -> Horizontal\&VerticalEdges -> Gradient -> Prewitt$$

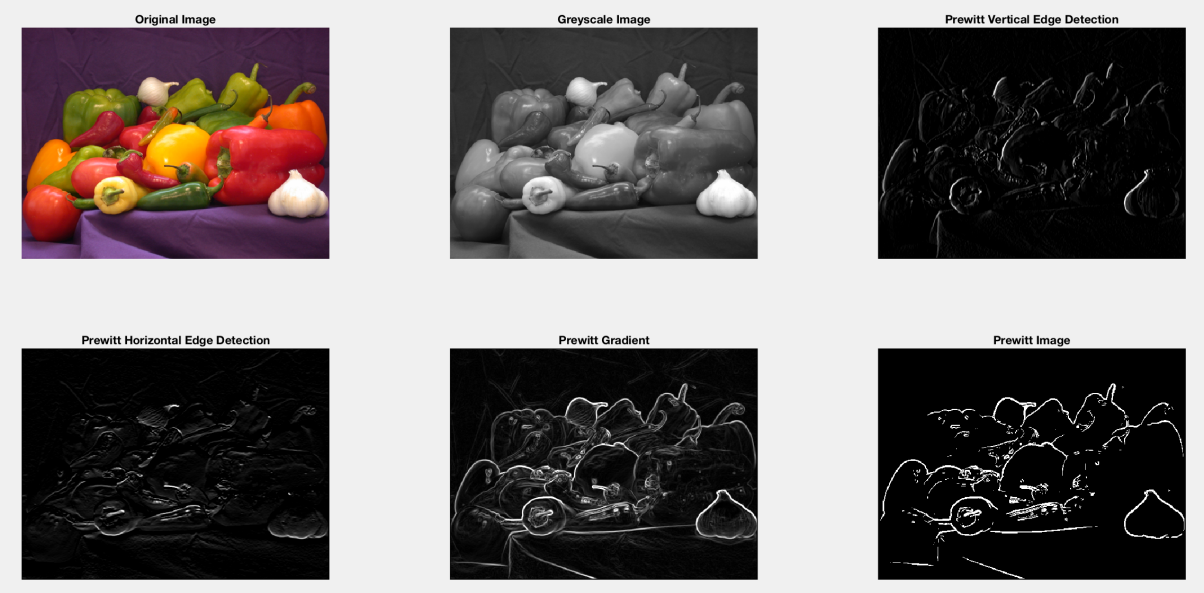

Figure 4: Prewitt Edge Detection

To conclude, both Sobel and Prewitt ensure edge detection with same features but different kernels though kernels are also similar. Despite kernel variation, end results of both algorithms are very similar given same parameters such that detecting their difference with naked eyes would be challenging.

Part 3: Laplacian of Gaussian

Laplacian of Gaussian, namely LoG, is an advanced edge detection algorithm in comparison to Sobel and Prewitt. LoG mainly consists of two steps, in which image is smoothed out and gradients of the image are found. These steps can be seen in Figure 5. To be more specific, first step of algorithm is to smooth out the given grey-scale image with Gaussian kernel. Although reducing noise in the given image is crucial for every edge detection algorithm, for LoG algorithm importance of this step is increased. Reason for reducing noise is mainly because of first order derivative operations in previous edge detection algorithms, though LoG is a second order derivative operation. Therefore, LoG is very noise sensitive. In second phase of LoG, a 3x3 laplacian kernel is used to filter image into its gradients. Gradient changes in the resulting images are indication of edges and perhaps even corners. However, as said noise must be reduced in order to witness full potential of LoG. Lastly, as said previously LoG is a second order derivative operation and uses a 3x3 filter. LoG filter can be seen in Figure 6.

