ENGN4528/6528 Computer Vision – 2015
Computer-Lab 4 (C-Lab4)

Sai Ma - u5224340

May 2015

1 Task-1: Implement K-means Clustering Function

In the first task, it asked us to implement K-means clustering algorithm and providing some existing codes. The implemented code would be used to test two images named *Peppers.png* and *mandm.png*. The more important thing is that assignment asked us to make sure each image is 24 bits RGB images. Then, we could use Matlab method *uint8* to make sure each layer of given image is 8 bits, and total image is 24 bits.

```
Img = imread(ImgName);
Img = uint8(Img);
```

1.1 Extract Vectors

In the method, we extract feature vectors with 5 dimensions include a pixel L, a, b color space values its coordinates values. Therefore, the first sub-task is to extract each pixel feature vector. Then, we could transform RGB image to LAB color space, and add coordinate of each pixel to its vectors. The following is how to implement this process:

```
cform   =  makecform('srgb2lab');
lab = applycform(Img, cform); % get lab value space
```

```

% build 5-dimensional feature vectors
features = im2feature(lab, factor, normFlag);
```

```
function [ features ] = im2feature( img, factor, normFlag )
% This method used to extract features from given image, ...
    becuase this used
% to image segement, then each pixel should have a feature

    [rows, cols, ¬] = size(img);
    npixels = rows * cols; % number of pixels

    % Each feature vector consists of [ L,a,b,x,y]
    [x,y] = meshgrid(1:cols, 1:rows); % get x, y values

    features = img;

    % add two dimensions for x, y coordinates in vectors
    features(:, :, 4) =  factor * x;
    features(:, :, 5) =  factor * y;

    features = reshape(features, [npixels 5]);
    features = double(features);

    if normFlag > 0

        disp('Normalize Features');
        pause(0.03);

        % Normalize the features, which benefits to get ...
            two vector distance
        for i = 1 : size(features,2)

            MaxValue = max(features(:, i));
            MinValue = min(features(:, i));

            gap = MaxValue - MinValue;

            features(:, i) = (features(:, i) - MinValue) / ...
                (gap + eps);

        end

    end
```

```

end
```

Notes that, the normalize process is a signal important pre-process for clustering, because it can ignore the range difference between different dimensions of a vector, and make calculated distance is accurate. For example, three vectors are: a = $[1, 2, 3, 500]$, b = $[1, 2, 3, 100]$, c = $[-1, -2, -3, 400]$. Once we did not perform normalized, the first three dimension will make tiny contribute to calculate distance between two vectors, and it will have influence on clustering results. Therefore, the normalized method aims to change these dimensions to same value ranges and make these dimensions weighted same in distance.

1.2 Implement K-means Function

The next job is to implement K-means clustering based on these extracted feature vectors. In the given codes (I changed some argument name to make its name definition clearer), it provides $nFeatures$ to define feature vector number, $ndims$ to represent to vector dimension (in the La*b*, it is 5), $random_{l}abels$ means the start condition on assign these features to class randomly, $cluster_{s}tats$ defines clustering result (its size is 6, first index number is the features number belong to this class, and next 5 numbers are *mean* values of assigned feature vectors), $data_{c}lusters$ saves relationship between feature number and its assigned class number. The most important argument is *distances* (I change it size to make it can used in my K-means version), it saved all distances information between each feature to all class *mean*. I will explain how to use *distances* in assignment feature to class in code comments.

In general, the K-means approach includes select start clustering result randomly, calculate *mean* of each class, calculate distances between each vector to each class, and assign this vector to the class with smallest distance to it. The existing code has first and second steps, and also include convergence condition. The other part are include in the following code:

```
function [data_clusters, cluster_stats] = my_kmeans( ...
    features, k )

% This function performs k-means clustering on data ,
% given (k) = the number of clusters.
```

```
%  Random Initialization
nFeatures = size(features, 1);
ndims = size(features, 2); % in the L a*b*, it is 5

% in this method, we random assign each feature to a cluster
random_labels = floor(rand(nFeatures, 1) * k) + 1;
% display(random_labels);
data_clusters = random_labels;
cluster_stats = zeros(k, ndims + 1); % centre of clusters
distances = zeros(nFeatures, k);

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 : k

        % Find all data points assigned to this cluster
        [ind] = find(data_clusters == c);

        num_assigned = size(ind, 1);

        % some heuristic codes for exception handling.
        if( num_assigned < 1 )
            disp('No points were assigned to this cluster, ...
                some special processing is given below');
            % Calculate the maximum distances from each ...
                cluster
            max_distances = max(distances);
            [maxx, cluster_num] = max(max_distances);
            [maxx, data_point] = max(distances(:, ...
                cluster_num));
            data_clusters(data_point) = cluster_num;
            ind = 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;

        % update centres of clusters
        if( num_assigned > 1 )

            summ = sum(features(ind, :));
            cluster_stats(c,2:ndims + 1) = summ / ...
                num_assigned;

        else
            cluster_stats(c,2:ndims + 1) = features(ind, :);
        end
    end

    % Exit criteria
    diff = sum(abs(cluster_stats(:) - last_clusters(:)));

    if( diff < 0.00001 )
        break;
    end

    % - Set each cluster center to the average of the ...
        points assigned to it.
    % - Assign each point to the nearest cluster center

    % first, get distances
    distances = getDistance(cluster_stats, features, ndims);

    % then, get smallest distance cluster number, and ...
        update the
    % membership assignment, i.e., update the ...
        data_clusters with current values.
    for featureCount = 1 :  nFeatures

        [¬, cluster_num] = min(distances(featureCount,:));
        % cluster_num is class number which has smallest ...
            distance to this
        % feature
        data_clusters(featureCount) = cluster_num;

    end
    % Display clusters for the purpose of debugging.
    cluster_stats
```

```
    %pause;
end
```

