Machine Learning

Homework 6

Report

Name: Sergiy Shelekh
ID: 0760822
Email: proxitrone@gmail.com

Introduction

The goal of this homework is to apply Kernel K-means and Spectral clustering techniques to perform image segmentation on given 2 images. As a similarity measure, we are given a kernel, which considers both spatial and color characteristics of our image data. Both Kernel K-means and Spatial clustering use K-means clustering, which is very sensitive to initialization, we need to show and compare different initialization strategies. For every clustering technique we use, it is required to show the clustering process in the form of GIF images, we should also see, that points within the same cluster have similar coordinates in the eigenspace of graph Laplacian, for spectral clustering. In this assignment I use MATLAB programming language, as it allows fast and efficient computations of things like Gram matrix and has good debugging capabilities.

Kernel K-means: Implementaion

In kernel K-means, the most important thing is to compute the Gram matrix, a matrix of pairwise distances between our datapoints.

```
%% Compute the Gram matrix first, we'll use it's elements in the
% distance computations. Regular for-loop implementation is too
% slow, use vectorized version for improved preformance
[Gram, Coord, Color] = compute_Gram(image_mat, hyper_params(1), hyper_params(2));
figure(1);
imshow(Gram);
title('Gram matrix');
```

We start off with extracting spatial and color information from a given image

```
function [Gram, Coordinates, Color] = compute_Gram(image_mat, gamma_s, gamma_c)
%COMPUTE_GRAM Compute pairwise distances between our datapoints
%   We use 2 RBF kernels multiplied together as the distance measure
    datapoints_num = size(image_mat, 1) * size(image_mat, 2);
    datapoints = 1:datapoints_num;
    Coordinates = spatial(datapoints, image_mat);

    Color = zeros(3, datapoints_num);
    for n=1:size(image_mat, 1)
        for m=1:size(image_mat, 2)
            Color(:, (n-1)*size(image_mat, 2)+m) = double(image_mat(m, n,:));
        end
    end
```

What we do here is creating a matrix, where rows contain spatial or color information (x, y coordinates, or R G B values) and columns represent each datapoints (10 000 in our case of 100x100 image). Next, for each RBF kernel, we take advantage of vectorization and expanding the 2-norm, in order to make computations much faster.

```
% Vectorized form
% spactial_rbf
A = dot(Coordinates, Coordinates, 1);
B = -2* (Coordinates'* Coordinates);
K = A+B+A';
spatial_rbf = exp(-gamma_s*K);

% color_rbf
A = dot(Color, Color, 1);
B = -2* (Color'* Color);
K = A+B+A';
color_rbf = exp(-gamma_c*K);

Gram_vec = spatial_rbf.*color_rbf;
```

This computation still takes some time, but we only need to compute this matrix once for each image. Having Gram matrix computed, we proceed to initialization of K-means on this data. In this homework, I use two different initialization strategies:

```
%% K-means initialization strategies
if init_type == 1
    % Choose random datapoints as initial cluster centers
    means_old = round(rand(cluster_num, 1)*datapoints_num+1);
    init_type_str = ['RNG', num2str(rngseed)];
```

Random initialization, where we choose each cluster mean uniformly from all datapoints, and k-means++

```
elseif init_type == 2
    % k-means++: Choose first point randomly, others as the weighted distribution
    % based on Gram
    means_old = zeros(cluster_num, 1);
    means_old(1) = round(rand*datapoints_num+1);
    means = [means_old(1)];
    for i=2:cluster_num
            shortest_distance=min(2-Gram(means, :), [], 1);
            [shortest_distance, ind] = sort(shortest_distance, 'descend');
            shortest_distance = shortest_distance/sum(shortest_distance, 2);
            threshold = rand;
            for n=1:datapoints_num
                if (threshold>shortest_distance(n))
                    means = [means, ind(n)];
                    break;
                end
            end
    end
    means_old = means';
```

In k-means++ we only choose the first cluster mean randomly, and then draw other means from a weighted probability distribution, based on the square of the distances from datapoints to their closest mean.

Before we start assigning datapoints to clusters, we still need to do some initialization. First, we record colors of our cluster means, for visualization.

```
%% Kluster colors
cluster_colors = zeros(size(image_mat,3), cluster_num);
for k=1:cluster_num
    cluster_colors(:, k) = color_vec(means_old(k), image_mat)/256;
end
clustered_image = zeros(size(image_mat));
clustered_image_gray = zeros(100, 100);

```

Assign a name to the GIF file we are going to store as our clustering process visualization.

```
%% Klustered GIF filename
file_path = 'Kernel K-means';
file_header = '/KKMeans';
image_num_str = ['Image',num2str(image_num)];
kluster_num_str = ['Klusters', num2str(cluster_num)];

filename = [file_path, file_header, image_num_str, init_type_str, kluster_num_str
```

Now we have everything we need to start K-means clustering alrorithm

```
%% Start K-means
for i=1:K_max
    disp(['--KKmeans iteration ', num2str(i), '--']);
    % Matrix to store assignment of datapoints to clusters
    % (1 if in cluster, 0 if not)
    clusters = zeros(cluster_num, datapoints_num);
    % E-step, assign points to clusters
    % Compute distances from datapoints to cluster means
    % Basically extract the approapriate rows of our Gram matrix
    cluster_distances = Gram(means_old, :);
    % Assignm minmum distance points to appropriate clusters, use
    % linear indexing
    [~, index] = max(cluster_distances, [], 1, 'linear');
    clusters(index) = 1;
    N_k = sum(clusters==1, 2);
```

We start with E-step, where we assign datapoints to closest clusters. This is very easy, since we already have our Gram matrix precomputed and it has exactly what we need, those distances. So we, basically, extract those distances to clusters from our Gram matrix and choose the smallest distance among them, for each datapoint (We take the maximum of Gram matrix entries, since our RBF kernels give us larger values, the closer points are to each other). The largest kernel distance between a datapoint and cluster mean gets that datapoint assign to the appropriate cluster. E-step is done, all datapoints are assigned to clusters, we now do the visualization part.

```
    % Visualize klusters
    for n=1:datapoints_num
        [~, k] = max(clusters(:,n), [], 1);
        [x, y] =ind2sub([100, 100], n);
        % Each datapoint get's it's cluster mean color
        clustered_image(x, y, :) = cluster_colors(:,k);
        clustered_image_gray(x, y) = sum(cluster_colors(:,k))/3;
    end
```

We first assign an appropriate color to each datapoint in a 100x100 image, then show the RGB and grayscale images during the process, and finally write GIF images for visualization.

```
    % Show cluster assignment at runtime
    figure(4);
    imshow(clustered_image);
    figure(3);
    imshow(clustered_image_gray);
    % Write current image frame to a GIF file
    [imind,cm] = rgb2ind(clustered_image, 255);
    if i ==1
        imwrite(imind,cm, filename, 'DelayTime', 1, 'Loopcount', inf);
    else
        imwrite(imind,cm, filename, 'DelayTime',0.5, 'WriteMode', 'Append');
    end
```

Now we need to proceed to the next part of K-means, which is the M-step, where we reassign cluster means.

```
% M-step, find new means
for k=1:cluster_num
    % For each datapoint in a cluster, find a point, that minimizes
    % the overall distance to every other point in a cluster, that
    % will be our new mean
    data_distances = zeros(datapoints_num, 1);
    for n=1:datapoints_num
        if clusters(k, n)
            % Compute distances from datapoint n to all
            % other datapoints in cluster k
            data_distances(n, 1) = Gram(n, :)*clusters(k, :)';
        end
    end
    [~, index] = max(data_distances);
    means_new(k, 1) = index;
end
```

Since we work in a high dimensional feature space indirectly via our kernel function, we can't simply average cluster points, so we compute the overall distance for each point in a cluster to all other points in that cluster, and choose the new mean to be a point with the minimum distance (maximum in our case, since RBF kernel is used). This process gives us new cluster means, so we also need to reassign our cluster colors for visualization purposes.

