159.793

Fingerprint Recognition

Supervisor: Andre L. C. Barczak

By: Bo LIU

Contents

1. Introduction

Fingerprints have been using for over a century. It can be used in forensic science to support criminal investigations, biometric systems such as civilian and commercial identification devices for person identification. It is one of the most significant biometric technologies which have drawn a substantial amount of attention recently [1, 3]. A fingerprint is comprised of ridges and valleys. The ridges are the dark area of the fingerprint and the valleys are the white area that exists between the ridges. The fingerprint of an individual is unique and remains unchanged of over a lifetime. The uniqueness of a fingerprint is exclusively determined by the local ridge characteristics and their relationships [1, 2].

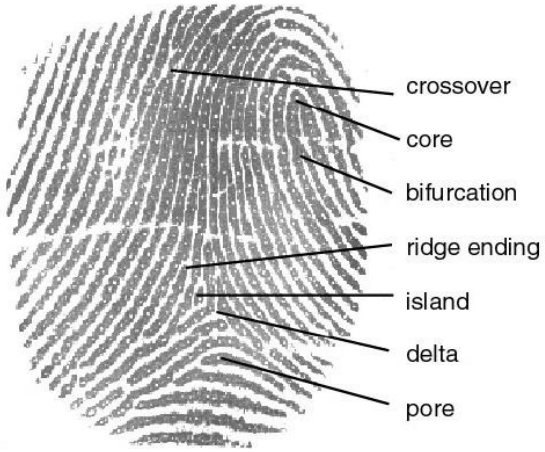

Figure 1: fingerprint structure

Figure 1.0 shows that the structure of a fingerprint, it consists of crossover, core, bifurcation, ridge ending, island, delta and pore. Source: [15]

The Fingerprint Recognition is a process of determining whether two sets of fingerprint ridge detail are from the same person. There are multiple approaches that are used in many different ways for fingerprint recognition which are minutiae, correlation, ridge pattern. These types of approaches can be broadly categorized as minutiae based or texture based.

Minutiae is the most popular approach that is used for fingerprint representation. It is based on local landmarks. The minutiae-based systems locate the points firstly. These points are called minutiae points which represent the fingerprint ridges either terminate or bifurcate in the fingerprint image, and then these minutiae points are matched in a given fingerprint and the stored template. While minutiae points perform fairly high accurate fingerprint matching for minutiae based verification systems [8, 9, 10, 11], they ignore the rich information in the ridge patterns which are used for improving the matching accuracy. In other words, further improvements are needed for acceptable performance, especially when large database involved. In addition, it is difficult to extract minutiae automatically and reliably from poor quality fingerprint, dried fingers or fingers with scars. For texture based approach, it uses the entire fingerprint image around minutiae points [12, 13, 14, 16]. The texture based fingerprint representation is limited due to the collection of local texture based on the minutiae points. Also, it performs depends upon the extraction of minutiae points.

In addition, another algorithm called Hu's Moment Invariant has been widely used in object recognition. Hu's moment invariants are invariant to translation, scaling and

rotation. However, it is not very robust for other transformations [27]. This algorithm was improved by [28] by using the formulas for 4^{th} order and 11 moments. This algorithm will be used to compare with SIFT in our experiment.

The aim of this report is to re-adopt an algorithm for fingerprint verification which is named Scale Invariant Feature Transform (SIFT) [4, 5, 7], and it's used for extending characteristic feature points of fingerprint beyond minutiae points. SIFT approach has been adopting in other object recognition problems. It transforms an image into a collection of local feature vectors and each of these feature vectors is invariant to image translation, scaling and rotation. These features are extracted to perform matching in scale space through a staged filtering approach, and are robust to changes in illumination, noise, occlusion and minor changes in viewpoint. Also, they are highly distinctive and allowed for correct object recognition with low probability of mismatch and are easy to match against database of local features [6].

It is expected that in the domain of fingerprint recognition, this method is also stable and reliable, effective and efficient, and the features points are robust to the fingerprint quality and deformation variation.

2. Literature Review

Scale Invariant Feature Transform (SIFT) is proposed by David G. Lowe [4, 5]. This method extracts distinctive invariant features from images that perform reliable matching between different views of an object or scene. The SIFT features are invariant to image scale and rotation, and are robust for matching across a substantial range of affine distortion, change in viewpoint, addition of noise and change in illumination. There are four major stages of computation that are used for generating the SIFT features – Scale-space extrema detection, Keypoint localization, Orientation assignment and Keypoint descriptor. In addition, the reference image has to be preprocessed before applying the computation of the SIFT features.

1. **Preprocessing:** There are two steps to perform preprocessing, adjusting the graylevel distribution and removing noisy SIFT feature points.

2. **Scale-space Extrema Detection:** It has been described by Koenderink [17] and Lindeberg [18] that the scale space of an image is produced from the convolution of a variable-scale Gaussian with an input image. Therefore, this stage identified interest key points that are invariant to scale and orientation in scale-space by using difference-of Gaussian (DoG) function [5]. Firstly, generating Gaussian pyramid, that is, the input image is firstly applied Gaussian smoothing using $\sigma = \sqrt{2}$ to give an image A. The value of σ is then increased to create the second smoothed image B. The DoG images are generated by subtracting two nearby scales which are separated by a constant multiplicative factor *k*. In other words, the DoG is obtained by

subtracting image B from A. After each octave, the Gaussian image is downsampled by a factor of 2 and the process is repeated until the entire DoG pyramid is built up. The number of scales in each octave is determined by a integer number, s, of intervals, so $k = 2^{1/s}$. Therefore, there are s + 3 images in the stack of blurred images for each octave. After the DoG pyramid has been produced, the local extrema is detected by comparing a pixel to its 26 neighbours in 3x3 region at the current and adjacent scales. These extrema are selected as candidate keypoints which will be filtered in the next stage.

3. **Keypoint Localization:** In this stage, the final keypoints are selected and determined based on the stability of the candidate keypoints. In order to perform a detailed fit to the nearby data for location, scale and ratio of curvatures, the candidate keypoints with low contrast or are poorly localized along an edge will be eliminated.

4. **Orientation Assignment:** Each keypoint location assigns one or more orientations based on local image gradient directions. All future operations are performed on transformed image data relative to the assigned orientation, scale, and location for each feature, thus the invariance to these transformations is provided.

5. **Keypoint Descriptor:** During the previous stages, a stable location, scale and orientation for each keypoint have been detected and determined. This stage measures the local image gradients at selected scale in the region around each keypoint, and computes a descriptor for the local image region that is highly distinctive. The approach proposed by [19] can be used efficiently in the computation of descriptor. This approach can be implemented by using the same pre-computed gradients and orientations for each level of pyramid that were used for orientation selection at stage 4. A weight is assigned to magnitude of each keypoint by using Gaussian weighting function which is to avoid sudden changes in the descriptor with small changes in position of the window. In addition, orientation histograms over 4x4 sample regions should be created for significant shift in gradient positions. Trilinear interpolation should be also applied, and it is used to avoid all boundary affects in which the descriptor abruptly changes and distribute the value of each gradient sample into adjacent histogram bins. Thus, a descriptor consists of a feature vector which contains the values of all the orientation histogram entries (the length of eight arrows for each orientation histogram). Finally, this feature vector is normalized to unit length. For image contrast, it changes each pixel value by multiplying gradients by a constant. Thus, the descriptor is invariant to affine changes in illumination.

6. **Keypoint Matching:** The final stage is to compare the keypoints that we have detected from previous stages. The best matching approach for each keypoint is to identify its nearest neighbor in the database of keypoints. The nearest neighbor is the keypoint with minimum Euclidan distance for the invariant descriptor vector. A more effective way to match the keypoints is to compare the distance of closest neighbor to the second-closest neighbor. The keypoint is said to be matched if the nearest neighbor is less than 0.8 times the distance to the second-nearest neighbor. Note that 0.8 is the ratio of distances, the experiment result from [5] shows that if the ratio of distances is greater than 0.8, 90% of the false matches will be eliminated while discarding less than 5% of the correct matches.