In the distance calculated approach, I selected squared Euclidean distance.

```
function [ distances ] = getDistance( cluster_stats, ...
    features, ndims )
% This method used to calucate distance between each ...
    feature to each
% clusters, then save these reuslts in a distances matrix, ...
    which size is
% Number of Feature * Number of Cluster.

% In order to save cacluated process, we use squared ...
    Euclidean distance.

    k = size(cluster_stats, 1);
    nFeatures = size(features, 1);
    distances = zeros(nFeatures, k);

    for featureCount = 1 : nFeatures
        for culsterCount = 1 : k
            sumDistance = 0;
            for dimCount = 1 : ndims
                distance = (features(featureCount, ...
                    dimCount) - ...
                    cluster_stats(culsterCount, dimCount + ...
                    1))^2;
                sumDistance = sumDistance + distance;
            end
            distances(featureCount, culsterCount) = ...
                sqrt(sumDistance);
        end
    end
end
```

After get these clustering result, I apply given method *displayclusters* to display clustering result.

1.3 Implement Clustering Results

In the K-means algorithm, I set k value to 10, which means output more colorful. As assignment ask, I changed $factor$ from 1, 10 and 20 display different results 1 3 5 2 4 6.

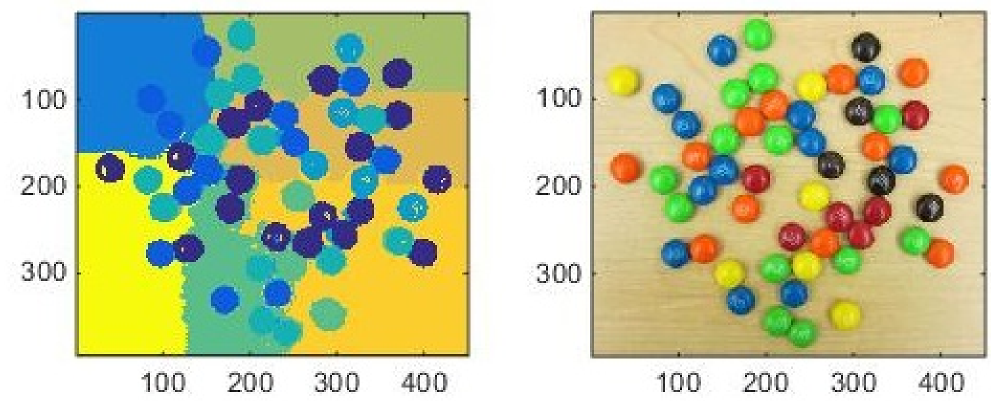

Figure 1: M & M with Factor is 1

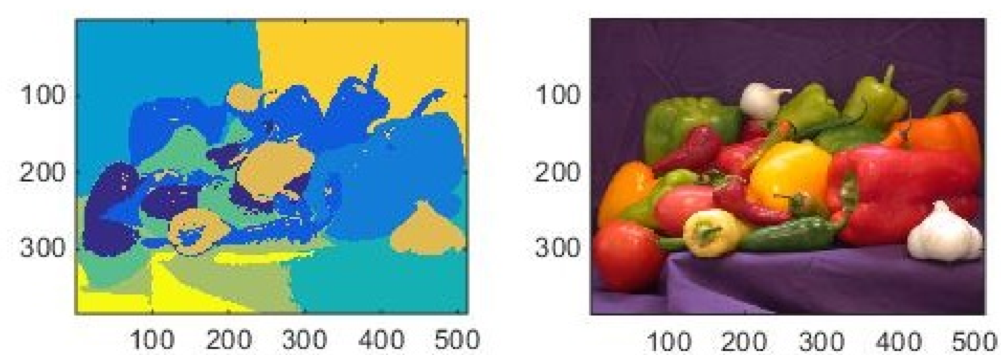

Figure 2: Peppers with Factor is 1

The different values of argument $factor$ cause three different displayed results. The less factor cause clustering result background (empty part) have