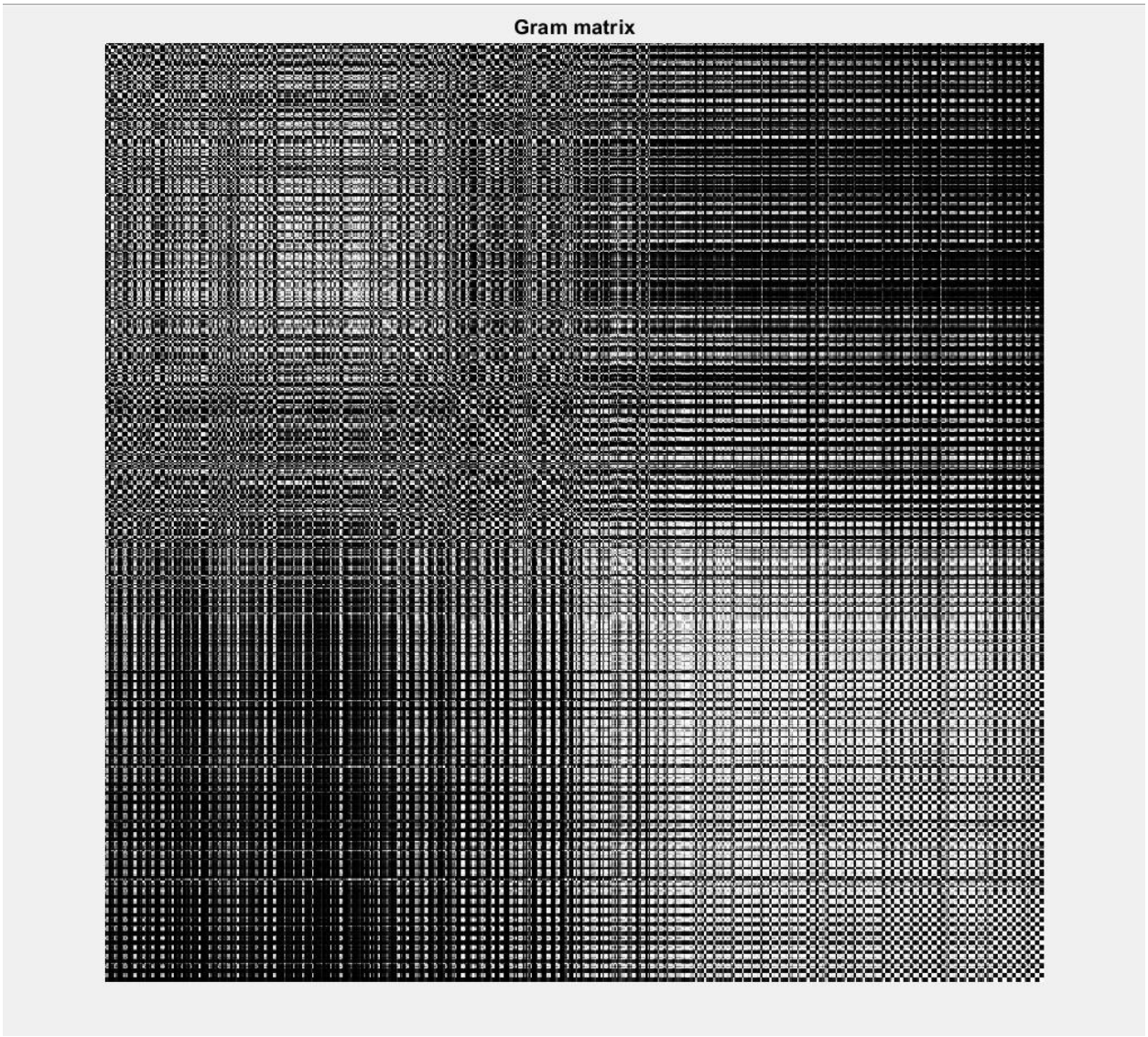
```
for k=1:cluster_num
    cluster_colors(:, k) = color_vec(means_new(k), image_mat)/256;
end
```

Finally, we check whether our termination condition has been reached, which is that the cluster means remain the same from previous iteration of K-means.

```
% Termination condition: Our cluster centers don't change from last
% iteration
if (sum(means_old - means_new ==0) == cluster_num )
    disp(['Number of iterations spend for KKmeans is ', num2str(i)]);
    break;
end
means_old = means_new;
```

Kernel K-means: Results

Experiments are run for different images, numbers of clusters, initialization strategies. Hyperparameters were pre-tuned in a heuristic manner, so that our Gram matrix has a more-or-less block-diagonal structure. Hyperparameters we use are $[\gamma_s = 10^{-6}, \gamma_c = 10^{-4}]$, since we want the color component of our image influence the kernel distance more than spatial component. First, look at gram matrix for image 1:



We can argue, that 3 clusters can be seen in this Gram matrix, it depends on the Kernel we use and it’s hyperparameters

Final clustering images are shown below, the GIF versions can be found in the folder Kernel K-means

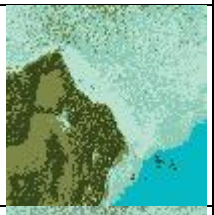
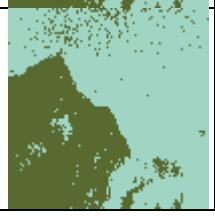
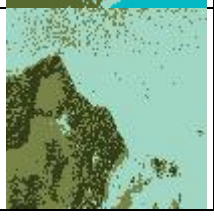
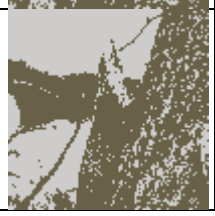
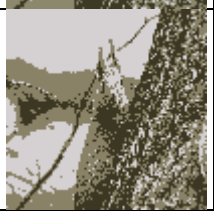
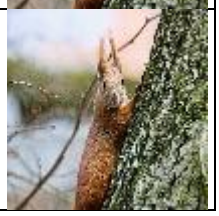
Image 1	2 Cluster	3 Clusters	5 Clusters	10 Clusters	Original
Random init					
k-means++					

Image 2	2 Cluster	3 Clusters	5 Clusters	10 Clusters	Original
Random init					
k-means++					

We see, that random initialization still gives bearable results, main difference can be seen in 3 clusters for image 1. Subjectively, k-means++ version looks better, but random initialization gives more information, since it also captures the blue part in the lower right corner of our image. However, the main advantage of k-means++ is that we need less iterations (on average) to converge.

