EE243: Advanced Computer Vision
Assignment #2

Ali Jahanshahi (862029266)

April 2019

1 Problem 1

1.1 Question 7.2

A mean approximation pyramid is created by forming 2×2 block averages. Since the starting image is of size 4×4, J = 2 and $f(x, y)$ is placed in level 2 of the mean approximation pyramid. The level 1 approximation (by taking 2×2 block averages over $f(x, y)$ and sub-sampling) is shown in Equation 1

$$\begin{vmatrix} 3.5 & 5.5 \\ 11.5 & 13.5 \end{vmatrix} \tag{1}$$

and the level 0 approximation is similarly $\begin{vmatrix} 8.5 \end{vmatrix}$. Equation 2 shows the completed mean approximation pyramid.

$$\begin{vmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{vmatrix} \begin{vmatrix} 3.5 & 5.5 \\ 11.5 & 13.5 \end{vmatrix} \begin{vmatrix} 8.5 \end{vmatrix} \tag{2}$$

Pixel replication is used in the generation of the complementary prediction residual pyramid. Level 0 of the prediction residual pyramid is the lowest resolution approximation,i.e $\begin{vmatrix} 8.5 \end{vmatrix}$. The level 2 prediction residual is obtained by

up-sampling the level 1 approximation and subtracting it from the level 2 approximation (original image). Thus, we have Equation 3.

$$\begin{vmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{vmatrix} - \begin{vmatrix} 3.5 & 3.5 & 5.5 & 5.5 \\ 3.5 & 3.5 & 5.5 & 5.5 \\ 11.5 & 11.5 & 13.5 & 13.5 \\ 11.5 & 11.5 & 13.5 & 13.5 \end{vmatrix} = \begin{vmatrix} -2.5 & -1.5 & -2.5 & -1.5 \\ 1.5 & 2.5 & 1.5 & 2.5 \\ -2.5 & -1.5 & -2.5 & -1.5 \\ 1.5 & 2.5 & 1.5 & 2.5 \end{vmatrix} \tag{3}$$

Similarly, the level 1 prediction residual is obtained by up-sampling the level 0 approximation and subtracting it from the level 1 approximation to yield Equation 4

$$\begin{vmatrix} 3.5 & 5.5 \\ 11.5 & 13.5 \end{vmatrix} - \begin{vmatrix} 8.5 & 8.5 \\ 8.5 & 8.5 \end{vmatrix} = \begin{vmatrix} -5 & -3 \\ 3 & 5 \end{vmatrix} \tag{4}$$

Therefore, the prediction residual pyramid is shown in Equation 5.

$$\begin{vmatrix} -2.5 & -1.5 & -2.5 & -1.5 \\ 1.5 & 2.5 & 1.5 & 2.5 \\ -2.5 & -1.5 & -2.5 & -1.5 \\ 1.5 & 2.5 & 1.5 & 2.5 \end{vmatrix} \begin{vmatrix} -5 & -3 \\ 3 & 5 \end{vmatrix} \begin{vmatrix} 8.5 \end{vmatrix} \tag{5}$$

1.2 Question 7.3

The number of elements in a $J+1$ level pyramid where $N = 2^J$ is bounded by $\frac{4}{3}N^2$ or $\frac{4}{3}(2^J)^2 = \frac{4}{3}2^{2J}$ is shown in Equation 6.

$$2^{2J}\Big(1 + \frac{1}{(4)^1} + \frac{1}{(4)^2} + ... + \frac{1}{(4)^J}\Big) \leq \frac{4}{3}2^{2J} \tag{6}$$

For $J > 0$, we can generate Table 1. As we can see, all but the trivial case, i.e. $J = 0$, are expansions. The expansion factor is a function of J and bounded by $4/3$ or 1.33.

Table 1

J	Pyramid Elements	Compression Ratio
0	1	1
1	5	5/4 = 1.25
2	21	21.16 = 1.312
3	85	85/64 = 1.328
⋮	⋮	⋮
∞		4/3 = 1.33

1.3 Question 7.15

With $j_0 = 1$ the approximation coefficients are $c_1(0)$ and $c_1(1)$, which are shown in Equation 7:

$$\begin{aligned} c_1(0) &= \int_0^{1/2} x^2\sqrt{2}dx = \frac{\sqrt{2}}{24} \\ c_1(1) &= \int_{1/2}^{1} x^2\sqrt{2}dx = \frac{7\sqrt{2}}{24} \end{aligned} \tag{7}$$

Therefore, we have the V_1 approximation in Equation 8. Figure 1 illustrates the plot of the V_1 approximation.

$$\frac{\sqrt{2}}{24}\varphi_{1,0}(x) + \frac{7\sqrt{2}}{24}\varphi_{1,1}(x) \tag{8}$$

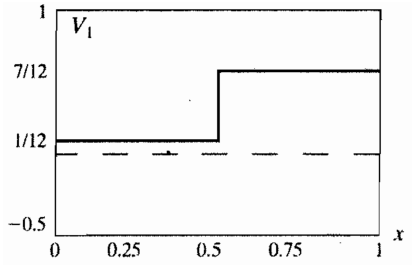


Figure 1: The V_1 approximation.

The last two coefficients are $d_1(0)$ and $d_1(1)$, which are computed as in the example. Thus, the expansion is shown in Equation 9.

$$y = \frac{\sqrt{2}}{24}\varphi_{1,0}(x) + \frac{7\sqrt{2}}{24}\varphi_{1,1}(x) + \Big[\frac{-\sqrt{2}}{32}\psi_{1,0}(x) - \frac{3\sqrt{2}}{32}\psi_{1,1}(x)\Big] + \dots \tag{9}$$

2 Problem 2

In this question we are asked to use the Haar wavelet to obtain a multi-resolution decomposition up to two levels of scale for the *house* image shown in Figure 2.

Figure 2: *house* original image.

Figure 3 shows the result of first level decomposition of Haar wavelet on the image. And, Figure 4 illustrates the second level decomposition of Haar wavelet on the image.