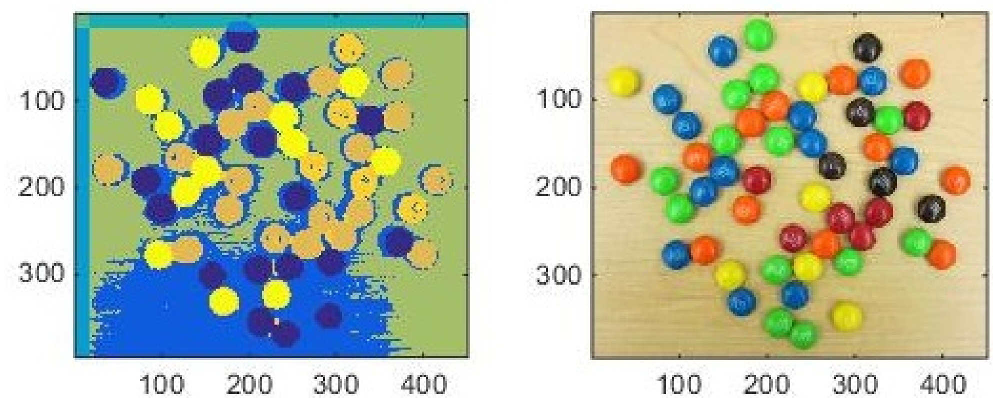

Figure 3: M & M with Factor is 10

many clearly segmentation. The reason is that in our implement, the value were saved in *uint8* data format, which value range from 0 to 255. The factor used to times coordinates of each feature vector, and make the values are bigger than before. Once we have a bigger $factor$, more values in last two dimensions will be set as maximum. For instance, when we use 10 to multiply original x value 30, it will become 255 in uint8 data format. Then, factor make many vector has same x, y coordinate values, and reduce the weighted of coordinated in clustering. As a result, in these displayed results above, the color difference effect on segmentation result when we have a bigger factor value.

1.4 Initial Assignment of the K Centers

In the *K-means* clustering algorithm, it aims to minimize the variance in data given clusters. In the other words, each feature will be calculated the distance between it and cluster centers, and then assign this feature it its nearest (smallest distance) cluster centers. Therefore, we should have the initial K centers, and apply them to compute distances with each features. For example, when we random features (they define the initial cluster centers) did not select well (too close), the finial clustering result will be bad. The following two figures 7 and 8 can display how the initial points (initial centers) can effect on cluster result.

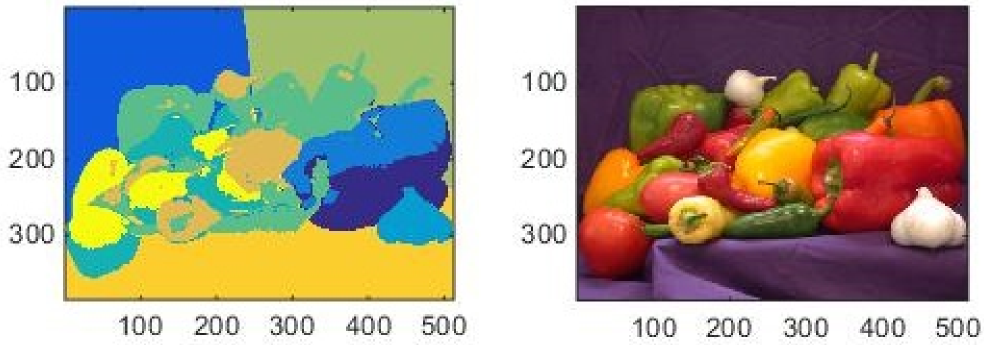

Figure 4: Peppers with Factor is 10

In the K-mean method write by my self, I use the codes to initial:

```
% in this method, we random assign each feature to a cluster
random_labels = floor(rand(nFeatures, 1) * k) + 1;
data_clusters = random_labels;
```

In the normal K-means approach, we usually select k features randomly, and assign them to k kinds of clusters. Then, at initial situation, each cluster will have one feature. In this K-means, we perform a different approach. For each features, we select one of k cluster randomly, and assign this feature to the random cluster. After assigned all features, we compute the cluster centers. This approach avoid the problem we displayed above because the initial cluster centers will located *far* from others. Then, the clustering result will be better.

