Computer Vision ENGN4528

CLAB 02

Yuge Shi
u5634555

Task 1: Implement your own Harris corner detector:

Figure 1.1: Results of corner detection with image *Right.png*

Figure 1.2: Results of corner detection with image *pepper.png*

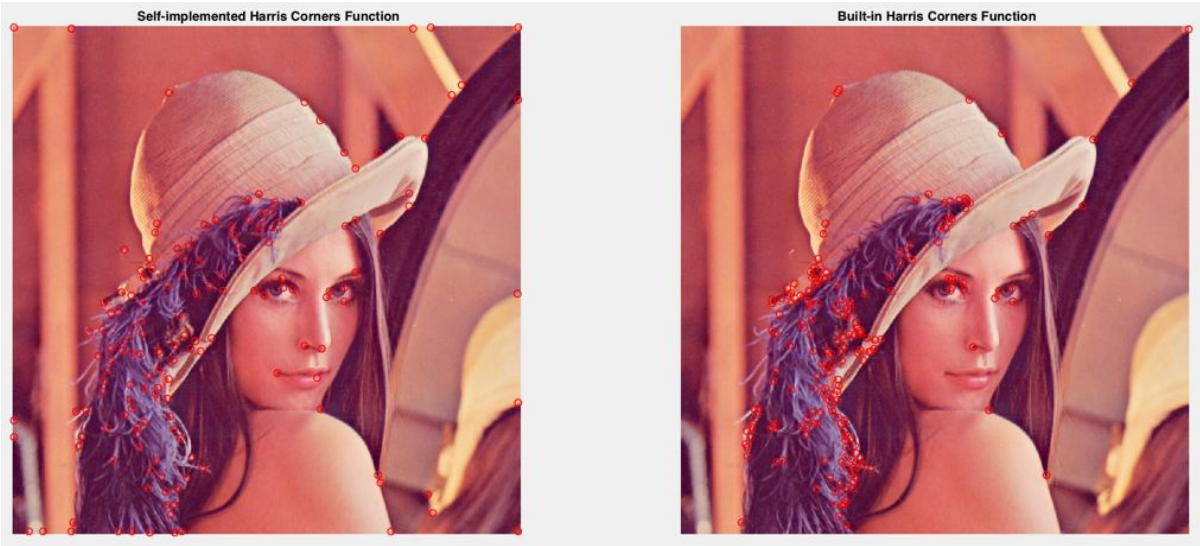

Figure 1.3: Results of corner detection with image *Lenna.png*

Figure 1.4: Results of corner detection with image *mandm.jpg*

It can be seen from the above images that the corners of the image are well-detected both in self-implemented function and built-in function. The change of any input of the corner detection function such as threshold, sigma of the Gaussian filter, radius(sze) could give rise to alterations to the result. The code of this task can be found in *Appendix A*.

Task 2: Implement your own k-means clustering function and use it for colour image segmentation