3. Methodology

3.1 Preprocessing

This stage is to initialize the original input image. In order to obtain better matching performance, the input image has to be processed in two steps: i) converting to grayscale, and ii) apply Gaussian-smoothing. In this report, we convert the input image into 8-bit grayscale, and then convert the scale from 8-bit to 32-bit with single precision floating-point numbers. In other words, all pixel values of the input image will be 32-bit floating point numbers, as it's easy to compare and detect the keypoints which will be done in the next stage. Finally, we apply Gaussian smoothing to the 32-bit image in order to reduce noise.

3.2 Scale – space Extrema Detection

This stage is to detect keypoints by using a cascade filtering approach which adopts efficient algorithms to identify candidate keypoint locations that are determined and examined in further stage. The keypoints are detected by identifying the locations and scales that is repeatedly assigned under different views of the same object. The purpose of detecting locations is to be invariant to scale change of the image. It is done by searching stable features across all possible scales, and this is known as scale space.

3.2.1 Gaussian Pyramid

It has been proposed by [16, 17] that the scale-space kernel is the Gaussian function. Therefore, the scale space of image can be defined as a function, $L(x, y, \sigma)$. It is generated from the convolution of a variable-scale Gaussian, $G(x, y, \sigma)$ with the preprocessed image, $I(x, y)$:

$$L(x, y, \sigma) = G(x, y, \sigma) * I(x, y) \quad (3.1)$$

where * is the convolution operation in x and y, and

$$G(x, y, \sigma) = 1/2\pi\sigma^2 * exp^{-(x2 + y2) / 2\sigma2} \quad (3.2)$$

Thus, $L(x, y, \sigma)$ is the first scale in the first octave of the Gaussian pyramid, to generate the next scale in the same octave. The σ is increased by a factor of k, which generates the next scale in the octave, $L(x, y, k\sigma)$. Once the complete octave has been processed, we down-sample the Gaussian image by a factor of 2, and this process is repeated applied across the entire of the pyramid which is built up by a number of octaves. The number of scales in each octave is determined by a constant number, s, of intervals, so $k = 2^{1/s}$ and the number of smoothed images (scales) for each octave is $s + 3$. Overall, the convolved images are grouped by octave which corresponds to doubling the value of σ, and the value of k is used to obtain a fixed number of smoothed images per octave and the number of DoG images per octave. The Gaussian pyramid is shown on the left of Figure 1.

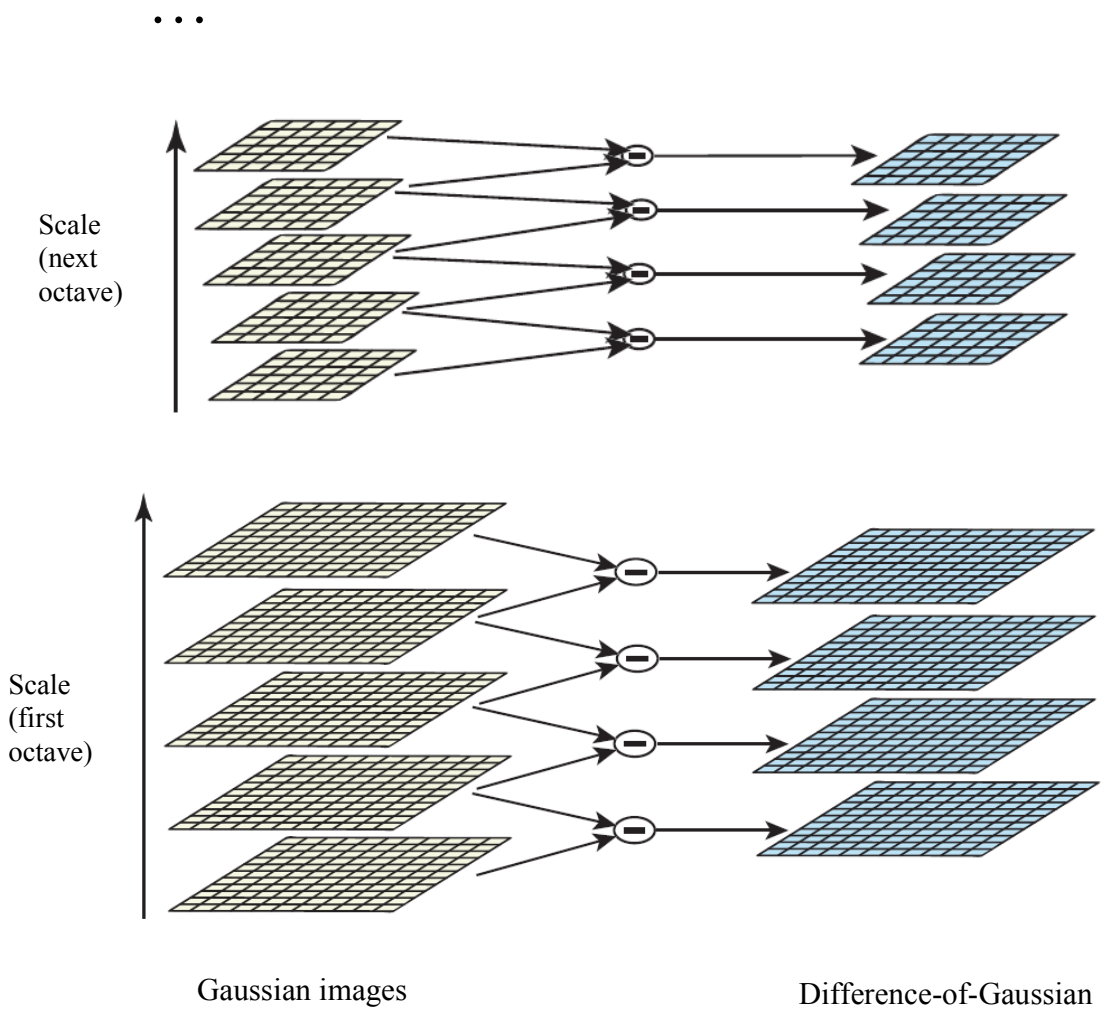

Figure 2: Gaussian (left) and Difference-of-Gaussian (right) pyramid. Gaussian images are generated by repeatedly convolved the initial image for each scale space in each octave. The DoG images on the right are produced by subtracting adjacent Gaussian images. Once the first octave has been processed, the Gaussian image is down-sampled by a factor of 2, and process is continues until all octaves are processed. Source: [4].

3.2.2 Difference-of-Gaussian Pyramid

It has been proposed by [5] that the efficient way to detect keypoint locations in scale space is to use scale-space extrema in the difference-of-Gaussian function convolved with image, $D(x, y, \sigma)$, which is shown on the right of Figure 1 is computed from the difference of adjacent scales which is separated by a constant factor k:

$$D(x, y, \sigma) = (G(x, y, k\sigma) - G(x, y, \sigma)) * I(x, y)$$

$$= L(x, y, k\sigma) - L(x, y, \sigma) \qquad (3.3)$$

This function is chosen because it is efficiently computed, as the smoothed images, L, which needs to be computed for scale space feature description, and therefore, D can be computed by simple image subtraction. Also, the difference-of-Gaussian function provides a close approximation to the scale-normalized Laplacian of Gaussian [18]. It is shown that the normalization of the Laplacian with the factor σ^2 is required for true scale invariance. In addition, the experimental comparison that was accompanied by [20] shows that the most stable image features are detected by the maxima and minima, compared to a range of other possible image functions such as Hessian, Harris corner function. Furthermore, for the constant factor k, there is almost no impact on the stability of extrema detection or localization for significant differences in scales, such as $k = 2^{1/2}$.