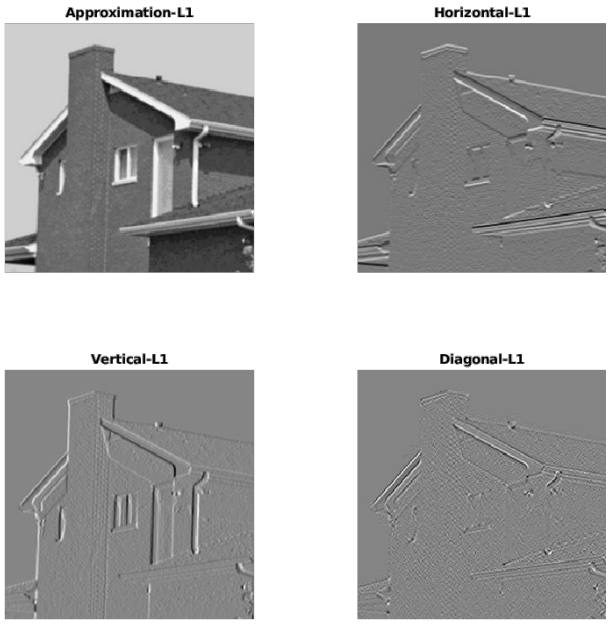


Figure 3: Haar wavelet first level decomposition.

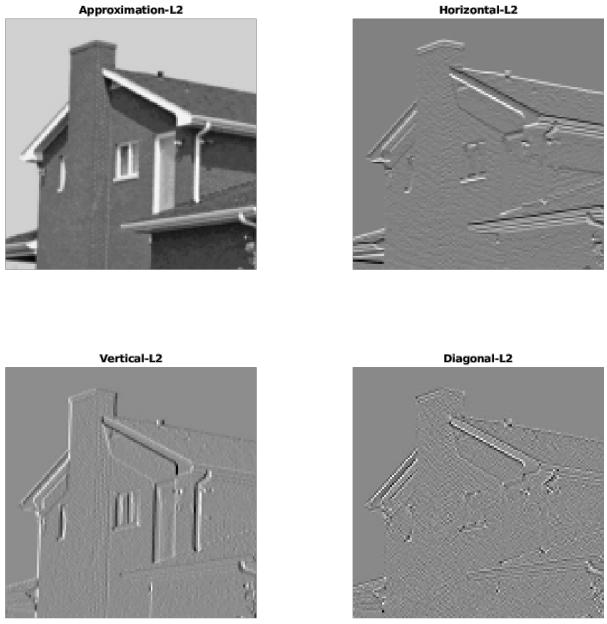


Figure 4: Haar wavelet second level decomposition.

In the next part of the question, we are asked to reconstruct back the original image using :

1) All the multi-resolution decompositions. For doing so, we used *idwt2* Matlab function with the result of previously mentioned decompositions. The result is shown in Figure 5.

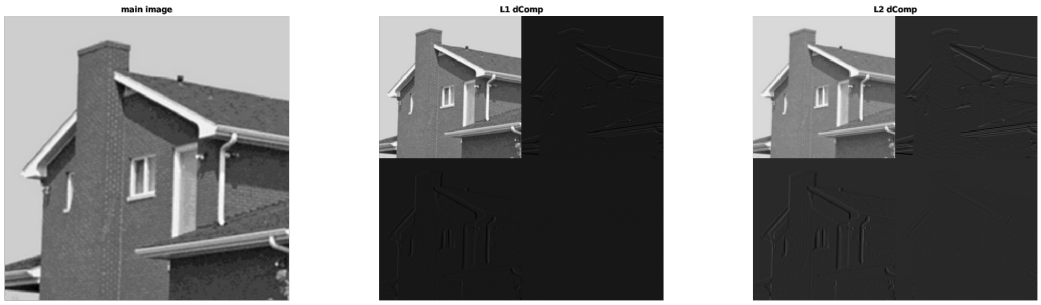


Figure 5: Reconstruction *house* image using all the multi-resolution decompositions.

2) Dropping high frequency components. We know that wavelet gives us four outputs: approximation (CA), vertical (CV), horizontal (CH), and diagonal (CD). Approximation output is the low (important) frequencies of a image.

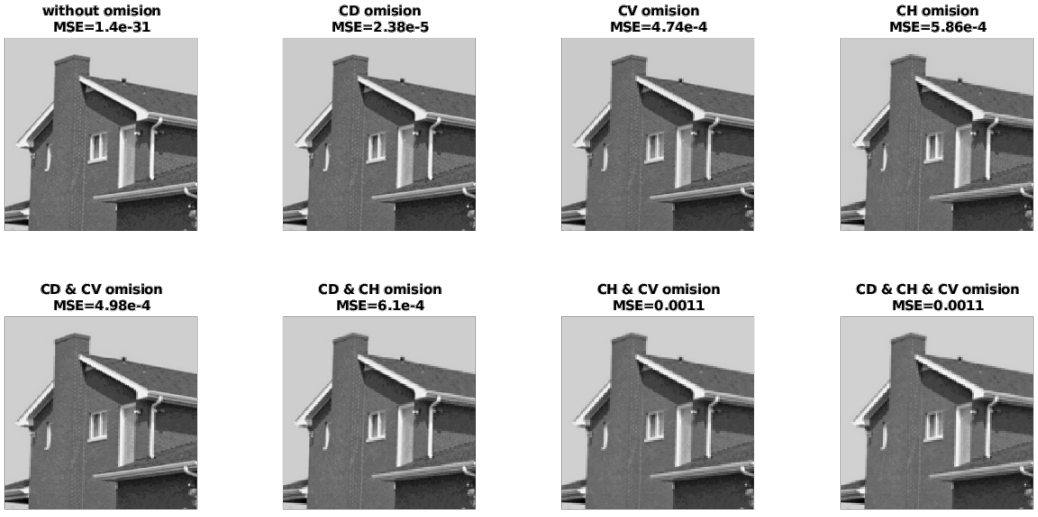


Figure 6: Reconstruction *house* image by dropping different permutations of high frequency components.

The rest of the outputs are high frequency components which can be dropped. Dropping each of the high frequency component introduces an error to the reconstructed image. We investigated the effect of them by dropping different permutations of high frequency components and measuring the reconstruction error by MSE metric. Figure 6 shows the result of image reconstruction in different scenarios.

Depending the image details, the component that can be removed with the lowest sacrifice much in reconstruction accuracy differs. As we can see, in this image, omitting CD (diagonal high frequency components) introduces the lowest reconstruction error. Based on how much compression we need, we can omit other high frequency components too.