0	-1	0
-1	4	-1
0	-1	0

Figure 5: LoG Filter

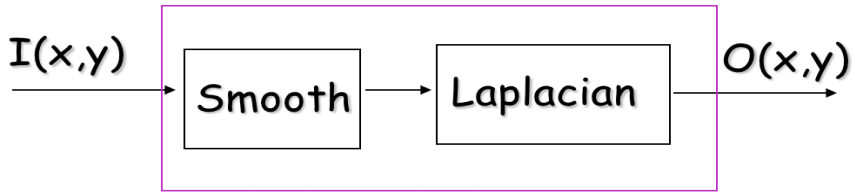

Figure 6: LoG algorithm

Results of LoG algorithm are depicted in Figure 7. Discussing Figure 7 is crucial in order to understand LoG. First image is the original image and second image is grey-scale version of the given image, third image is the resulting image of LoG. In other words, second order derivative filtering on the given grey-scale image leads us to the third image, in which certain lines are drawn. Main motivation of the drawn lines are to detect changes in gradient of the LoG Filtered image. For sake of simplicity, gradient magnitude over the given lines are plotted into three graphs. Correspondence between graphs and lines are done with coloring. Line drawings are approximations of exact pixel locations, yet exact pixel locations can be seen in Listing 3. It is obvious that red line is plotted over a single edge such that change of gradient magnitude over this edge is depicted in first graph. On the other hand, green line is not drawn over an edge, therefore there must be no huge change in gradient as in red line. As seen in second graph, green line is more consistent given the same gradient magnitude interval. Lastly, blue line is drawn over two edges in y-axis. Expected behaviour would therefore be as in red line with an additional gradient fluctuation. As expected, third graph demonstrates multiple fluctuation that is caused by multiple edges. It can be also seen that multiple edges that are nearby cause much higher fluctuation in gradient magnitude, such that nearby corner may be another reason of this fluctuation though line does not pass over the exact corner.