1.5 Key Steps of K-means Clustering Algorithm

In the K-means algorithm, it includes the key steps on:

- 1) Choose k, initialize cluster centers
- 2) Assign each feature to its closest center
- 3) Update the cluster center as the mean points

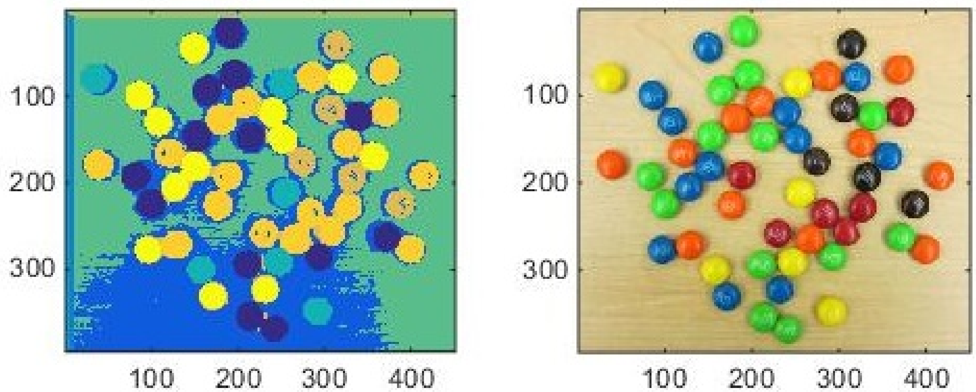

Figure 5: M & M with Factor is 20

- 4) Repeat 2 and 3 step until clustering result is converged

If we write it in pseudocode (based on my initial approach discussed above) algorithm 1 and algorithm 2.

1.6 Key Steps of Mean-shift Clustering Algorithm

In the mean-shift clustering, it aims to segment clustering by the color distribution space. Its main task is to find the *peaks* in the density distribution. Its main tasks are:

- 1) Convert image into feature space (based on color density)
- 2) Set a fix size search windows distributed over feature space
- 3) Compute MEANs of data within each windows
- 4) Sift windows based on computed MEANs above
- 5) Repeat step 4 until all windows not moved

In the processes, $MEAN$ is the difference between weighted mean of neighbors xi around x and the current value of x.

Algorithm 1 K-means

input: k, *features)*
output: $\{cluster_1, ...cluster_k\}$

function K-MEANS($k, features$)
    for $feature$ in $features$ **do**
        $i = random(1, k)$
        $cluster_i \leftarrow feature$
    end for
    for $cluster_i$ in from $cluster_1$ to $cluster_k$ **do**
        $clusterCenter_i = getClusterCenter(cluster_i)$
    end for
    while No Re-assign Features **do**
        for $feature$ in $features$ **do**
            $i = argmin(distance(feature, cluster_1), ...distance(feature, cluster_k))$
            $cluster_i \leftarrow feature$
        end for
        for $cluster_i$ in $cluster_1$ to $cluster_k$ **do**
            $clusterCenter_i = getClusterCenter(cluster_i)$
        end for
    end while
    return $\{cluster_i, ...clusterk\}$
end function

Algorithm 2 getClusterCenter

input: $cluster_i$
output: $clusterCenter_i$

function SIFT($cluster_i$)
    $cluster_i$ has assigned $features$
    n is number of $features$
    m is the dimension of each $feature$
    for c in $[1, m]$ **do**
        $clusterCenter_c = \sum_{1,n}(feature_c) \times \frac{1}{n}$
    end for
    return $clusterCenter$
end function

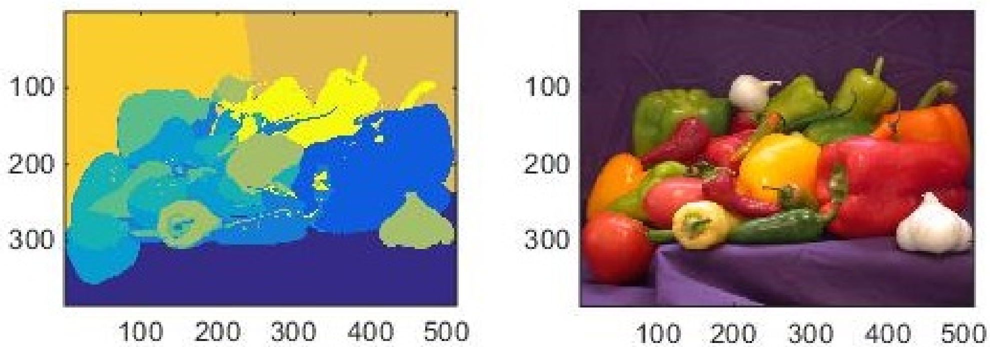

Figure 6: Peppers with Factor is 20

2 Task-2: Implement DLT Based Homograph Estimation

The next task is to build a method to explore *DLT* for homography estimation. As we known, in order to implement *Direct Linear Transforms* approach, we need 4 or more pairs of points to estimate the 3×3 homography matrix. In the assignment, it asked us to input 6 pairs of points to estimate the homography matrix. As a result, the first job is to get 6 pairs of points. I used Matlab *ginput(12)* (12 means this method will get 12 points locations). Before run *ginput*, the images $left$ and $right$ should be displayed on screen. Then, the approach to get 6 pairs of points locations can write in Matlab as following:

```
imgLeft = imread('Left.jpg');
imgRight = imread('Right.jpg');

figure('name', 'Left Image and Right Image');
subplot(1,2,1);
imshow(imgLeft), title('Left Image');
subplot(1,2,2);
imshow(imgRight), title('Right Image');

hold on