3 Problem 3

3.1 Edge Detection

In this part we are asked for applying Laplacian of Gaussian and Canny edge detector on *house* and *lena* images which are shown in Figure 7a and Figure 7b, respectively.

(a) House (b) Lena

Figure 7: *House* and *Lena* original images.

Figure 8 and Figure 9 show the result of applying the Laplacian of Gaussian

edge detector with different filter width size and different standard deviations for *house* and *lena* images, respectively. We used *fspecial* Matlab function with *log* type to produce a filter. Then, we applied the filter on the image using *imfilter* functions.

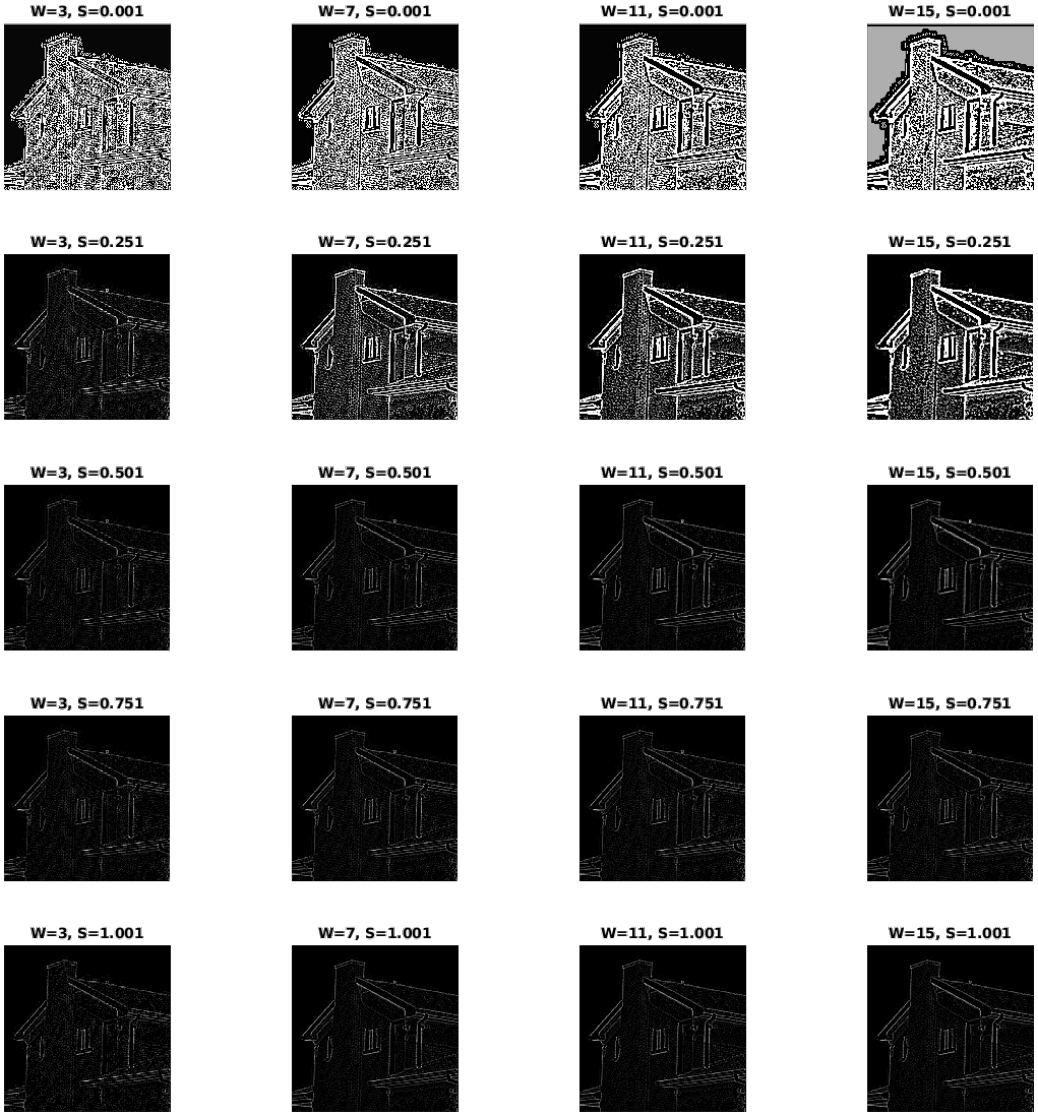


Figure 8: Laplacian of Gaussian edge detection with different filter size and thresholds for *House*.