Figure 2.1.1: Boundary image of *Right.png*

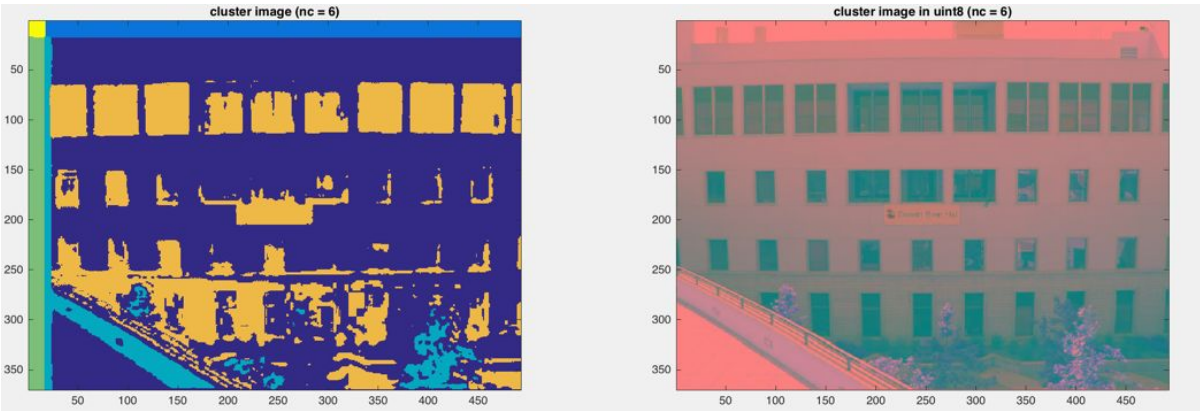

Figure 2.1.2: Cluster images of *Right.png*

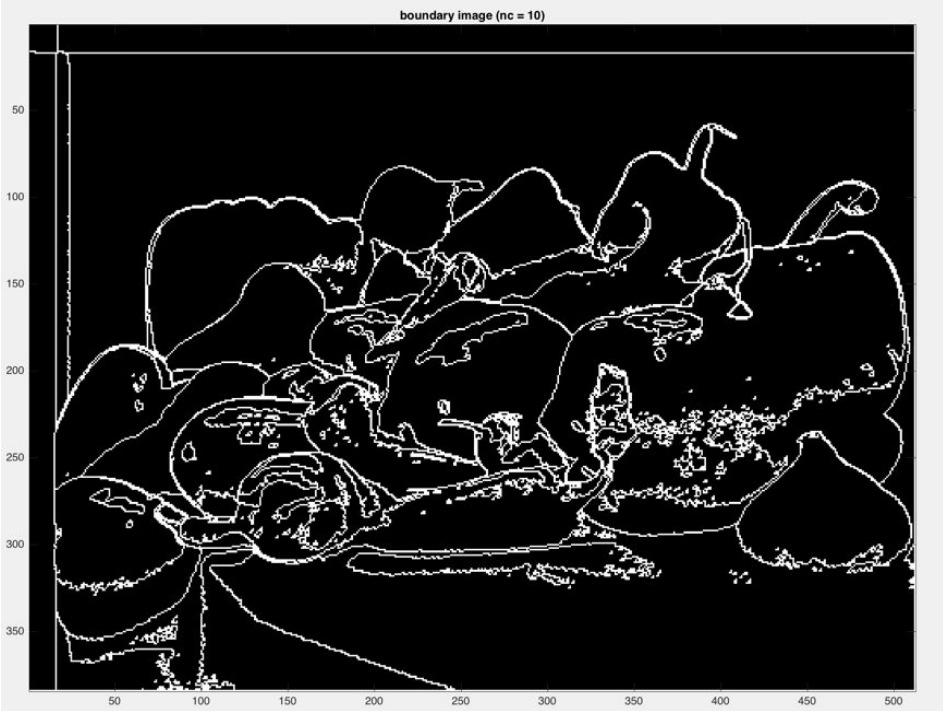

Figure 2.2.1: Boundary of image *pepper.png*

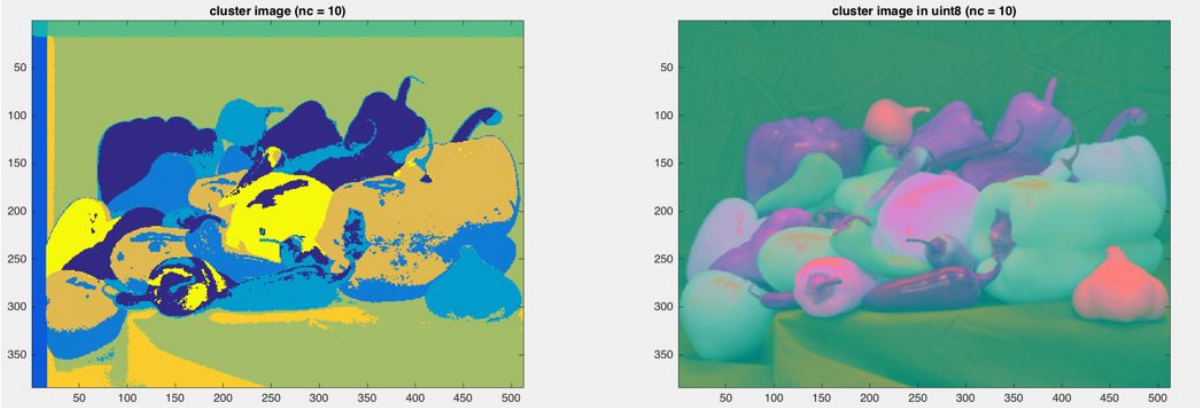

Figure 2.2.2: Cluster images of *pepper.png*

Figure 2.3.1: Boundary of image *Lenna.png*

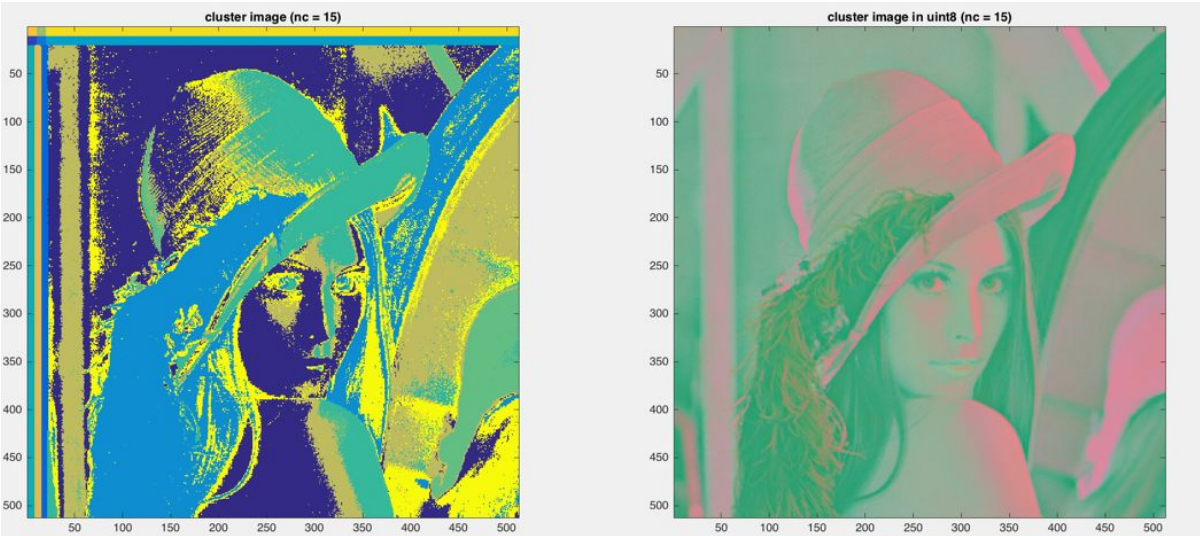

Figure 2.3.2: Cluster images of *Lenna.png*

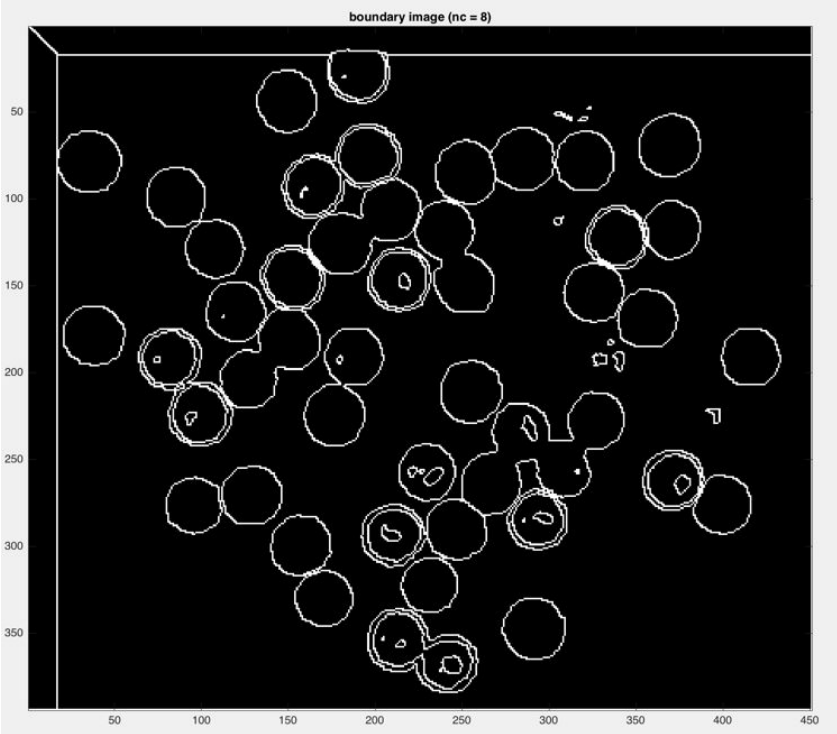

Figure 2.4.1: Boundary of image *mandm.jpg*

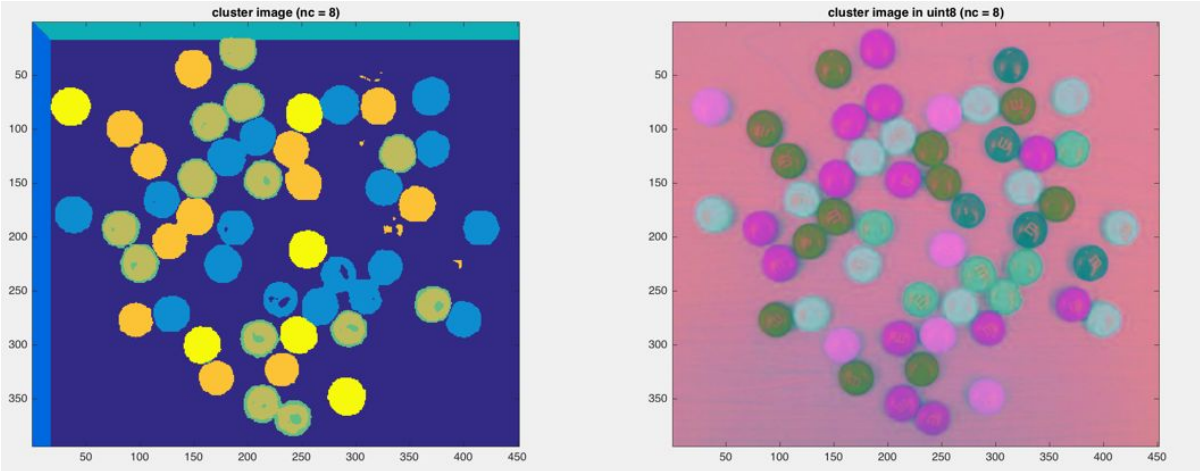

Figure 2.4.2: Cluster images of *mandm.jpg* (nc = 8)

It can be seen from the above images that different parts/colours of the images get assigned to different clusters. It is also noted that:

1. When the weighting for position increases, the position feature would be taken more into consideration when clustering and thus would give rise to different results;
2. A change in *nc*, i.e. the number of clustering centre, could also give very different outcomes;
3. The results of clustering can vary in different runnings. This is resulted from the fact that the clustering centre are assigned randomly at first.