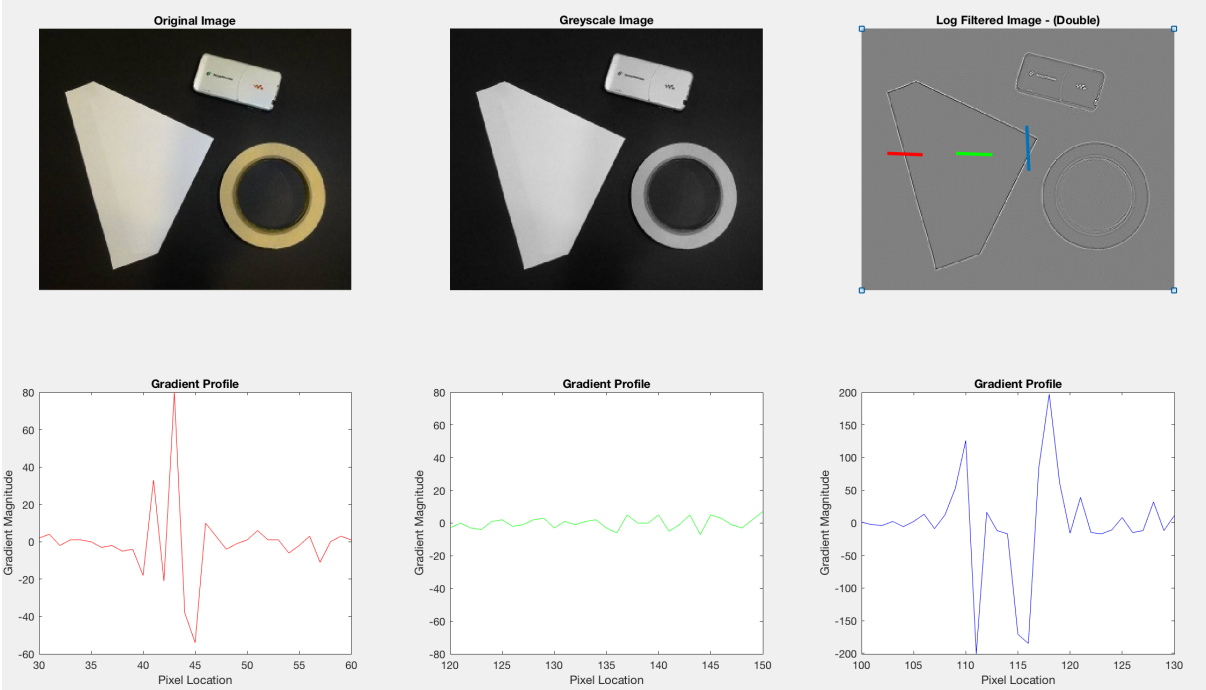

Figure 7: Results of LoG algorithm

In Listing 3, corresponding MatLab code for LoG is demonstrated with necessary plot functions.

Listing 3: MatLab Code for LoG

```
function J = lab3log(img)
    [row, col, ch] = size(img);
    k = 1;
    log_filter = [0 1 0; 1 -4 1; 0 1 0];

    if(ch == 3)
       newimg = rgb2gray(img);
    end

    newimg = double(newimg);
    J = zeros(size(newimg));

    for i = k + 1: 1: row - k -1
         for j = k + 1: 1 : col - k - 1

             subimg = newimg(i-k: i+k, j-k:j+k) ;
             J(i,j) =  sum(sum(subimg .* log_filter));
         end
    end

   Y1 = J(128, 30:60);
   Y2 = J(128, 120:150);
   Y3 = J(100:130, 175);
   subplot(2,3,1);imshow(img);title('Original Image');
   subplot(2,3,2);imshow(uint8(newimg));title('Greyscale Image')
      ;
   subplot(2,3,3);imshow(J, []);title('Log Filtered Image - (
      Double)');
   subplot(2,3,4);plot(30:60,Y1);title('Gradient Profile');
   xlabel('Pixel Location');ylabel('Gradient Magnitude');
   subplot(2,3,5);plot(120:150,Y2);ylim([-80 80]);title('
      Gradient Profile');
   xlabel('Pixel Location');ylabel('Gradient Magnitude');
   subplot(2,3,6);plot(100:130,Y3);title('Gradient Profile');
   xlabel('Pixel Location');ylabel('Gradient Magnitude');

end
```

Part 4: Kanade-Tomasi Corner Detection

Kanade-Tomasi Corner Detection algorithm is a well-known corner detection algorithm that uses image gradients and eigenvalues to compute corners of an image. Corners of an image indicate change in gradient magnitude both in x and y direction. To be more

specific, calculated gradients are written into a 2x2 matrix in the following format:

$$E(x, y) = x^T H x$$

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

After gradients are calculated, eigenvalues of H are computed that are positive values. More the eigenvalues are, more is the chance to be a corner. Therefore, given threshold is used. If minimum of eigenvalues is greater than threshold, it concluded that given location is a corner. Although the algorithm may fail to work perfect, surprising results are achieved. by tuning the parameters such as window size and threshold. However, for every different image, parameter tuning has to be done from start. Results of this algorithm in various images are depicted below. Figure 8 demonstrates Corner Edge Detection on various blocks with threshold = 250000 and window size = 11. On the other hand, Figure 9 demonstrates corner detection on checkerboard with threshold = 50 and window size = 11. Lastly, Figure 11 depicts corner detection with window size = 13 and threshold = 1000000. As said previously, parameters are constant that require tuning.
In Listing 4, MatLab code for Kanade-Tomasi Corner Detection can be seen with necessary plot functions for all three figures with fine-tuned parameters. However, parameters may require some additional tuning for better performance. To conclude, it is shown that instead of pixelwise windowing, namely moving window pixel by pixel, in corner detection window movement is same size with window instead of 1. Otherwise, algorithm works very slow, making it very time consuming.

