# ELEC408-Computer Vision
# HW2-Hybrid Images
# Report
# 7.11.2016

EKİN AYÜREK
ID:31459

# Table of Contents

## 1. Introduction

In this assignment we are required to implement SIFT-like (Scale Invariant Feature Transform) algorithm in *Matlab*. I had a skeleton project code in the beginning and I should implement *get_interest_points.m*, *get_features.m*, *match_features.m* functions. In addition, we will prove that correlation matrix in colour gradients is an optimization problem for gradients. I also tried to make my algorithms scale-invariant for the bonus part of the homework. I will briefly explain my algorithm and after that I will show my results which are better than the result in the assignment website [1].

## 2. get_interest_points.m

I use Harris corner detection from our textbook [2]. I converted the image double image for more robust calculations.

```
image = im2double(image)
```

After that I defined my Gaussian filter for smoothing the image. It was very dependent to image. I found the best sigma values empirically and I will show them in the results section. The filter size is recommended to be feature_width. However, I found that feature_width+1 is works better than feature_width+1. I assume that, odd number of filter size is better because it centres the pixel.

```
sigma = 0.7;
gaussian_filter = fspecial('Gaussian', feature_width+1, sigma);
```

If the image colored, I use collar gradients in order to calculate correlation matrix. If the image I user intensity gradients. I use Matlab's *imgradientxy* function to calculate gradients and I use *imfilter* function filter images. If the images is colored, then I get the color gradients.

```
[g_ch1x, g_ch1y] =imgradientxy( imfilter( squeeze(image(:,:,1) ), gaussian_filter));
[g_ch2x, g_ch2y] = imgradientxy( imfilter( squeeze(image(:,:,2) ), gaussian_filter));
```

```
[g_ch3x, g_ch3y] = imgradientxy( imfilter( squeeze(image(:,:,3) ), gaussian_filter));
```

And our correlation matrix is:

$$C = D^T D = \begin{bmatrix} g_{ch1x} & g_{ch2x} & g_{ch3x} \\ g_{ch1y} & g_{ch2y} & g_{ch3y} \end{bmatrix} \begin{bmatrix} g_{ch1x} & g_{ch2y} \\ g_{ch2x} & g_{ch2y} \\ g_{ch3x} & g_{ch3y} \end{bmatrix}$$

$$= \begin{bmatrix} {g_{ch1x}}^2 + {g_{ch2x}}^2 + {g_{ch3x}}^2 & g_{ch1x} * g_{ch1y} + g_{ch2x} * g_{ch2y} + g_{ch3x} * g_{ch3y} \\ g_{ch1x} * g_{ch1y} + g_{ch2x} * g_{ch2y} + g_{ch3x} * g_{ch3y} & {g_{ch1y}}^2 + {g_{ch2y}}^2 + {g_{ch3y}}^2 \end{bmatrix}$$

Thus, I calculate the elements of c matrix:

```
weighting = fspecial('average', [3,3]);
c11 = imfilter(g_ch1x.*g_ch1x +  g_ch2x.* g_ch2x + g_ch3x.*g_ch3x, weighting);
c12 = imfilter(g_ch1x.*g_ch1y +  g_ch2x.* g_ch2y + g_ch3x.*g_ch3y, weighting); % =a21
c22 = imfilter(g_ch1y.*g_ch1y +  g_ch2y.* g_ch2y + g_ch3y.*g_ch3y, weighting);
```

In the above, I also filter the elements of c matrix to weighting over neighbourhoods. Basically, correlation matrix is averaged over neighbourhood pixels. It is recommended Gaussian weighting in the textbook [2]. However, I found that average weighting better works than Gaussian weighting. Note that, I first tried that with a for loop in which I was calculation C matrix for each (i,j) pairs but process time is very long. After I calculated it with a matrix multiplication only, process time decreased almos 10 times.

If the image is gray, the process is easier:

$$C = \begin{bmatrix} Ix^2 & Ix * Iy \\ Ix * Iy & Ix^2 \end{bmatrix}$$

Then,

```
[Ix, Iy] = imgradientxy( imfilter(image, gaussian_filter));
weighting = fspecial('average',[3,3]);
c11 = imfilter(Ix.^2, weighting);
c12 = imfilter(Ix.*Iy , weighting); % =a21
c22 = imfilter(Iy.^2, weighting);
```

I use the Harris corner function from textbook [2] as:

$$h = \det(C) - \alpha trace(C)^2$$

I define alpha is recommended one, $\alpha = 0.04$

```
alpha = 0.04;
%Determinan of 2x2 matrix: ad-bc
determinants = c11.*c22 - c12.*c12;
%Trace of 2x2 matrix = a+d
traces = c11 + c22;
```

```
%Harris corner function
h = determinants - alpha*traces.*traces;
```

After finding h, we should determine a threshold. I found that threshold may drastically change picture to picture. Therefore, I decided to use adaptive threshold. First, I determine threshold value according to mean, however it turns out that mean of h can be negative. Thus I define threshold as a constant times mean of the absolute value of h.

$$threshold = \beta mean(abs(h))$$

After that, I tried some constants 5,7,8,9,10 for β. I found that β = 10 is a good threshold. Thus,

```
threshold = h > 10 * mean2(abs(h));
%Applying threshold
h= h.*threshold;
```

Then, I do nom-maximal suppression by using Matlab's *imregionalmax* function.

```
h = imregionalmax(h,8;
```

After that, I found the coordinates of corners by using *find* function.

```
%Coordinates of non zero h elements=our corners.
[x,y] = find(h);
```

However, there can be corners near the edges for which we cannot calculate feature vector, because their size may extend image dimension. Thus, I return only valid indices.

```
image_sizex = size(image,1);
image_sizey = size(image,2);
%Valid indicies for a given featurewidth
valid_indicies = (x > feature_width/2+1) & (x+feature_width/2 < image_sizex +1) & (y >
feature_width/2+1) & (y+feature_width/2 < image_sizey + 1);
%Valid harris corners
x = x(valid_indicies);
y = y(valid_indicies);
```