Spectral Clustering: Implementation

For spectral clustering we are, once again, using the Gram matrix computed before. However, now it acts as a graph representation of our data, we can call it the weighted adjacency matrix. Terminology aside, we use this matrix to first compute the degree of each node (datapoint) and graph Laplacian.

```
%% Degree matrix
D = diag(sum(Gram, 1));
%% Laplacian for ratio cut
L_ratio = D - Gram;
%% Laplacian for normalized cut
D_sqrt= diag(1./sqrt(sum(Gram, 1)));
L_norm = D_sqrt*L_ratio*D_sqrt;
```

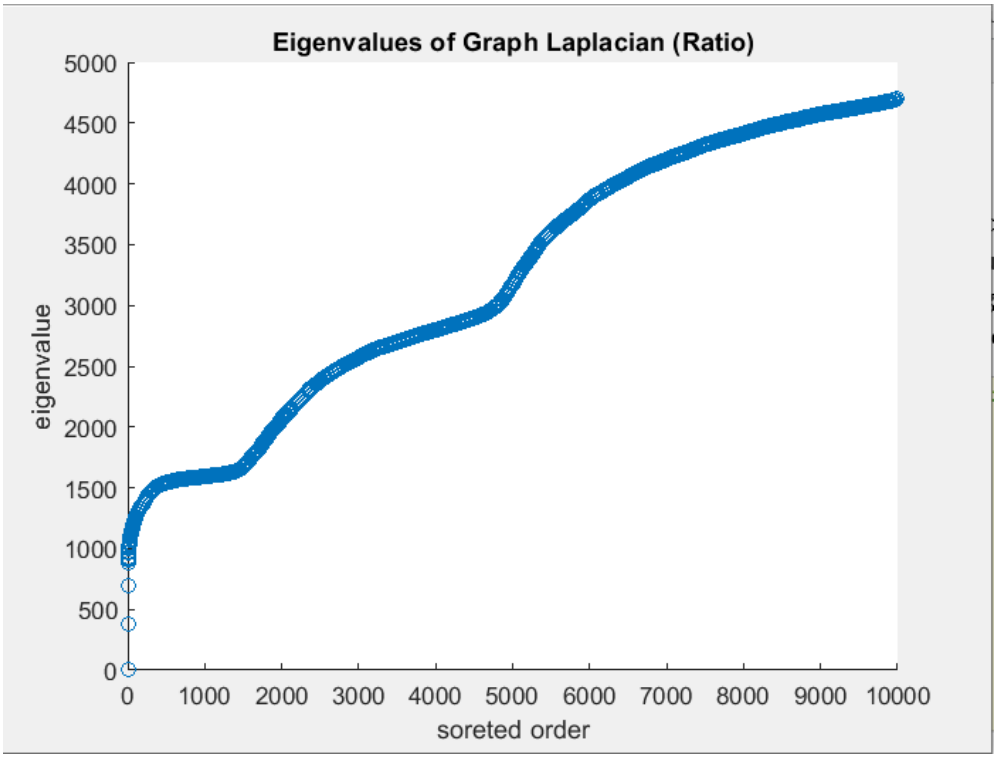
Here comes the first difference between the RatioCut and NormalizaedCut, in the first case Laplacian is simply given as the difference between the degree matrix and weight matrix, but in the second case we also normalize the previous Laplacian with the squared inverse of the degree matrix. Next, we use Laplacian matrices to compute corresponding eigenvalues and eigenvectors.

```
%% Get the eigenvalues and vectors of our graph Laplacian (Ratio)
[eigVec_ratio, eigVal] = eig(L_ratio);
[d, ind] = sort(diag(eigVal));
eigVal = eigVal(:, ind);
eigVec_ratio = eigVec_ratio(:, ind);
figure(2);
scatter(1:numel(d), d);
title('Eigenvalues of Graph Laplacian (Ratio)');
ylabel('eigenvalue');
xlabel('soreted order');
```

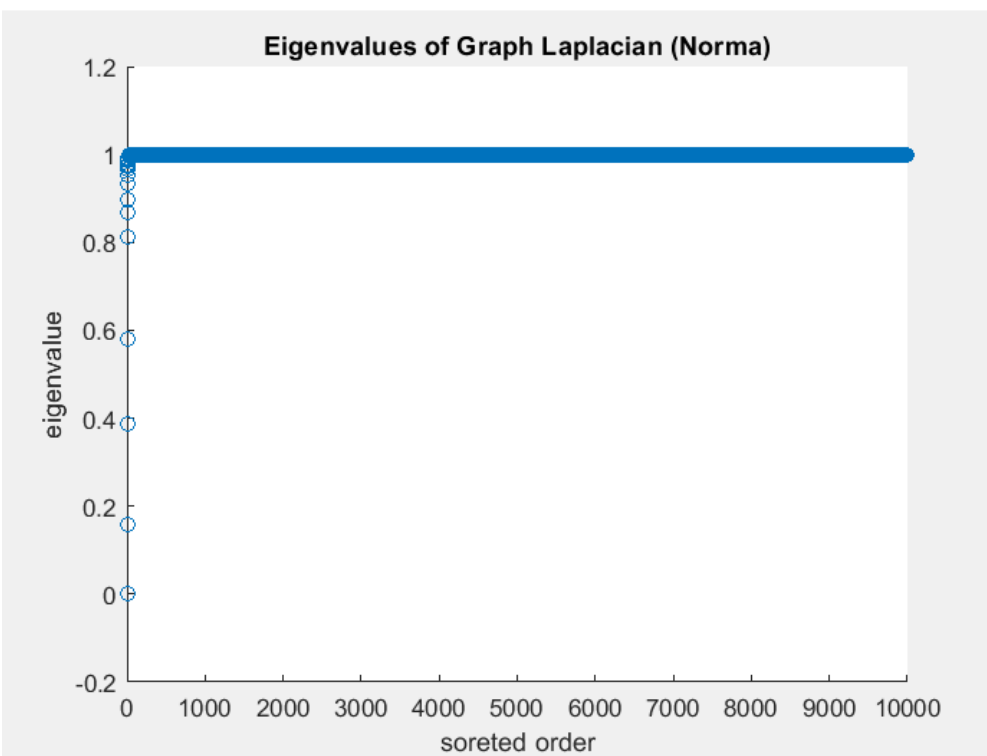
```
%% Get the eigenvalues and vectors of our graph Laplacian (Normal)
[eigVec_norm, eigVal] = eig(L_norm);
[d, ind] = sort(diag(eigVal));
eigVal = real(eigVal(:, ind));
eigVec_norm = real(eigVec_norm(:, ind));
% Normalize the rows with norm 1
eigVec_norm = eigVec_norm./sqrt(sum(eigVec_norm.^2, 2));
figure(5);
scatter(1:numel(d), real(d));
title('Eigenvalues of Graph Laplacian (Norma)');
ylabel('eigenvalue');
xlabel('soreted order');
```

