HOMEWORK 4

Jiazhi Li

jiazhil@usc.edu

March 18, 2019

1 Texture classification

1.1 Abstract and Motivation

Classification refers to as assigning a physical object or incident into one of a set of predefined categories. In texture classification the goal is to assign an unknown sample image to one of a set of known texture classes [1]. In this problem, the texture images have some repeated and similar structure. It is very important for extraction of texture feature to provide the enough information and distinguish details for texture segmentation.

1.2 Approach and Procedures

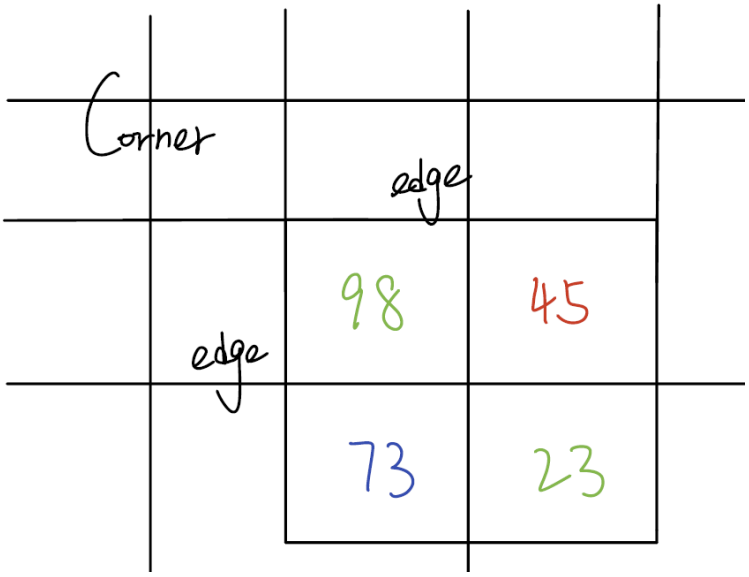
There are some necessary procedures to followed for texture classification.

1.2.1 Pre-processing

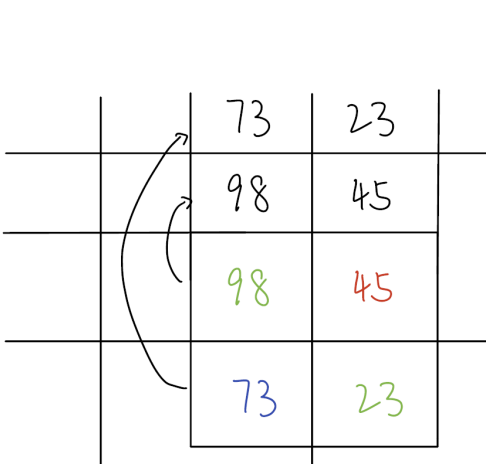
At very beginning, for each texture image, we calculate the average of gray level and subtract image mean to reduce illumination effects. And then, we use matrix multiplicity to generate 25 5*5 Laws filters according 5 1D kernels.

1.2.2 Feature extraction

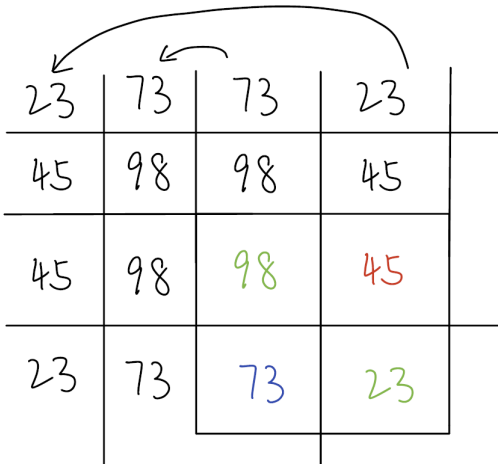
First, I use boundary extension by mirror reflection shown in **Figure** 1.



(a) Origin image



(b) Bourdary extension by row



(c) Bourdary extension by column

Figure 1: Boundary extension

After that, I use 25 2-D Laws Filters to make correlation with texture image pixel by pixel so that for each texture, we have 25 filtered images. The following formula is used to calculate average energy for each filtered images,

$$average = \frac{1}{M * N} \sqrt[2]{\sum_{m,n=1}^{M,N} k_{mn}^2}$$

By this way, for each texture image, I calculate average energy for 25 filtered result so that a 25-D feature vector is obtained.

1.2.3 Feature Averaging and normalization

Before the next step about PCA, I use the following formula to normalize it,

$$newterm = \frac{term - \mu}{\sigma}$$

where μ represents the mean of this 25D vector and σ represents the standard deviation of the 25D vector. By this way, the 25D vector have 0 as mean and 1 as variance.

1.2.4 PCA for dimension reduction

Having got 12 25D feature vectors, we combine them into a 25*12 matrix and then apply PCA to this matrix in order to obtain a 3*12 matrix, which is shown in **Figure** 2. The main procedures of PCA are:

- Calculate the empirical mean and the deviation from the mean
- Find the covariance matrix
- Find the eigenvectors and eigenvalues of the covariance matrix
- Rearrange the eigenvectors and eigenvalues and set the first three eigenvectors as basis vectors
- Project the data set to the subspace of these three eigenvectors

```
projection in subspace
[-0.2426486408876566, 0.04120779941750741, 0.05998476373419785;
 -0.1602579936080147, 0.07880183125636445, -0.0822523887954773;
 0.5158273777870547, 0.03389723663690294, -0.08241562894070406;
 0.009053015124806812, -0.1075268699446111, -0.06792684281601996;
 -0.2258207677076098, -0.01079999291358591, -0.001158124248809796;
 0.003825070828102838, -0.1140364635678763, -0.0403438152939329;
 -0.2268509825355209, -0.008156214361016342, -0.005580452109070945;
 -0.1384123004879579, 0.07238170207805462, -0.0582738447032015;
 0.2245220054506157, 0.09852825793484196, 0.05154853495745104;
 0.1429280853830075, 0.09129139573979307, 0.08635985384695326;
 0.1870904805423009, -0.1395372189402012, 0.08556572889622485;
 -0.089255349889128 55, -0.0360514633361738, 0.05449221553950795]
```

Figure 2: Feature vectors after PCA

And then, I plot the reduced 3-D feature vector in the feature space, which is shown in **Figure** 3.

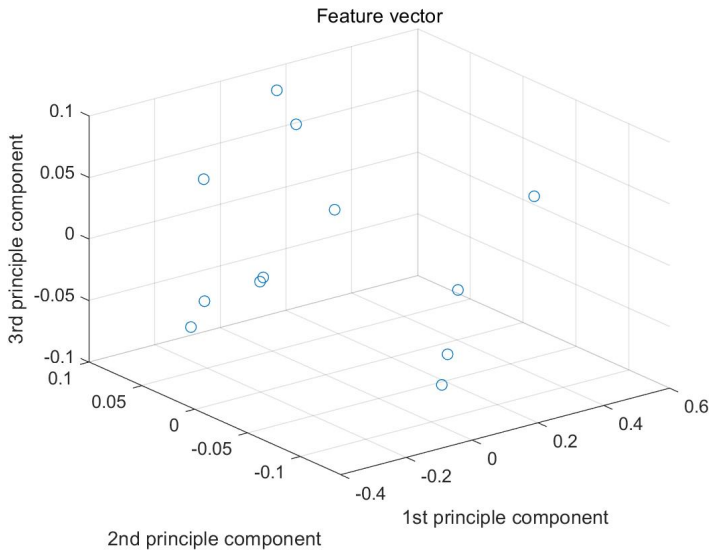


Figure 3: Reduced 3-D feature vector in the feature space

1.2.5 K-means algorithm for image clustering

Last step is to apply the K-means cluster algorithm for image clustering. The more details about K-means will be introduced in discussion section.

1.3 Experimental Results

The cluster result are shown in **Figure** 4 and **Figure** 5. The result contain two parts. They are centroid selected and texture number. The centroids is a 3D vector with using of PCA. If not, it will be a 25D vector. The result is not very robust, which I will discuss in next section.

```
Cluster 0 :
        Centroid:
                [ 0.309147 -0.00237057 0.0182329 ]
        Samples:
                [ 0.515827 0.0338972 -0.0824156 ]3
                [ 0.224522 0.0985283 0.0515485 ]9
                [ 0.18709 -0.139537 0.0855657 ]11
Cluster 1 :
        Centroid:
                [ -0.231773 0.0074172 0.0177487 ]
        Samples:
                [ -0.242649 0.0412078 0.0599848 ]1
                [ -0.225821 -0.0108 -0.00115812 ]5
                [ -0.226851 -0.00815621 -0.00558045 ]7
Cluster 2 :
        Centroid:
                [ -0.149335 0.0755918 -0.0702631 ]
        Samples:
                [ -0.160258 0.0788018 -0.0822524 ]2
                [ -0.138412 0.0723817 -0.0582738 ]8
Cluster 3 :
        Centroid:
                [ 0.0166377 -0.0415809 0.00814535 ]
        Samples:
                [ 0.00905302 -0.107527 -0.0679268 ]4
                [ 0.00382507 -0.114036 -0.0403438 ]6
                [ 0.142928 0.0912914 0.0863599 ]10
                [ -0.0892553 -0.0360515 0.0544922 ]12
```

Figure 4: Cluster result1

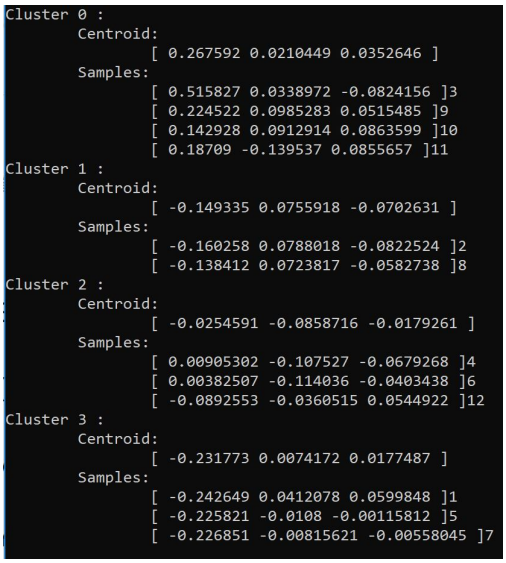


Figure 5: Cluster result2

1.4 Discussion

1.4.1 Strongest discriminant power feature

In order to find the strongest discriminant power feature vector and the weakest one, we need to consider each feature vector separately. This time, we try to use one single feature vector to make cluster. The result using whole bunch of features is shown in **Figure** 6.

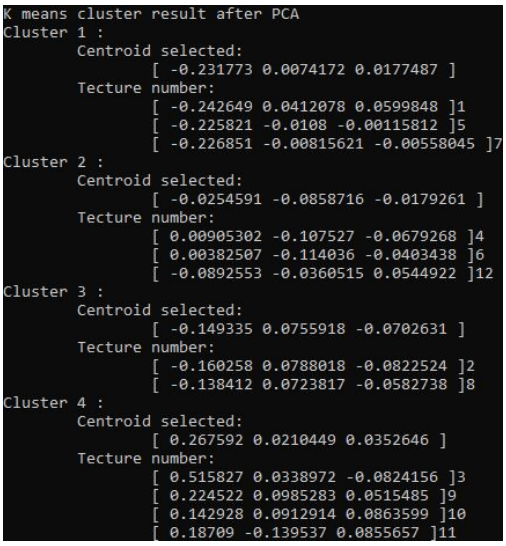
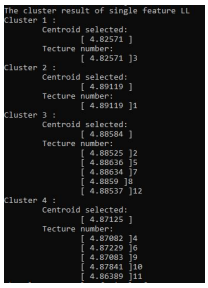
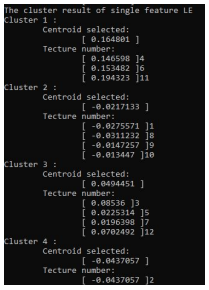


Figure 6: Cluster result using 25 feature

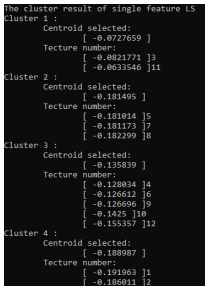
The goal is to find the similar cluster result to the above mentioned one, which will be the strongest discriminant feature vector. I just present all cluster result by single feature vector in **Figure** 7.