*The verification of these observations are displayed in *Appendix D.*

The code for this task is included in *Appendix B.*

Task 3: Play with SIFT code, and implement an image match algorithm

Step 1: Understand the provided functions and draw SIFT key feature points using the functions

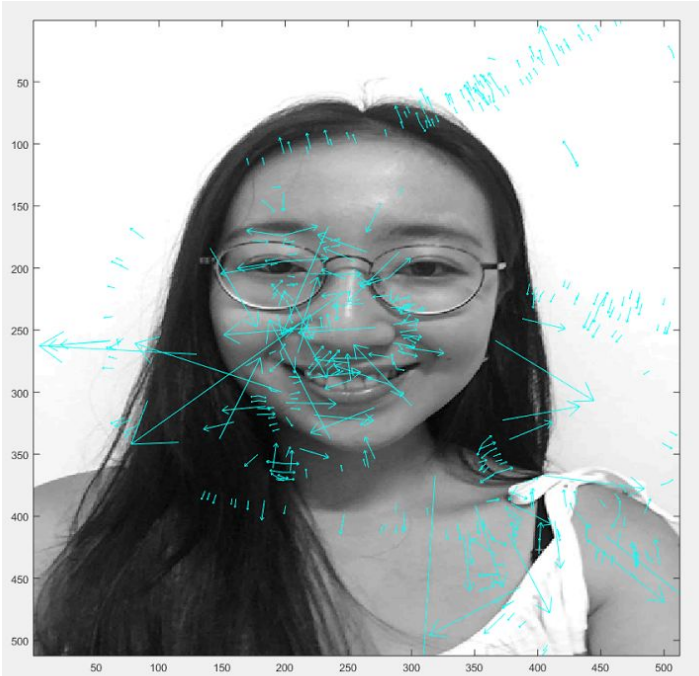

Figure 3.1.1: Key points of original image

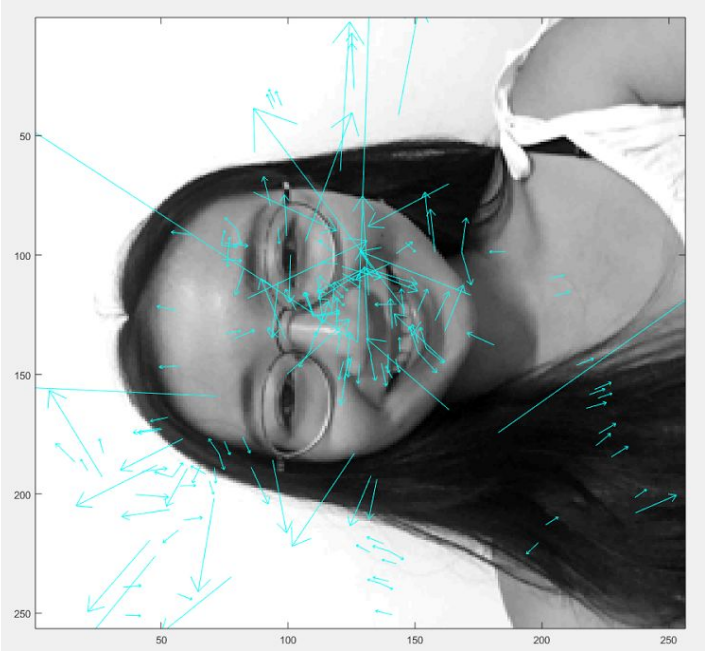

Figure 3.1.2: Key points of modified image *img2* (rotate 90° counter-clockwise, scale 0.5)

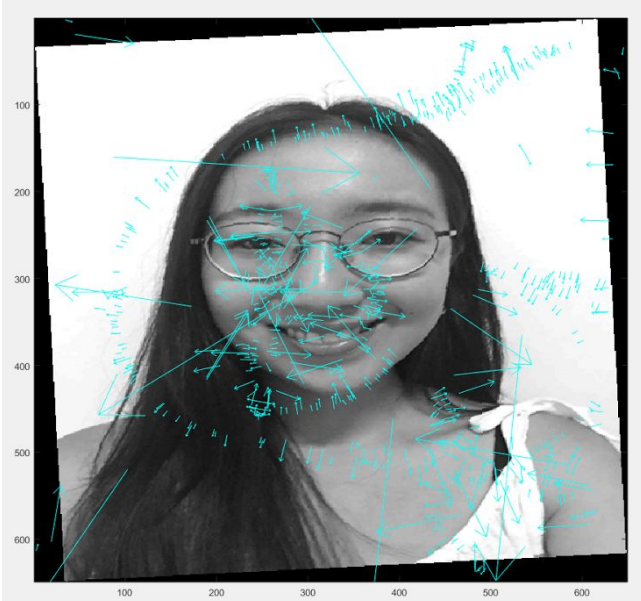

Figure 3.1.3: Key points of modified image *img3* (rotate 3° counter-clockwise, scale 1.2)

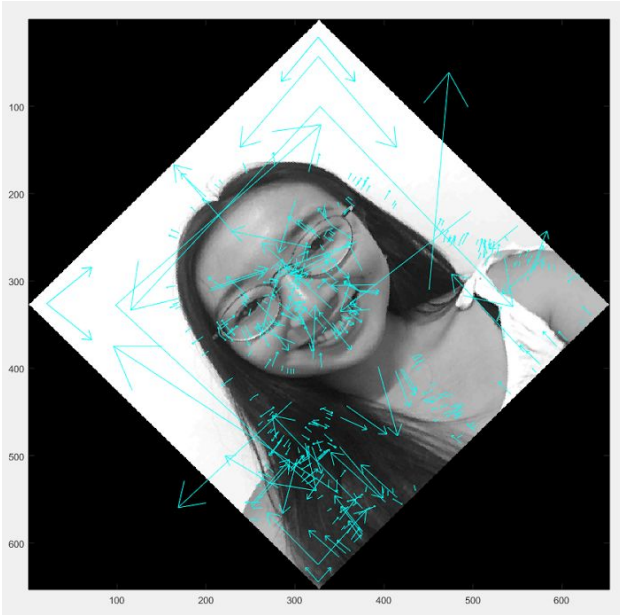

Figure 3.1.4: Key points of modified image *img2* (rotate 45° counter-clockwise, scale 0.9)

Step 2: Display two images side-by-side on a 1x2 image panel. Compute their SIFT feature points, find feature matches between the two images, and draw lines connecting the matched SIFT feature points.