3.2.3 Local Extrema Detection

Once the DoG images have been produced across the entire pyramid, the local maxima and minima can be detected based on these DoG images. For each DoG image in each octave, it compares each sample point to its eight neighbours in the current image and nine neighbours in the scale above and below (Figure 2). The sample point is only selected if its pixel value is less than or larger than all of its neighbours pixel value. Another thing has to be stated is that in each octave, the top and bottom DoG images are only used for comparing. In other words, the middle DoG images are used for keypoint selection. This process has to be applied across the entire pyramid until all extrema are selected as candidate keypoints.

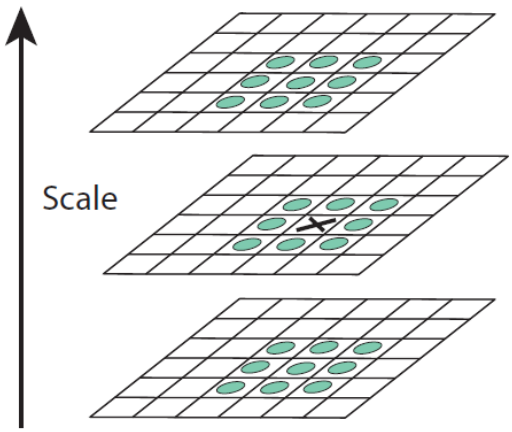

Figure 3: local extrema of DoG image are detected by comparing a pixel value (marked X) with its 26 neightbours (marked circle) in 3x3 regions from current image and adjacent scales. Source: [4].

3.3 Keypoint Localization

This stage is to remove all unreliable keypoints. After all candidate keypoints has been selected during the previous stage, this stage performs a detailed fit to the nearby data for location, scale, and ratio of principal curvatures, that is, for each candidate keypoint, it will be eliminated if it has low contrast or is poorly localized along an edge.

3.3.1 Low-contrast Keypoints Elimination

A method that was developed by [21] used for fitting 3D quadratic function to the local sample points to determine the interpolated location of the maximum. The experiment result from [21] has shown that this method provides a significant improvement to matching and stability, and uses the Taylor expansion up to quadratic terms of the scale-space function, D (x, y, σ), shifted to the sample point:

$$D(\mathrm{x}) = D + \frac{\partial D^T}{\partial \mathrm{x}}\mathrm{x} + \frac{1}{2}\mathrm{x}^T\frac{\partial^2 D}{\partial \mathrm{x}^2}\mathrm{x} \tag{3.4}$$

Where D and its derivatives are evaluated at the sample point and x = $(x, y, \sigma)^T$ is the offset from this point. The location of the extremum, $\hat{\mathrm{x}}$, is determined by taking the derivative of this function with x and setting it to zero,

$$\hat{\mathrm{X}} = -\frac{\partial^2 D}{\partial \mathrm{x}^2}^{-1}\frac{\partial D}{\partial \mathrm{x}} \tag{3.5}$$

The Hessian and derivative of D are approximated by using difference of neighbouring sample points. In this case, if $\hat{x}$ is greater than 0.5 that means the extremum lies closer to a different sample point, and the sample point is modified and the interpolation performed instead that point. $\hat{x}$ is finally added to the location of its sample point to get the interpolated estimate for the location of the extremum.

However, in order to reject the unstable extrema with low contrast, the function value at extremum, $D(\hat{x})$ is applied and obtained by subtracting equation (3.5) from equation (3.4) :

$$D(\hat{x}) = D + \frac{1}{2}\frac{\partial D^T}{\partial x}\hat{x} \qquad (3.6)$$

$|D(\hat{x})|$ can be considered as a threshold to reject the sample points with low contrast. The experiment result from [5] shows that the all extrema with a value of $|D(\hat{x})|$ less than 0.04 are discarded. This value is also adopted in other SIFT – based object recognition problem.

3.3.2 Eliminating Edge Responses

The principal curvatures that are used to measure the poorly defined peak in DoG function can be computed from a 2x2 Hessian matrix, H, computed at the location and scale of the keypoint:

$$H = \begin{bmatrix} D_{xx} & D_{xy} \\ D_{xy} & D_{yy} \end{bmatrix} \tag{3.7}$$

The derivatives are calculated based on the difference of neighbouring sample points.

The eigenvalues of H are proportional to the principal curvatures of D. An approach was proposed by [22] stated that the eigenvalues computation could be avoided, and their ratio should focused. Thus, let α be the eigenvalue with the largest magnitude, and β be the smaller one. We compute the sum of the eigenvalues from the trace of H and their product from the determinant:

$$Tr(H) = D_{xx} + D_{yy} = \alpha + \beta,$$

$$Det(H) = D_{xx}\, D_{yy} - (D_{xy})^2 = \alpha\, \beta \tag{3.8}$$

Let *r* be the ratio between the largest magnitude eigenvalue and the smaller one, so that $\alpha = r\beta$. Therefore:

$$\frac{Tr(H)^2}{Det(H)} = \frac{(\alpha+\beta)^2}{\alpha\beta} = \frac{(r\beta+\beta)^2}{r\beta^2} = \frac{(r+1)^2}{r} \tag{3.9}$$

This only depends on the ratio of the eigenvalues. If the two eigenvalues are equal and it increases with r, the quantity $(r+1)^2/r$ is at a minimum. Thus, the ratio of principal curvatures is estimated with a threshold, *r*, given:

$$\frac{Tr(H)^2}{Det(H)} < \frac{(r+1)^2}{r} \tag{3.10}$$

This is used to estimate whether the location of sample point along the edge is poorly determined. From the experiment result [5], set $r = 10$, which eliminates keypoints that have a ratio between the principal curvatures greater than 10.

3.4 Orientation Assignment

In order to achieve invariant to image rotation, each keypoint is assigned a consistent orientation based on local image properties. This approach contrasts with the approach from [23], in which each image property is based on a rotationally invariant measure. Therefore, this approach limits the descriptors that can be used and discards image information.

For the most stable results, the scale of the keypoint is used to select the Gaussian smoothed image, *L*, with the closet scale, thus all computations are performed in a scale-invariant manner. For each image sample point, *L(x, y)*, at this scale, the gradient magnitude, *m(x, y)*, and orientation, $\theta(x, y)$, is pre-computed using pixel difference:

$$m(x,y) = \sqrt{(L(x+1,y) - L(x-1,y))^2 + (L(x,y+1) - L(x,y-1))^2}$$

$$\theta(x,y) = tan^{-1}\left((L(x,y+1) - L(x,y-1)) / (L(x+1,y) - L(x-1,y))\right) \quad (3.11)$$

An orientation histogram is formed from the gradient orientations of sample points within a region around the keypoint. There are 36 bins covering 360 degree range of orientation in the orientation histogram. Each sample point that is added to the histogram is weighted by its gradient magnitude and a Gaussian-weighted circular window with a σ that is 1.5 times that of the scale of the keypoint.

Peaks in the orientation histogram correspond to dominant directions of local gradients. In the orientation histogram, we use the highest peak and any other local peak within 80% of the highest peak to create a keypoint with that orientation. However, multiple keypoints will be assigned with different orientations at the same location and scale if there are multiple peaks of similar magnitude at keypoint locations. A Gaussian distribution is fit to the 3 histogram values closet to each peak to interpolate the peaks position for better accuracy. The stability of location, scale and orientation assignment is shown in Figure 4.