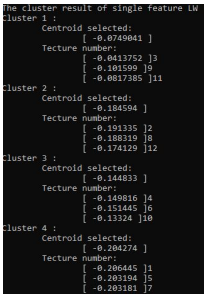
(a) LL



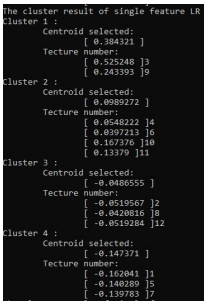
(b) LE



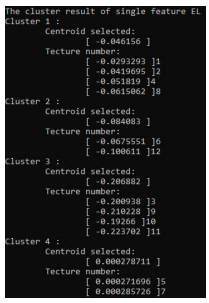
(c) LS



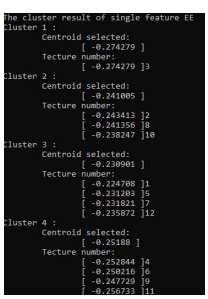
(d) LW



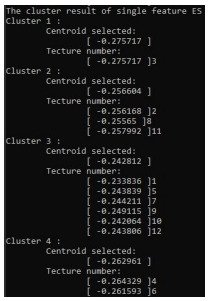
(e) LR



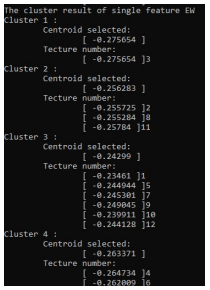
(f) EL



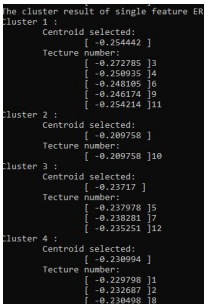
(g) EE



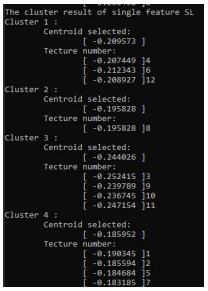
(h) ES



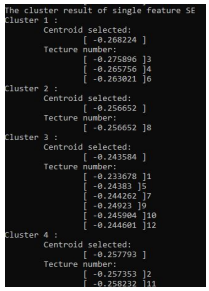
(i) EW



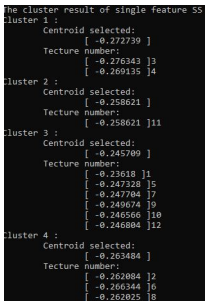
(j) ER



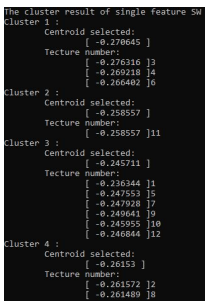
(k) SL



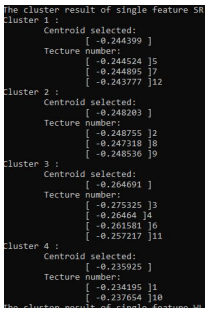
(l) SE



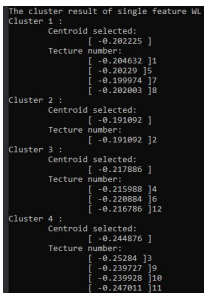
(m) SS



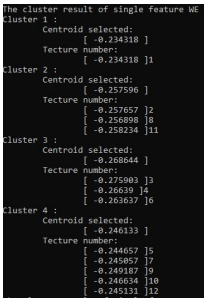
(n) SW



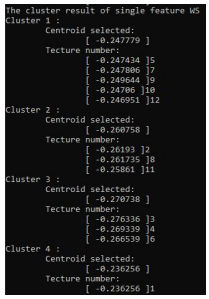
(o) SR



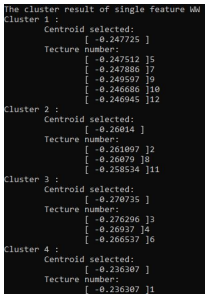
(p) WL



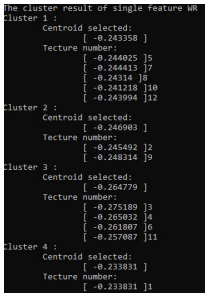
(q) WE



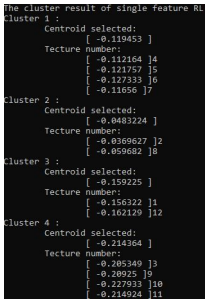
(r) WS



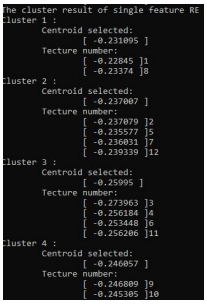
(s) WW



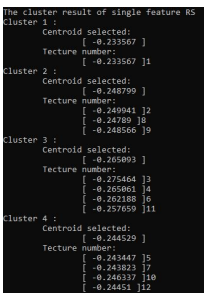
(t) WR



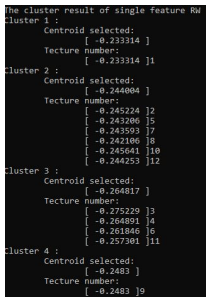
(u) RL



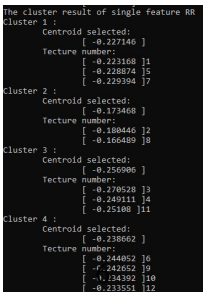
(v) RE



(w) RS



(x) RW



(y) RR

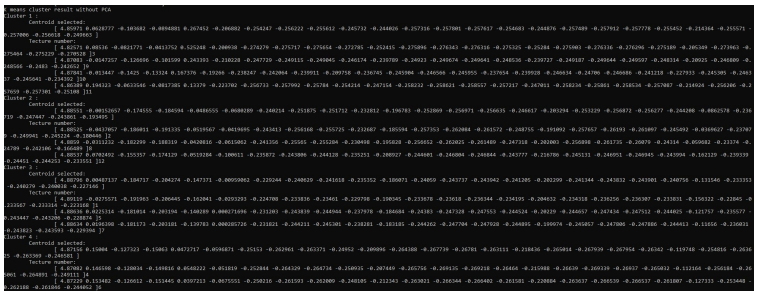
Figure 7: Cluster result using single feature vector

The above mentioned cluster result is only one trial of result. Actually, the five trials are made to get the general performance of 25 different single feature vector. The statistics suggest that LW has the strongest discriminant power and RS, RL, RE, RS, RR almost every 2D kernel with Ripple kernel are very weak, which means Ripple Kernel is not quite appropriate for these texture. Among them, compared with other trials of result, I think RW is the weakest one. By using Fourier transform, we can get the frequency function of each 1-D kernels. Level is low pass filter with cut off frequency at 0.5. Edge, Spot and Wave are all band pass filters. And, Ripple is high pass filter. What's more, the tensor product results of each 1D kernel can be considered as figure out the texture from x and y two axis separately. To justify this conclusion, we can easily find that in **Figure** 7, the cluster result of single filter LW is very close to the result obtained by 25 feature vectors. But the result obtained by single filter RW is the worst cluster result.

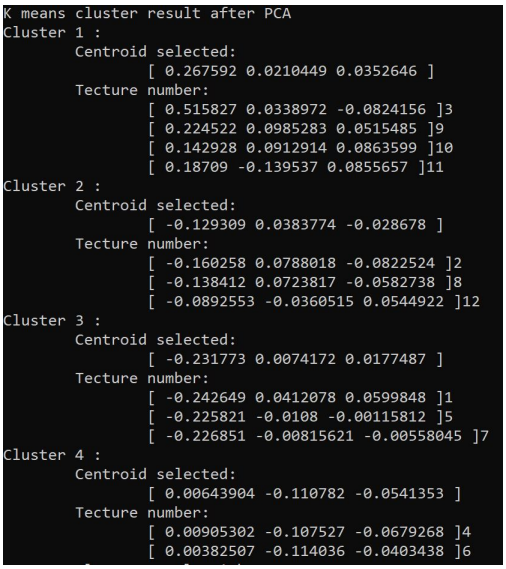
From the prospective of theory, Level kernel will work well in difference in brightness. Since we have already substrate mean, this kernel will not be the strongest one. Edge kernel performs very well in texture with edge whose is vertical edge in texture 3, texture 9 and texture 11. We can found that LE, EE, SE, WE and RE make good division for these textures with edge. Also, Spot kernel work well in the texture with spots. However, for these kernels designed for specific texture, they can only have strong discriminant power in specific class. Instead, they don't have the strongest discriminant power for whole bunch of textures. According to exclusive method, only wave kernel is left. Actually, all 2D kernels with wave component have good discriminant result. Among them, compared with other trials of result, I think LW is the best one.

1.4.2 Effectiveness of feature dimension reduction

The cluster result for 3D vector and 25D vector are shown in **Figure** 8 and **Figure** 9.

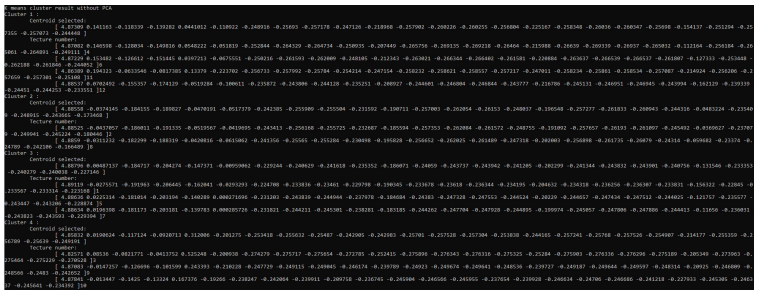


(a) 25D

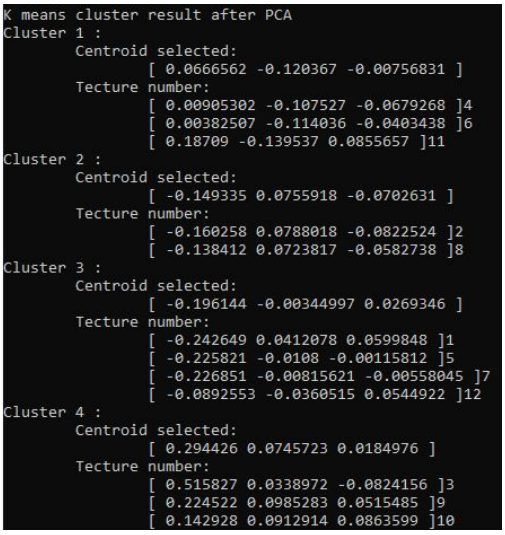


(b) 3D

Figure 8: Comparison1 between 3D vector and 25D vector



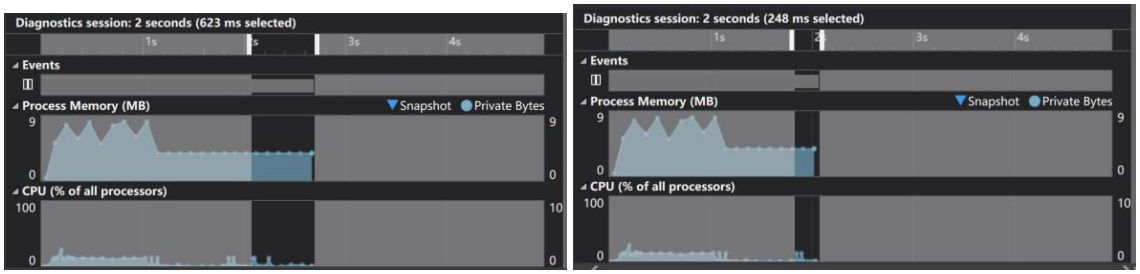
(a) 25D



(b) 3D

Figure 9: Comparison2 between 3D vector and 25D vector

From these two comparisons, we can find the using of PCA has no changes in the cluster result. The only change is about saving of time, which is shown in **Figure** 10.



(a) Time without PCA

(b) Time with PCA

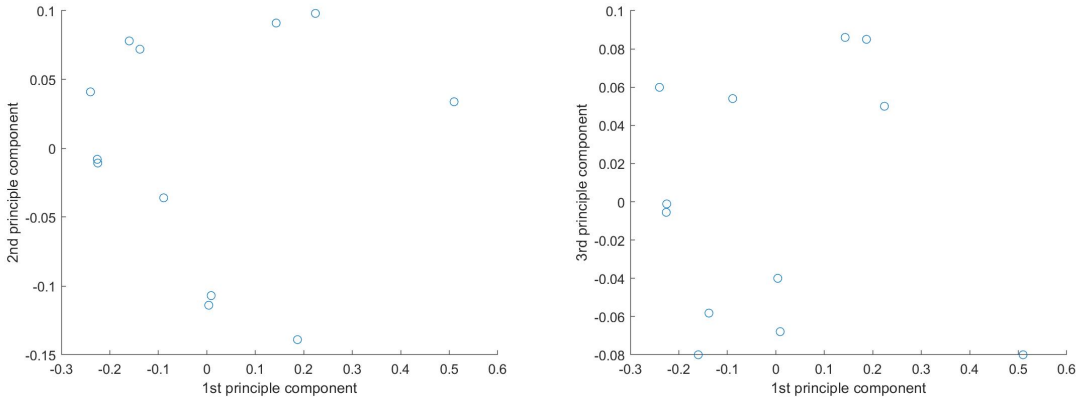
Figure 10: Time using without or with PCA

1.4.3 K-means and K-means++

The difference between K-means and K-means++ is the selection of initial cluster centers. The K-means++ follows the procedure:

- Choose one center randomly
- Compute distance between x and the nearest center for remaining data point x
- Choose one new data point at random as a new center, where a point x is chosen with probability proportional to distance
- Repeat steps 2 and 3 until k centers have been chosen

The K-means algorithm just ends at the first step, but the K-means++ follows procedure and make the initial centroid far away from other cluster centroids. However, the cluster for feature vector is more suitable to use K-means than K-means++ from the observation of cluster result. From the **Figure** 11, we can make some explanations.