For the normalized version, we also normalize eigenvectors of Normalized Laplacian

Example of sorted eigenvalues of graph laplacian for image 1:



Notice the gap between the first three eigenvalues



After having these eigenvectors, we have everything we need to start clustering our datapoints

```
function [means_new, objective] = spectral(image_num, image_mat, cluster_num, init_ty
%spectral Perform spectral clustering
%   Build a similarity graph (Gram matrix), graph Laplacian, look at it's
%   eigenvalues and eigenvectors
    rng(rngseed);
    %% Initialize
    K_max = 100;
    objective = [];

    % Total number of data_points
    datapoints_num = size(image_mat, 1)*size(image_mat, 2);

    %% Extract first cluster_num eigenvectors
    data_vectors = eigVec(:, 1:cluster_num);
```

Depending on the number of clusters we want to partition our graph into, we work only with the first number of eigenvectors. Then we do similar initialization, assigning cluster colors, building GIF image name as in the Kernel K-means part, and can start the clustering procedure. Start as before:

```
    %% Perform K-means on the first cluster_num eigenvectors

    for i=1:K_max
        disp(['--Spectral Kmeans iteration ', num2str(i), '--']);
        % Matrix to store assignment of datapoints to clusters
        % (1 if in cluster, 0 if not)
        clusters = zeros(cluster_num, datapoints_num);
```

Next, compute distances from each eigenvector datapoint to cluster means. We don't use Gram matrix, but compute direct Euclidean distances.

```
    % E-step, assign points to clusters
    % Compute distances from datapoints to cluster means
    cluster_distances = pdist2(data_vectors, data_vectors(means_old, :), 'euclidean')';

    [~, index] = min(cluster_distances, [], 1, 'linear');
    clusters(index) = 1;
    N_k = sum(clusters==1, 2);
```

Again, we choose the minimum distance points to be assigned to appropriate clusters and proceed to visualization part. Visualization is the same as in Kernel K-means,

```
        % Visualize klusters
        for n=1:datapoints_num
            [~, k] = max(clusters(:, n), [], 1);
            [x, y] =ind2sub([100, 100], n);
            clustered_image(x, y, :) = cluster_colors(:,k);
            clustered_image_gray(x, y) = sum(cluster_colors(:,k))/3;
        end
        % Show cluster assignment at runtime
        figure(4);
        imshow(clustered_image);
        figure(3);
        imshow(clustered_image_gray);
        % Write current image frame to a GIF file
        [imind,cm] = rgb2ind(clustered_image, 255);
        if i ==1
            imwrite(imind,cm, filename, 'DelayTime', 1, 'Loopcount', inf);
        else
            imwrite(imind,cm, filename, 'DelayTime',0.5, 'WriteMode', 'Append');
        end
```

Next comes the M-step part. Idea is still the same, we compute distances from each point in a cluster to all the other points in a cluster, but now, instead of using Gram matrix as the distance measure, we use distances between eigenvector values.

```
    % M-step, find new means
    for k=1:cluster_num
        % For each datapoint in a cluster, find a point, that minimizes
        % the overall distance to every other point in a cluster, that
        % will be our new mean
        data_distances = ones(datapoints_num, 1)*1000;
        for n=1:datapoints_num
            if clusters(k, n)
                % Compute distances from datapoint n to all
                % other datapoints m in cluster k
                data_distances(n, 1) = pdist2(data_vectors,data_vectors(n,:), ...
                    'euclidean')'*clusters(k, :)';
            end
        end
        [~, index] = min(data_distances);
        means_new(k, 1) = index;
    end
```

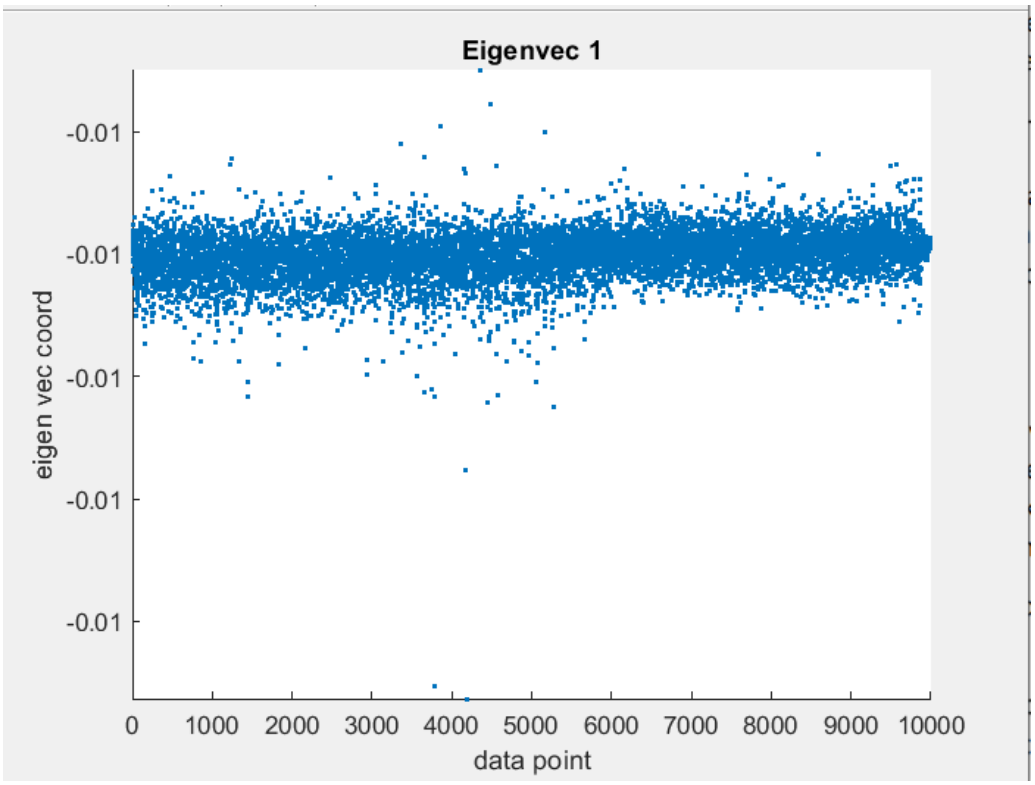
Last part in this K-means, as before, is clusters colors and termination criteria

```
% Reassign cluster colors
for k=1:cluster_num
    cluster_colors(:, k) = color_vec(means_new(k), image_mat)/256;
end
% Termination condition: Our cluster centers don't change from last
% iteration
if (sum(means_old - means_new ==0)==cluster_num)
    disp(['Number of iterations spend for KKmeans is ', num2str(i)]);
    break;
end
means_old = means_new;
```