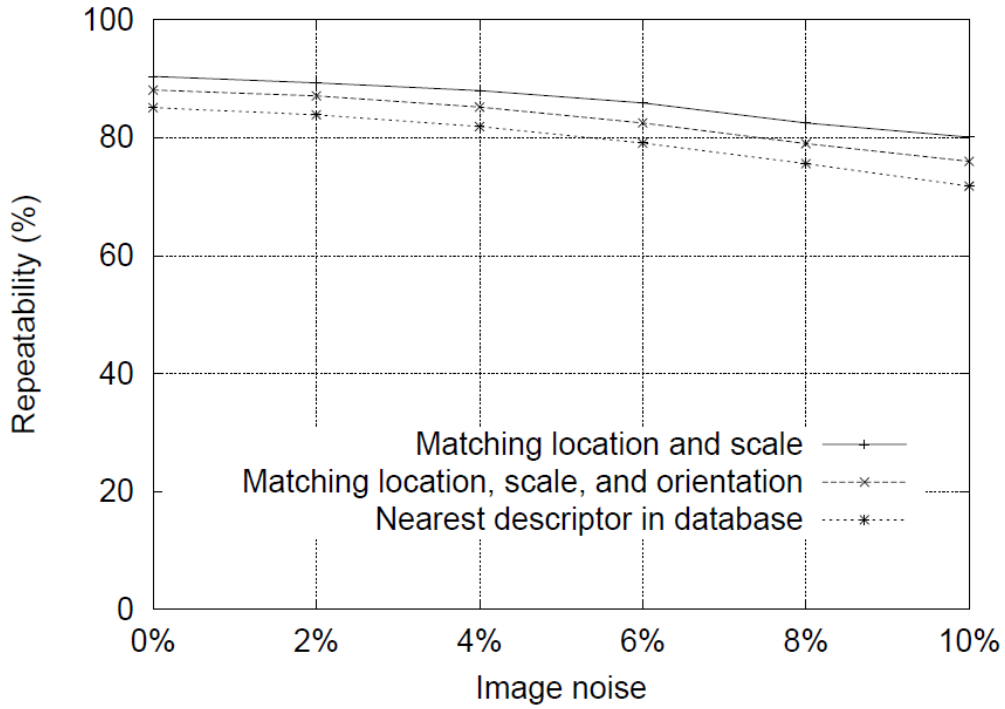

Figure 4: shows the stability of location, scale, and orientation assignment under differing amounts of image noise. The top line shows the percentage of stability of keypoint location and scale assignment that are detected as a function of pixel noise. The second shows the responsibility and stability of matching after orientation assignment is applied. The bottom line

shows the finial percent of accuracy of correctly matching a keypoint in a large database. Source: [4].

3.5 Keypoint Descriptor

An image location, scale and orientation to each keypoint have been assigned during the previous operations. In this stage, a descriptor is computed for the local image region that is highly distinctive for each keypoint.

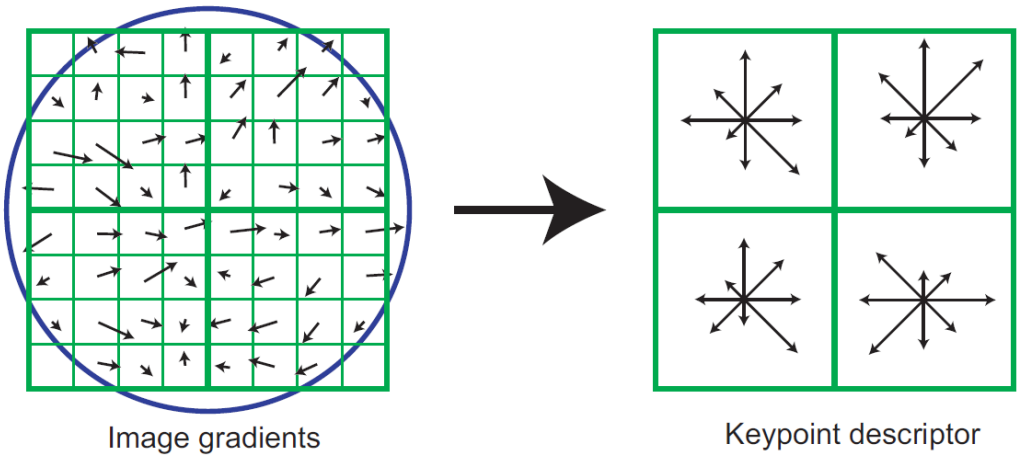

Figure 5: A keypoint descriptor is produced by computing the gradient magnitude and orientation at each sample point in a region around the keypoint location (shown on the left). The overlaid circle is a Gaussian window. The keypoint descriptor is shown on the right side. Source: [4].

In order to achieve orientation invariance, the coordinates of descriptor and the gradient orientations are rotated relative to the keypoint orientation, and these gradients are performed during the previous stage. The image gradient magnitudes and orientations are sampled around keypoint location. These are illustrated with small arrows at each sample location on the left side of Figure 5. A Gaussian weighting function with σ related to the scale of the keypoint, that is, one half the width of the descriptor window is used for assigning a weight to the magnitude of each sample point. A circular window on the left side of Figure 5 shows the Gaussian window. There are two main aims for Gaussian window. One is to avoid sudden changes in the descriptor with small changes in the position of the window. Another purpose is for the gradients that are far from the centre of the descriptor.

The right side of Figure 5 shows the keypoint descriptor. The orientation histogram is created over 4x4 sample region (shown on the left of the Figure 5), as it allows vital shift in gradient position. There are eight directions for each orientation histogram. The length of each arrow corresponds to the magnitude of that histogram entry. The same histogram will be produced if a sample gradient is shifted less than 4 sample positions.

There is another important thing has to be considered is about the boundary affects in which the descriptor changes, as a sample point shifts smoothly from one histogram to another or from one orientation to another. Thus, tri-linear interpolation can be used for distributing the value of each gradient sample point into adjacent histogram bins. That

means each entry of a bin is multiplied by a weight of 1 – d for each dimension, where d is the distance of the sample from the central value of the bin.

Overall, the descriptor constitutes from a vector which contains the values of all the orientation histogram entries, which corresponds to the lengths of the arrows (shown on the right side of Figure 5). The Figure 5 shows a 2x2 array of orientation histograms. In this report, we use 4x4 array of histograms, each of which contains 8 orientation bins. Therefore, there are 4x4x8 = 128 element feature vector for each keypoint. Finally, we have to normalize the feature vector to unit length in order to reduce the effects of illumination change. If an image contrast has changed, each pixel value in the image will be multiplied by a constant and will multiply gradients by the same constant, so the contrast will be canceled by vector normalization. The brightness change will not affect the gradient values, as they are compute from pixel differences. So, the descriptor is invariant to affine changes in illumination.

3.6 Object Recognition

From previous stages, we have detected keypoint location, orientation and descriptor for each scale and octave. All of the geometric information around each keypoint is what we need for object recognition. This stage will discuss the algorithm for keypiont matching and SIFT feature indexing.

3.6.1 Keypoint Matching

The best way to match each keypoint is to identify its nearest neighbor in the keypoints database. The nearest neighbour is defined as the keypoint with minimum Euclidean distance for the invariant descriptor vector. However, due to the features arise from background clutter, there are many features from an image will not have correct match. Therefore, we have to discard some features that do not have any good match to the database. A better measurement is adopted by comparing the distance of the closest neighbour to that of the second-closest neighbour. In order to achieve reliable matching, correct matches need to have the closest neighbour significantly closer than the closest incorrect match which supports the performance of this measurement. The probability of matching is determined by the ratio of distance from the closest neighbour to the distance of the second closet. The ratio of distance is selected between 0.0 and 1.0. For example, two images, I_1 and I_2, need to be matched. Given a feature point p_1 in I_1, its closest point p_2, second closest point p_3 and their distances d_1 and d_2 are calculated in I_2, if the d1 is less than d2 times the ratio of distance, p_1 is considered to match with p_2.

3.6.2 Nearest Neighbour Indexing

The keypoint descriptor that we computed from previous stage has 128-dimentional feature vector (4x4 array x 8 orientations), so that in order to search nearest neighbours of points in high dimensional spaces, a suitable algorithm that is called the Best-Bin-First (BBF) algorithm [24] can be adopted.