I plot the corners on to image for visualization.

```
%Plotting for visualization of corners.
figure, imagesc(image), axis image, hold on
plot(y,x,'ro'), title('corners');
```

## 3. get_features.m

I use gray image in order to determine feature vectors, because It works well without color information. In addition, it speeds up the process time.

```
if length(size(image)) == 3
gray_image = rgb2gray(image);
else
gray_image = image;
end
```

We first filter image with a Gaussian filter for smoothing. I found that sigma=1.2 works well in that filter independent from images. The filter size for that Gaussian is again

feature_width+1. Then, I calculated gradient magnitudes and directions for filtered gray image by using Maltabs's *imgradient* function.

```
sigma = 1.2;
gaussian_filter = fspecial('Gaussian', feature_width+1, sigma);
[grad_magnitudes, grad_directions] =imgradient(imfilter(gray_image,gaussian_filter));
```

I order to find feature vectors, for each point I get patches from magnitudes of gradients and directions of gradients in the size of feature_width. As it is recommended in the textbook, I also multiple gradients with a Gaussian filter whose sigma is equal to 8. It increases feature matching drastically.

```
%x and y coordinates of keypoint
x_coordinate = x(i);
y_coordinate = y(i);
feature_part_gradients = grad_magnitudes(x_coordinate-feature_width/2 :
x_coordinate+feature_width/2-1, y_coordinate-feature_width/2 : y_coordinate+feature_width/2-
1).*fspecial('Gaussian', feature_width, 8);
feature_part_directions= grad_directions(x_coordinate-feature_width/2 :
x_coordinate+feature_width/2-1, y_coordinate-feature_width/2 : y_coordinate+feature_width/2-
1);
```

After that I partition the patch in to 4x4 arrays via for loop and with them I created 8 binned histograms. Then I concatenated those histograms to get 128 length feature vector.

```
for j=1:4
    for k=1:4
        % When we get the histogram of a cell we will append the
        % histogram array. However, I preallocate histogram vector,
        % therefore I find the location of new histogram and add the
        % histogram to that location.
         histogram( ((j-1)*32+(k-1)*8 +1): ((j-1)*32+k*8)  ) = getHistogram(
feature_part_directions((j-1)*feature_width/4 + 1: j*feature_width/4,   (k-1)*feature_width/4 + 1:
k*feature_width/4 )   ,   feature_part_gradients( (j-1)*feature_width/4 + 1: j*feature_width/4,   (k-
1)*feature_width/4 + 1: k*feature_width/4) );
    end
  end
features(i,:) = histogram';
```

getHistogram function first transform matrices to column vectors to be able to calculate histogram weights by just using inner product. I partition [-180,180] interval to 8 part. For each bin, I found the gradients whose directions lies in that bin and I summed them up by inner product.

```
function histogram = getHistogram(Gdir, Grad)
%To speed up I trasform matricies to column vectors.
Gdir = Gdir(:);
Grad = Grad(:);
%I use conditions on arrays to determine a gradients' histogram locations.
%For examples (Gdir>=-180 & Gdir<-135) is 1 for the gradients whose
%directions between [-180,-135] and gives zero for others. And I inner
%product that with Grad vector to find sum of the gradients whose
%directions  between [-180,-135].
%I do same thing for other 7 intervals.
histogram = [(Gdir>=-180 & Gdir<-135)'*Grad; (Gdir>=-135 & Gdir<-90)'*Grad; ( Gdir>=-90 &
Gdir<-45)'*Grad; (Gdir>=-45 & Gdir<0)'*Grad; (Gdir>=0 & Gdir<45)'*Grad; (Gdir>=45 &
Gdir<90)'*Grad; (Gdir>=90 & Gdir<135)'*Grad; (Gdir>=135 & Gdir<180)'*Grad];
```

```
%End of the function
end
```

After that I normalize feature vectors and clip them to 0.2 and normalize again.

```
%Normalizing features vector.
features = normc(features);
%Clipping excessive histogram values to 0.2. And renormalizing it to 1.
features =normc(features.*(features<0.2) + 0.2*(features>=0.2) );
%End of the function
```

## 4. match_features.m

In order to match features I used ratio test. Ratio test said that:

$$if \ \frac{distance\ over\ nerast\ feature\ vector}{distance\ over\ second\ nerast\ feature\ vecotr} < Threshold$$

Then accept that matching.

If the ratio is bigger then threshold we are not confident it is a good matching. In order to implement this, first I should find distance between each (i,j) matching from features of image1 to features of image2. Matlab has a built in function *pdist2* does that. From the help command of pdist2:

> *D = pdist2(X,Y) returns a matrix D containing the Euclidean distances between each pair of observations in the MX-by-N data matrix X and MY-by-N data matrix Y. Rows of X and Y correspond to observations, and columns correspond to variables. D is an MX-by-MY matrix, with the (I,J) entry equal to distance between observation I in X and observation J in Y.*

I use Euclidian distance,

```
distances = pdist2(features1, features2, 'euclidean');
```

After that I sort those distances in ascending order,

```
[distances_sorted, index_matrix] = sort(distances, 2, 'ascend');
```

Then ratio becomes:

```
ratio = (distances_sorted(:,1)./distances_sorted(:,2));
```

I eliminate ratios greater then threshold.

```
test = ratio < threshold;
```

I also find the confidences for good matches,

```
confidences = (1./ratio(test,:));
matches = zeros(size(confidences,1), 2);
```

Then I find the matching points by using find function.

```
%Find that elements' indexes for the first image
matches(:,1) = find(test);
%Find the corresponding element in second image
matches(:,2) = index_matrix(test, 1);
```