(a) Relationship between first and second principle component (b) Relationship between first and third principle component

Figure 11: Relationship between different principle components

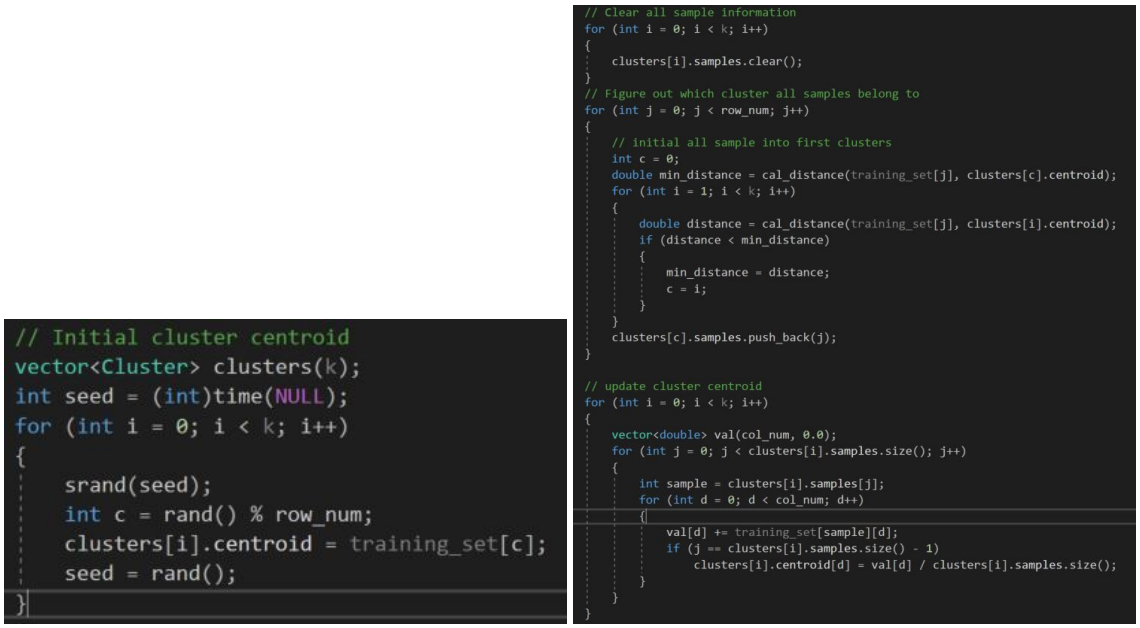
From these figures, we can find some feature vectors are far from all other feature vectors maybe because of noise, which make it become a cluster itself in K-means++. With these cluster only contained one feature vector, the result is not good. Thus, I still choose K-means as cluster algorithm.

1.4.4 Discussion for K-means result

I implement this algorithm by myself without using any toolbox. The pipeline of this algorithm is:

- Generate the initial cluster centroids at random
- Figure out which cluster all samples belong to
- Update the centroids according to the distance between itself and other samples
- Repeat step 2 and step 3 until the cluster converge

The screenshots for this part are shown in **Figure** 12.



```
// Initial cluster centroid
vector<Cluster> clusters(k);
int seed = (int)time(NULL);
for (int i = 0; i < k; i++)
{
    srand(seed);
    int c = rand() % row_num;
    clusters[i].centroid = training_set[c];
    seed = rand();
}
```

(a) Initial part

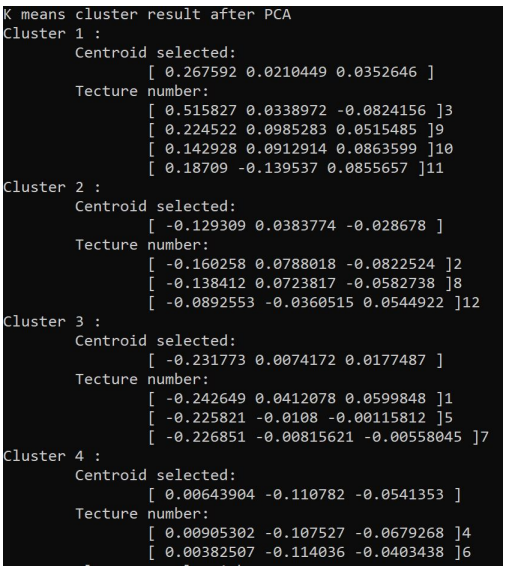
```
// Clear all sample information
for (int i = 0; i < k; i++)
{
    clusters[i].samples.clear();
}
// Figure out which cluster all samples belong to
for (int j = 0; j < row_num; j++)
{
    // initial all sample into first clusters
    int c = 0;
    double min_distance = cal_distance(training_set[j], clusters[c].centroid);
    for (int i = 1; i < k; i++)
    {
        double distance = cal_distance(training_set[j], clusters[i].centroid);
        if (distance < min_distance)
        {
            min_distance = distance;
            c = i;
        }
    }
    clusters[c].samples.push_back(j);
}

// update cluster centroid
for (int i = 0; i < k; i++)
{
    vector<double> val(col_num, 0.0);
    for (int j = 0; j < clusters[i].samples.size(); j++)
    {
        int sample = clusters[i].samples[j];
        for (int d = 0; d < col_num; d++)
        {
            val[d] += training_set[sample][d];
            if (j == clusters[i].samples.size() - 1)
                clusters[i].centroid[d] = val[d] / clusters[i].samples.size();
        }
    }
}
```

(b) Iteration part

Figure 12: K-means function

Since the initial cluster centroids are selected at random and it is very important for the next procedure, actually, the good choice of initial cluster centroids tend to get good cluster result. By this way, the result is not quite robust. Except for the two output results presented in experiment section, I just present other two results in **Figure** 13 and **Figure** 14.



```
K means cluster result after PCA
Cluster 1 :
        Centroid selected:
                [ 0.267592 0.0210449 0.0352646 ]
        Tecture number:
                [ 0.515827 0.0338972 -0.0824156 ]3
                [ 0.224522 0.0985283 0.0515485 ]9
                [ 0.142928 0.0912914 0.0863599 ]10
                [ 0.18709 -0.139537 0.0855657 ]11
Cluster 2 :
        Centroid selected:
                [ -0.129309 0.0383774 -0.028678 ]
        Tecture number:
                [ -0.160258 0.0788018 -0.0822524 ]2
                [ -0.138412 0.0723817 -0.0582738 ]8
                [ -0.0892553 -0.0360515 0.0544922 ]12
Cluster 3 :
        Centroid selected:
                [ -0.231773 0.0074172 0.0177487 ]
        Tecture number:
                [ -0.242649 0.0412078 0.0599848 ]1
                [ -0.225821 -0.0108 -0.00115812 ]5
                [ -0.226851 -0.00815621 -0.00558045 ]7
Cluster 4 :
        Centroid selected:
                [ 0.00643904 -0.110782 -0.0541353 ]
        Tecture number:
                [ 0.00905302 -0.107527 -0.0679268 ]4
                [ 0.00382507 -0.114036 -0.0403438 ]6
```

Figure 13: Cluster result3

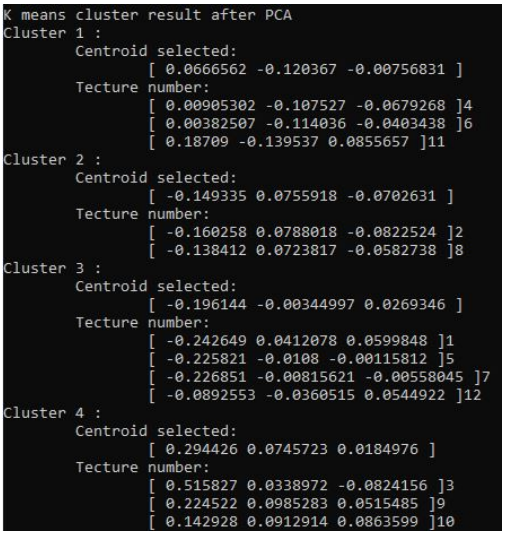
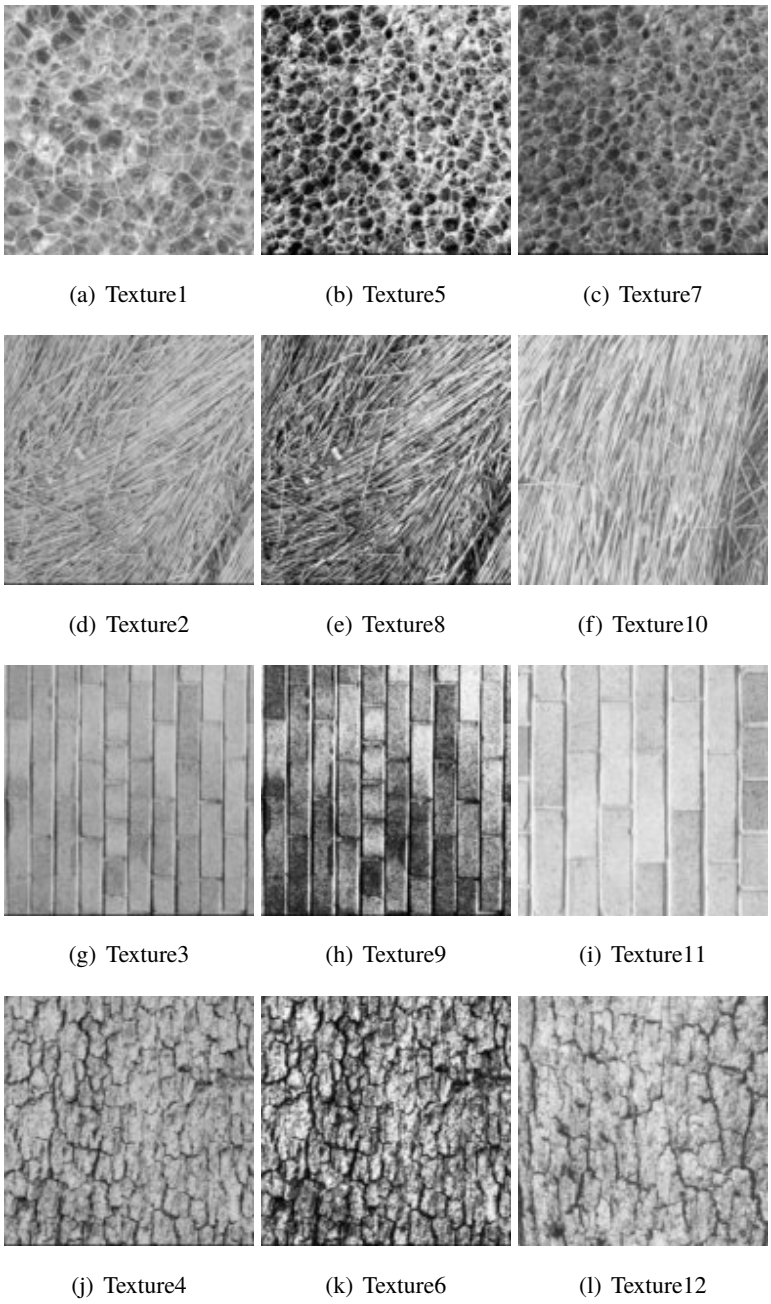


Figure 14: Cluster result4

And then, I make the following **Table** 1. All clusters are shown in **Figure** 15. The cluster for 1 5 7 can be classified correctly every time. The problem always happens in texture 10 and texture 12. For 10, it looks a little bit white than 2 and 8. What's worse, the direction of texture is quite different. And for 12, the size of texture is greater than 4 and 6 so that it is difficult to cluster them into one same class. Also, I have made a trail for five clusters rather than four cluster. By this way, the groups are correct with texture 10 being in a group by itself.



(a) Texture1 (b) Texture5 (c) Texture7

(d) Texture2 (e) Texture8 (f) Texture10

(g) Texture3 (h) Texture9 (i) Texture11

(j) Texture4 (k) Texture6 (l) Texture12

Figure 15: Classes for textures

Table 1: Statistics for four outputs

Cluster number	Actual label	Output1	Output2	Output3	Output4
1	1 5 7	1 5 7	1 5 7	1 5 7	1 5 7 12
2	2 8 10	2 8	2 8	2 8 12	2 8
3	3 9 11	3 9 11	3 9 10 11	3 9 10 11	3 9 10
4	4 6 12	4 6 10 12	4 6 12	4 6	4 6 11

2 Texture Segmentation

2.1 Abstract and Motivation

Image segmentation is the technique and process of dividing an image into specific regions with unique properties and proposing objects of interest. It is a key step from image processing to image analysis. The existing image segmentation methods are mainly divided into the following categories: threshold-based segmentation methods, region-based segmentation methods, edge-based segmentation methods, and segmentation methods based on specific theories. From a mathematical point of view, image segmentation is the process of dividing a digital image into regions that do not intersect each other. The process of image segmentation is also a marking process, that is, assigning the same number to the pixels belonging to the same area [2].