BBF algorithm is designed, and used for efficiently searching and finding an approximate solution to the nearest neighbour search problem in high dimensional spaces. The BBF algorithm is based on the k-d tree search algorithm which makes indexing higher dimensional spaces possible so that bins in feature space are searched in the order of their closest distance from the query location. Thus, the BBF algorithm is an approximate algorithm which returns the nearest neighbour for a large fraction of queries and a very close neighbour [24].

The performance of matching is determined by the number of nearest-neighbour candidates, especially for a large database of keypoints. In order words, we have to stop searching after the first *N* nearest-neighbour candidates have been checked. This provides an efficient speedup over exact nearest neighbour, and only results in a small percent of loss in the number of correct matches. In the implementation, we select 0.8 as the ratio of distance. That means the keypoint is matched if its nearest neighbour is less than 0.8 times the distance to the second-neighbour. This is the main reason that the BBF algorithm works especially fine for this problem.

4. Experiment Results and Discussion

4.1 Fingerprint Database

The performance of the SIFT based fingerprint recognition was evaluated on FVC2002 DB1a [26]. The database contains images of 100 different fingers with 8 versions for each finger – totally 800 images. The fingerprint image is named with the format XX_YY.tif. XX and YY represent the person ID and the fingerprint impression respectively.

Table 1 Description of FVC 2002 DB1a

	Sensor Type	Image Size	Number of images	Resolution
DB1a	Optical Sensor	388x374(142K pixels)	100x8	500 dpi

Figure 6: fingerprint image 1_1.tif which will apply SIFT algorithm. Source: [26].

4.2 Scale-space Extrema Detection

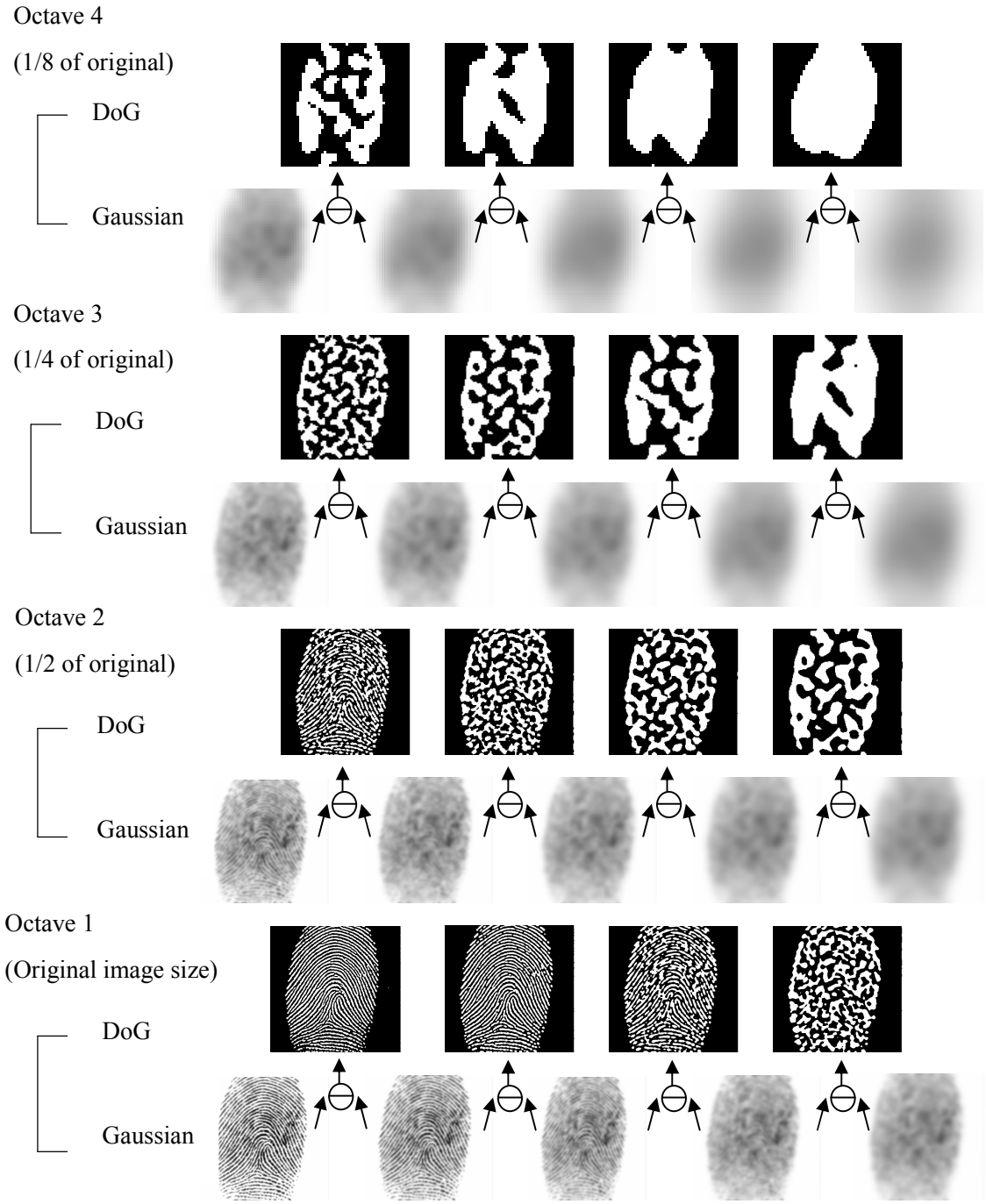

Figure 7: Gaussian and DoG pyramid

Figure 7 shows the Gaussian pyramid and DoG pyramid which is created by subtracting two adjacent Gaussian images. We used the number of octaves 4 (a constant s = 2, of intervals + 2), number of scales 5 (s+3), and the standard deviation of Gaussian kernel 1.8. The constant k in Equation (3.3) is set to $2^{1/intervals}$, in this case, k = $\sqrt{2}$.

After the Gaussian and DoG pyramid has been created, the next step is to find the local minimum and maximum – candidate keypoints selection which is shown on Figure 8.

Figure 8: 198 candidate keypoints

Figure 8 shows that 198 keyponts are selected based on comparing the 26 neighbours of a given sample point at current, above and below scale in DoG pyramid. These candidate keypoints will be detected and might be eliminated in the next stage.

4.3 Keypoints Localization

Based on the candidate keypoints we have been detected from previous stage, this stage is to eliminate the candidate keypoints which have low contrast and poorly localized along an edge.

There are two thresholding values for eliminating the keyponts with low contrast. The first one (0.5) is used to determine whether the offset $\hat{x}$ (Equation 3.5) lies closer to a different sample point. If $\hat{x}$ is larger than 0.5 in any dimension, that means the extremum located closer to a different sample point. Therefore, the sample point is modified and the interpolation performed instead about that point, Then the offset $\hat{x}$ is added to the location of its sample point to get the interpolated estimate for the location of the extremum. To consider a keypoint have low contrast, simply compares Equation (3.6) with another threshold value 0.04. All extrema with value from Equation (3.6) is less than 0.04 will be discarded. (The threshold is 0.03 in Lowe's paper)

Figure 9: 163 remaining keypoints after the removal of the candidate keyponts with low contrast.

For eliminating a keypoints that are poorly located along an edge another constant value called curvature threshold is used as an input *r* in Equation (3.10). If the Equation (3.10) is false, we can say that the keypoint is located along an edge.

Figure 10: 99 keypoints left after the stage 2 finished.

Figure 9 and 10 shows that the effects of keypoints removals based on the candidate keyponts which were selected from previous stage. The Figure 9 shows the 163 keyponts that remain following removal of those with a value of Equation 3.6 is less than 0.04. The transition from Figure 9 to 10 shows the effects of Equation (3.10).

4.4 Orientation Assignment

Figure 11: 188 SIFT feature orientation.