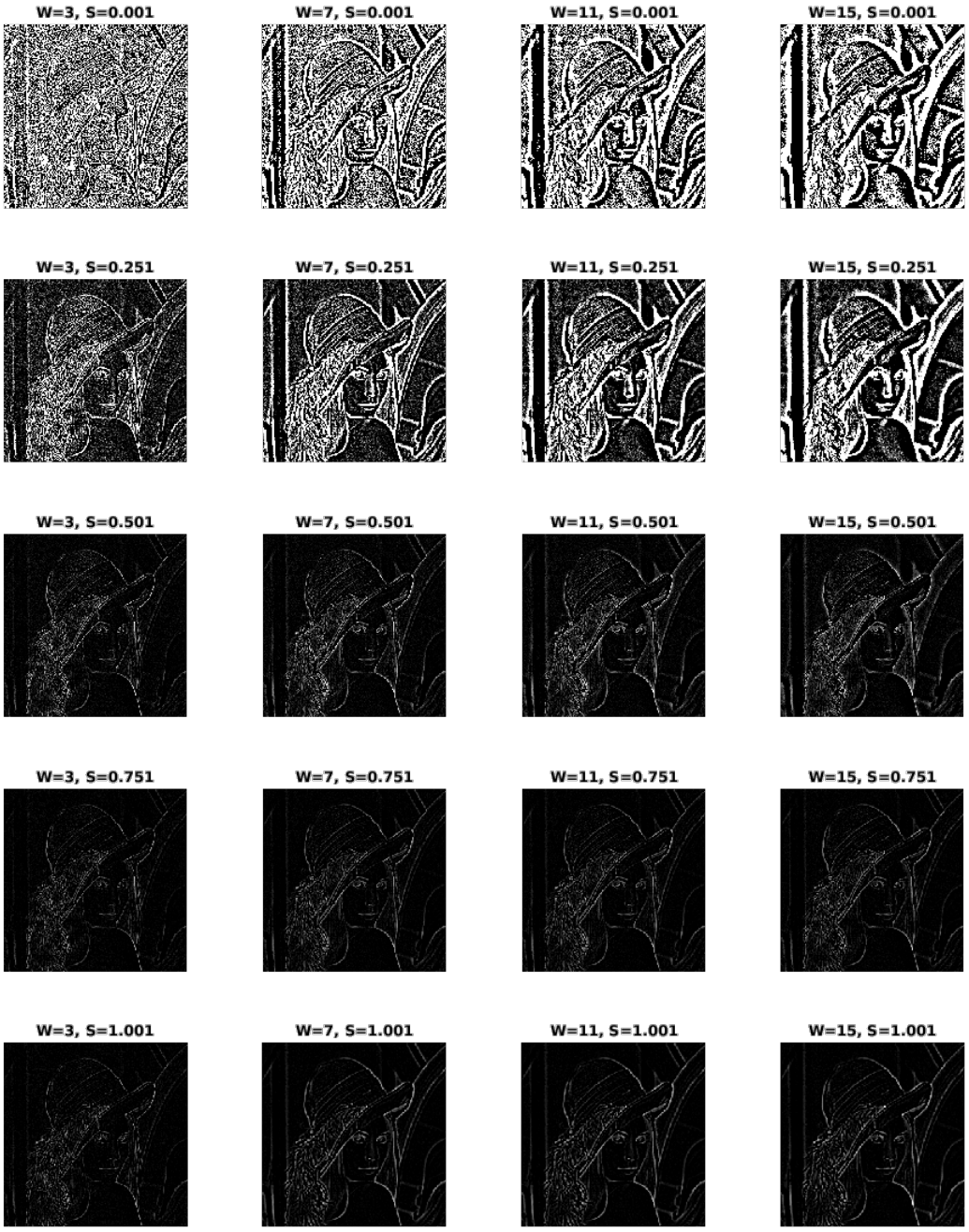


Figure 9: Laplacian of Gaussian edge detection with different filter size and thresholds for *Lena*.

In the next step, we applied Canny edge detector on *house* and *lena* images that is shown in Figure 10 and Figure 11, respectively.

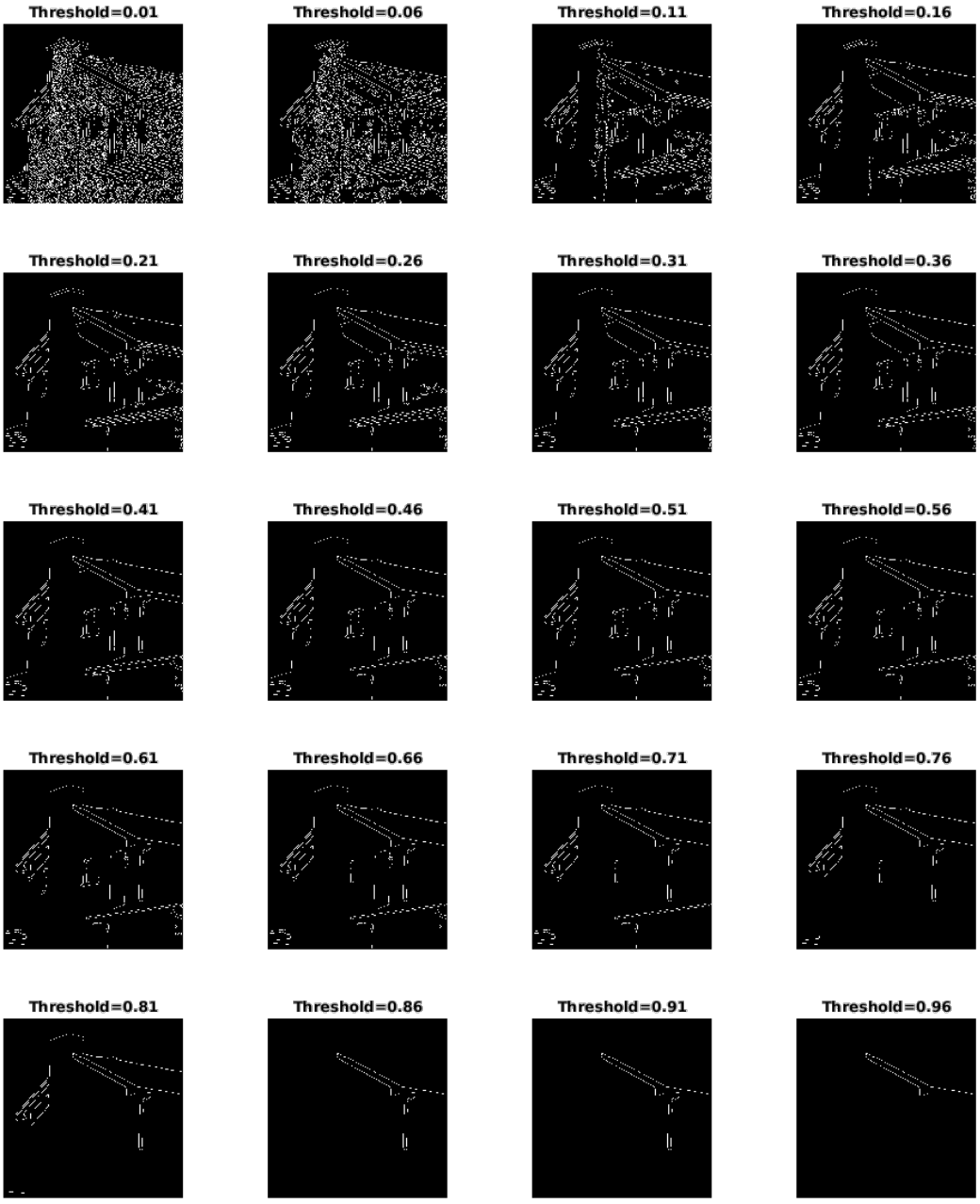


Figure 10: Canny edge detection with different thresholds for *House*.