2.2 Approach and Procedures

There are some necessary procedures to followed for texture segmentation.

2.2.1 Laws feature extraction

The first step is to apply 25 Laws filters to the input image and get 25 gray-level images. 25 gray-level images are shown in **Figure** 16.

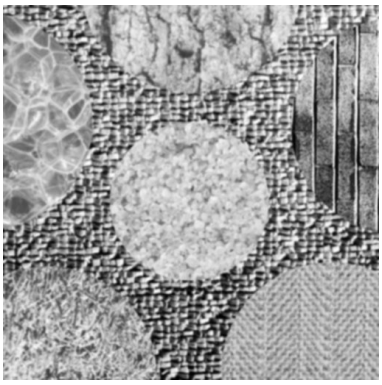


Figure 16: Gray image after filter LL

This image is from gray-scale image after LL filter by normalizing with mean. Since all kernels have a zero-mean except for LL, I cannot find a proper normalized operator to make normalizing and the gray-scale of these images don't make sense.

2.2.2 Energy feature computation

At this step, I use lots of window with different window size to compute the energy measure for each pixel based on the results from step 1. And then, I substrate the average of the gray-level of all pixels in this window

to the pixel calculated the energy feature. The flow diagram is shown in **Figure** 17.

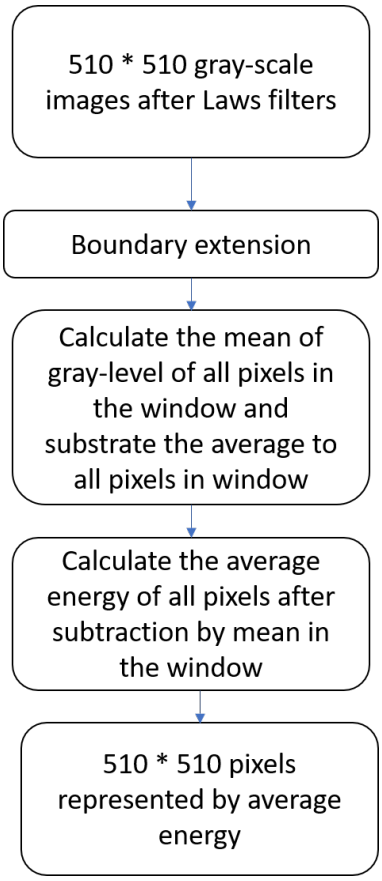


Figure 17: Flow diagram

To be mentioned, the using of subtraction with mean is to eliminate the effect of illumination of images. Not like the first problem subtraction with the mean of whole image, this time, I just substrate the local mean in the corresponding window. After that, I use the following formula to calculate average energy,

$$average = \frac{1}{M * N} \sqrt[2]{\sum_{m,n=1}^{M,N} k_{mn}^2}$$

2.2.3 Energy feature computation

At this time, we get the 25D energy feature vector because of 25 features for each pixels so that we has 510 * 510 energy feature vectors. And then, all other 24 values of rows are divided by the value of first row which is the value of LL. By this way, its energy is normalized at each pixel.

2.2.4 Segmentation

The last step apply K-means algorithm to cluster these 510 * 510 feature vectors into 7 clusters. In C++, I use the data structure of pair to record the (x,y) location and energy feature vector of this pixel so that there are 510 * 510 pairs. Each feature vector is a 25D vector. According to the energy feature vector of each pixel, k-means cluster is used to give each pair a label from A to G. And then, we traversal all these 510 * 510 pixels and set this pixel with seven different gray-levels from 0 to 255.

2.3 Experimental Results

I use the following notation for each texture in the comb.raw, shown in **Figure** 18.

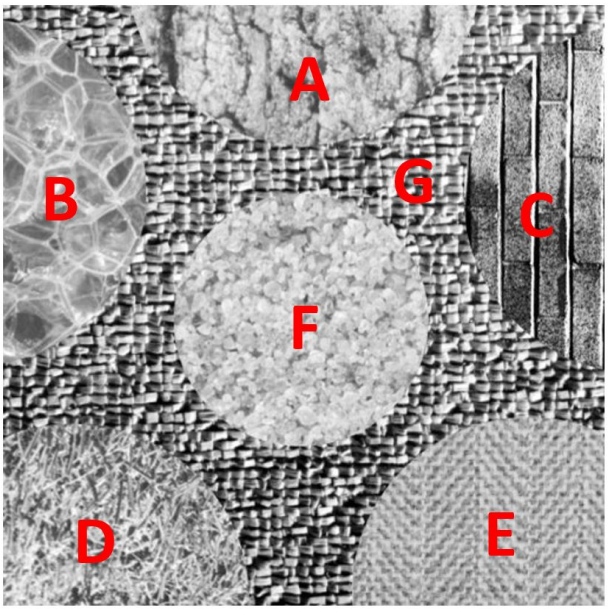
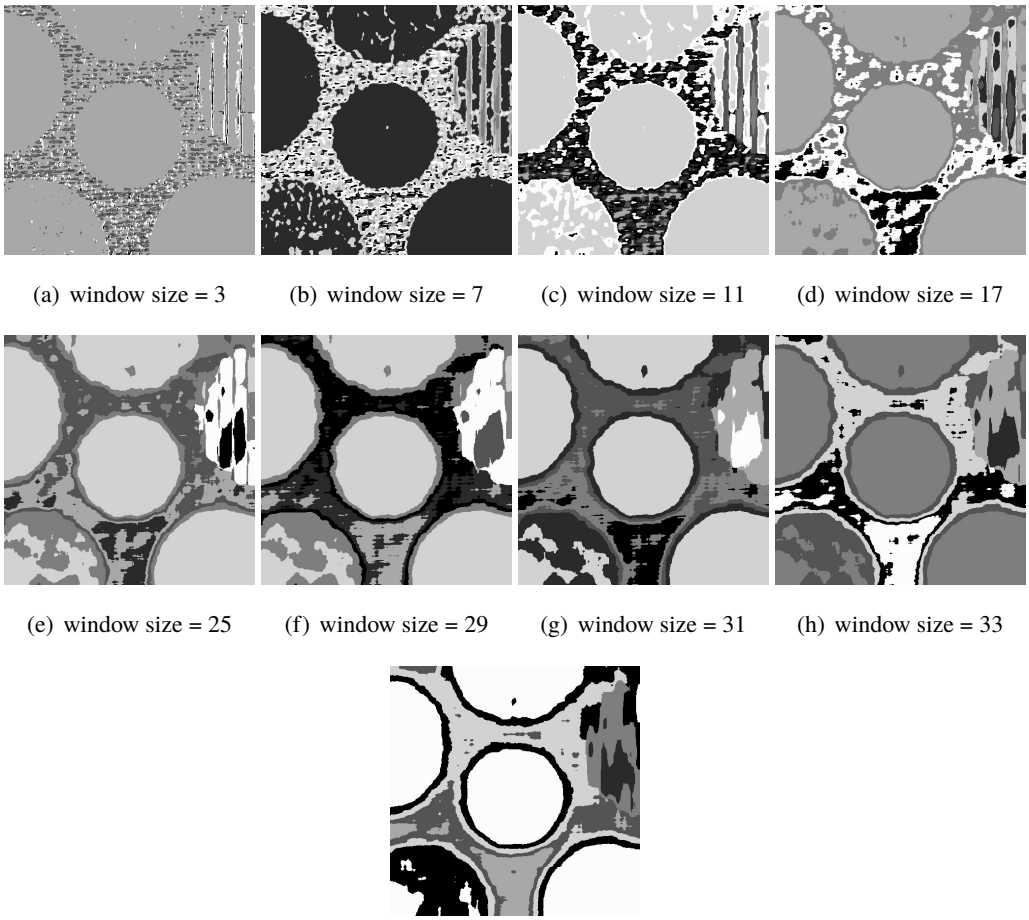


Figure 18: Notation for textures

For representation, I use 7 gray levels to do denotation. And the result with different window size are shown **Figure** 19.



(a) window size = 3 (b) window size = 7 (c) window size = 11 (d) window size = 17

(e) window size = 25 (f) window size = 29 (g) window size = 31 (h) window size = 33

(i) window size = 41

Figure 19: Segmentation with different window size

2.4 Discussion

2.4.1 Evaluation of window size

We can find that the texture in the corner and edge cannot be capture correctly when the window size is small. In order to capture the texture in the edge between two different textures, we need to choose the larger window size. By this way, the edges in that texture will be considered in the large window. In large window, more pixel will be considered so that in the corner and edge, the texture will be regarded as another texture with the combination of two different textures. From the image, we can see that between two texture, there is another layer for edge.

However, the segmentation results for all window size are not quite good. As the window size is small, we can consider that the average energy of each pixel doesn't contain enough information for the texture. At this time, only texture C looks different because it contains a line from up to down. As the window size going up, there are discriminating result between G, C and D. However, we cannot distinguish A, B, F and E from the

result maybe because the size of those textures are quite familiar. The idea to distinguish them by different window size is not good so that we need to find some another ways to make it. Actually, even from our eye, these four texture are not quite different especially for E and F.

2.4.2 Frequency response for Laws filters

The frequency response for each Laws Filter is shown in **Figure** 20.

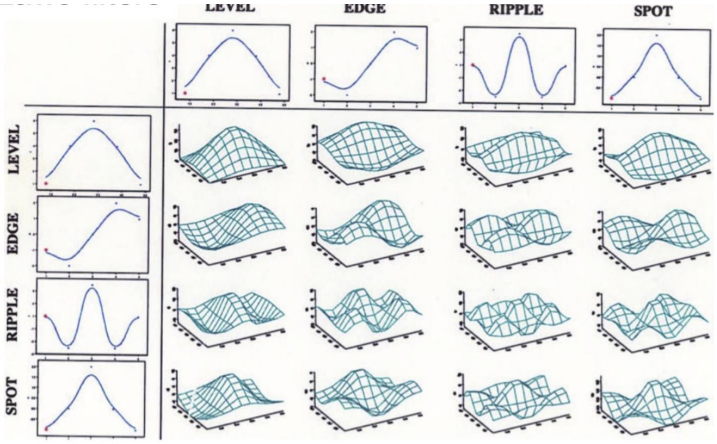


Figure 20: Frequency response

From the figure, we can find that different Laws filter perform different roles in feature extraction. For example, LL is a band pass filter so that only specific texture can get high frequency respond by this filter. And high pass filter will filter out some low frequency component of image and the image with many edges which are the high frequency component in frequency domain will obtain the high frequency response. Some filters represent more complicated frequency function in frequency domain. For each texture image, if a particular filter response is fitting in this image, it will get high frequency response and high energy value. Also, all 25 Laws filter represents 15 different kinds of filters. LE, ES, LS, EW, LW, ER, LR, SW, WR and SR are the same as EL, SE, SL, WE, WL, RE, RL, WS, RW and RS in frequency domain.Thus for each 25 dimension feature vector, we can make dimension reduction by this way to be a 15D vector without changing the cluster result.

3 Advanced Texture Segmentation Techniques

At this part, I make the following improvements to enhance my segmentation result.

3.1 PCA for feature reduction

From the comparisons, we can find the using of PCA has no changes in the cluster result. The only change is about time saving, which is a very useful improvement for examination of other improvement method. It

really saves time in the way to make experiments.

3.2 Mahalanobis distance

The Mahalanobis distance is a measure of the distance between a point P and a distribution D, introduced by P. C. Mahalanobis in 1936. It is a multi-dimensional generalization of the idea of measuring how many standard deviations away P is from the mean of D. This distance is zero if P is at the mean of D, and grows as P moves away from the mean along each principal component axis. The Mahalanobis distance measures the number of standard deviations from P to the mean of D. If each of these axes is re-scaled to have unit variance, then the Mahalanobis distance corresponds to standard Euclidean distance in the transformed space. The Mahalanobis distance is thus unitless and scale-invariant, and takes into account the correlations of the data set [3].
Since the K-means algorithm is wrote by myself, which I have introduced in last section, this time I just replace the origin euclidean distance calculation between centroids and sample vector with Mahalanobis distance. The formula is shown as followed,

$$D_M(x) = (x - \mu)^T S^{-1} (x - \mu)$$

where x represents sample, μ represents sample mean and S represents covariance matrix of sample.