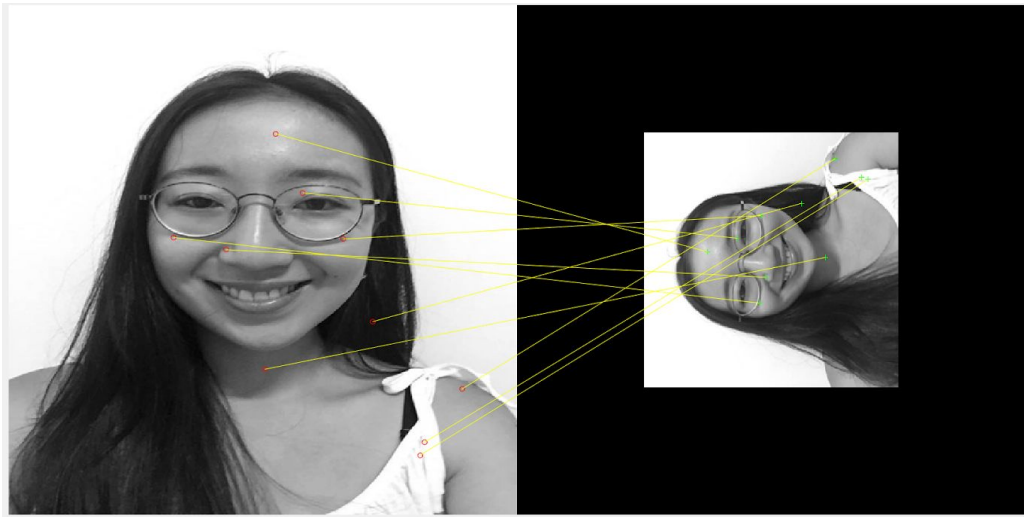

Figure 3.2.1: Connecting the matched SIFT feature points between *img1* and *img2*

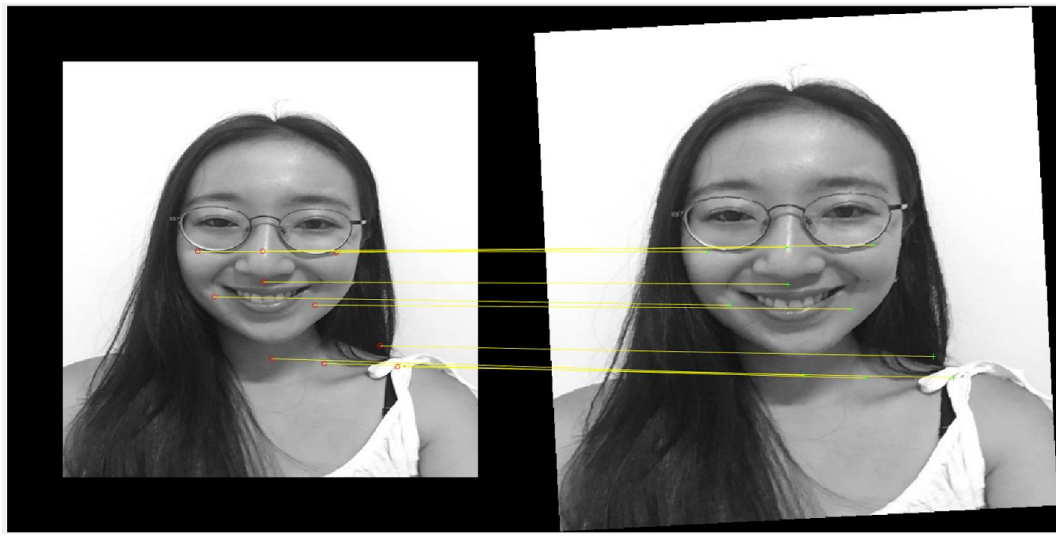

Figure 3.2.2: Connecting the matched SIFT feature points between *img1* and *img3*