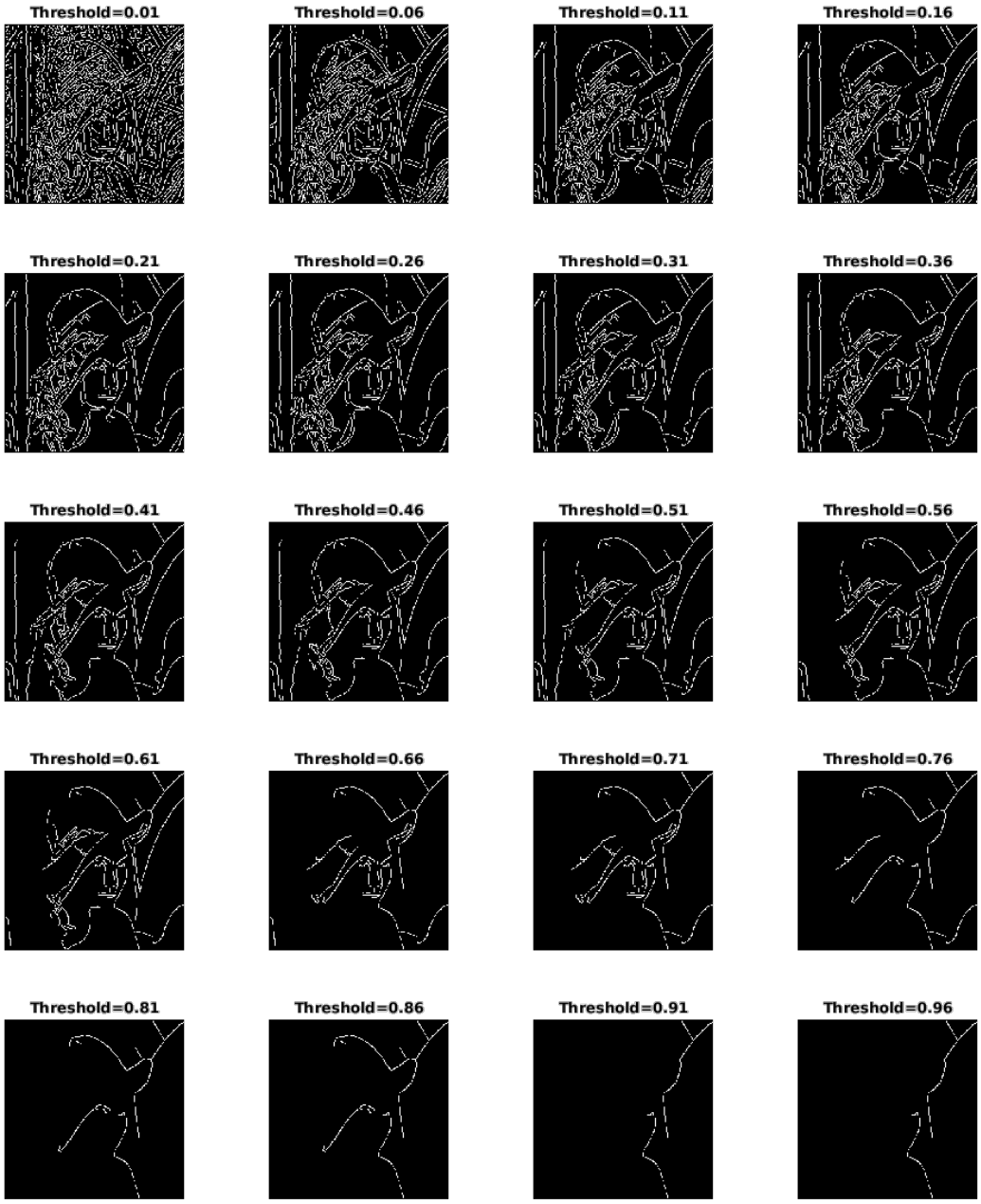


Figure 11: Canny edge detection with different thresholds for *Lena*.

In order to get apply Canny edge detector we used *edge* function with *Canny* as the type. Canny edge detector in *edge* function uses two thresholds. *edge* function

disregards all edges with edge strength below the lower threshold, and preserves all edges with edge strength above the higher threshold. We can specify threshold as a 2-element vector of the form ['low' 'high'] with low and high values in the range [0 1]. In our script, we set 'low'=0 and changed 'high' from 0.01 to 1. The captions on top of each subfigure shows the result for each 'high' threshold.

In addition, we applied Canny edge detector function (*edge*(image, 'Canny')) with default arguments on both images. Figure 12a and Figure 12b show the result of edge detector application on *house* and *lena* images, respectively. By comparing Figure 12a and Figure 10, we can see that the default Canny edge detector acts similar to when 'high' threshold is in [0.06,0.11]. For *lena*, the range is [0.11, 0.21].

(a) House (b) Lena

Figure 12: *House* and *Lena* Canny edge detector with default arguments.

3.2 Hough transform

In this problem, we are asked to use the output of the canny edge detector, and implement a Hough transform to detect the lines in the `house.tif` image. Figure 13 shows the result of the lines detected by the algorithm.