3.3 Gaussian window to calculate average energy

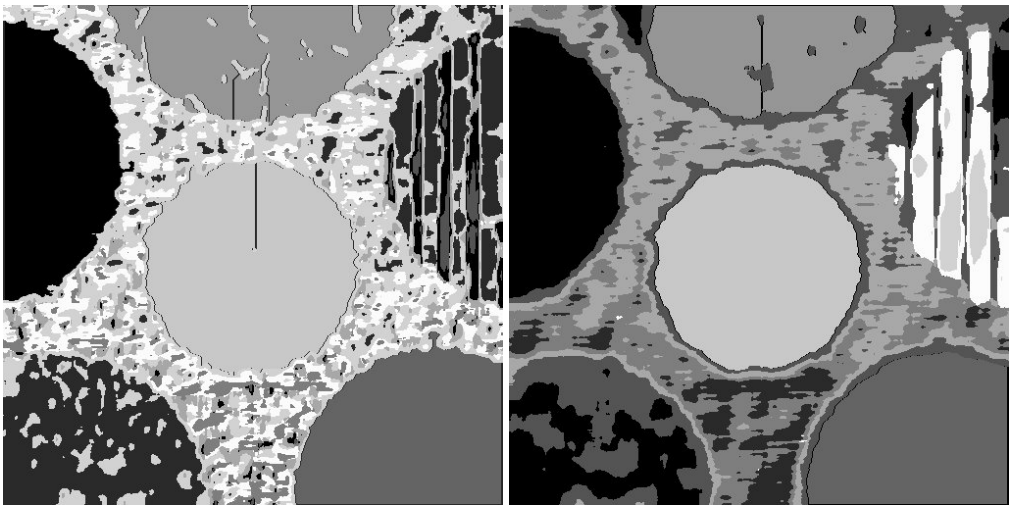
Actually, in the previous procedure to calculate the average energy, we use the uniform window, which may not contain the details of textures. This time, instead, I use Gaussian weighted window to calculate the average energy. For some specific texture, this changing works well. And the formula goes to,

$$average = \frac{1}{M * N} \sqrt[2]{G_{mn} * \sum_{m,n=1}^{M,N} k_{mn}^2}$$

where G_{mn} represents Gaussian weighted value of Gaussian filters.

3.4 Enhancement for the boundary of two adjacent regions

In some cases, the edge of two adjacent regions is quite vague. These segmentation with vague boundary are not good for further processing. By this way, we need to focus on the texture properties in these two regions only. One way is that using different larger size window and changing the weight of window. The comparison between the results with this enhancement are shown in **Figure** 21.



(a) Without edge enhancement (b) With edge enhancement

Figure 21: Comparison about edge enhancement

3.5 Post-processing technique to merge small holes

In some cases, the region for texture contains some unexpected small holes. In order to eliminate these holes, the following flow diagram is shown in **Figure** 22.

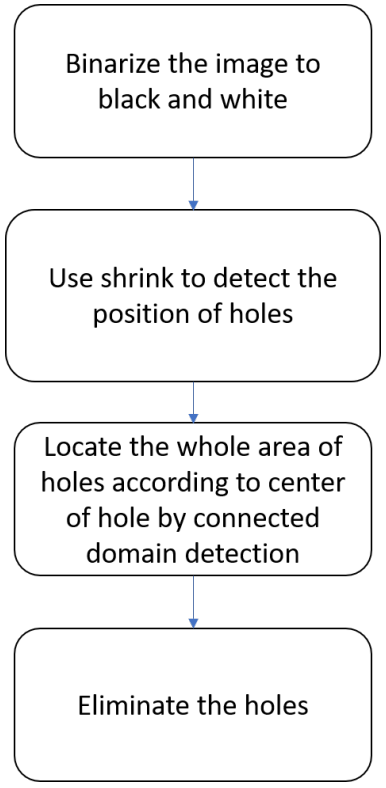
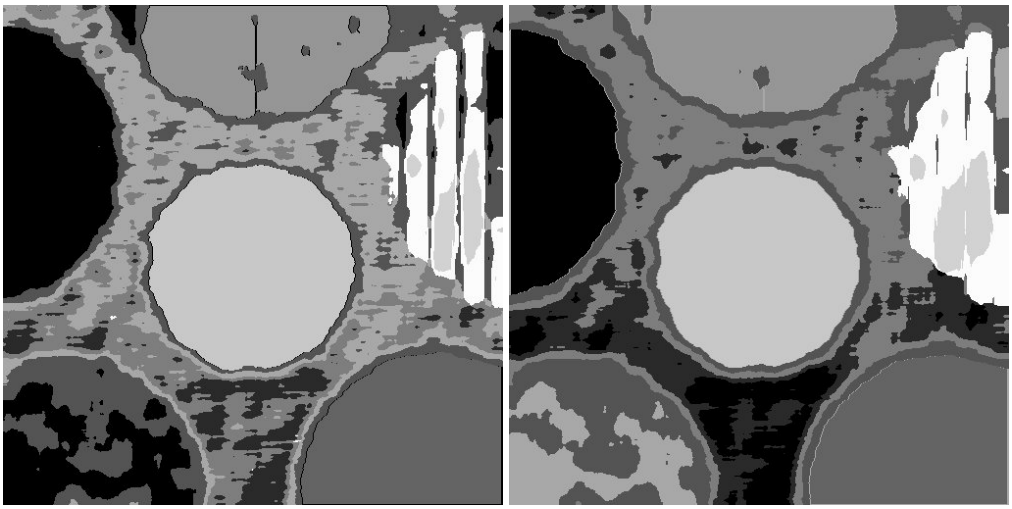


Figure 22: Flow diagram

The comparison is shown in **Figure** 23. We can see the image become a little bit cleaner.

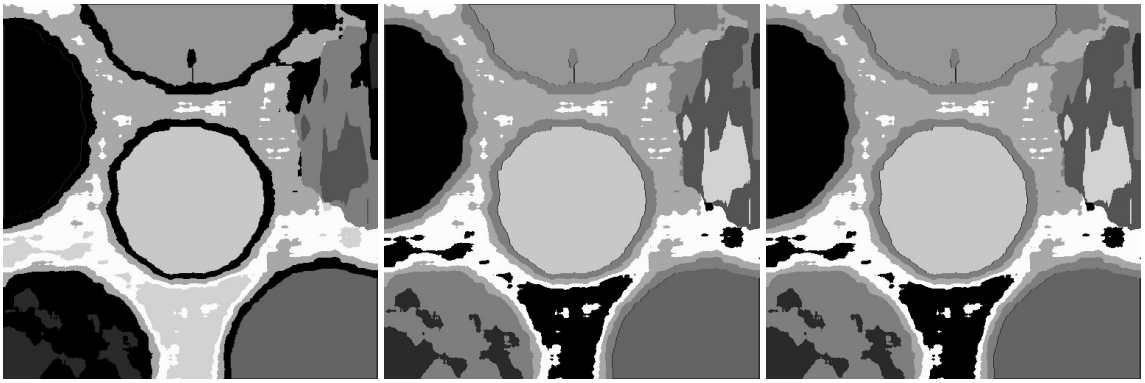


(a) without holes elimination (b) with holes elimination

Figure 23: Comparison about holes elimination

3.6 Experiment results

Having using the above mentioned enhancements, I get the result after enhancement shown in **Figure** 24.



(a) result 1 (b) result 2 (c) result 3

Figure 24: Final result after all enhancements

4 Image Feature Extractor

4.1 Abstract and Motivation

Feature extraction is a dimensionality reduction process, where an initial set of raw variables is reduced to more manageable groups (features) for processing, while still accurately and completely describing the

original data set. And for image, even for a small size of image, the time complexity to process this image is huge. So, image feature extraction is introduced to maintain the basic and important features and reduce the dimension of image [4].
When the input data to an algorithm is too large to be processed and it is suspected to be redundant, then it can be transformed into a reduced set of features (also named a feature vector). Determining a subset of the initial features is called feature selection. The selected features are expected to contain the relevant information from the input data, so that the desired task can be performed by using this reduced representation instead of the complete initial data.
And in this part, SIFT is used as a very robust image feature extractor.

4.2 Approach and Procedures

Matching features across different images in a common problem in computer vision. When all images are similar in nature (same scale, orientation, etc) simple corner detectors can work. But when you have images of different scales and rotations, you need to use the Scale Invariant Feature Transform. The following subsection is going to introduce the procedures of SIFT.

4.2.1 Establish Gaussian scale space

Within a certain range, no matter whether the object is large or small, the human eye can distinguish it. However, it is not so easy for computers to have the same capabilities. In an unknown scene, computer vision does not provide the size of objects. One way is to provide images with objects at different scales to the machine. In the process of establishing unified cognition, it is necessary to consider the feature points that exist in images at different scales. We can also simulate the imaging process of an object on the retina when the distance from the object is far from the object by the degree of blurring of the image. The closer the distance is to the object, the larger the image is, the more blurred the image is. This is the Gaussian scale space, using different parameter to blur to obtain the image with the same resolution in different blur level.
In order to keep key points invariant in scale, we generate the input images in different scale. By this way, we get Gaussian pyramid, which is also the scale space of the input image. This is done by applying Gaussian smoothing with different values of sigma. In each octave, the image is in the same scale but with different sigma value of Gaussian filter that is a multiple of k. And, between the adjacent octave, the image is down sampled by a factor of 2.

4.2.2 Extract key point by DoG extreme point

The purpose of constructing the scale space is to detect feature points that exist at different scales. Laplace of Gaussian works very well in maintaining the feature point of image. However, the time complexity is very large. Instead, we use difference of Gaussian. The DoG gives us an edge map. In order to detect the key point from the edge map and find the extreme points of the scale space, each pixel is compared with all adjacent

points of its image domain (same scale space) and scale domain (adjacent scale space). When it is greater than (or less than) all neighbouring phases, the change point is the extreme point.

4.2.3 Discard some key points

By comparing the detected local extrema points of DoG in a discrete spatial search, since the discrete space is the result of sampling the continuous space, the extrema points found in the discrete space are not necessarily the extreme points in the true sense. So, it is necessary to get rid of points that don't meet the conditions. You can find the extreme points by curve fitting through the scale space DoG function. The essence of this step is to remove the points where the local curvature of the DoG is very asymmetrical.
Two kind of key point to discard are:

- Low contrast feature points
- Unstable edge response points

4.2.4 Assign orientation for each key point

After the above steps, the feature points existing at different scales have been found. In order to achieve image rotation invariant, it is necessary to assign values to the direction of the feature points. The gradient distribution characteristics of the neighbourhood pixels of the feature points are used to determine the direction parameters, and then the gradient histogram of the image is used to obtain the stable direction of the local structure of the key points. The range to calculate the angle and amplitude is the area centred on the feature point and radius by $3 * 1.5 * \sigma$. The orientation of each pixel is weighted by its gradient magnitude. Then a histogram of 36 bins are created that represent 360 degrees and the direction of maximum value in histogram is selected as te main direction.
What's more, After calculating the gradient direction, the histogram is used to count the gradient direction and amplitude corresponding to the pixels in the neighbourhood of the feature points. The horizontal axis of the histogram is the angle of the gradient direction (the range of the gradient direction is 0 to 360 degrees, the histogram is 10 columns for every 36 degrees of the histogram, or 8 columns for one column without 45 degrees), and the vertical axis is the accumulation of the gradient magnitude corresponding to the gradient magnitude, and the peak of the histogram is the main direction of the feature point.
The paper also mentions the use of a Gaussian function smooth the histogram and enhance the effect of the neighbourhood points of the feature points on the direction of the key points and also reduce the effects of the mutations. In order to get a more precise direction, it is usually also possible to interpolate the discrete gradient histograms.
Also, in the gradient histogram, when there is a column value equivalent to 80% of the main peak energy, this direction can be considered as the feature point auxiliary direction. Therefore, a feature point may detect multiple directions (it may also be understood that one feature point may generate multiple feature points with the same coordinates but different directions).

4.2.5 Generate key point descriptors

Through the above steps, the SIFT feature point position, scale and direction information have been found. Next, we need to use a set of vectors to describe the key points, which is, to generate feature point descriptors. This descriptor not only contains feature points, but also contains pairs of feature points of its contributing pixels. The descriptor should have a high degree of independence to ensure a matching rate. The generation of feature descriptors has roughly three steps:

- Correct the main direction of rotation to ensure rotation invariance.
- Generate a descriptor to form a 128-dimensional feature vector normalization processing.
- Normalizing the length of the feature vectors to remove the influence of illumination.

4.2.6 Key point matching

With the above mentioned procedures, we can get lots of features shown in **Figure** 25 and **Figure** 26. And then, we apply Brute Force matcher and FLANN matchers in OpenCV to get the matching image. More details about these two matcher will discussed in section discussion.