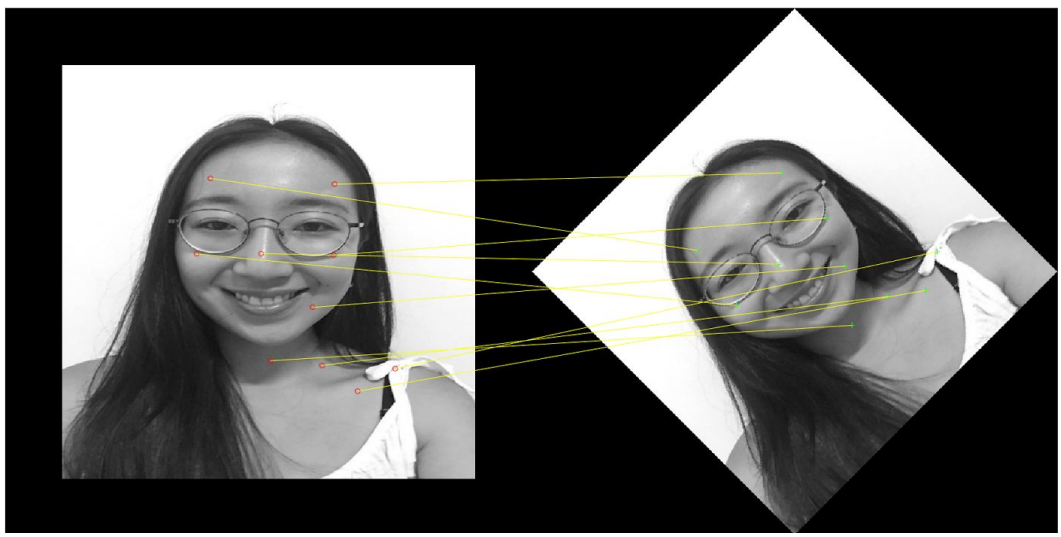

Figure 3.2.3: Connecting the matched SIFT feature points between *img1* and *img4*

It is clear from the above images that the function is capable of matching feature points no matter how the image is rotated or resized, which again verified that the SIFT descriptor is invariant with rotation and scaling. The code for this part can be found in *Appendix C.*

Appendix A: Code for Task 1

```
close all;
clear all;
im = imread(input('input the image name:'));
bw = rgb2gray(im);  % convert to grey scale image

% Harris Corner detection
sigma = 2; thresh = 30000000; sze = 11; disp = 0;

% derivative masks
dy = [-1 0 1; -1 0 1; -1 0 1]; % derivative mask in y direction
dx = dy'; % derivative mask in x direction

% image derivatives
Ix = conv2(double(bw), dx, 'same'); % find derivative of x direction using convlution
Iy = conv2(double(bw), dy, 'same'); % find derivative of y direction using convolution

% Calculating the gradiant of the image Ix and Iy
g = fspecial('gaussian', max(1,fix(6^sigma)), sigma); % construct Gaussian filter
Ix2 = conv2(Ix.^2, g, 'same'); %smooth image drivatives with Gaussian filter
Iy2 = conv2(Iy.^2, g, 'same');
Ixy = conv2(Ix.*Iy, g, 'same');

% Compute the cornerness
k = 0.04; % range of k: 0.01~0.1
cornerness = (Ix2.*Iy2 - Ixy.^2) - k*(Ix2 + Iy2).^2; % R =
lambda1*lambda2-k*(lambda1+lambda2)^2
% the equation above gets a similar enough result with the eigen value one,
% it is implemented here to save computation time

% Non-maximum suppression and threshold
mx = ordfilt2(double(cornerness), sze^2, ones(sze));       % Grey-scale dilation
% function ordfilt2:
%       replaces each element in cornerness by the last element of
%       its 11*11 sorted neighbour (ranking from smallest to largest),
%       which would be 1 as long as there is one pixel in its
%       neighbour is 1. The dilation effect is thus achieved.
cornerness = (double(cornerness) == mx) & (double(cornerness) > thresh);  % Find local
maxima

% self-implemented function plot
[rws, cols] = find(cornerness); % extract the row and column number from cornerness
subplot(1,2,1),imshow(im);
hold on;
p = [cols, rws];
plot(p(:,1),p(:,2), 'or'); % plot the corner points on the image
```

```
title('\bf Self-implemented Harris Corners Function')

% built-in function plot
coor = corner(bw,'Harris');
subplot(1,2,2),imshow(im);
hold on;
plot(coor(:,1),coor(:,2), 'or');
title('\bf Built-in Harris Corners Function')
```

Appendix B: Code for Task 2

Main function:

```
close all
clear all
% read color image and convert to 24 bits
img = imread('Right.jpg');
% uncomment following if the image is 16 bits:
% img = uint8(img/256);

cform = makecform('srgb2lab');
lab = applycform(img, cform);

% calling the k-mean and display functions
features = im2feature(lab);
[data,stats] = my_kmeans(features,6);
displayclusters(lab, data);
```

K-mean function:

```
function [data_clusters, cluster_stats] = my_kmeans( data, nc )
% This function performs k-means clustering on data ,
% given (nc) = the number of clusters.
% Random Initialization
data = double(data);
ndata = size(data,1);
ndims = size(data,2);
% randomly distribute the data to 1~nc
random_labels = floor(rand(ndata,1) * nc) + 1;
data_clusters = random_labels;
% statistic of each cluster (nc+data)
cluster_stats = zeros(nc,ndims+1);
% distance of each point to its own cluster centre
distances = zeros(ndata,nc);
while(1)
   pause(0.03);
   % Make a copy of cluster statistics for
   % comparison purposes.  If the difference
   % is very small, the while loop will exit.
   last_clusters = cluster_stats;
   % For each cluster
   for c=1:nc
      % Find all data points assigned to this cluster
      [index] = find(data_clusters == c);
      num_assigned = size(index,1); % count index (rows)
      % some heuristic codes for exception handling.
      if( num_assigned < 1 )
```

```
            disp('No points were assigned to this cluster, some special processing is given
below');
            % assign the data that's the furthest to all the other clusters
            % to the unassigned cluster
            [maxx,cluster_num] = max(max(distances));
            [maxx,data_point] = max(distances(:,cluster_num));
            data_clusters(data_point) = cluster_num;
            index = data_point;
            num_assigned = 1;
        end   %% end of exception handling.

        % Save number of points per cluster,  plus the mean vectors.
        cluster_stats(c,1) = num_assigned;
        if( num_assigned > 1 )
            summ = sum(data(index,:));
            cluster_stats(c,2:ndims+1) = summ ./ num_assigned;
        else % if = 1
            cluster_stats(c,2:ndims+1) = data(index,:);
        end

    end

    % Exit criteria
    diff = sum(abs(cluster_stats(:) - last_clusters(:)));
    if( diff < 0.00001 )
        break;
    end

    %  Assign each point to the nearest cluster center
    cluster_position = cluster_stats(:,2:end);
    for i = 1:nc
        a = data-repmat(cluster_position(i,:),ndata,1);
        distances(:,i) = sqrt(sum(a.^2,2));
    end

    %update the membership assignment, i.e., update the data_clusters with current values.
    for i = 1:ndata
        [x,data_clusters(i)] = find(distances(i,:) == min(distances(i,:)),1);

    end
 end
```