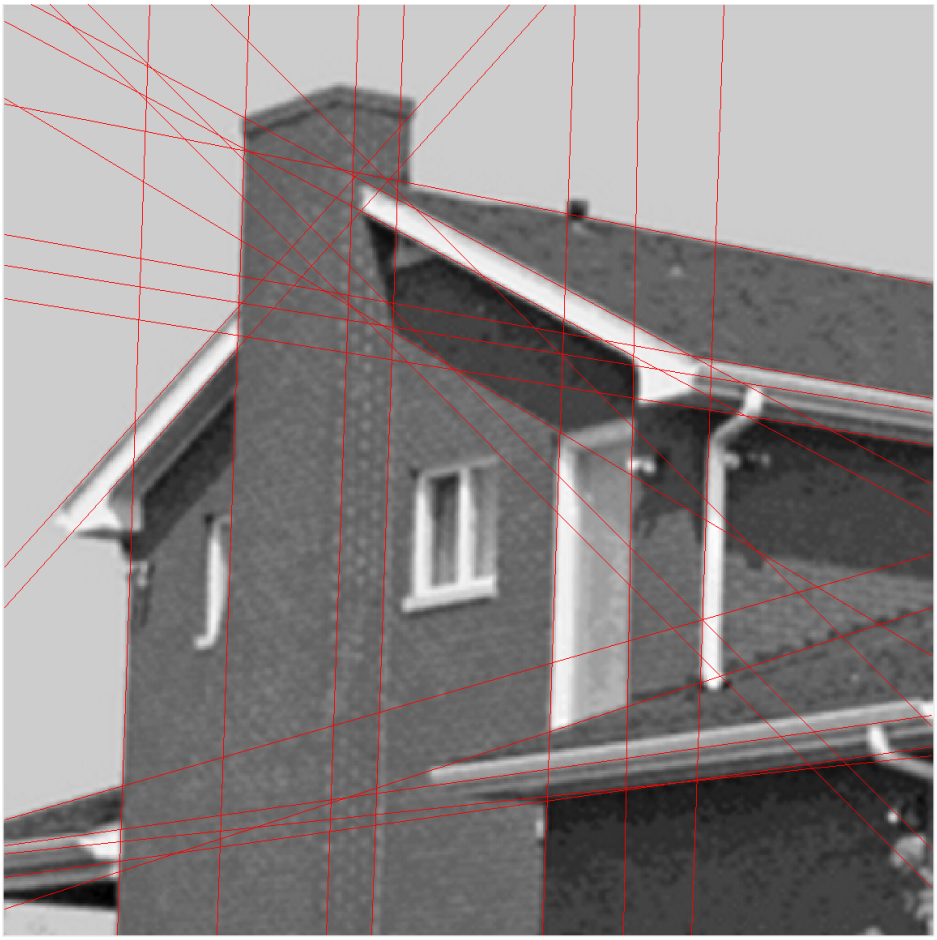


Figure 13: Line detection using Hough transform for *house* image.

4 Problem 4

4.1 Part A

In this part, we are asked to implement the Shi-Tomasi corner detector as a function. We used following:

- Sobel operator to obtain the gradients.
- Window size of 5×5 for w.
- Suppress points which are not local maxima within a 5×5 window.

We applied the corner detector on *house* image shown in Figure 7a. Figure 14 shows the the minimum eigen value of the image.

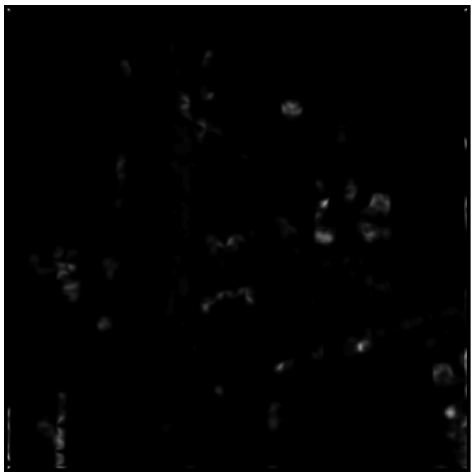
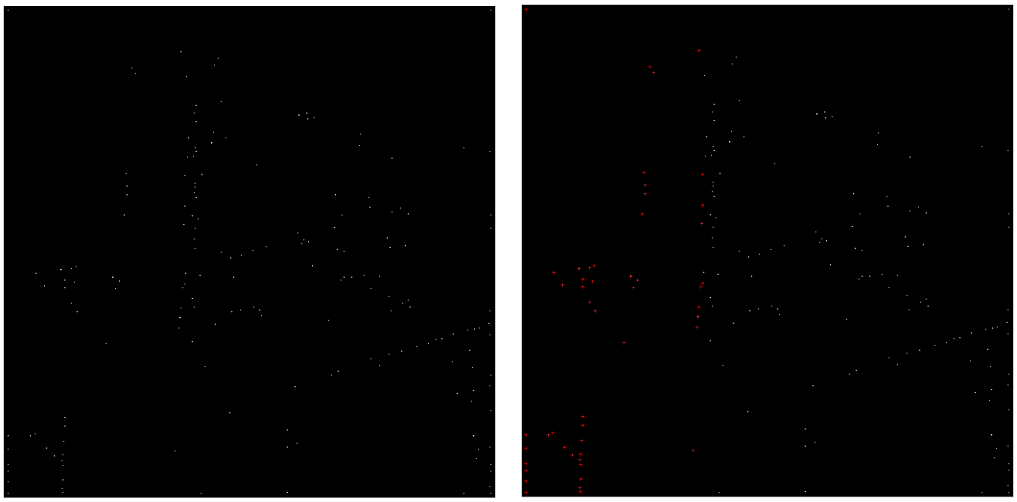


Figure 14: *house* image minimum eigen value.

Figure 15a illustrates the obtained corners by suppressing points which are not local maxima within a 5×5 window. And, Figure 15b shows the top 50 corners according to the minimum eigen value criterion in the algorithm.



(a) All corners. (b) Top 50 coreners.

Figure 15: Selected corners according to the minimum eigen value criterion.

Figure 16 shows the selected corners on the original image.

Figure 16: Selected top 50 corners by Shi-Tomasi on the original *house* image.

4.2 Part b

This part of the problem asks for HoG and SIFT features. We are asked to complete different functions used in `featurematching.m`, which is the main code for this problem. The broken down of the code is as follows:

1. The code loads the `blocks.png` image and then applies one of the two affine transforms `tform1` or `tform2` to obtain a transformed image. Figure 17 and Figure 19 show `blocks.png` image and its transformed version for `tform1` or `tform2`, respectively. Then, it calls the `getCorners.m` function to obtain the corner points, which is shown in Figure 18 and Figure 20.

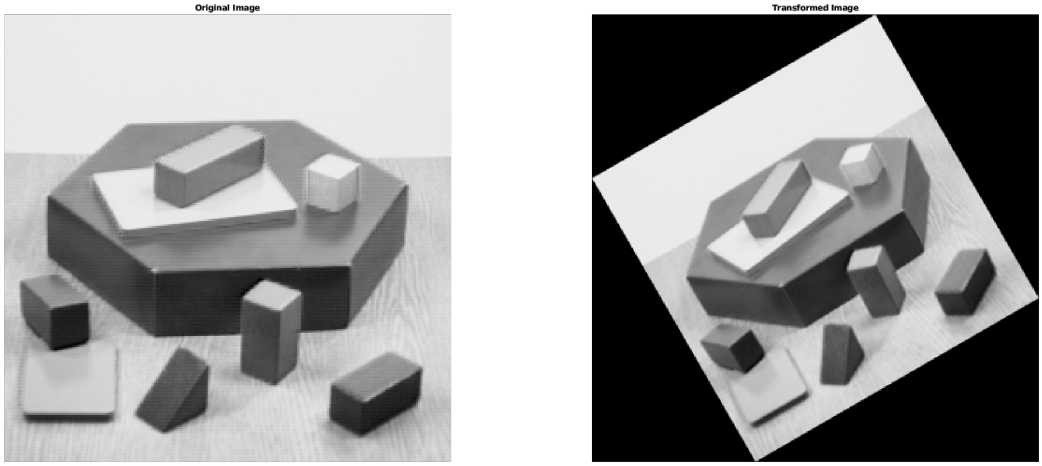


Figure 17: `blocks.png` image and its transformed version `tform1`.