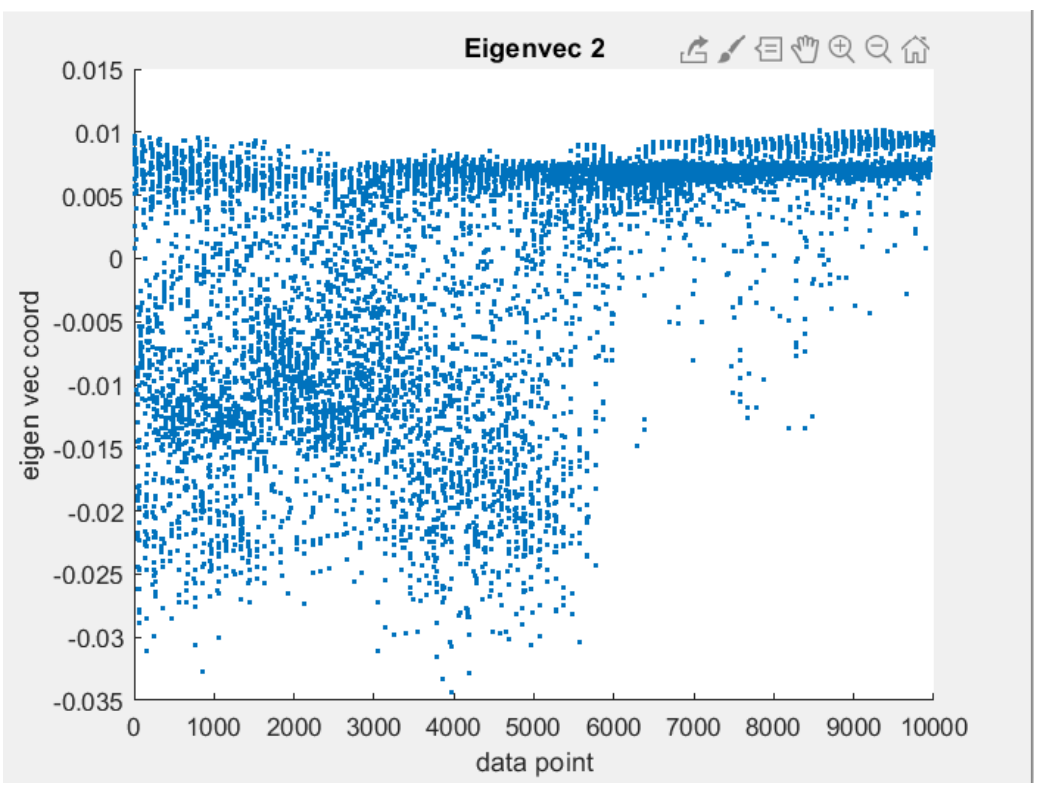
Finally, we plot the eigenvector values for each coordinate (same as number of clusters we choose) and look at whether points in the same cluster are have similar coordinates in this eigenspace.

```
%% Plot the eigenspace coordinates of points in same clusters
% Basically plot the eigenvectors for each cluster
for k=1:cluster_num
    figure(5+k);
    scatter(1:datapoints_num, data_vectors(:, k), '.');
    title(['Eigenvec ', num2str(k)]);
    xlabel('data point');
    ylabel('eigen vec coord');
end
```

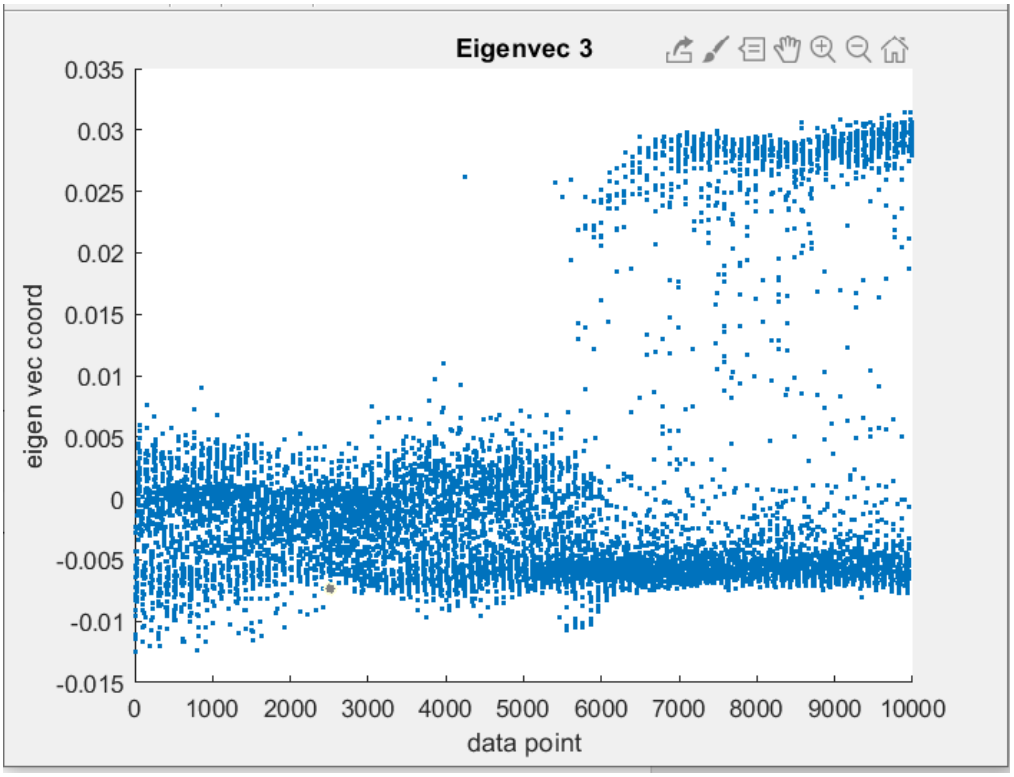
Eigenvectors for RatioCut graph Lapalcian are quite messy:



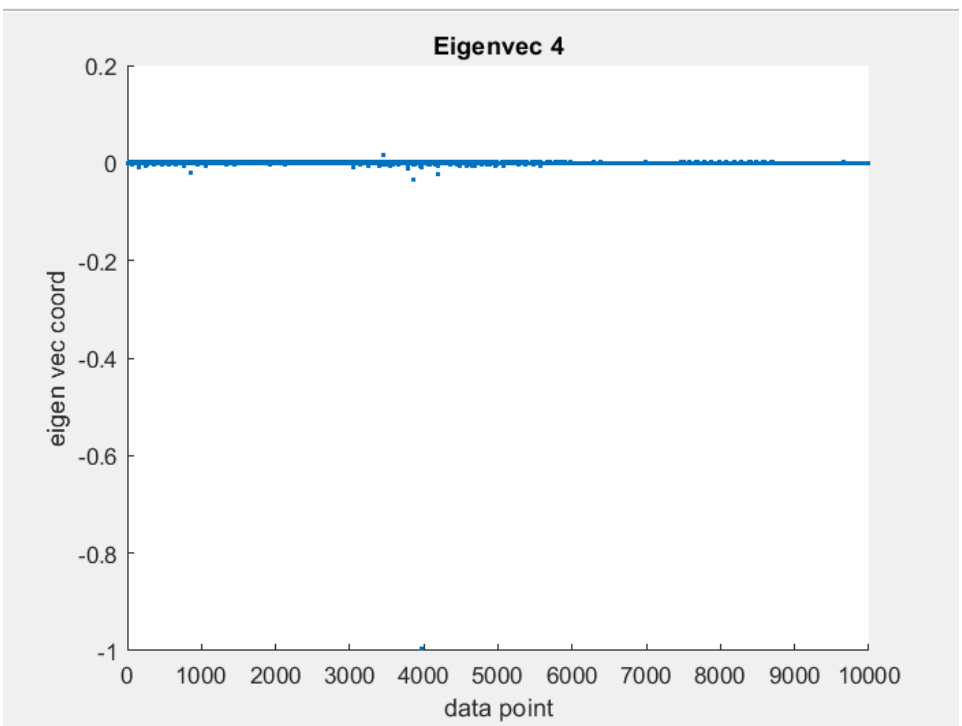
Every datapoint is the same, notice how the y-axis values don't change