Figure 8: Kanade-Tomasi Corner Detection

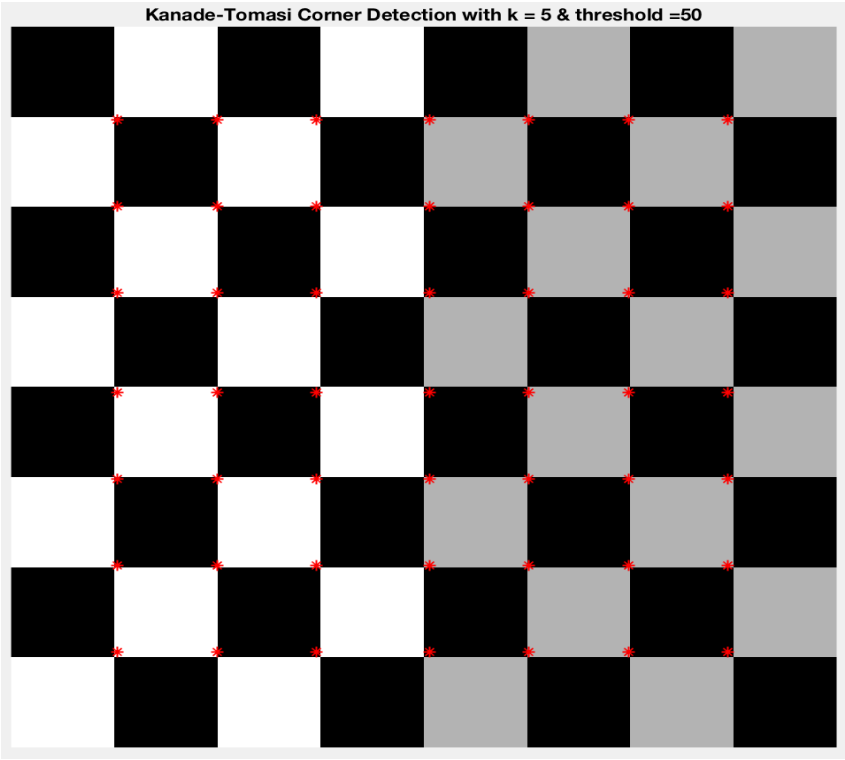

Figure 9: Kanade- Tomasi Corner Detection

Figure 10: Kanade-Tomasi Corner Detection

Lastly, Kanade-Tomasi Edge Detector on various images with and without certain corners. Thus, it would be more reasonable to discuss the results from a different view. With motivation of testing our algorithm on various images, Figure 11 demonstrates results of the Corner Detection on a famous meme. It can be seen that image does have very certain corners as well as hard to detect corners. By this means, there is a lot of different corners. Kanade-Tomasi corner detector with threshold = 320000 and k = 5 found almost all of these corners. Although there are faulty detection, some of these faulty detection are caused by non-linearity in image itself. With better tuning, algorithm may give a better result.