Figure 11 shows that the gradient and magnitude within a region around the keypoint. This is done based on the Equation (3.11). However, we have to find the peaks in the orientation histogram, and it corresponds to dominant directions of local gradients. Therefore, the highest peak in the orientation is determined, and any other local peak that is within 0.8 of the highest peak is adopted to create a keypoint with that orientation. In other words, there are multiple keypoints created at the same location and scale but different orientations for those keypoint locations with multiple peaks of similar magnitude.

4.5 Keypoint Descriptor

The descriptor vector contains all orientation histogram entries. The experiment of this paper is used 4x4 array of orientation histograms rather than 2x2 (8x8 samples) which was shown in Figure 5. A 4x4 array means 4x4 descriptors computed from 16x16 sample array. Therefore, the descriptor vector contains 16x16 array of histograms with 8 orientation bins in each, the total element in descriptor is 16x16x8 = 2048 which is used to generate a histogram of gradient orientation around each local extremum. This descriptor vector contains invariant descriptors which are used later for keypoints matching.

Figure 12: 28 keypoints matched between 1_1.tif and 1_2.tif. 1_1 has 99 features, and 1_2 has 94.

4.6 Keypoint Matching

(a) 1_1.tif (b) 1_2.tif

Figure 13: SIFT features between matched fingerprint images.

The Figure 12 shows the matching points between fingerprint 1_1 and 1_2. There were 28 keypoints matched. The Figure12 (a) has 99 keypoints and (b) has 94. 99 keypoints from Figure 12 (a) matched 28 keypionts in Figure 12 (b). The number of keypoints matched ignores the same location having more than one feature (Orientation Assignment).

Note that the fingerprint 1_1 and 1_2 are from same person but different ridge structure.

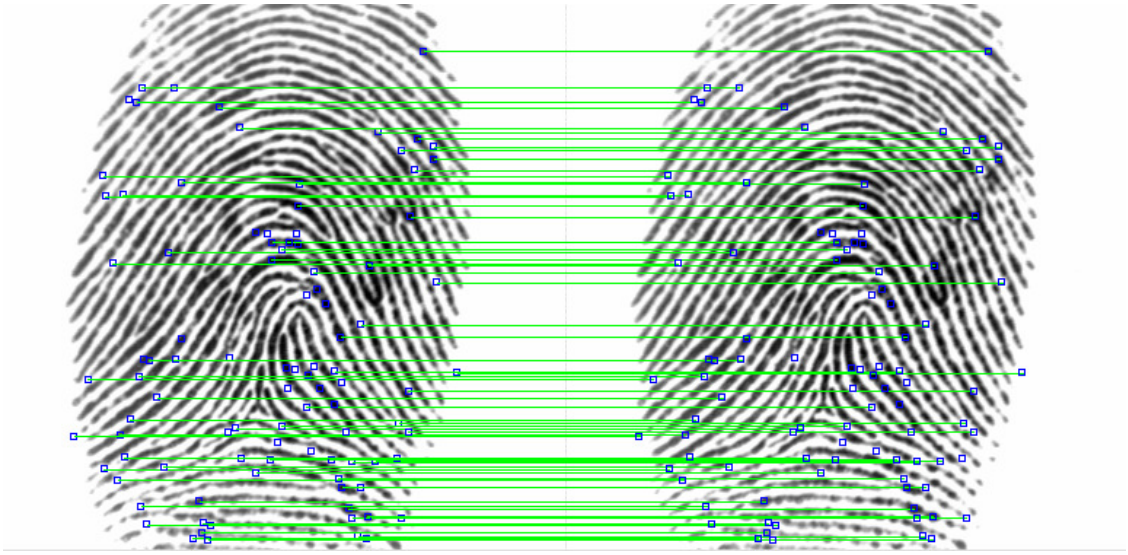

Figure 14: Fingerprint matched between same images. 72/99 keypionts matched on 1_1.tif

4.7 Searching In a Database

When searching a fingerprint in a database, the threshold used to check whether a fingerprint is matched is quite hard to be determined, especially when we use a fingerprint to match another one with different version (e.g x_1.tif, x_5.tif). From Figure 12 and Figure 14, we can easily see that the different matching score for exactly two same fingers, and same fingers but different finger version. This is because of the number of keypoints we detected. In other words, the number of keypoints directly affects the final matching score. In order to increase the number of keypoints we detect during the stage 1, the fingerprint quality plays a major role, which means the fingerprint enhancement should be applied before applying SIFT algorithm. Furthermore,

thresholding values used across the whole algorithm such as the low contrast and curvature threshold also influences the final matching, as these values are used for detecting and creating SIFT features. Therefore, we have to adjust the thresholds based on the quality of the fingerprint image, especially when we deal with a number of fingerprint images, each of which has different quality, and ensure the fingerprint images have similar quality which could be done by image enhancement.

For Moment Invariant [28], firstly we have to calculate all 11 moments up to 4^{th} order for original and matched fingerprint image, and then we have to find the sum of the difference between each moment. Finally, we square root the sum. This is the distance between two images. The smaller the distance, the images are said to be more similar. In our experiment, we used a threshold 0.000015 to determine whether two fingerprints were matched. However, this threshold is different based on different databases, as the quality of fingerprint image might be different.

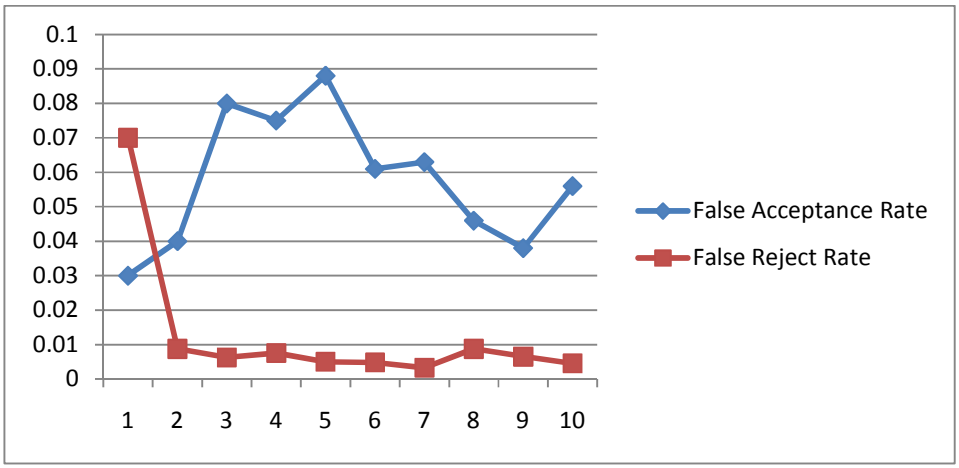

Figure15: False acceptance and Reject Rate based on 10 samples for SIFT.

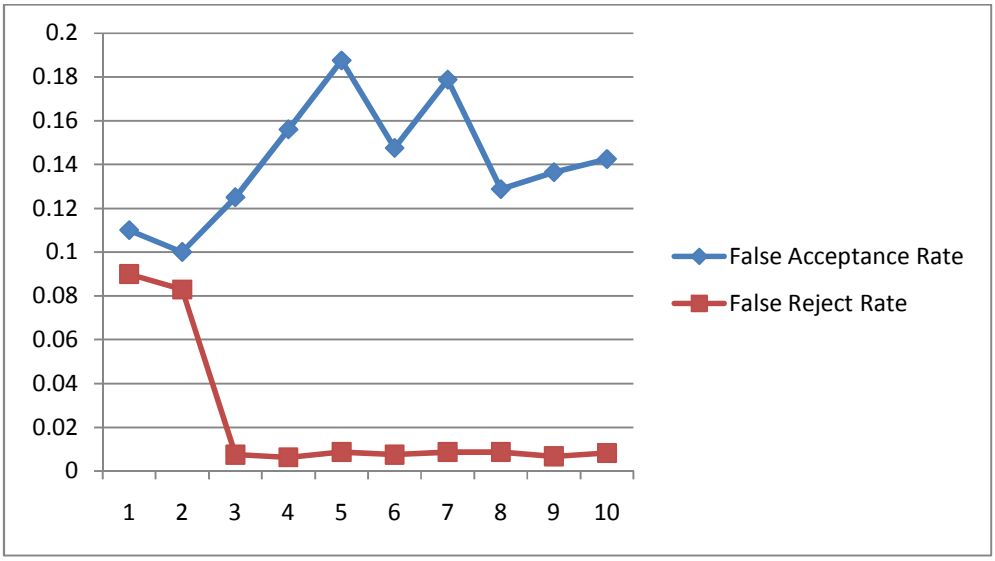

Figure 16: False acceptance and Rejection Rate based on 10 samples for Moment Invariant.