```

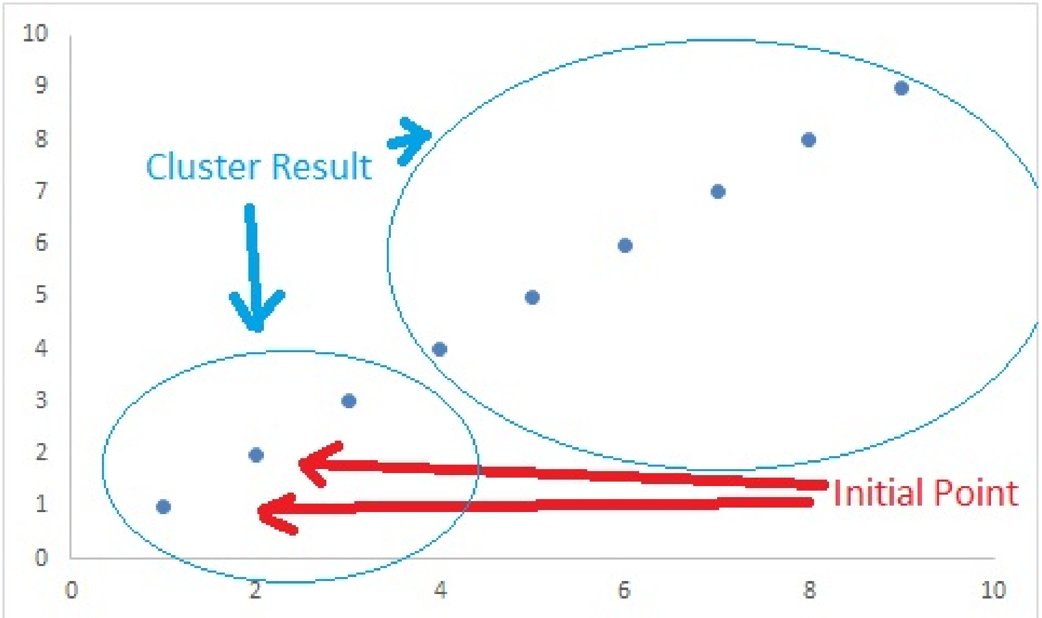

Figure 7: Initial Centers Exampel 1

```
% select 6 pairs points from left and right images, left ...
    iamge first
[x, y] = ginput(12);
```

After these codes, the 6 pairs of points locations will saved in the argument $[x, y]$. In the assignment, the DLT method has 4 input arguments, and they are $u2Trans$, $v2Trans$, $uBase$, and $vBase$. According to its name definitions, we can easily know that they are the 6 pairs of points y-scale and x-scale values lists. In the process to get these points, we input $left$ image point firstly. Therefore, we assume that points in the $left$ are $u2Trans$ and $v2Trans$. Then, the next code are how to divided $[y, x]$ to these four input arguments of DLT.

```
% get 6 left points from selected points
u2Trans = y(1:2:end,:)'; % odd matrix
v2Trans = x(1:2:end,:)'; % odd matrix

% get 6 right points from selected points
uBase = y(2:2:end,:)'; % even matrix
vBase = x(2:2:end,:)'; % even matrix
```

Figure 8: Initial Centers Exampel 2

After that, we prepare these input arguments well. I continue to transform these input DLT method. According to definition, $(uBase, vBase, 1)' = H * (u2Trans, v2Trans, 1)'$ in this method, and we aims to get H. From knowledge on lecture, the H is eigenvector of A^TA with smallest eigenvalue. Therefore, in the direct linear transform approach, its main task is to construct A. According to its definition, the A matrix is following 2

$$\begin{bmatrix} x_1 & y_1 & 1 & 0 & 0 & 0 & -x'_1x_1 & -x'_1y_1 & -x'_1 \\ 0 & 0 & 0 & x_1 & y_1 & 1 & -y'_1x_1 & -y'_1y_1 & -y'_1 \\ ... \\ x_n & y_n & 1 & 0 & 0 & 0 & -x'_nx_n & -x'_ny_n & -x'_n \\ 0 & 0 & 0 & x_n & y_n & 1 & -y'_nx_n & -y'_ny_n & -y'_n \end{bmatrix}$$