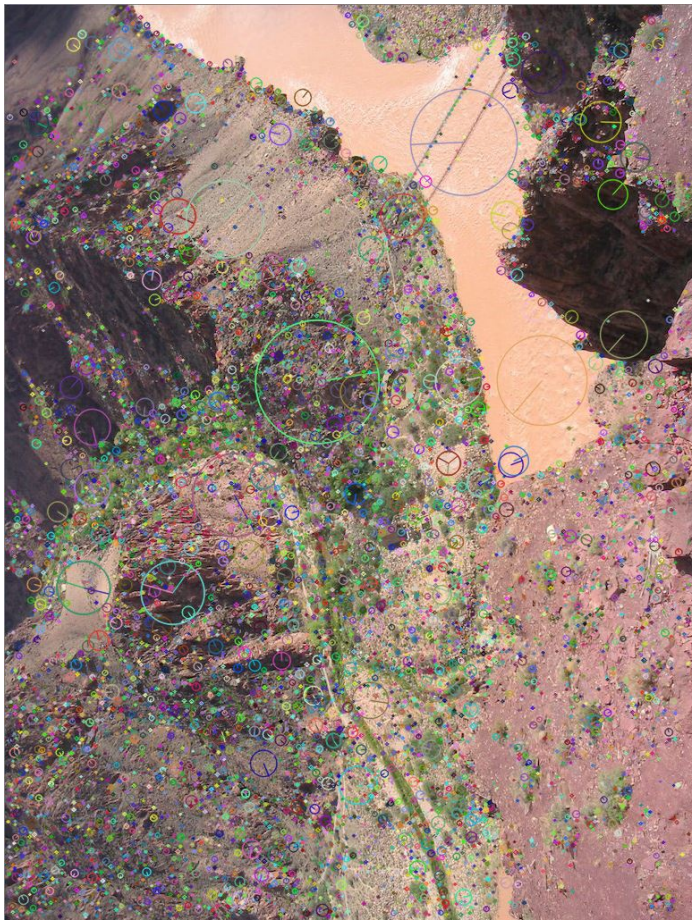


Figure 25: Features for river1

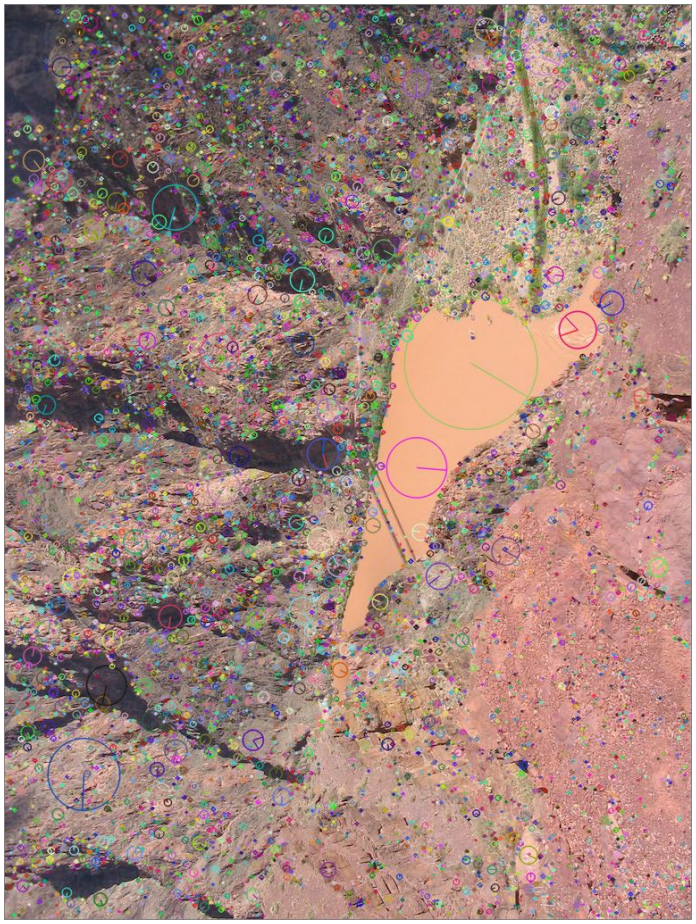


Figure 26: Features for river2

And then, we make matching according to the high dimension distance between features. Without any post processing, the whole bunch of feature matching is shown in **Figure** 27.

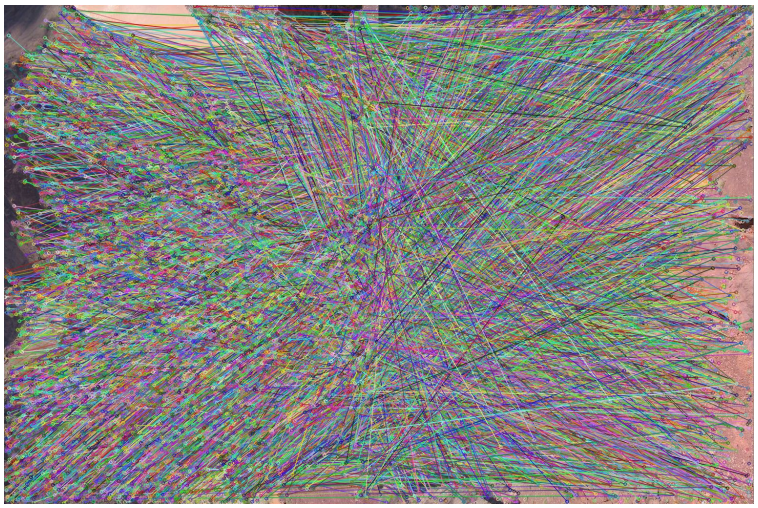


Figure 27: Matching without feature selection

The matching result consists many unreal matching, and we need to discard them. First, I find the minimum distance in whole bunch of feature matching set and set a parameter a to multiplex this minimum to become a threshold. The formula is shown as followed,

$$threshold = a * minimum$$

In this report, I choose a as 1.5 and 2 to present in **Figure** 28 and **Figure** 29.

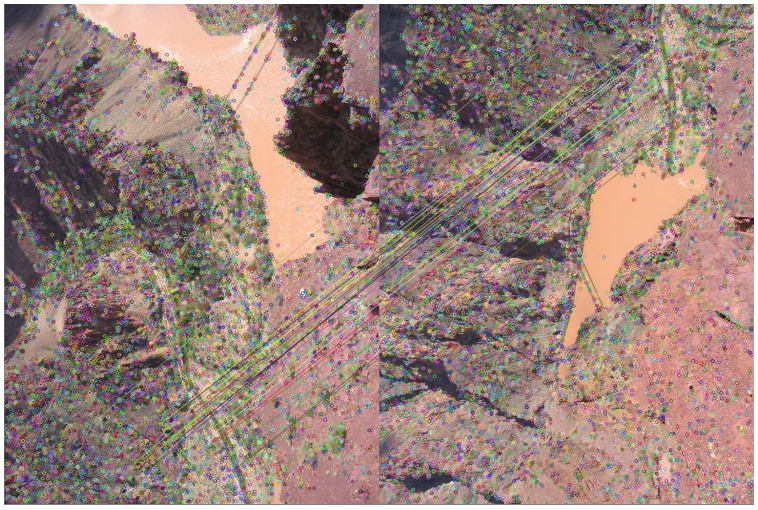


Figure 28: Matching with a = 1.5

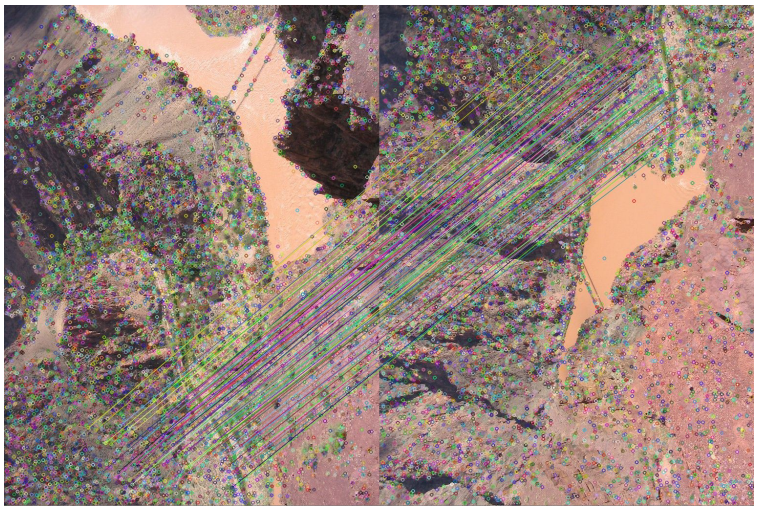
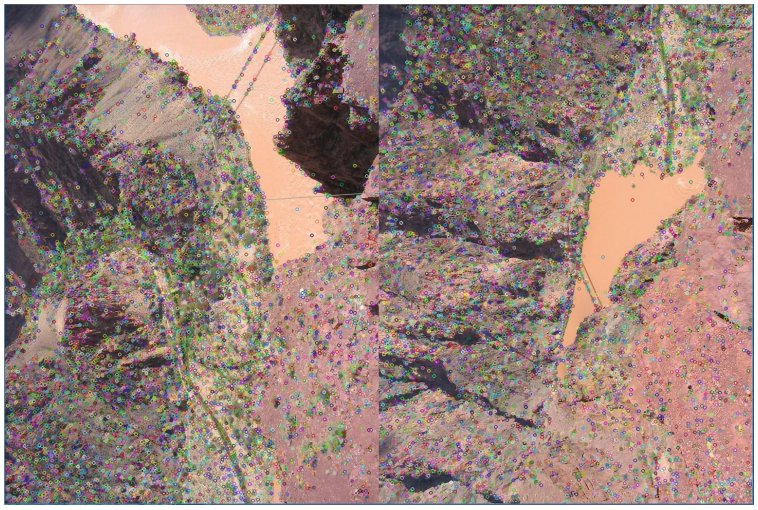


Figure 29: Matching with a = 2

4.3 Experimental Results

The key-point with the largest scale mean the largest l2 norm distance for Brute Force matcher. So, I filter all match points and get the match with largest l2 norm distance, which is shown in **Figure** 30.



(a) Key-point with largest l2 norm distance

(b) Details about Key-point with largest l2 norm distance

Figure 30: key point with largest distance

As I understand, the key-point with the largest scale in river 1 means the keypoint obtained in the smallest image in octave. It is shown in **Figure** 31 and its closest neighboring key-point in river image 2 is shown in **Figure** 32.

Figure 31: Key-point with largest scale

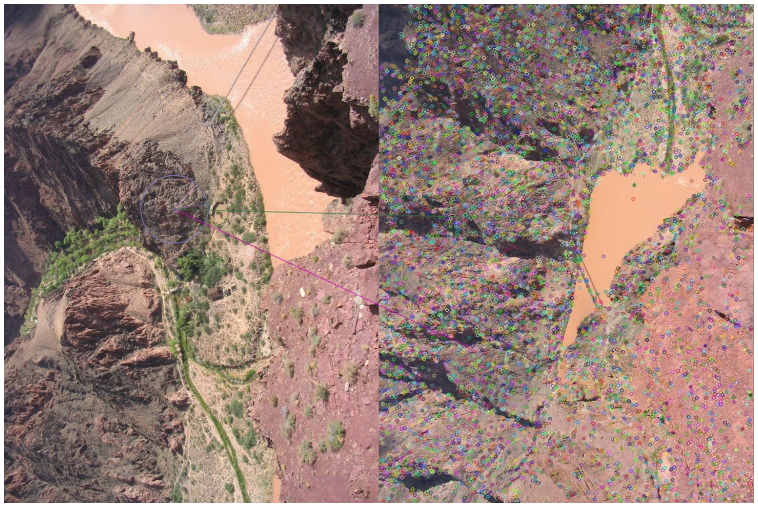


Figure 32: Match for Key-point with largest scale

The key-point with the smallest scale in river 1 is shown in **Figure** 33 and its closest neighboring key-point in

river image 2 is shown in **Figure** 34.

Figure 33: Key-point with smallest scale

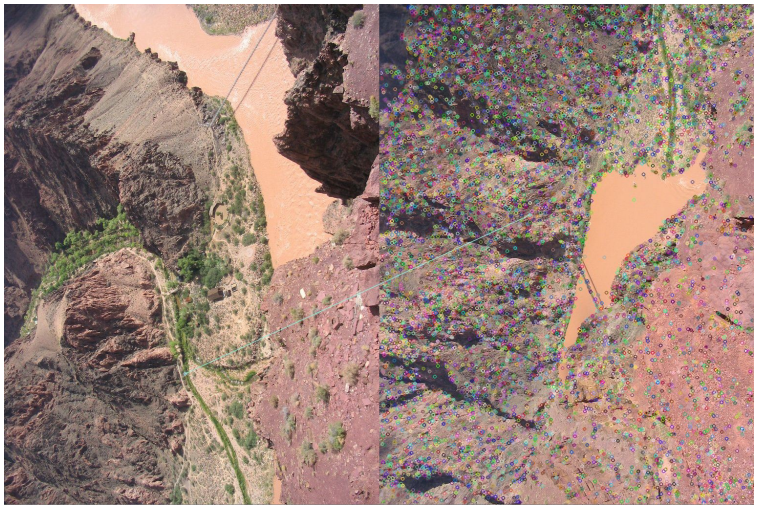


Figure 34: Match for Key-point with smallest scale

The key-point in first octave in river 1 and its closest neighboring key-point in river image 2 is shown in **Figure** 35.