The Figure 15 and 16 show the percentage of False Acceptance and Rejection Rate for SIFT and Moment Invariant respectively based on 10 sample fingerprint images which are selected from the database with different impression and quality. These sample images are matched to the database without applying any image enhancement approach. The Average time for SIFT and Moment Invariant are 1749.4 and 116.23 seconds. We can see that SIFT approach performs more accuracy than Moment Invariant. However, Moment Invariant [28] is 15 times faster than SIFT approach. When Moment Invariant algorithm applies, the initial image is not binarized and thinned, because if the thinning algorithm is not optimized, the process of Moment Invariant algorithm will take longer and finish with less accurate. Also, the quality of thinning fingerprint is totally based on the quality of original image. This is also shown by [29]. Therefore, SIFT and Moment Invariant approaches have a common problem which is the fingerprint preprocessing in terms of image enhancement. These two algorithms require the initial image with a high quality in order to achieve high matching accuracy. SIFT approach is not the one which could be used to

match a fingerprint from a large database in real time, especially if we apply image enhancement before applying SIFT, it will takes even longer to complete the matching even though it performs more accurate matching.

5. Conclusion

The SIFT features discussed in this paper are widely used for any object recognition such as face and fingerprint object recognition, and they are invariant to image scaling, rotation, addition of noise. They are quite useful due to their highly distinctiveness which enables the correct match for keypoints among fingerprints. These keypoints are extracted by crating Gaussian and DoG pyramid. The keypoints with low contrast and are poorly located along an edge will then be removed. The distinctiveness is accomplished by using a high dimensional vector which represents the image gradients within a local region of the image.

When using SIFT in fingerprint recognition, the number of keypoints extracted based on the quality of a fingerprint image. Therefore, the image enhancement is a key step in preprocessing stage, as it can improve the fingerprint ridge structure and remove all noise around the boundary region which is different for every fingerprint impression even for the same finger. The preprocess is really significant for SIFT-based object recognition, as it influences the keypoints detection which has direct association with the final matching result because the gradient and magnitude are calculated based on the keypoint

location, particularly when we compare same fingerprint with different impression. Thus, The solid SIFT features we detect, the more accurate matching the SIFT approach will perform. In real time fingerprint recognition, SIFT algorithm should be applied with other approach according to the huge computation, which means it is better to use other approach such as Moment Invariant to target all matched fingerprints from database, then applying SIFT algorithm to filter these matched fingerprints, and targeting the matched fingerprint by using SIFT eventually.

References

1. H. C. Lee and R. E. Gaensslen. *Advances in Fingerprint Technology*. Elsevier, New York, 1991.
2. A. Moenssens. *Fingerprint Techniques*. Chilton Book Company, London, 1971.
3. E. Newham. *The Biometric Report*. SJB Services, New York, 1995.
4. D. Lowe, "Object Recognition from Local Scale-Invariant Features," *International Conference on Computer Version*. September, 2004.
5. D. Lowe, "Distinctive image features from scale-invariant key points," International Journal of Computer Vision, 60(2), 91-110, 1999.
6. Bicego, M.; Lagorio, A.; Grosso, E. & Tistarelli, M. (2006). On the Use of SIFT Features for Face Authentication, Proceedings of CVPRW06, pp. 35, 0-7695 – 2646-2, IEEE Computer Society, New York, NY.
7. Park, U.; Pankanti, S. & Jain, A.K. (2008). Fingerprint Verification Using SIFT Features. *Proceedings of SPIE Defense and Security Symposium*, 0277-786X, SPIE, Orlando, Florida.
8. F. Pernus, S. Kovacic, and L. Gyergyek, Minutiae-based fingerprint recognition, *Proceedings of the Fifth international Conference on Pattern Recognition*, 1380-1382, 1980.
9. A. K. Jain, S. Prabhakar, and S. Chen, Combining multiple Matchers for a High Security Fingerprint Verification System, *Pattern Recognition Letters*, 20(11-13), 1371-1379, 1999.
10. A.K.Jain, L. Hong, and R. Bolle, On-line fingerprint verification, IEEE *Transactions on Pattern Analysis and Machine Intelligence*, vol. 19, 302-314, 1997.

11. N. K. Ratha, K. Karu, S. Chen, and A. K. Jain, A Real-Time Matching System for Large Fingerprint Databases, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 18(8), 799-813, 1996.

12. D. Roberge, C. Soutar, and B. V. Kumar, High-Speed fingerprint verification using an optical correlator, in Proceedings SPIE, vol. 3386, 242-252, 1998.

13. S. Chikkerur, S. Pankanti, A. Jea, N. Ratha, and R. Bolle, Fingerprint Representation Using Localized Texture Features, *International Conference on Pattern Recognition*, 521-524, 2006.

14. A. J. Willis and L. Myers, A Cost-Effective Fingerprint Recognition System for Use with Low-Quality prints and Damaged Fingertips, Pattern Recognition, 34(2), 255-270,2001.

15. Biometric Education/Fingerprints, Rosistem Romania, http://www.barcode.ro/tutorials/biometrics/fingerprint.html, accessed 02 December 2005

16. A. K. Jain, S. Prabhakar, L. Hong, and S. Pankanti, Filterbank-based Fingerprint Matching, IEEE Transactions on Image Processing, 9(5), 846-859, 2000.

17. J.J. Koenderink, the structure of images. Biological Cybernetics, 50: 363-396. 1984.

18. T. Lindeberg, Scale-space theory: A basic tool for analyzing structures at different scales. Journal of Applied Statistics, 21(2): 224-270. 1994.

19. Edelman, Shimon, Nathan Intrator, and Tomaso Poggio, "Complex cells and object recognition," Unpublished Manuscript.

20. K. Mikolajczyk, *Detection of local features invariant to affine transformations*, Ph.D. thesis, Institute National Polytechnique de Grenoble, France. 2002.

21. M. Brown, and D.G. Lowe, Invariant features from interest point groups. In British Machine Vision Conference, Cardiff, Wales, pp. 656-665.

22. C. Harris, and M. Stephens. A combined corner and edge detector. In Fourth Alvey Vision Conference, Manchester, UK, pp. 147-151. 1988.

23. C. Schmid, and R. Mohr. Local grayvalue invariants for image retrieval. IEEE Trans. on Pattern Analysis and Machine Intelligence, 19(5): 530-534. 1997.

24. J. Beis, and D.G. Lowe, Shape indexing using approximate nearest-neighbor search in high-dimensional spaces. In conference on Computer Vision and Pattern Recognition, Puerto Rico, pp.1000-1006. 1997.

25. http://en.wikipedia.org/wiki/Best_Bin_First

26. D. Maio, D. Maltoni, J. L. Wayman, and A. K. Jain, FVC2002: Second Fingerprint Verification competition, *International Conference on Pattern Recognition*, 811-814, 2002.

27. A. Barczak (2009). Machine Vision Study Guide. Institute of information and Mathematical sciences Massey University. 67-70.

28. A. Barczak, Feature-based Rapid Object Detection: From Feature Extraction to Parallisation. PhD thesis, Institute of Information and Mathematical Sciences, Auckland, New Zealand, November 2007.