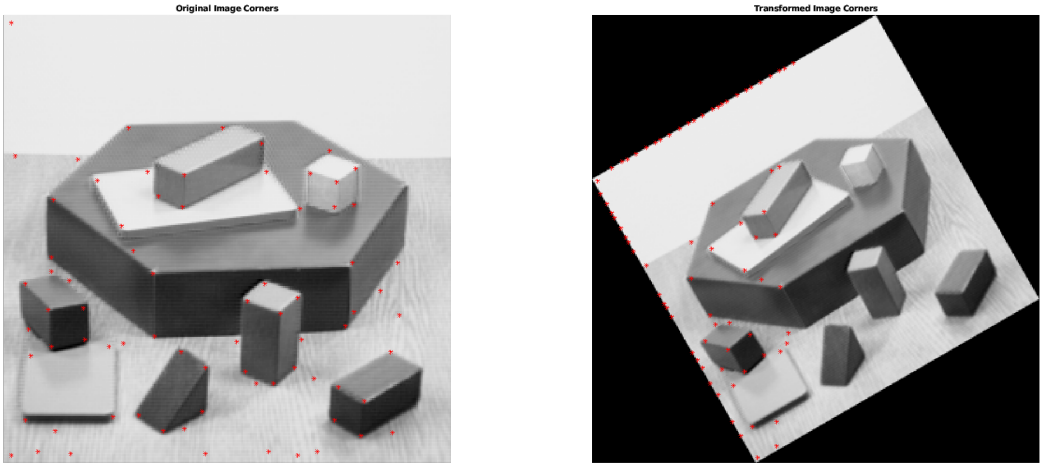


Figure 18: The corners of `blocks.png` image by running `getCorners.m`.

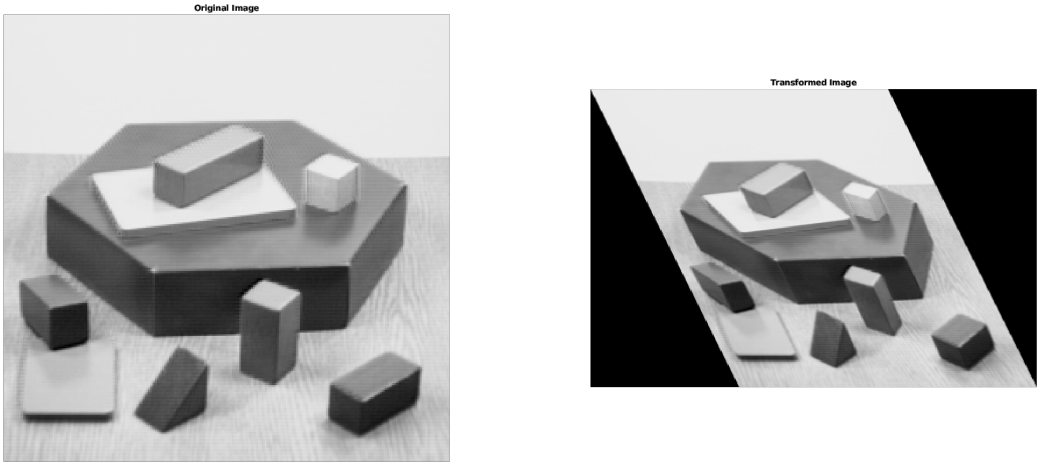


Figure 19: `blocks.png` image and its transformed version `tform2`.

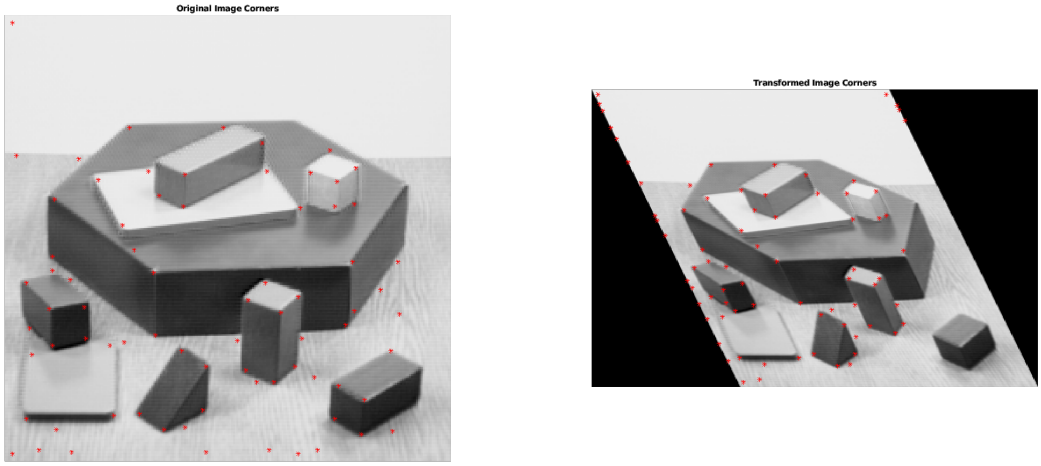


Figure 20: The corners of `blocks.png` image by running `getCorners.m`.

2. The images along with their corresponding corners are used as inputs to the `getFeatures.m` function. This function extracts the Histogram of Gradient (HoG) features for 8×8 patches around each of the corner points. We used 16 bins to obtain the HoG features.

3. The matches between the feature points of the original and transformed images are obtained using `getMatches.m` function. For each feature point,

we choose the nearest (according to HoG feature) as its match if the distance measure is greater than a certain threshold (hand picked). We used the normalized cross-correlation as a distance measure. Figure 21 and Figure 22 shows the matched features for `tform1` or `tform2`, respectively.