## 5. proj2.m

When I started to code I calculated [x,y] pairs in revers order. I realize it later. Thus, I changed the order in *show_correspondance* part and *evaluate_correspodance* functions.

```
num_pts_to_visualize = size(matches,1);
%num_pts_to_visualize = 20;
show_correspondence(image1, image2, y1(matches(1:num_pts_to_visualize,1)), ...
                            x1(matches(1:num_pts_to_visualize,1)), ...
                            y2(matches(1:num_pts_to_visualize,2)), ...
                            x2(matches(1:num_pts_to_visualize,2)));
```

```
num_pts_to_evaluate = size(matches,1);
% All of the coordinates are being divided by scale_factor because of the
% imresize operation at the top of this script. This evaluation function
% will only work for the particular Notre Dame image pair specified in the
% starter code. You can, however, use 'collect_ground_truth_corr.m' to
% build ground truth for additional image pairs and then change the paths
% in 'evaluate_correspondence' accordingly. Or you can simply comment out
% this function once you start testing on additional image pairs.
evaluate_correspondence(y1(matches(1:num_pts_to_evaluate,1))/scale_factor, ...
                        x1(matches(1:num_pts_to_evaluate,1))/scale_factor, ...
                        y2(matches(1:num_pts_to_evaluate,2))/scale_factor, ...
                        x2(matches(1:num_pts_to_evaluate,2))/scale_factor);
```

## 6. Results

### Notre Dame

I found that best sigma in *get_interest_points.m* is 1. And best threshold in matches is threshold =0.665. For that sigma and thresh old I got 67 good points and 0 bad points. If I increase threshold, Then, it starts to do some bad matches. However, until 67 points there is no bad matches. I also increased feature width to 20 and It results with 78/0 good/bad points.

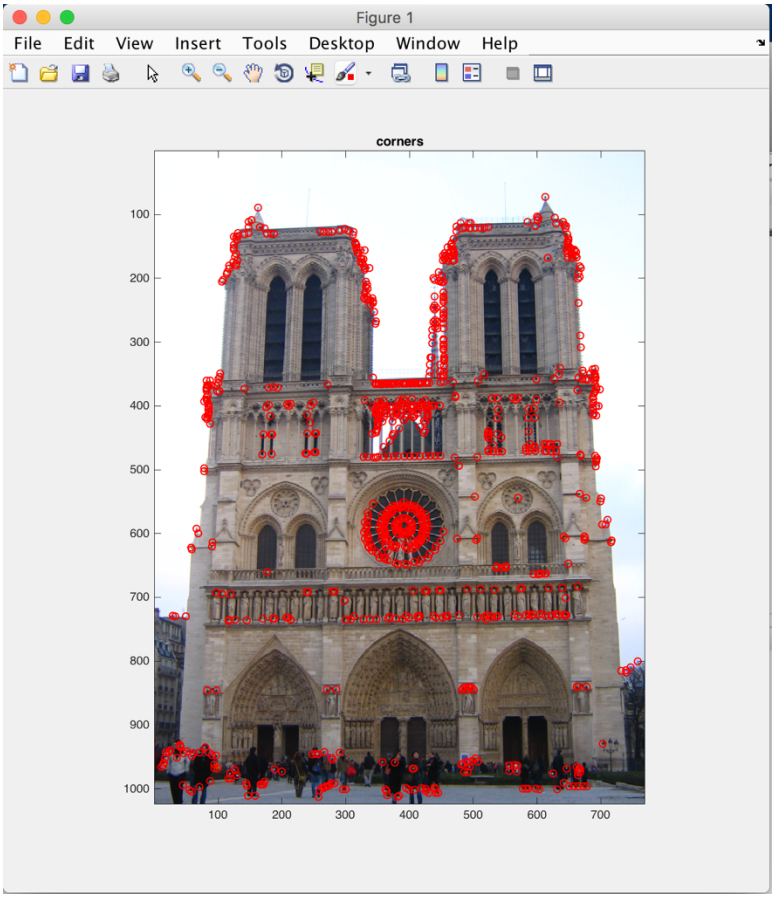


Fig1. Corners in NotreDame picture(sigma=1)

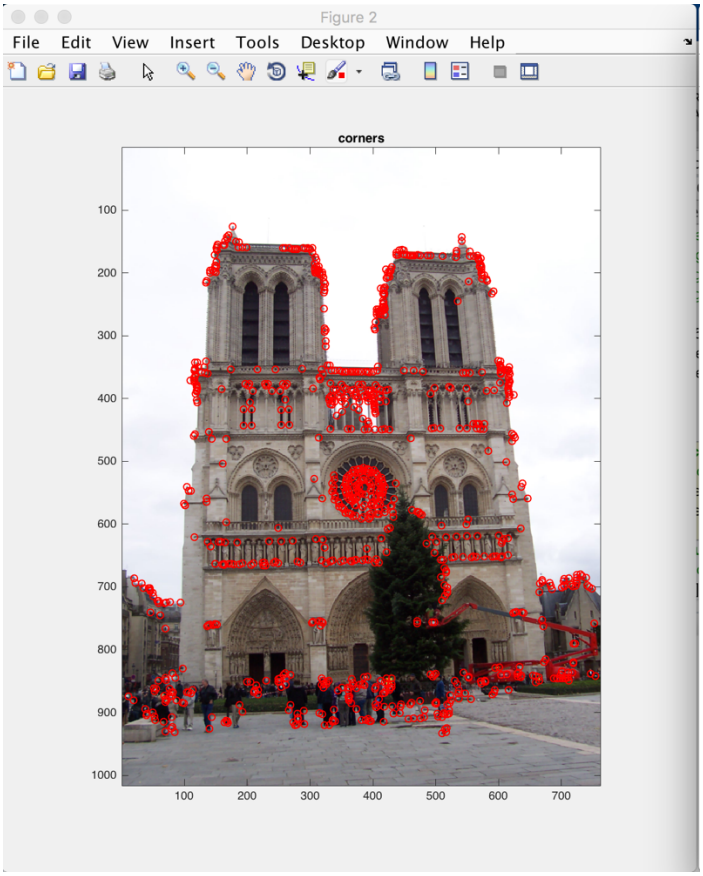

Fig2. Corners in the NotreDame2 picture (sigma=1)