We see different values laying below and above zero, those are our different clusters

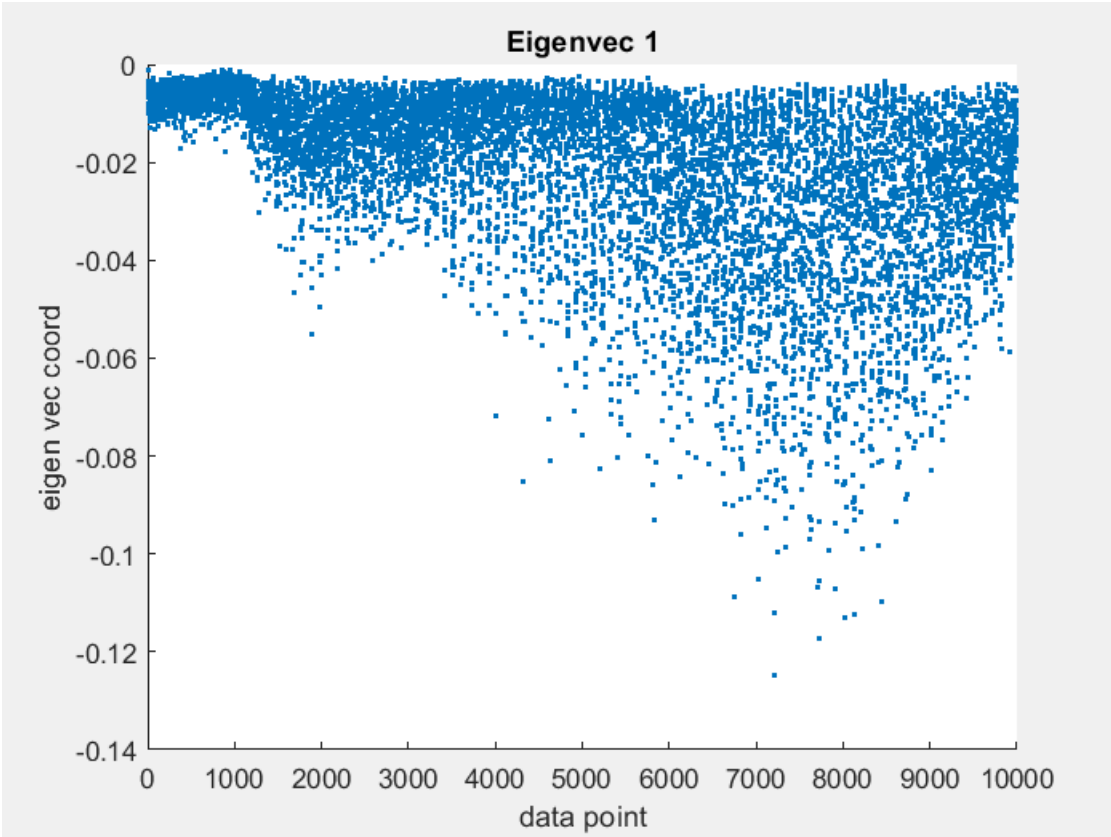


Same thing here, we decide which cluster our points belong to using these eigenvectors, difference in 0 level.