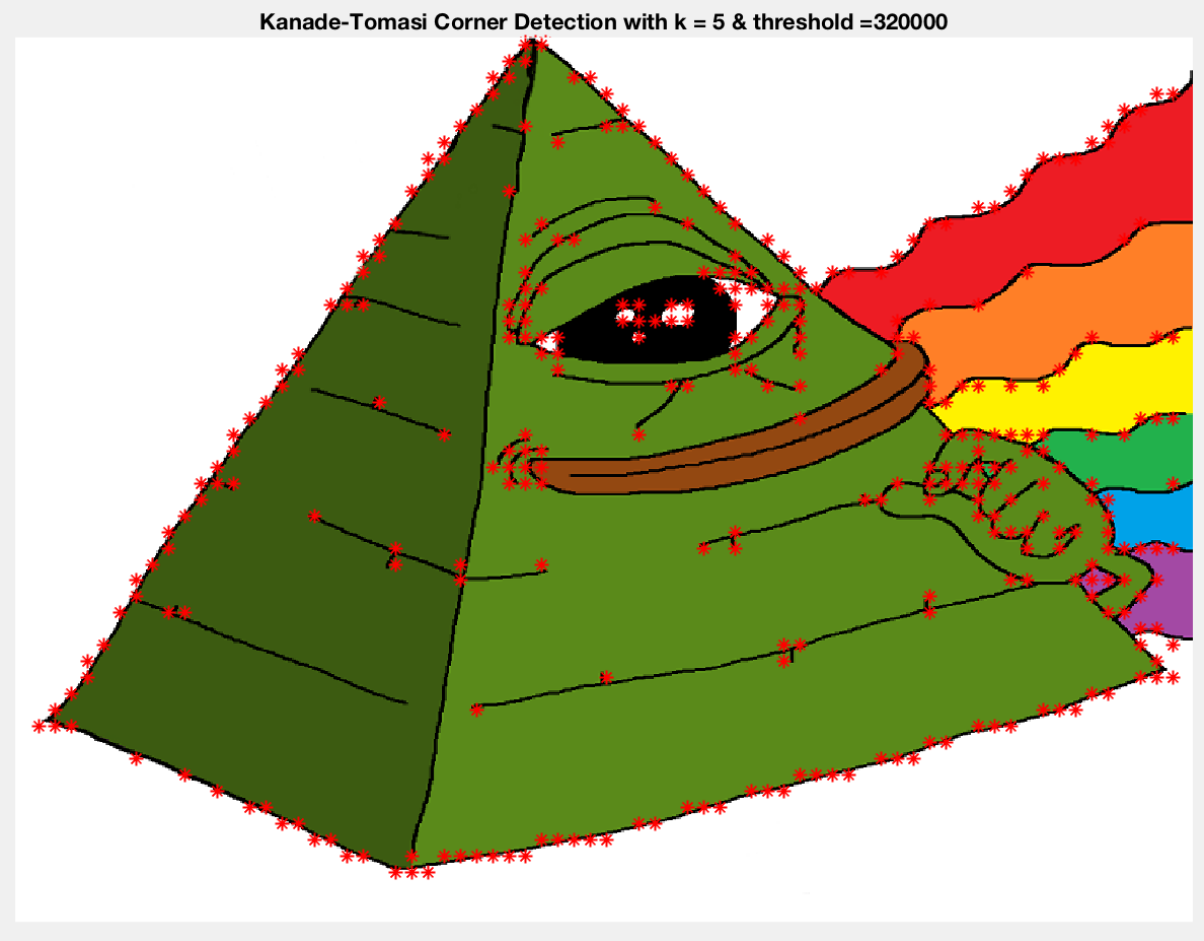

Figure 11: Kanade-Tomasi Corner Detection

On the other hand, Figure 12 depicts corner detection on an image from Matrix Reloaded. As it can be seen, all bullets are detected with threshold = 200000 and k= 5. There are also additional corner detections, such as corners of protagonists glasses. For this reason, success of Kanade-Tomasi Edge Detector achieves success. As said previously, better corner detection is possible with better tuning.