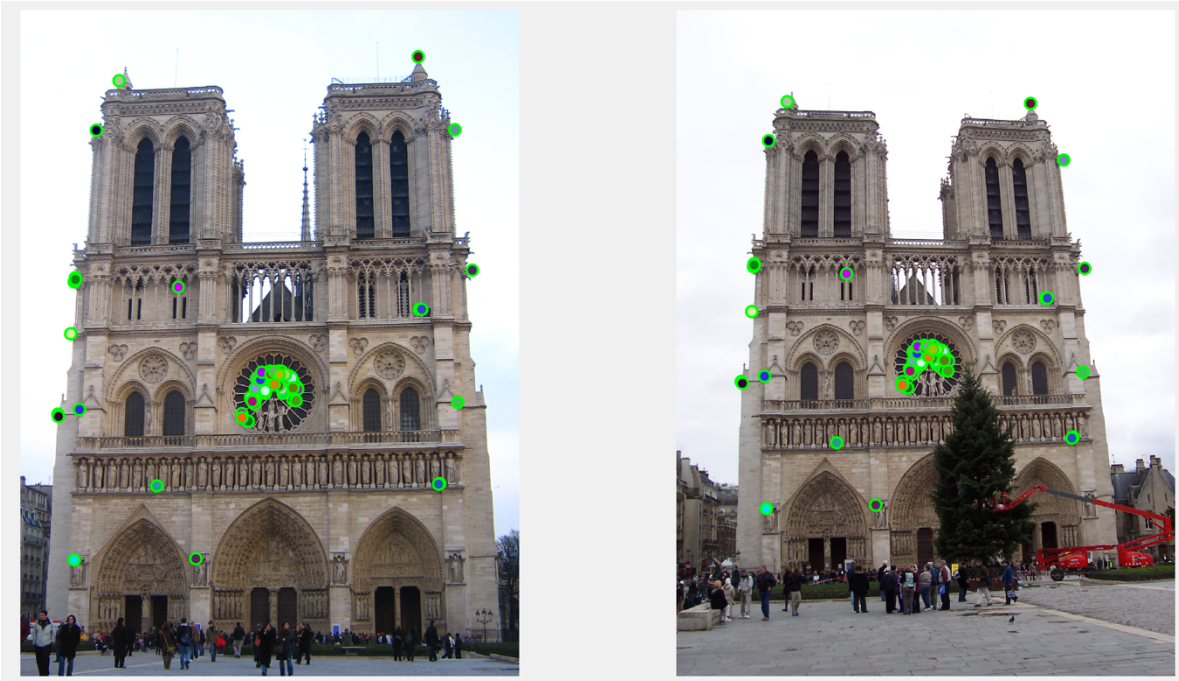

Fig3. Matching in the NotreDame pictures (67 good/0 bad matching)
**(%100 accuracy**), (sigma=1, threshold=0.665)

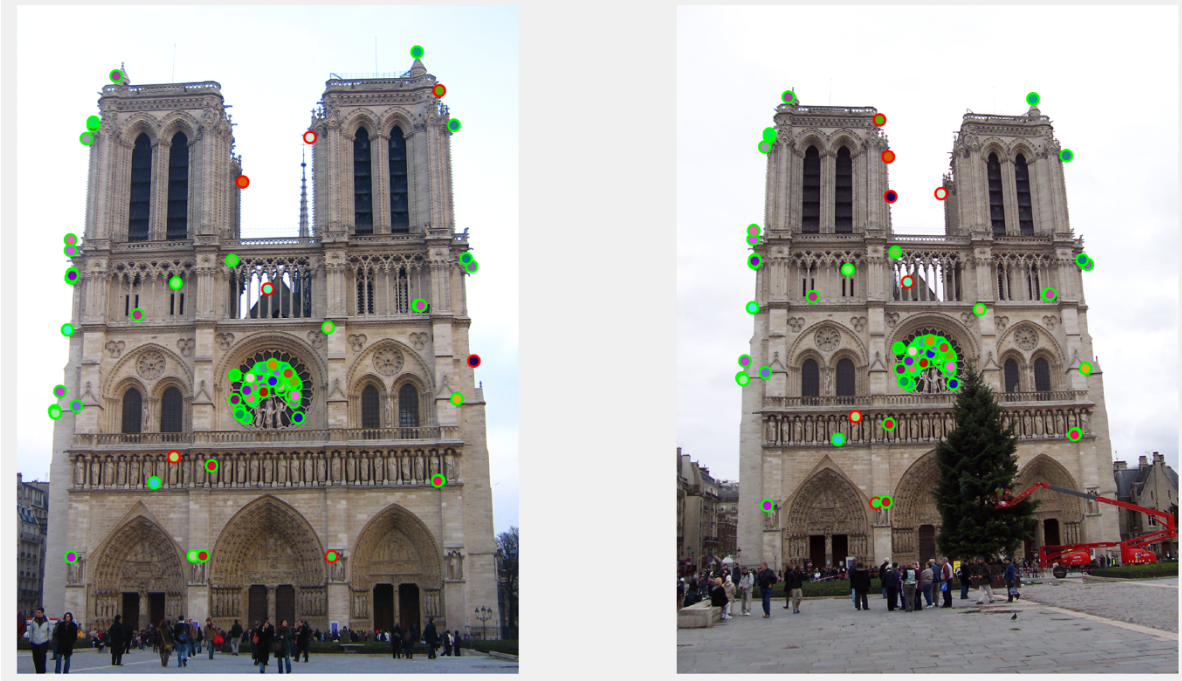

Fig4. Matching in the NotreDame pictures (94 good/7 bad matching)
(% 93.1 accuracy), (sigma=1, threshold=0.72)

## Keble

I found that best sigma value for Keble is 0.3 and best threshold values is 0.62. In that value it has %100 good matches.