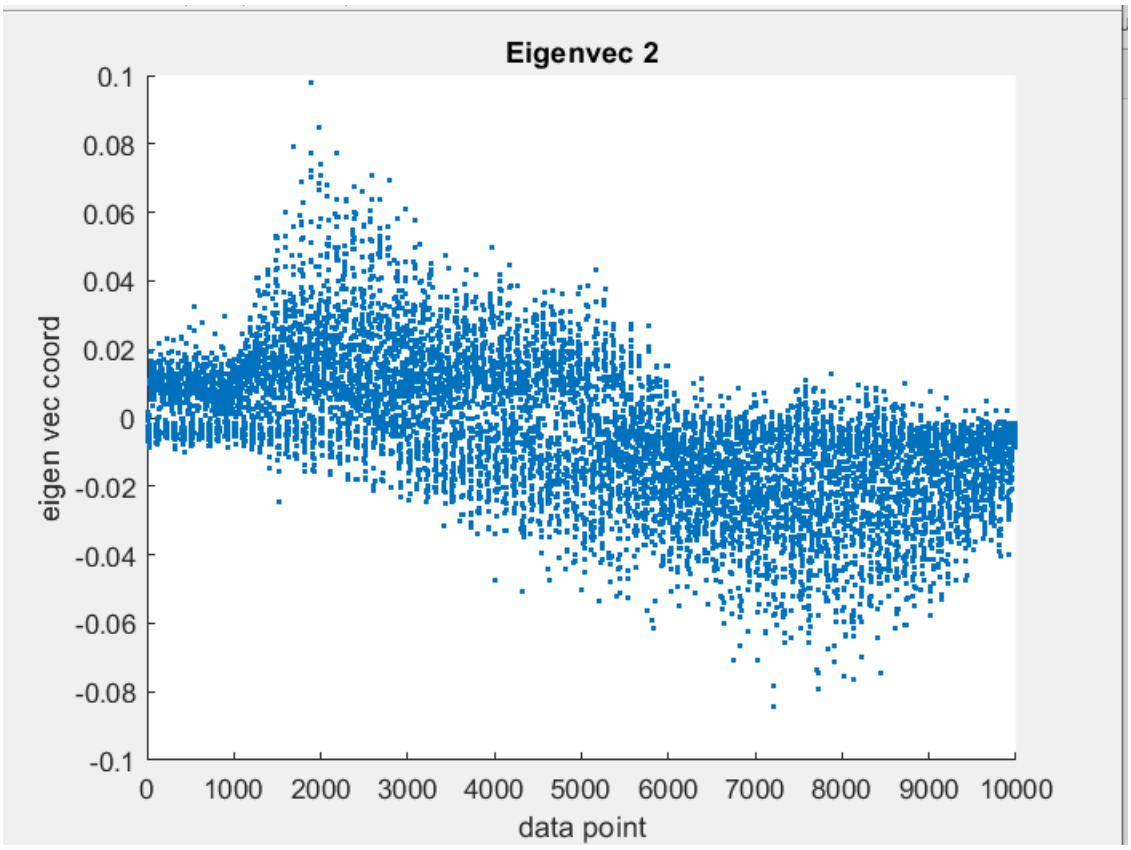


Next vectors essentially don’t influence our clustering procedure

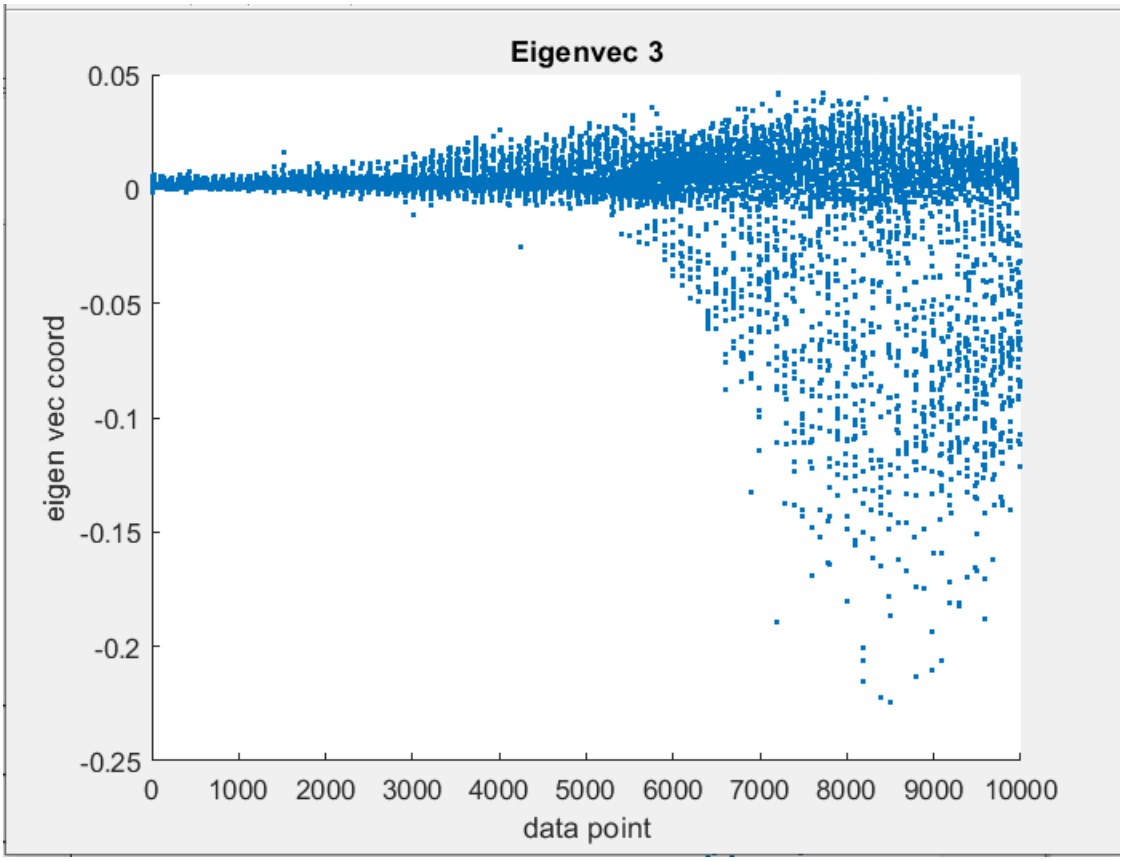
Now look at the normalized cut Laplacian eigenvectors:



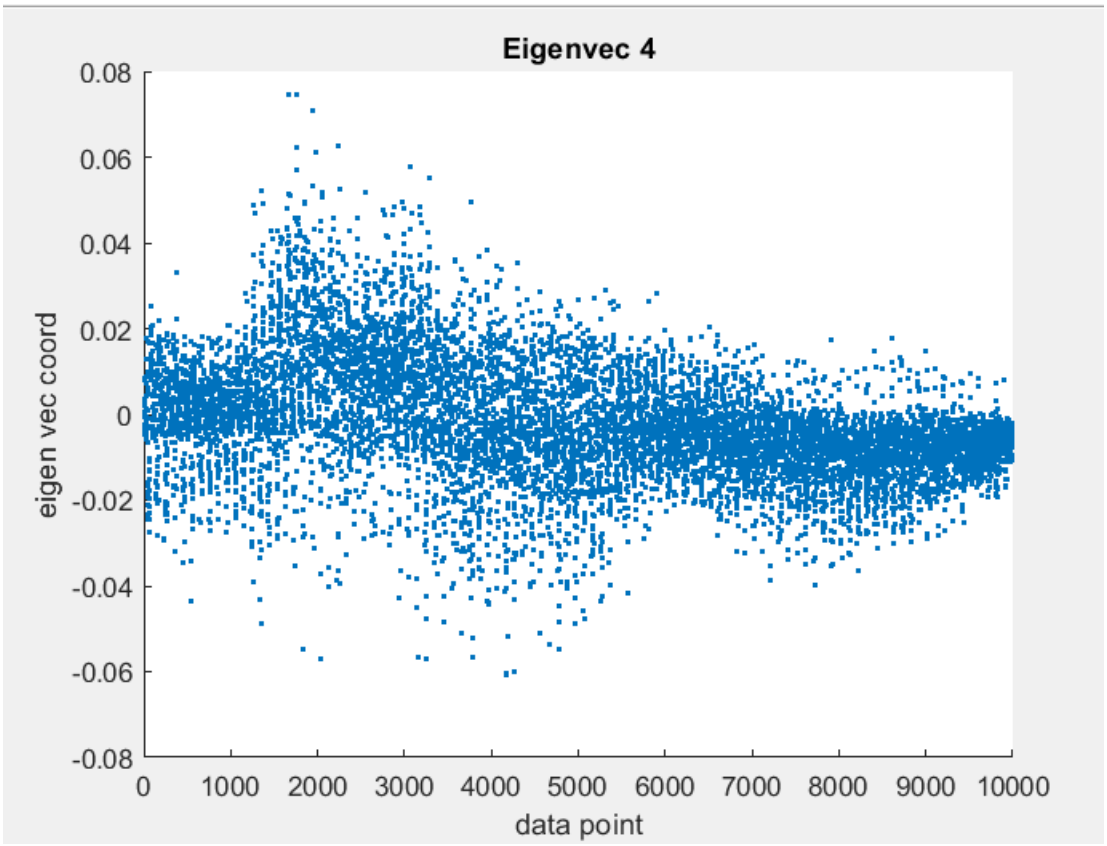
We see more structure here, but still all values are below 0, so the first eigenvector doesn't influence our clustering