Consider we have 6 pairs of points, these n is equal to 6. Then, we should construct this matrix A as its definition. Based on the definition, $u2Trans$ is x, $v2Trans$ is y, $uBase$ is x' and $vBase$ is y'. Then the following is how to construct matrix A:

```
% get the vector size, consider of these four arguments ...
    have same size,
    % the sizes all are 6
    [pointNums, ¬] = size(u2Trans);

    % then, consturct matrix A
    A = zeros(2*pointNums, 9);

    for count = 1 : pointNums

        xPrime = uBase(:, count);
        yPrime = vBase(:, count);

        x = u2Trans(:, count);
        y = v2Trans(:, count);

        A(2*count - 1,1:9) = [x, y, ones(1, 1), zeros(1, ...
            3), -xPrime*x, -xPrime*y, -xPrime];
        A(2*count, 1:9) = [zeros(1, 3), x, y, ones(1, 1), ...
            -yPrime*x, -yPrime*y, -yPrime];

    end
```

After that, we perform Matlab method *svd* to get its eigenvector with smallest eigenvalue.

```
[¬, ¬, V] = svd(A);
H = V(:, end); % get the smallest eigenvalue mapped ...
    eigenvector
H = reshape(H, 3, 3)'; % change it shape to make used to ...
    test solution
leftDown = H(3, 3)';
H = H/leftDown; % make the left down equals to 1
```

2.1 Normalize Points

However, it we use this H to test whether it is correct H, it will not pass. The reason is that we input 6 pairs of coordinates, and we should normalize the $ubase, vbase, uTransandvTrans$ values to minimize sum of squared residuals and get the accurate H. The following is method on normalize.

```
function [ Points, Transform ] = getNormalize( xValues, ...
    yValues )
% get normalize transform for given coorderinates

    [¬, pointsSize] = size(xValues);

    Points = ones(3, pointsSize);
    Points(1, :) = xValues;
    Points(2, :) = yValues;

    % get the mean of x and y
    xMean = mean(xValues); % get centers
    yMean = mean(yValues);
    xDistance = xValues - xMean;
    yDistance = yValues - yMean;

    % get the scale
    xScale = mean(abs(xDistance));
    yScale = mean(abs(yDistance));

    % construct transform matrix and normalization
    Transform = [1/xScale, 0, -xMean/xScale; 0, 1/yScale, ...
        -yMean/yScale; 0, 0, 1];

end
```

Then, we use these normalized transform to multiply our coordinate values, and get stable H.

```
% perform normalize approach
[pointBase, transformBase] = getNormalize(uBase, vBase);
[pointTrans, transformTrans] = getNormalize(u2Trans, v2Trans);

% normalize these points
normBase = transformBase*pointBase;
uBase = normBase(1,:);
vBase = normBase(2,:);

normTrans = transformTrans*pointTrans;
u2Trans = normTrans(1,:);
v2Trans = normTrans(2,:);

% then, get the 3*3 homography matrix between these two images
H = DLT(u2Trans, v2Trans, uBase, vBase);
```

```
H = (transformBase\H)*transformTrans;
```

2.2 Test Homograph

In order to prove our H is correct, I write some code to transform image based on our calculated H.

```
tform = projective2d(H');
imageNorm = imwarp(imgRight, tform);

figure('name', 'Transform Right Image by H');
subplot(1,3,1), imshow(imageTransform), title('Transform ...
    Without Normalized');
subplot(1,3,2), imshow(imageNorm), title('Transform With ...
    Normalized');
subplot(1,3,3), imshow(imgLeft), title('Transformed Left ...
    Image');
```

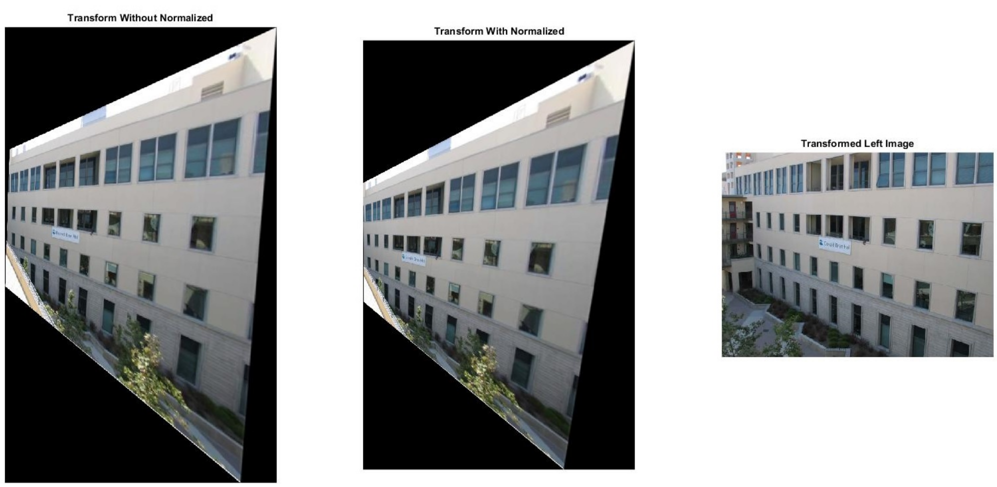

Figure 9: Transformed Result

After get this figure 9 above to test H, I make sure I get the correct H from my *DLT* method.