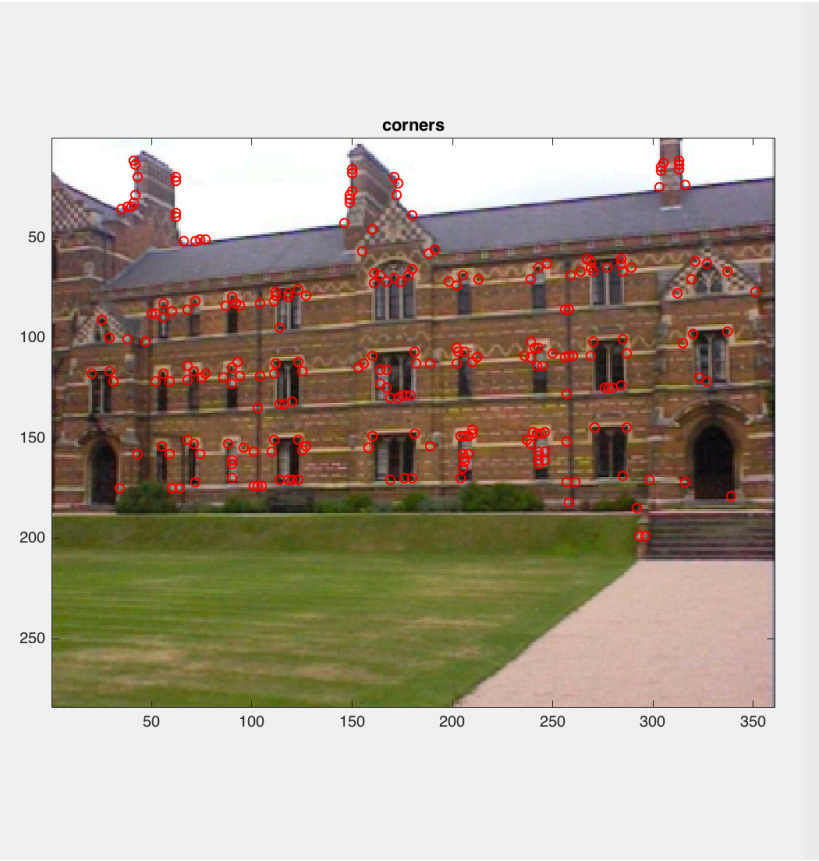


Fig5. Corners in Keble1(sigma=0.3)

Fig6. Corners in Keble2(sigma=0.3)

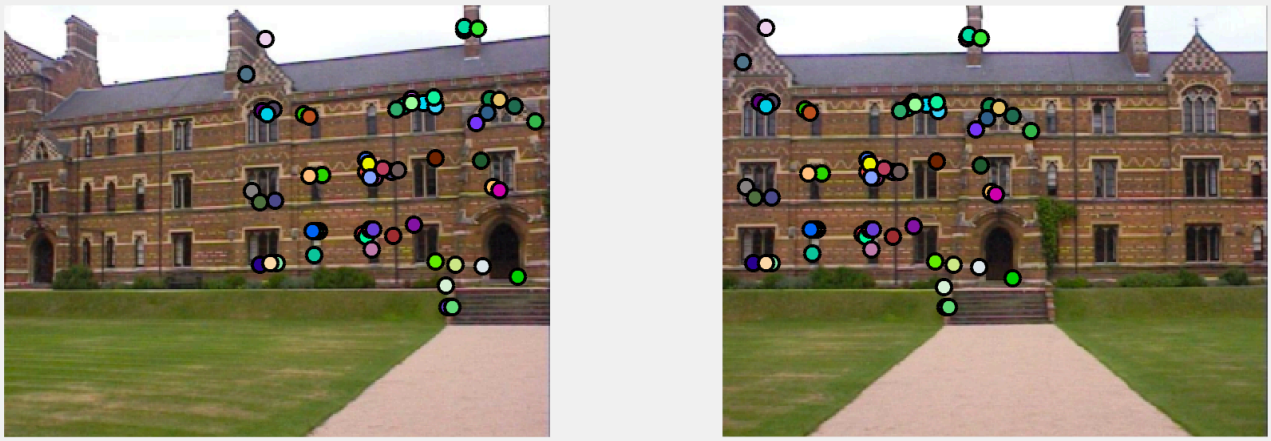

Fig7. Matchin in Keble Images(sigma=0.3, threshold=0.62)(61 Good/0 Bad matches)(%100 accuracy)

## Library:

I took photo of same scene with different positions in the SKL in the university. I test my algorithm at those images.


Fig8. Corners in library1(sigma=0.7)