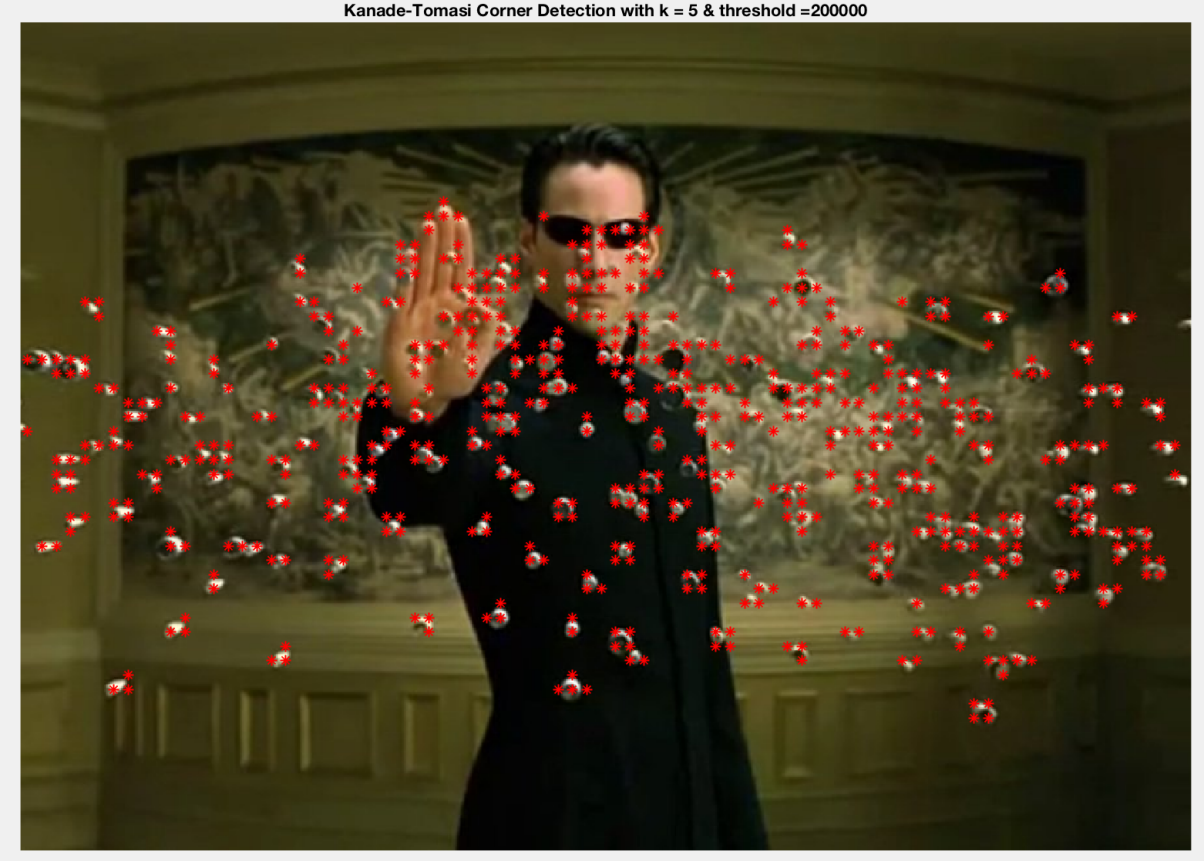

Figure 12: Kanade- Tomasi Corner Detection

Listing 4: MatLab Code for Kanade-Tomasi Corner Detector

```
function J = lab3ktcorners(img, threshold, k)
    [row, col, ch] = size(img);
    if(ch == 3)
        newimg = rgb2gray(img);
    else
        newimg = img;
    end

    newimg = double(newimg);
    [Gx,Gy] = imgradientxy(newimg);
    J = [];

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

            Gxt = Gx(i-k: i+k, j-k:j+k) ;
            Gyt = Gy(i-k: i+k, j-k:j+k) ;
            H = [sum(sum(Gxt.^2)) sum(sum(Gxt.*Gyt));sum(sum(Gxt
                .*Gyt)) sum(sum(Gyt.^2))];
            eigen = eig(H);

            if (min(eigen) > threshold)
                J = [J; i j];
            end
        end
    end

    %Blocks
    %J = lab3ktcorners(img, 250000, 5);
    %imshow(img);
    %hold on;
    %plot(J(:,2), J(:,1) , 'r*');
    %title(['Kanade-Tomasi Corner Detection with k = ', num2str(
        k), ' & threshold =', num2str(threshold)]);

    %checkerboard(80) - 400x400 double
    %J = lab3ktcorners(img, 50, 5);
    %imshow(img);
    %hold on;
    %plot(J(:,2), J(:,1) , 'r*');
    %title(['Kanade-Tomasi Corner Detection with k = ', num2str(
        k), ' & threshold =', num2str(threshold)]);

    %Monastry
    %J = lab3ktcorners(img, 1000000, 6);
    imshow(img);
    hold on;
    plot(J(:,2), J(:,1) , 'r*');
    title(['Kanade-Tomasi Corner Detection with k = ', num2str(k
        ), ' & threshold =', num2str(threshold)]);
end
```