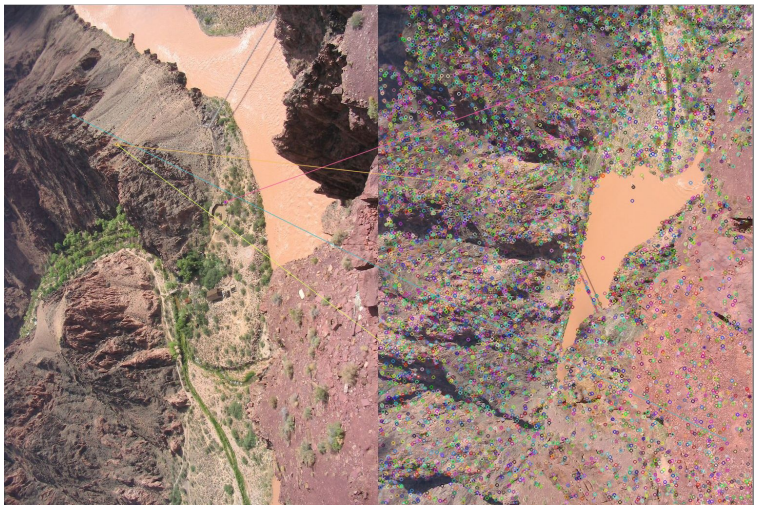


Figure 35: Match for Key-point in first octave

The key-point in last octave in river 1 and its closest neighboring key-point in river image 2 is shown in **Figure** 36.

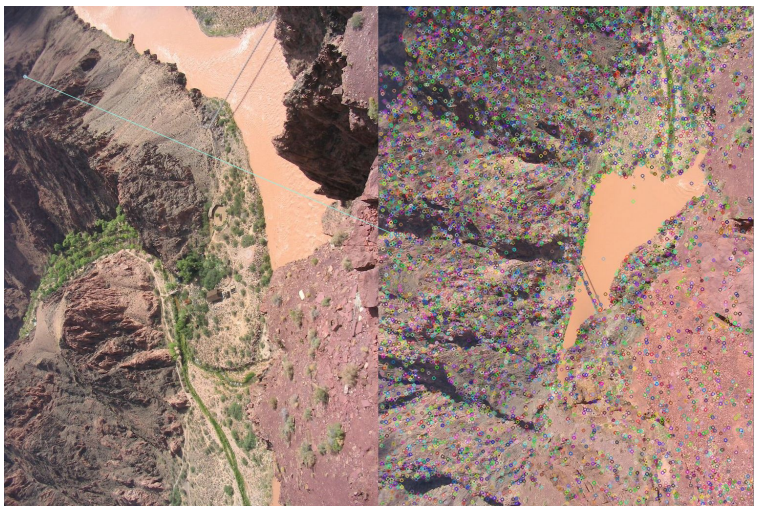


Figure 36: Match for Key-point in last octave

4.4 Discussion

4.4.1 Several questions

1. SIFT is robust to scaling, rotation and translation.

2. Scale-invariant through multi-scale filtering. In detail, the procedure to get the key point is through finding the extrema in Gaussian scale space. In order to keep key points invariant in scale, the input images in

different scale are generated. By this way, we get Gaussian pyramid, which is also the scale space of the input image. This is done by applying Gaussian smoothing with different values of sigma. In each octave, the image is in the same scale but with different sigma value of Gaussian filter that is a multiple of k. And, between the adjacent octave, the image is down sampled by a factor of 2.
And rotation- and translation-invariant through locating key-points as extreme of difference-of-Gaussian function. What's more, each key-point is assigned a canonical orientation. Having found the main orientation of each key point, the histogram is used to count the gradient direction and amplitude corresponding to the pixels in the neighborhood of the feature points, and the peak of the histogram is the main direction of the feature point. What's more, before calculating the 128 dimension vector, SIFT correct the main direction of rotation to ensure rotation invariance. The more details about rotation-invariant is explained in section orientation.
3. Illumination robustness achieved by thresholding the gradient magnitudes at a value of 0.1 times the maximum possible gradient value and canonical orientation is determined by the peak in a histogram of local image gradient orientation. And in the procedure to generate the descriptor, the 128 dimension feature vector is normalizing to remove the influence of illumination.
4. DoG is faster than LoG. The advantage of DoG against LoG is speed. The time complexity of LoG is large. Instead, the using of DoG which is the approximation of LoG saves lots of time.
5. The SIFT's output vector size in original paper is $8 * 4 * 4 + 8 * 2 * 2 = 160$. But in the toolbox of OpenCV, the size of output vector is 128.

4.4.2 Analysis

As we seen in the result about key point with the largest scale, which is a key point in the mountain, it really make sense because this key point is found in the smallest size of image in the image pyramid. The smallest size of image represents the largest octave of pyramid. From the **Figure** 25, we can see the size of keypoint which represents the diameter of meaningful keypoint neighborhood, which means that it's obtained from the small image. The same keypoint with large size is in river and some smooth texture. These keypoints cannot be found in image with large size because they are not sharp. But in high octave and with the small image size, as the smooth contour becoming sharp edge, these keypoint can be detected. However, these image is very blur so that these keypoint is not very good in scale-invariant.

4.4.3 Comparison about different matcher

The result shown in experiemtn result result are obtained by Brute Force matcher. Since the Brute Force matcher computes a match for every point, we need to wipe out some noisy and unreal matches. As we give the l2 norm distance threshold in the above mentioned procedure, the final result is shown in **Figure** 29. In this result, I filter out matches that were not relevant. However, this matcher filtering may create other extra errors when an object is compared with different objects because Brute Force matcher tend to match all

keypoint even irrelevant points. This is the difference between these two matchers. Flann based matcher uses the nearest neighbor's technique, which has faster speed and can improve accuracy by change parameters.

4.4.4 Orientation

After the main direction of the feature points is obtained, three pieces of information (x, y, σ, θ), which is position, scale and direction, can be obtained for each feature point. From this, a SIFT feature region can be determined. One SIFT feature region is represented by three values, the center represents the feature point position, the radius represents the scale of the key point, and the arrow represents the main direction. Key points with multiple directions can be copied multiple times, and then their direction values are assigned to the copied feature points respectively. One feature point produces multiple feature points with same coordinates and dimensions but different directions. The key point with the largest scale in river image is a key point with different two direction.

In order to ensure the rotation invariant of the feature vector, the coordinate axis is rotated by θ (the main direction of the feature point) in the vicinity of the feature point, that is, the coordinate axis is rotated to the main direction of the feature point.

After the rotation, a window of 8 × 8 is taken centering on the main direction. As shown in **Figure** 37, the center of the left image is the position of the current key point. Each small cell represents a pixel of the scale space where the key point neighborhood is located. The gradient amplitude and gradient direction of each pixel are obtained, and the arrow direction represents the gradient direction of the pixel and the length representing the gradient magnitude, weighting it using a Gaussian window. Finally, a gradient histogram of 8 directions is drawn on each 4×4 small block, and the accumulated value of each gradient direction is calculated to form a seed point, as shown in the right figure. Each feature point consists of 4 seed points, each of which has vector information in 8 directions. This kind of neighborhood directional information joint enhances the anti-noise ability of the algorithm, and provides reasonable fault tolerance for feature matching with positioning error.

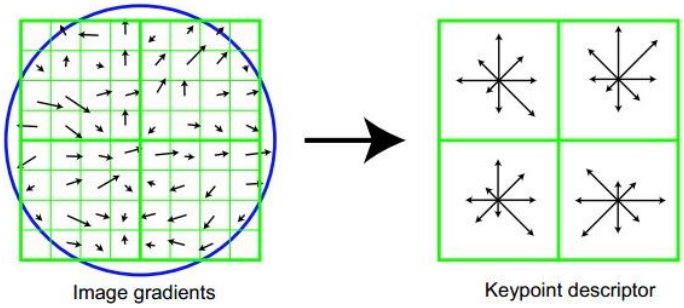


Figure 37: Generate direction descriptor

Different from the main direction, the gradient histogram of each seed region is divided into 8 direction intervals from 0-360, and each interval is 45 degrees, that is, each seed point has gradient intensity information

of 8 directions. A total of 16 seed points of 4×4 are used for each key point to describe, so that a key point can generate a 128-dimensional SIFT feature vector.

By segmenting the pixels around the feature points, the intra-block gradient histogram is calculated to generate a unique vector, which is an abstraction and uniqueness of the image information of the region. The procedure to get the 128 dimension vector is shown in **Figure** 38.

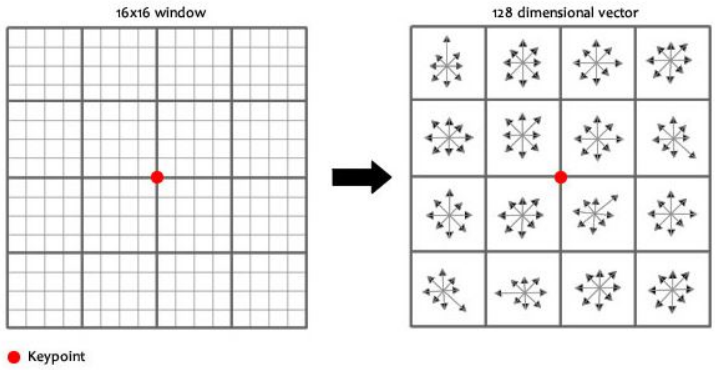


Figure 38: 128 dimension vector

From the **Figure** 25 and **Figure** 26, we can easily make some intuitive explanation. The orientation of each key-point is perpendicular to the side if the point is located in edge.

4.4.5 Matching

According to the result shown in **Figure** 25 and **Figure** 26, we get the following jointly image shown in **Figure** 39.

Figure 39: Joint of two images

Since after we use the threshold to wipe off some non-real key points, the matching result is really good. What's more, SIFT performs very well in scale-, rotation- and translation-invariant, which are the advantage of SIFT.

5 Bag of Words

5.1 Abstract and Motivation

The bag-of-words model is a simplifying representation used in natural language processing and information retrieval (IR). In this model, a text (such as a sentence or a document) is represented as the bag of its words, disregarding grammar and even word order but keeping multiplicity [5]. Also, for images, we can establish the codebook including some image vocabulary.

5.2 Approach and Procedures

5.2.1 Extract SIFT key points for each image

Ten training images are read. We extract their SIFT key points and generate descriptor for each feature or key point. The number of features is shown in **Table** 2.

Table 2: Number of features for each training image

	Number of features
zero1	3
zero2	4
zero3	3
zero4	6
zero5	0
one1	12
one2	8
one3	3
one4	13
one5	9

What's more, the feature extracted images are shown in **Figure** 40.

(a) one1 (b) one2 (c) one3 (d) one4 (e) zero1 (f) zero2 (g) zero3 (h) zero4 (i) zero5

Figure 40: Images with features

5.2.2 K means cluster

After we get the 61 features, we use k means cluster to separate them into 2 classes. The clusters histogram is shown in **Figure** 41.

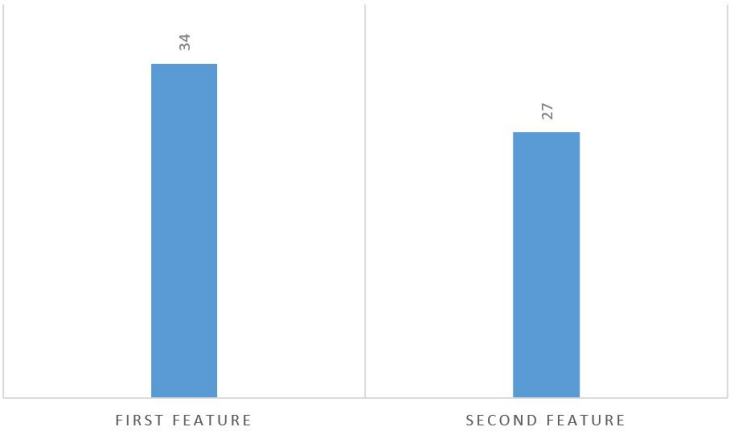


Figure 41: Clusters for features

5.2.3 Establish vocabulary

By this way, we use these features to establish a codebook with bin size of 2.

5.2.4 Testing

To be mentioned first, the whole bunch of feature and centroids are represented by 128 dimension vector. In this step, we read eight.jpg to extract the its features. The image is shown in **Figure** 42.

Figure 42: Features of eight

For this image, we get 12 features. And then, we calculate high dimension distance from this 128-Dimension feature vector to two different centroids in two clusters and judge which cluster the feature belongs to.

5.3 Experimental Results

Finally, we get the histogram of the bag of word for this image, which is shown in **Figure** 43.