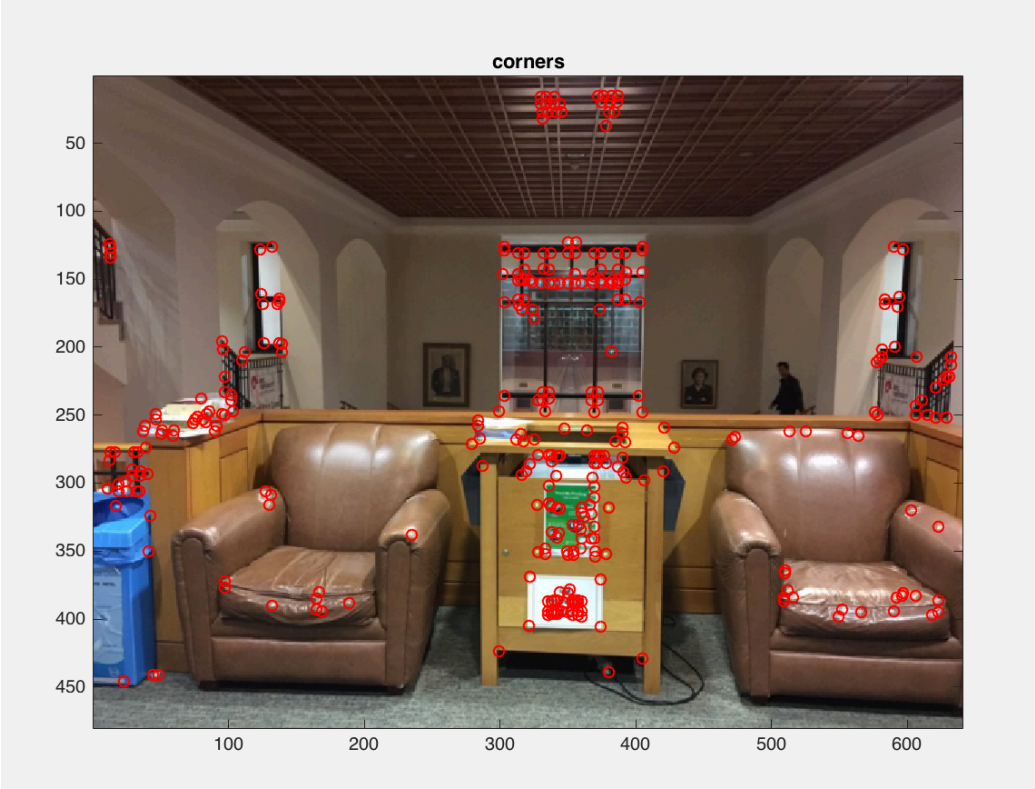


Fig9. Corners in library2(sigma=0.7)

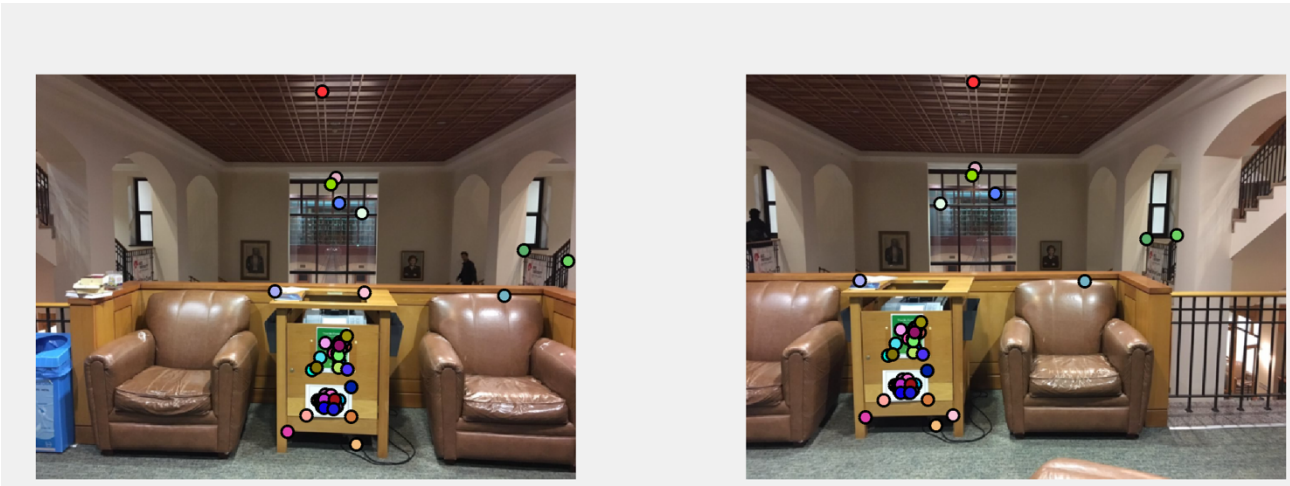

Fig10. Matching in Library pictures (sigma=0.7, threshold=0.7)(~43good matches/~3 bad matches)

## Me:

I took my photo and I tried algorithm on my pictures.