2.3 Linear Equations in Solving Homograph

In the process of getting homograph, our first task is to get a H that fulfill the equation is:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} h_{00} & h_{01} & h_{02} \\ h_{10} & h_{11} & h_{12} \\ h_{20} & h_{21} & h_{22} \end{bmatrix} \times \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

After expose this equation, we get following equations 1

$$\begin{aligned} x_i' \times (h_{20} \times x_i + h_{21} \times y_i + h_{22}) = h_{00} \times x_i + h_{01} \times y_i + h_{02} \\ y_i' \times (h_{20} \times x_i + h_{21} \times y_i + h_{22}) = h_{10} \times x_i + h_{11} \times y_i + h_{12} \end{aligned} \tag{1}$$

Then, the problem become to solve the matrix A eigenvector with its smallest eigenvalue:

$$\begin{bmatrix} x_1 & y_1 & 1 & 0 & 0 & 0 & -x_1'x_1 & -x_1'y_1 & -x_1' \\ 0 & 0 & 0 & x_1 & y_1 & 1 & -y_1'x_1 & -y_1'y_1 & -y_1' \\ ... \\ x_i & y_i & 1 & 0 & 0 & 0 & -x_i'x_i & -x_i'y_i & -x_i' \\ 0 & 0 & 0 & x_i & y_i & 1 & -y_i'x_i & -y_i'y_i & -y_i' \end{bmatrix}$$

Then, we perform singular value decomposition to get our target H.

2.4 Minimally Points to Compute Homograph

In the task, we use 6 pairs of correspondences to compute homograph. In fact, different 4 pairs of correspondences is enough to compute the homograph. As we analysis before, *projective* degree of freedom is 8, which means there are 8 unknown arguments in the transform matrix (because the value at left down of 3×3 matrix is 1). For each pair of coordinate, $x_i' = \frac{h_{00}x_i + h_{01}y_i + h_{02}}{h_{20}x_i + h_{21}y_i + h_{22}}$, $y_i' = \frac{h_{01}x_i + h_{11}y_i + h_{12}}{h_{20}x_i + h_{21}y_i + h_{22}}$. In order to solve the 8 unknown arguments from h_{00} to h_{21} (h_{22} is 1). We need 4 pairs of $[x, y]$, $[x', y']$ (8 given arguments) to construct polynomial and solve 8 unknown arguments.