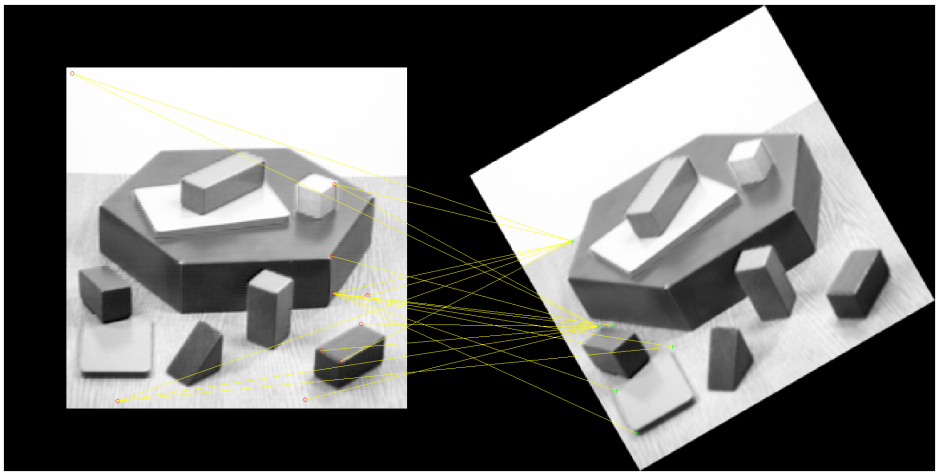


Figure 21: HoG feature matching on `blocks.png` image `tform1`.

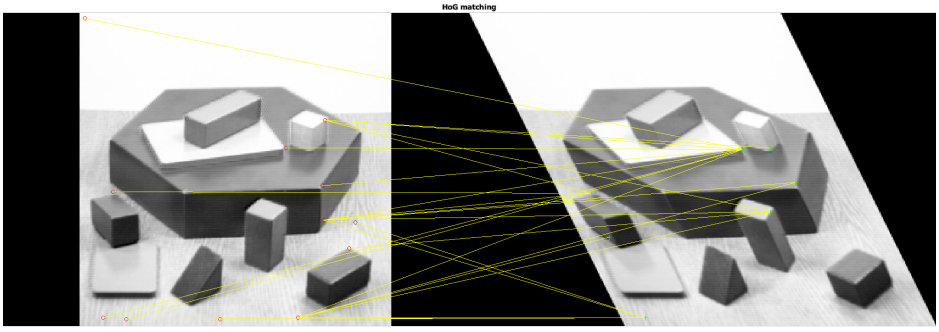


Figure 22: HoG feature matching on `blocks.png` image `tform2`.

4. Finally, the `featurematching.m` code extracts the SIFT features using the `vlfeat` package (you do not need to implement this), which is shown in Figure 23 and Figure 24 for `tform1` or `tform2`, respectively.

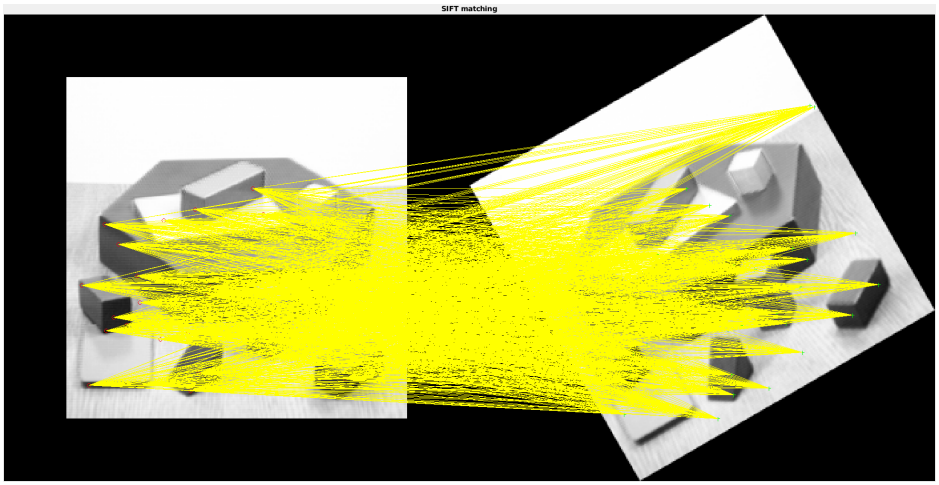


Figure 23: SIFT features of `blocks.png` image `tform1`.

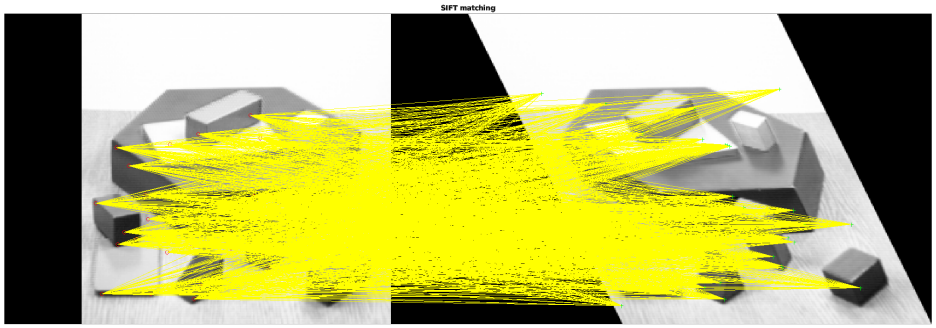


Figure 24: SIFT features of `blocks.png` image `tform2`.

Performance of features: Here are some specifications of HoG and SIFT:

- HOG is used to extract global feature whereas SIFT is used for extracting local features.
- SIFT is used for identification of specific objects using bag of words model for example. HOG is used for classification. For example classifying patches as pedestrians.
- SIFT, as it is named, is scale and rotation invariant whereas HoG is not. However, SIFT generally doesn't work well with lighting changes and blur, which is not the case in this assignment.

In conclusion, since SIFT transforms an image into a large collection of local feature vectors (local descriptors called SIFT keys), it is invariant to scaling, rotation and translation. Thus, it performs better on the transformed images.