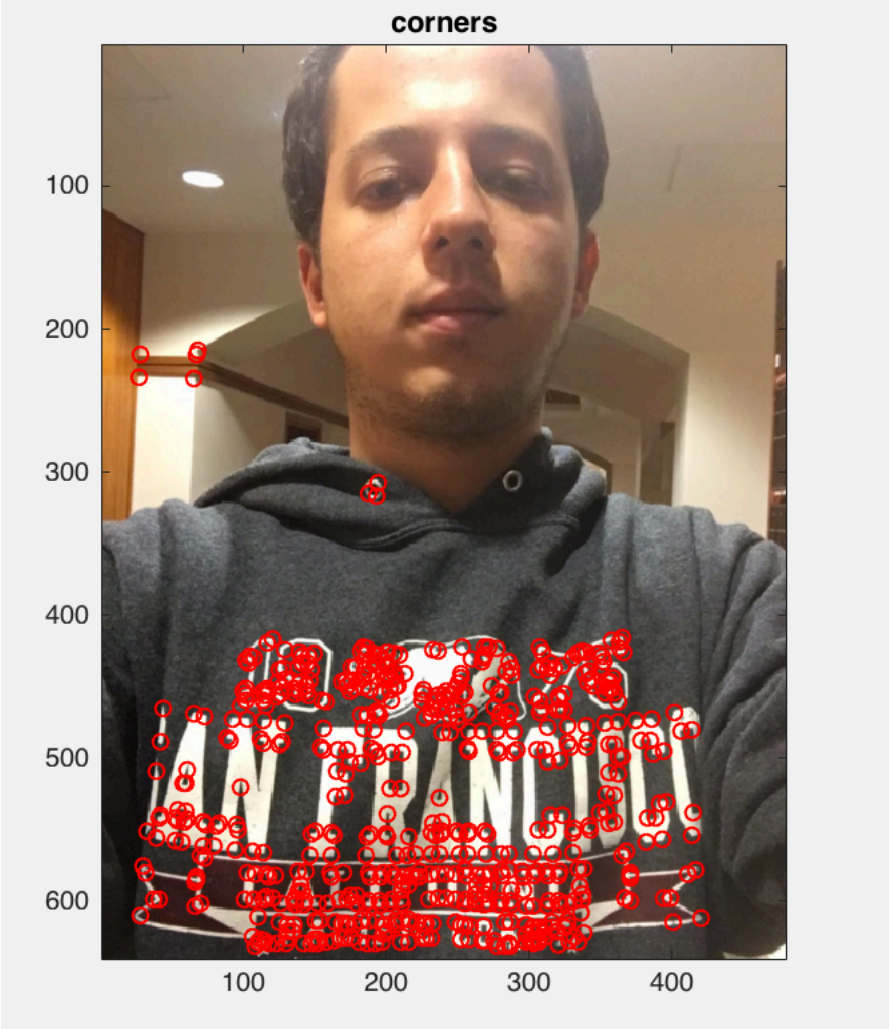


Fig11. Corners in Me1.(sigma=0.5)


Fig12. Corners in Me2(sigma=0.5)

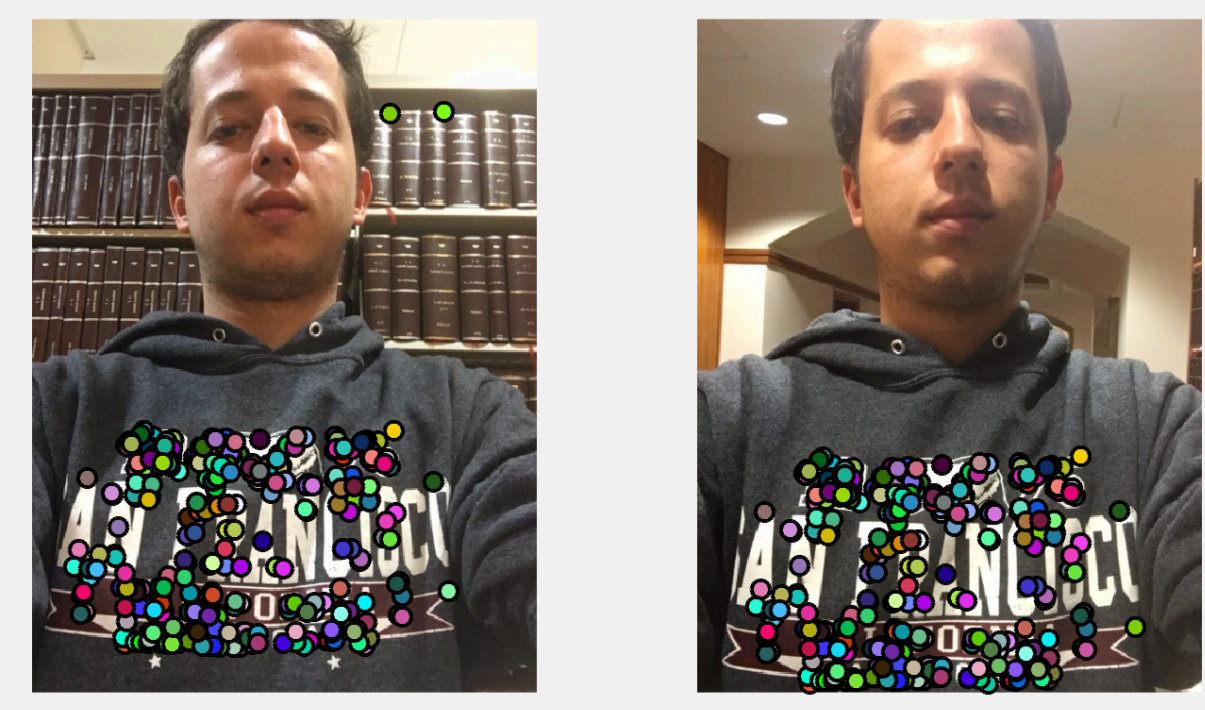

Fig13. Matching in Me pictures. (sigma=0.5,threshold=0.65)