3 Appendix: Matlab Code

```
% use for task 1: implement own k-means clustering function
```

```
clc;
clear;

% read png image, if not in 3 * 8 bits RGB, transform it
mandmImgName = 'mandm.png';
% mandmImgName = 'ANUbuilding1.jpg';
factor = 1.0; % use to set weight of x and y values
kValue = 10;
normFlag = 1;

runK_means(mandmImgName, factor, kValue, normFlag);

mandmImgName = 'peppers.png';
runK_means(mandmImgName, factor, kValue, normFlag);

factor = 10.0;
runK_means(mandmImgName, factor, kValue, normFlag);

mandmImgName = 'mandm.png';
runK_means(mandmImgName, factor, kValue, normFlag);

factor = 20.0;
runK_means(mandmImgName, factor, kValue, normFlag);

mandmImgName = 'peppers.png';
runK_means(mandmImgName, factor, kValue, normFlag);
```

```
function [  ] = runK_means( ImgName, factor, kValue, ...
    normFlag )
% This method used to run K means algorithm with given ...
    arguments

    Img = imread(ImgName);
%     figure('name', 'Input Colour Image');
%     imshow(Img);
%     Img = uint8(Img);
    display(size(Img));

    % convert it to La*b* colour space
    cform   =  makecform('srgb2lab');
    lab = applycform(Img, cform); % get lab value space

    % build 5-dimensional feature vectors
    features = im2feature(lab, factor, normFlag);
```

```

    % perform k-means algorithm
    [k_clusters, ¬] = my_kmeans(features, kValue);

    % display results
    displayclusters(Img, k_clusters, factor, ImgName);

    fprintf('Finish Run %s image with factor is %d\n', ...
        ImgName, factor);
    pause(0.03);


end
```

```
function displayclusters( img, clusters, factor, imageName )

    % Just take size information and then make
    % a new display image.
    [rows, cols, ¬] = size(img);

    % Reshape cluster information into image
    cluster_img = reshape( clusters, [rows cols] );

%     % Find boundaries
%     boundary_img_x = filter2( [-1 1], cluster_img, ...
    'same' );
%     boundary_img_y = filter2( [-1 1]', cluster_img, ...
    'same' );
%     boundary_img = (abs(boundary_img_x) + ...
    abs(boundary_img_y)) > 0;

    str = sprintf('Image %s with factor is: %d', ...
        imageName, factor);
    figure('name', str);
    subplot(1,2,1);
    imagesc(cluster_img); axis image;
    subplot(1,2,2);
    imagesc(img); axis image;

    drawnow;
    figure(gcf);

%     str = sprintf('%s boundary image', imageName);
%     figure('name', str);
```

```
%     imagesc(boundary_img); axis image; colormap(gray);

%pause;
```

```
% use for task 2: DLT based  homography estimation
clc;
clear;

imgLeft = imread('Left.jpg');
imgRight = imread('Right.jpg');

figure('name', 'Left Image and Right Image');
subplot(1,2,1);
imshow(imgLeft), title('Left Image');
subplot(1,2,2);
imshow(imgRight), title('Right Image');

hold on

% select 6 pairs points from left and right images, left ...
    iamge first
[x, y] = ginput(12);

% get 6 left points from selected points
uBase = x(1:2:end,:)'; % odd matrix
vBase = y(1:2:end,:)'; % odd matrix

% get 6 right points from selected points
u2Trans = x(2:2:end,:)'; % even matrix
v2Trans = y(2:2:end,:)'; % even matrix

% get H
H = DLT(u2Trans, v2Trans, uBase, vBase);

tform = projective2d(H');
imageTransform = imwarp(imgRight, tform);

% perform normalize approach
[pointBase, transformBase] = getNormalize(uBase, vBase);
[pointTrans, transformTrans] = getNormalize(u2Trans, v2Trans);

% normalize these points
normBase = transformBase*pointBase;
uBase = normBase(1,:);
```

```
vBase = normBase(2,:);

normTrans = transformTrans*pointTrans;
u2Trans = normTrans(1,:);
v2Trans = normTrans(2,:);

% then, get the 3*3 homography matrix between these two images
H = DLT(u2Trans, v2Trans, uBase, vBase);
% % get real matrix by normalized matrix
H = (transformBase\H)*transformTrans;

tform = projective2d(H');
imageNorm = imwarp(imgRight, tform);

figure('name', 'Transform Right Image by H');
subplot(1,3,1), imshow(imageTransform), title('Transform ...
    Without Normalized');
subplot(1,3,2), imshow(imageNorm), title('Transform With ...
    Normalized');
subplot(1,3,3), imshow(imgLeft), title('Transformed Left ...
    Image');
```

```
function [ H ] = DLT( u2Trans, v2Trans, uBase, vBase )

% Computes the homography H applying the Direct Linear ...
    Transformation
% The transformation is such that
% p = H p' , i.e.,:
% (uBase, vBase, 1)'=H*(u2Trans , v2Trans, 1)'
%
% INPUTS:
% u2Trans, v2Trans - vectors with coordinates u and v of ...
    the transformed image point (p')
% uBase, vBase - vectors with coordinates u and v of the ...
    original base image point p
%
% OUTPUT
% H - a 3x3 Homography matrix
%
% Sai Ma, 04/23/2015

    % check two vectors size, if they are not match, ...
        return false
    [¬, SizeU2Trans] = size(u2Trans);
```

```
    [¬, SizeV2Trans] = size(v2Trans);
    [¬, SizeUBase] = size(uBase);
    [¬, SizeVBase] = size(vBase);

    if SizeU2Trans ≠ SizeV2Trans || SizeUBase ≠ SizeVBase
        error('x and y coordinater vectors must have same ...
            size');
    elseif SizeU2Trans ≠ SizeUBase
        error('points from two images must have same number');
    end

    pointNums = SizeU2Trans; % get number of points

    % then, consturct matrix A
    A = ones(2*pointNums, 9);

    for count = 1 : pointNums

        xPrime = uBase(:, count);
        yPrime = vBase(:, count);

        x = u2Trans(:, count);
        y = v2Trans(:, count);

        A(2*count - 1,1:9) = [x, y, ones(1, 1), zeros(1, ...
            3), -xPrime*x, -xPrime*y, -xPrime];
        A(2*count, 1:9) = [zeros(1, 3), x, y, ones(1, 1), ...
            -yPrime*x, -yPrime*y, -yPrime];

    end

    % The right singular vectors of A are the eigenvectors ...
        of A'*A
    [¬, ¬, V] = svd(A);
    H = V(:, end);
    H = reshape(H, 3, 3)'; % change it shape to make used ...
        to test solution

    leftDown = H(3, 3)';
    H = H/leftDown; % make the left down equals to 1

end
```