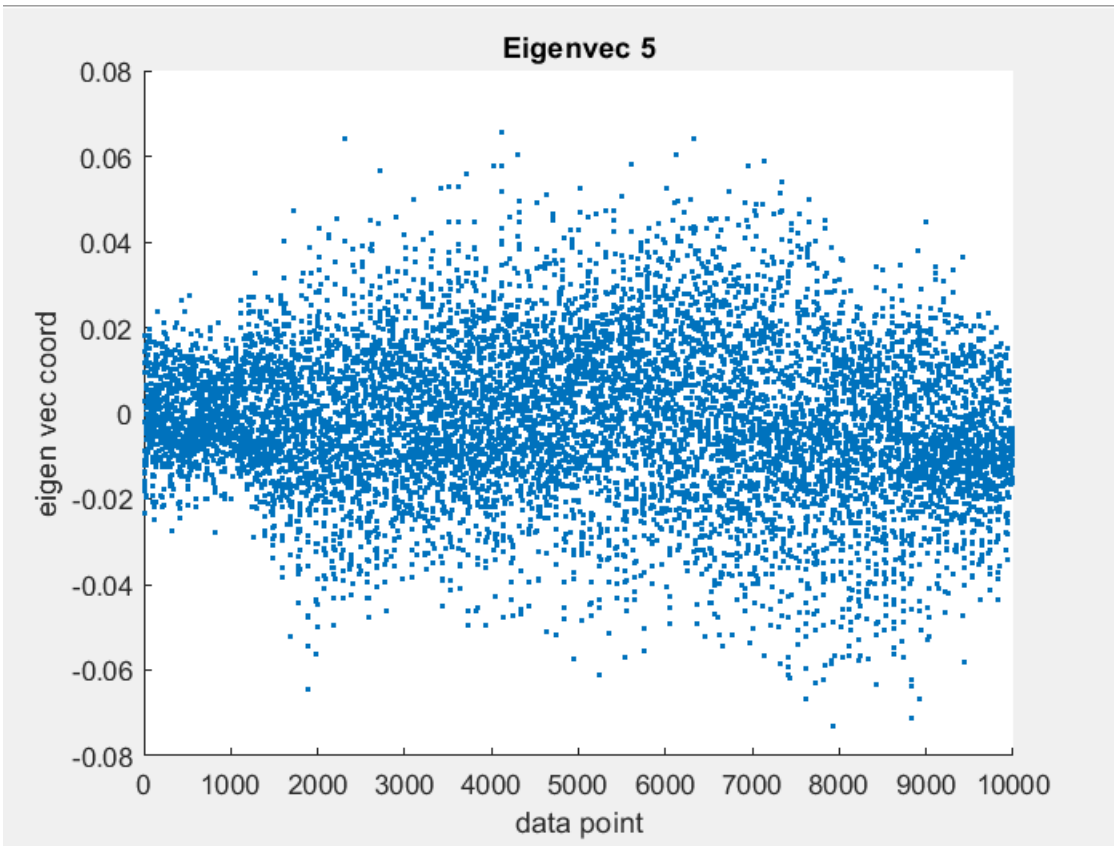


Here we can clearly see the difference in clusters



Here we can see once more the which points belong to which clusters, or rather by which criteria they are divided


Eigenvec 4
0.08
0.06
0.04
0.02
0
-0.02
-0.04
-0.06
-0.08
eigen vec coord
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
data point


Eigenvec 5
0.08
0.06
0.04
0.02
0
-0.02
-0.04
-0.06
-0.08
eigen vec coord
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
data point

Spectral Clustering: Results

For each image, we look at clustering results for different number of clusters and initialization strategies

RatioCut:

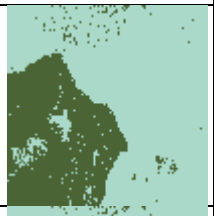
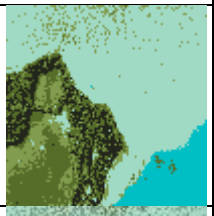
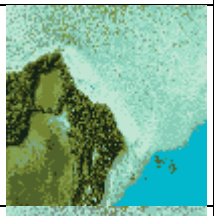
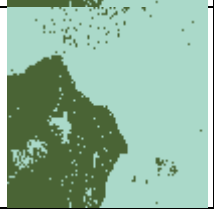
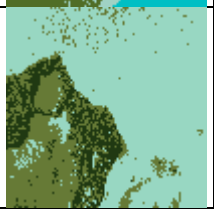
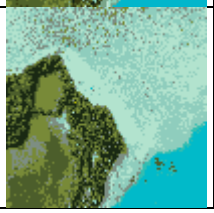
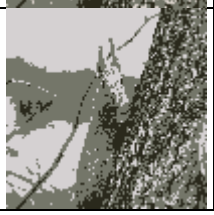
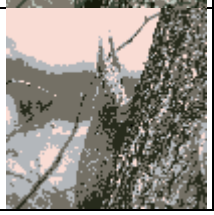
Image 1	2 Cluster	3 Clusters	5 Clusters	10 Clusters	Original
Random init					
k-means++					

Image 2	2 Cluster	3 Clusters	5 Clusters	10 Clusters	Original
Random init					
k-means++					

NormalCut:

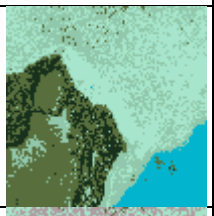
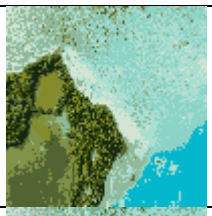
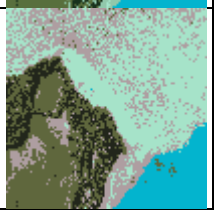
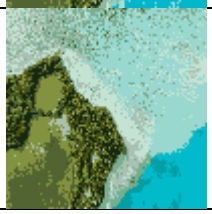
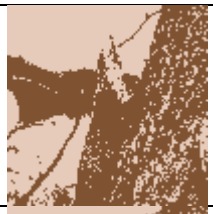
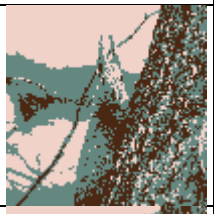
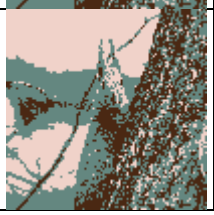
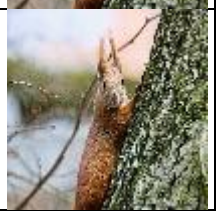
Image 1	2 Cluster	3 Clusters	5 Clusters	10 Clusters	Original
Random init					
k-means++					

Image 2	2 Cluster	3 Clusters	5 Clusters	10 Clusters	Original
Random init					
k-means++					

Overall, both methods preform quite good, however we can still see some images falling into local minimum of K-means.

Conclusion

In this homework, three different clustering approaches were implemented: Kernel K-means and Spectral Clustering (both ratio and normal cut). All these techniques use k-means algorithm at some point. Kernel K-means directly uses kernel similarity measure between datapoints (Gram matrix) to cluster data and recompute means, while in spectral clustering we build a graph representation of our data first (graph Laplacian) and then perform clustering in the eigenspace of that graph representation. Spectral clustering approach allows us to get more insight into how many natural clusters there are in our dataspace (largest eigengap) at the expense of taking extra computational time to find those eigenvalues and eigenvectors. I liked the Kernel K-means more, since it works faster and together with k-means++ initialization strategy can give great results.