## 7. PCA for Color Gradient

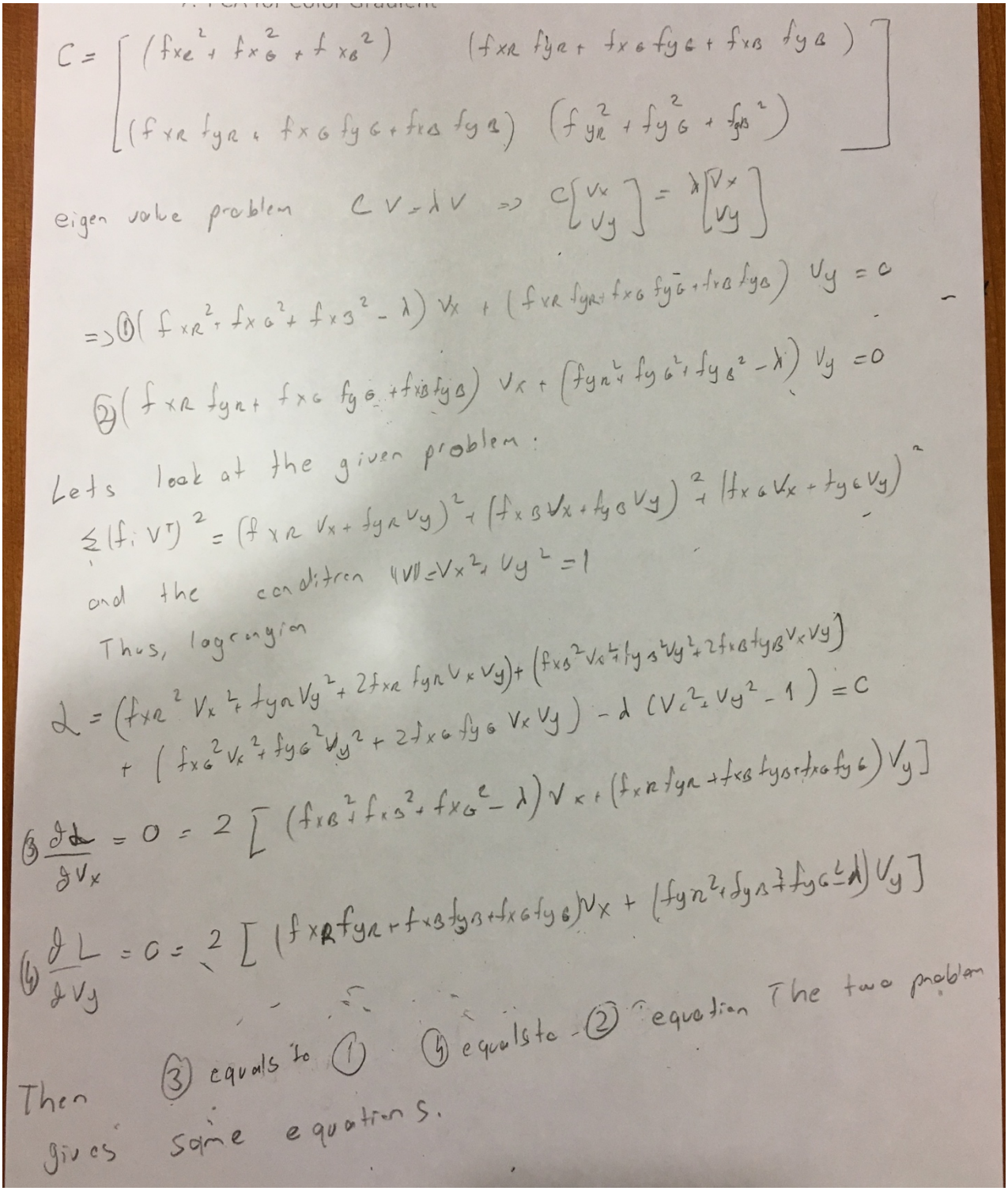


Fig14. Solution

## 8. Conclusion

My algorıthm works very well on given two images and images that I took. However, it didn't work properly on Shrek image in the dataset. Because my algorithm is not rotational invariant. I started to implement scale and rotational invariant sift but there was no enough time to complete it. I add normal and scale invariant part sperately.

## 9. Bonus Part

### Scale Invariant Feature Detection

I completed the scale invariant part in my project. In the get_interest points, I start from sigma=0.3 and I multiply sigma with $2^{1/4}$ in each loop as suggested in lecture slides. In each loop, I find the harris corner function and I look at the points.

```
for n = 1:15
sigma = scale_factor*sigma;
```

If they are local maximum in LoG kernels of the image, then I add that point as a corner. The results for Shrek image is shown in the below.

```
image_above = (sigma/scale_factor)^2*imfilter(gray_image, fspecial('log', [3,3], sigma/scale_factor) );
image_current =(sigma)^2 * imfilter(gray_image, fspecial('log', [3,3], sigma));
image_belove =(sigma*scale_factor)^2 * imfilter(gray_image, fspecial('log', [3,3], sigma*scale_factor));
local_maximums =  image_current > image_above & image_current > image_belove;
h= h.*local_maximums;
scale_xy = scale_xy + (h-xy>0).*sigma;
xy = (xy+h)>0;
```


Fig15. Corners in Schrek reference image.

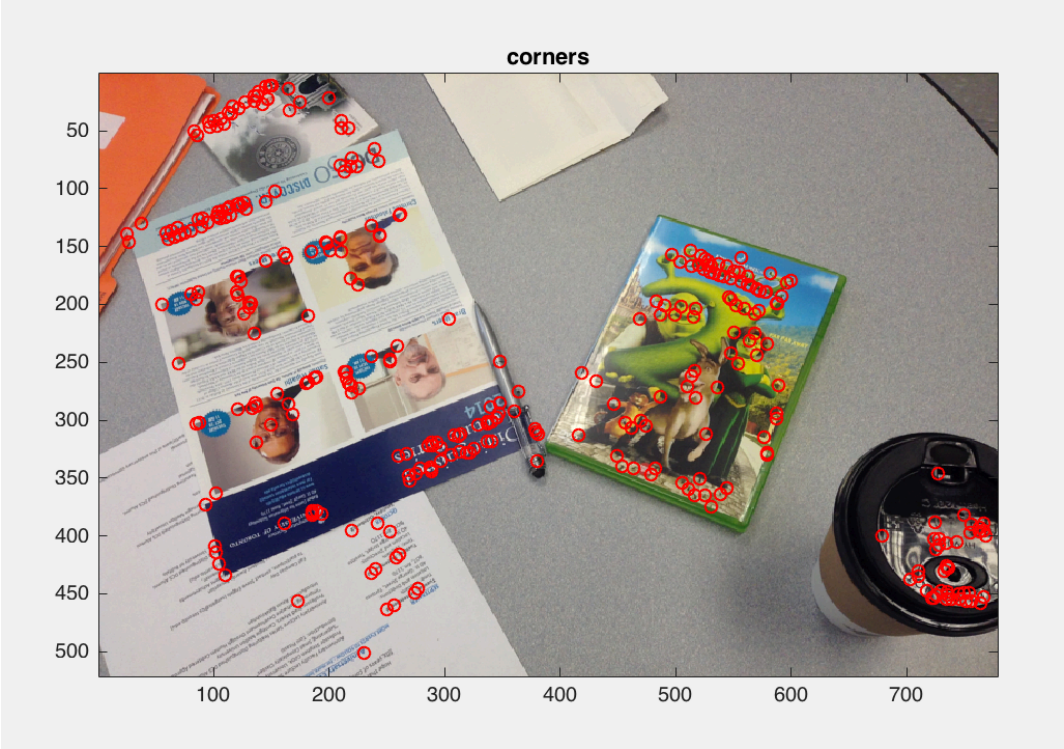


Fig16. Corners in shrek image2.