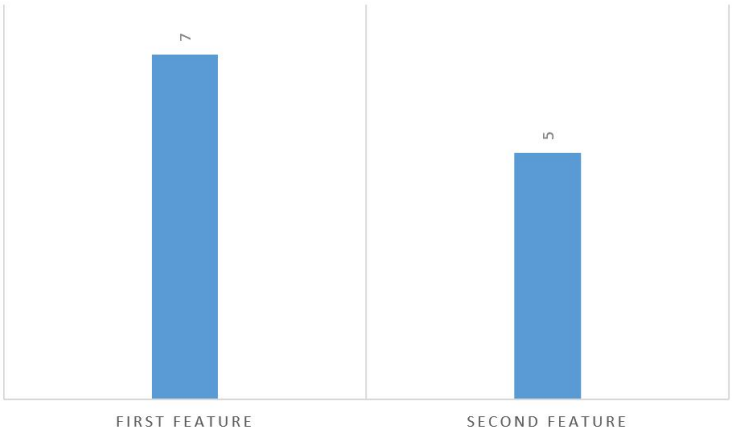


Figure 43: Histogram of the bag of word for eight

5.4 Discussion

5.4.1 Evaluation of codebook

In order to evaluate the image vocabulary in code book, we need to find the real representation of each cluster. As we know, there are two kinds of training image, one and zero. By this way, we read them again and calculate the high dimension distance between each feature of zero or one from the centroids of two clusters. And if the feature is belonged to first label cluster, we make first label score plus 1. By this way, we get the following **Figure** 44. The first five scores are for all one.jpg and then the following five scores are for all zero.jpg. What's more, the last one is for eight.jpg.

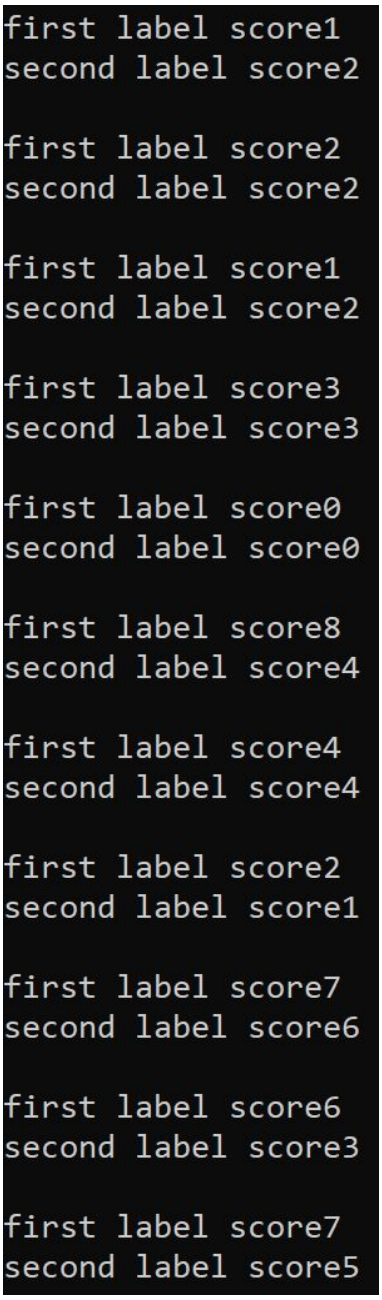
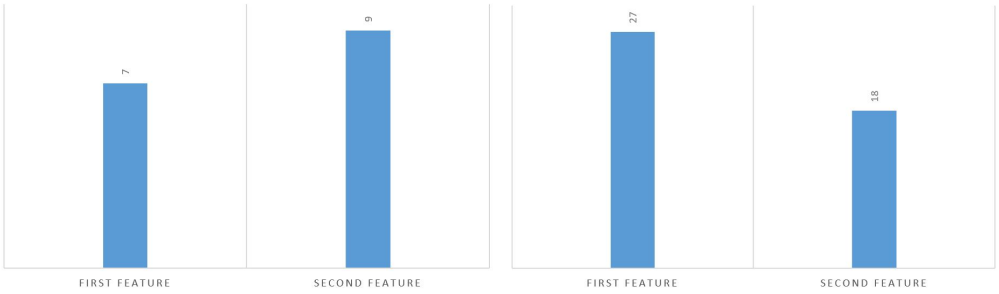


Figure 44: Label scores

By this way, we can count the histogram for one and zero separately, which is shown in **Figure** 45.



(a) Histogram for all one.jpg (b) Histogram for all zero.jpg

Figure 45: Histograms for different set of images

What's more, the whole diagram is shown in **Figure** 46.

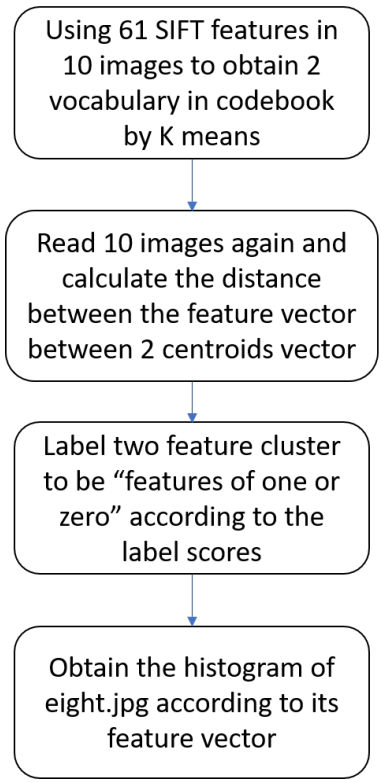


Figure 46: Flow diagram

From the histogram, we can call the first feature to be "features of one" and the second feature to be "features of zero". Both of them are what we get in codebook. Let's observe the result of eight. From human sighting, we think eight is more like zero because it actually consists of two zeros. Thus, the histogram of eight makes sense because it have more "features of zero" than "features of one".

5.4.2 K means

As I mentioned before, since the initial cluster centroids are selected at random and it is very important for the next procedure, actually, the good choice of initial cluster centroids tend to get good cluster result. By this way, the result is not quite robust. From the result shown before, the codebook with two vocabulary perform almost well in this experiment.

Reference

[1] Unser M. Texture classification and segmentation using wavelet frames[J]. IEEE Transactions on Image Processing A Publication of the IEEE Signal Processing Society, 1995, 4(11):1549-60.

[2] Arivazhagan S, Ganesan L. Texture segmentation using wavelet transform[J]. Pattern Recognition Letters, 2003, 24(16):3197-3203.

[3] Maesschalck R D, Jouan-Rimbaud D, Massart D L. The Mahalanobis distance[J]. Chemometrics and Intelligent Laboratory Systems, 2000, 50(1):1-18.

[4] Yan T W, Garcia-Molina H. SIFT: a tool for wide-area information dissemination[C]// Usenix Technical Conference. 1995.

[5] Wallach H M. Topic modeling:beyond bag-of-words[C]// International Conference on Machine Learning. 2006.

Appendix

In this part, I want to introduce the structure of program in details. There are three classes and one structure for the homework, Imagedata, kernel, algorithm and cluster.

Class Imagedata

In class Imagedata, there are six kinds of function such as basic function, Image operation, texture classification and texture segmentation.

```
//basic function
Imagedata(int h, int w, int p); //constructor
~Imagedata(); //destructor
void set_doubledata(); //initialize double data for shot noise
void initialize(int d); //initialize all data with value d
void read(unsigned char* buff); //read data from buff
void load(string path); //load data from path
unsigned char* write(); //write data to buff
void save(string path); //save data to path
int convert(int h, int w, int p); // convert the position in a the height * width * byteperpixel
int get_height(); // return height
int get_width(); // return width
int get_byteperpixel(); // return byteperpixel
vector<vector<vector<double>>> get_data_double(); // get double data of imagedata
```

```
//image operation
Imagedata Boundaryextension(int ex); //Boundary extension
Imagedata Crop(int N); //crop operation
Imagedata Boundaryextension_double(int ex); //Boundary extension for double data
Imagedata DN_Gaussion_double(int N, double sigma); // Gaussian filter for double data
Imagedata Convolution_double(Kernel k); // convolution for double data
void double2usignedc(); //change double data to unsigned char data
Imagedata Crop_double(int N); // crop operation for double data
Imagedata Save_double(string path); // save double data to path
double* write_double(); // write double data to buff
void pure(unsigned char color); // set the whole image to the specific color
void set_color_data(int i, int j, vector<unsigned char> c); //set color data
vector<unsigned char> get_color_data(int i, int j); //get color data
Imagedata RGB2GRAY(); //convert RGB image to gray-level image
Imagedata Binary(int threshold); // convert gray-level image to binary image in 0 and 255
Imagedata Binarize(); //convert range from 0 and 255 to 0 and 1
Imagedata RBinarize(); //convert range from 0 and 1 to 0 and 255
Imagedata label_pixel(vector<pair<int, int>> v); // label some specific pixels
Imagedata DN_Median(int N); // Denoise filter (Median)
void Color_merge(Imagedata R_component, Imagedata G_component, Imagedata B_component); // Combine separable RGB channels
Imagedata get_RGB(int i); //get single channel data
```

```
// Texture analysis
// Texture classification
vector<double> Feature_vector(); // get the 25-D feature vector of a texture
Imagedata Sub_Mean(); // subtract mean to the image
vector<Imagedata> Feature_Extraction(); // get a set of imagedata of the image after laws filter
vector<Imagedata> Feature_Extraction_filter_normalized();   // get a set of imagedata of the image after laws filter
double Average_energy(); // calculate the average energy of the filtered image
```

```
// Texture Segmentation
void segmentation(vector<vector<int>> label); // segment the image according to the label
void Emphasize(pair<int, int> centre,unsigned char s); // label specific connected domain with s graylevel
```

Class Kernel

```
Kernel(); // default constructor
Kernel(int h, int w); //constructor
~Kernel(); //destructor
void set_data(double d, int i, int j); //set elements(i,j) as value d
void set_wholedata(double d);
double sum(); //get sum
vector<vector<double>> get_wholedata(); //return a two dimension vector as kernel
double get_data(int i, int j); //get the value of element(i,j)
int get_height();
int get_width();
Kernel EQ(); //equalization
void Print(); // Print the kernel
void resize(int h, int w); // Resize the kernel
void Mark_Pattern(int a[]); // convert a array to a kernel
void generate(int a[]); // convert a array to kernel
Kernel Multiplication(Kernel other); // this * other (Matrix multiplication
Kernel Transpose(); // Transpose opeartion for matrix
void Vector2Kernel(vector<vector<double>> vector_set); // combine several row vectors to be a kernel
vector<double> Kernel2vector(); // change a kernel to be a vector
vector<vector<double>> Kernel2vector2D(); // change a kernel to be a 2-D vector
Kernel normalize(); // normalize by each row to be a mean = 0 and variance = 1
Kernel getRow(int i); // return row
Kernel getColumn(int i); // return column
```

Class OpencvAlgorithm

In this part, there are four classes of algorithm, SIFT, Texture classification, K means and Bag of words.

```
// SIFT
vector<string> Laws_label(); // return a string vector with Laws filter labels
void SIFT(Mat img_1, Mat img_2, double thres); // The warp for SIFT and Brute Force matching
void SIFT_largestscale(Mat img_1, Mat img_2); // key point with largest scale, this time image with samllest scale
void SIFT_smallestscale(Mat img_1, Mat img_2); // key point with smallest scale, this time image with largest scale
void SIFT_firstoctave(Mat img_1, Mat img_2); // key point in first octave
void SIFT_lastoctave(Mat img_1, Mat img_2); // key point in last octave
void SIFT_largestdistance(Mat img_1, Mat img_2); // the match point with the largest l2 norm distance
void SIFT_feature(Mat image,string path); // show the iamge and save the image to path
```

```
// Texture classification
vector<vector<double>> PCA(cv::Mat pcaSet); // wrapping for PCA
vector<vector<double>> PCA_no_print(cv::Mat pcaSet); // wrapping for PCA without print
cv::Mat Kernel2Mat(Kernel k); // convert kernel to cv::Mat
Kernel Mat2Kernel(cv::Mat mat); // convert cv::Mat to kernel
```

```
// K means
void k_mean_cluster(vector<vector<double>> data); // wrapping for k-means cluster
// Kmean with centroid without print
vector<vector<int>> k_mean_cluster_no_print(vector<vector<double>> data, int cluster_number, int iteration_number);
// Kmean with centroid as return value
vector<vector<double>> k_mean_cluster_centroid(vector<vector<double>> data, int cluster_number, int iteration_number);
// Energy feature vector calculation
vector<vector<double>> Energy_feature_computation(vector<Imagedata> Image, int Window_size);
// Energy feature vector normalized
vector<vector<double>> Energy_feature_normalizationwithLL(vector<vector<double>> Energy_feature);
```

```
// Bag of words
vector<int> hist_cal(Mat test, vector<vector<double>> centroids); // calculate the histogram for test according to ce
Mat BOWKMEANSTRAINER(); // return the centroids of two cluster of bag of words
void BOWHIST_STATISTIC(Kernel Centroids); // Count the score for first feature and second feature for each eight.jpg
```

Struct cluster

```
struct Cluster
{
    vector<double> centroid;
    vector<uint> samples;
};

double cal_distance(vector<double> a, vector<double> b); // calculate the distance of two vectors
vector<Cluster> k_means(vector<vector<double>> trainX, int k, int maxepoches); // return cluster result according to traini
```