Appendix C: Code for Task 3

```
close all;
clear all;

%% read, modify and write the image
img1 = imresize(rgb2gray(imread('2.jpg')), [512 512]);
img2 = imrotate(imresize(img1,0.5),90);
img3 = imrotate(imresize(img1,1.2),3);
img4 = imrotate(imresize(img1,0.9),45);

imwrite(img1,'img1.png');
imwrite(img2,'img2.png');
imwrite(img3,'img3.png');
imwrite(img4,'img4.png');

%% implement sift and show keypoints
[img_1,descriptors1,locs1] = sift('img1.png');
[img_2,descriptors2,locs2] = sift('img2.png');
[img_3,descriptors3,locs3] = sift('img3.png');
[img_4,descriptors4,locs4] = sift('img4.png');

showkeys(img_1,locs1);
showkeys(img_2,locs2);
showkeys(img_3,locs3);
showkeys(img_4,locs4);

%% matching keypoints
% except for the self-implemented function, built-in function matchFeatures
% can readily be used with the descriptors as input
min_position = [];
for i = 1:size(descriptors1,1)
    descriptor1_row = repmat(descriptors1(i,:),size(descriptors2,1),1); %map one row of
descriptor1 to the size of descriptor2
    % calculate the distance using matrix operation
    difference = descriptor1_row - descriptors2;
    distances = sqrt(sum(difference.^2,2));
    min_distance = min(distances); %find minimum distance
    % find second smallest distance
    distances1 = distances;
    distances1(distances1 == min_distance) = 10;
    min2_distance = min(distances1);
    % when minimum distance smaller than k of the second minimum,
    % the two keypoints match. k lies in 0.6~0.8.
    if min_distance <= 0.6*min2_distance
```

```
        % store the keypoint position of minimum distance (in descriptor)
        min_position = [min_position; [i, find(distances == min_distance,1)]];
    end
end
% find the location of keypoint
matchpoints1 = locs1(min_position(1:10,1),1:2);
matchpoints2 = locs2(min_position(1:10,2),1:2);
% plot & connecting matching keypoints
figure; showMatchedFeatures(img1, img2,matchpoints1, matchpoints2, 'montage', 'parent',
axes);

% the code below are pretty much the same with the above chunk, just with
% different images.
min_position = [];
for i = 1:size(descriptors1,1)
    descriptor1_row = repmat(descriptors1(i,:),size(descriptors3,1),1);
    difference = descriptor1_row - descriptors3;
    distances = sqrt(sum(difference.^2,2));
    min_distance = min(distances);
    distances1 = distances;
    distances1(distances1 == min_distance) = 10;
    min2_distance = min(distances1);
    if min_distance <= 0.6*min2_distance
        min_position = [min_position; [i, find(distances == min_distance,1)]];
    end
end
matchpoints1 = locs1(min_position(1:10,1),1:2);
matchpoints2 = locs3(min_position(1:10,2),1:2);
figure; showMatchedFeatures(img1, img3,matchpoints1, matchpoints2, 'montage', 'parent',
axes);

min_position = [];
for i = 1:size(descriptors1,1)
    descriptor1_row = repmat(descriptors1(i,:),size(descriptors4,1),1);
    difference = descriptor1_row - descriptors4;
    distances = sqrt(sum(difference.^2,2));
    min_distance = min(distances);
    distances1 = distances;
    distances1(distances1 == min_distance) = 10;
    min2_distance = min(distances1);
    if min_distance <= 0.6*min2_distance
        min_position = [min_position; [i, find(distances == min_distance,1)]];
    end
end
matchpoints1 = locs1(min_position(1:10,1),1:2);
matchpoints2 = locs4(min_position(1:10,2),1:2);
```

figure; showMatchedFeatures(img1, img4,matchpoints1, matchpoints2, 'montage', 'parent', axes);

Appendix D: Verification for observations in Task 2

The observation of output under change of position weighting constant is as shown in Figure 4.1:

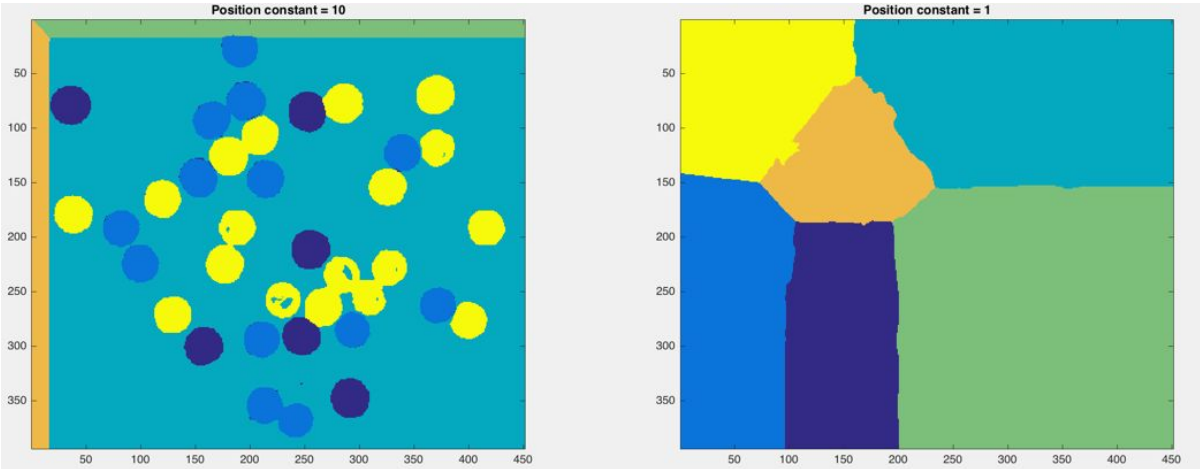

Figure 4.1

The observation of output under change of *nc* is as shown in Figure 4.2:

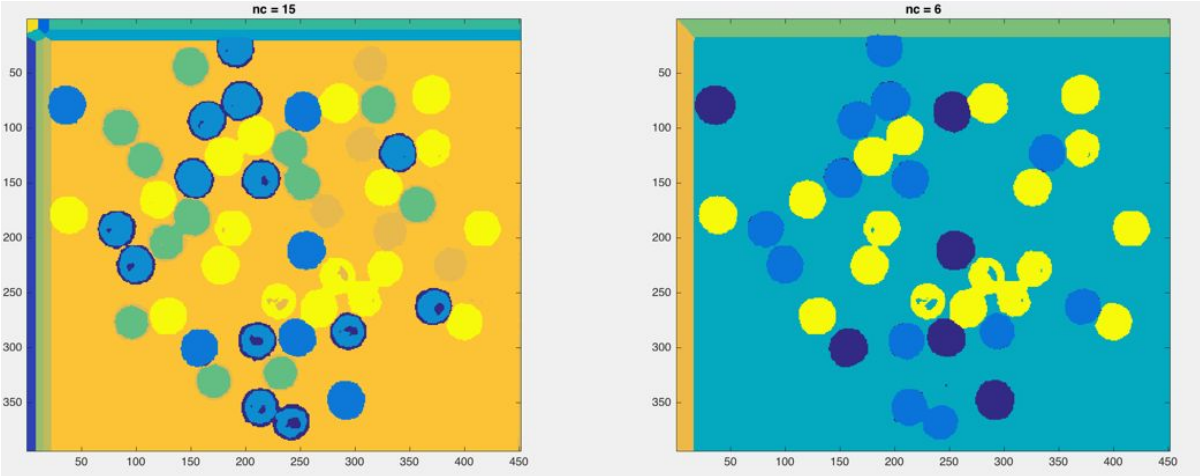

Figure 4.2

The observation of output in different runnings is as shown in Figure 4.3:

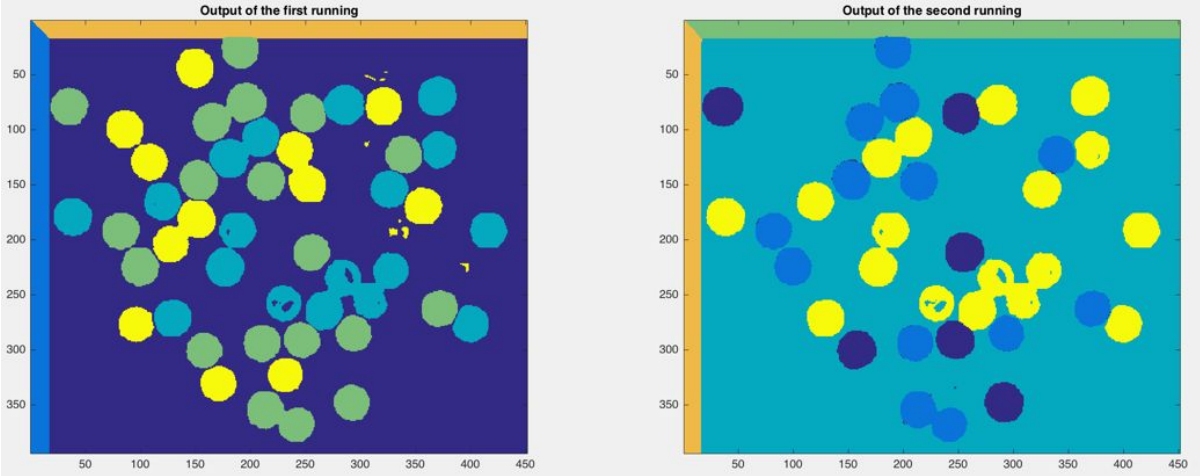
Output of the first running
Output of the second running
50
100
150
200
250
300
350
400
450

Figure 4.3