29. R. Thai. "Fingerprint Image Enhancement and Minutiae Extraction, " 2003.

Appendix

Appendix A – kdmatching.c

```
/** To compile this program, using the following command
g++ kpmatching.c `pkg-config opencv --libs --cflags gtk+-2.0`-o kpmatching
There are two main functions, one for matching two given fingerprint images
and another one for database matching, use only one at a time.
For executing the program
./programCompiledFileName initialImage imageToBeMatched
OR
./programCompiledFileName initialImage imageDatabaseDirectory
*/

#include <stdio.h>
//#include "sift.c"
#include "kpstruct.c"
#include "cv.h"
#include "cxcore.h"
#include "highgui.h"
#include <iostream>
#include <time.h>
#include <vector>
#include <gdk/gdk.h>
#include <gtk/gtk.h>
#include <sys/types.h>
#include <dirent.h>
#include <string>
```

```
#include <errno.h>
#include <sstream>

using namespace std;

// used in searching in database
//if number of keyponts matched large than this value
//program asks user to stop or continue.
#define KEYPOINT_MATCHED_THRESHOLD 25
struct coord{
  int x;
  int y;
};
clock_t start, end;

IplImage* merge_imgs( IplImage* img1, IplImage* img2 ){
  IplImage* merge = cvCreateImage( cvSize( (img1->width + img2->width),
                                           MAX(img1->height, img2->height)),
                                   IPL_DEPTH_8U, 3 );
  cvZero( merge );
  cvSetImageROI( merge, cvRect( 0, 0, img1->width, img1->height ) );
  cvAdd( img1, merge, merge, NULL );
  cvSetImageROI( merge, cvRect(img1->width, 0, img2->width, img2->height) );
  cvAdd( img2, merge, merge, NULL );
  cvResetImageROI( merge );

  return merge;
}
```

```
void display_merged_img( IplImage* img, char* title )
{
  IplImage* small;
  GdkScreen* scr;
  int scr_width, scr_height;
  double img_aspect, scr_aspect, scale;

  /* determine screen size to see if image fits on screen */
  gdk_init( NULL, NULL );
  scr = gdk_screen_get_default();
  scr_width = gdk_screen_get_width( scr );
  scr_height = gdk_screen_get_height( scr );

  if( img->width >= 0.90 * scr_width  ||  img->height >= 0.90 * scr_height )
    {
     img_aspect = (double)(img->width) / img->height;
     scr_aspect = (double)(scr_width) / scr_height;

     if( img_aspect > scr_aspect )
           scale = 0.90 * scr_width / img->width;
     else
           scale = 0.90 * scr_height / img->height;

     small = cvCreateImage( cvSize( img->width * scale, img->height * scale ),
                                    img->depth, img->nChannels );
     cvResize( img, small, CV_INTER_AREA );
    }
  else
```

```
    small = cvCloneImage( img );

  cvNamedWindow( title, 1 );
  cvShowImage( title, small );
  cvReleaseImage( &small );
}
int getdir (string dir, vector<string> &files){
    DIR *dp;
    struct dirent *dirp;
    if((dp  = opendir(dir.c_str())) == NULL) {
        cout << "Failed to (" << errno << ") open " << dir << endl;
        return errno;
    }

    while ((dirp = readdir(dp)) != NULL) {
        files.push_back(string(dirp->d_name));
    }
    closedir(dp);
    return 0;
}

void draw_sift_feature( IplImage* img, struct feature* feat, CvScalar color )
{
        int len, hlen, blen, start_x, start_y, end_x, end_y, h1_x, h1_y, h2_x, h2_y;
        double scl, ori;
        double scale = 5.0;
        double hscale = 0.75;
        CvPoint start, end, h1, h2;
```

```
        /* compute points for an arrow scaled and rotated by feat's scl and ori */
        start_x = cvRound( feat->x );
        start_y = cvRound( feat->y );
        scl = feat->scale;
        ori = feat->orientation;
        len = cvRound( scl * scale );
        hlen = cvRound( scl * hscale );
        blen = len - hlen;
        end_x = cvRound( len *  cos( ori ) ) + start_x;
        end_y = cvRound( len * -sin( ori ) ) + start_y;
        h1_x = cvRound( blen *  cos( ori + CV_PI / 18.0 ) ) + start_x;
        h1_y = cvRound( blen * -sin( ori + CV_PI / 18.0 ) ) + start_y;
        h2_x = cvRound( blen *  cos( ori - CV_PI / 18.0 ) ) + start_x;
        h2_y = cvRound( blen * -sin( ori - CV_PI / 18.0 ) ) + start_y;
        start = cvPoint( start_x, start_y );
        end = cvPoint( end_x, end_y );
        h1 = cvPoint( h1_x, h1_y );
        h2 = cvPoint( h2_x, h2_y );

        cvLine( img, start, end, color, 1, 8, 0 );
        cvLine( img, end, h1, color, 1, 8, 0 );
        cvLine( img, end, h2, color, 1, 8, 0 );
}

void draw_features( IplImage* img, struct feature* feat, int n )
{
        CvScalar color = CV_RGB( 255, 0, 0 );
```

```
        int i;


        if( img-> nChannels > 1 )
                color = CV_RGB( 0, 0, 255 );;
        for( i = 0; i < n; i++ )
                draw_sift_feature( img, feat + i, color );
}
int checkExist(vector<coord> vec, int x, int y){
  for(int i = 0; i < vec.size();i++)
    if (vec[i].x == x && vec[i].y == y)
      return 1;
  return 0;
}


int main(int argc, char** argv){
  vector<coord> coords = vector<coord>();
  struct coord _coord;
  int total_feat1, total_feat2, t1,t2;
  IplImage* img1, *img2, * merged;
  struct feature* feat1, * feat2, * feat;
  struct feature** neighbours;
  struct kd_node* root;
  CvPoint pt1, pt2, pt3,pt4,pt5,pt6;
  double d0,d1;
  int n1,n2,k,i,no_of_match = 0;
  char str[80] = "Fingerprint Recognition";
  if(argc != 3){
    printf("The program needs two parameters to start with.\n");
```

```
  exit(1);
}
start = clock();
if( (img1 = cvLoadImage(argv[1],1)) == 0) {
  printf("Failed to load Image %s\n", argv[1]);
  return -1;
}
if( (img2 = cvLoadImage(argv[2],1)) == 0) {
  printf("Failed to load Image %s\n", argv[2]);
  return -1;
}

merged = merge_imgs(img1,img2);

total_feat1 = SIFT_feature(img1, &feat1,&t1);

printf("%s has %d features\n", argv[1],t1);
total_feat2 = SIFT_feature(img2, &feat2,&t2);
printf("%s has %d features\n", argv[2],t2);



root = kd_build(feat2,total_feat2);
int c = 0;
for(i = 0; i < total_feat1; i++){
  feat = feat1 + i;
  // find feat's closest point neighbours[0] and second closest point neighbours[1].
  k = nearest_neighbour(root,feat,2, &neighbours,200);
  if(k == 2){
```

```
    // calculate their distances d0 and d1.
    d0 = descriptor_distance_squard(feat,neighbours[0]);
    d1 = descriptor_distance_squard(feat,neighbours[1]);
    pt3 = cvPoint(feat->x-2,feat->y-2);
    pt4 = cvPoint(feat->x+2,feat->y+2);
    cvRectangle(merged,pt3,pt4,CV_RGB(0,0,255),1,8,0);
    if (d0/d1 < 0.8){
          pt1 = cvPoint(cvRound(feat->x),cvRound(feat->y));
          pt2 = cvPoint(cvRound(neighbours[0]->x),cvRound(neighbours[0]->y));
          pt2.x += img1->width;
          cvLine(merged,pt1,pt2,CV_RGB(0,255,0),1,8,0);
          _coord.x = feat->x;
      _coord.y = feat->y;
          if(!checkExist(coords,feat->x,feat->y))
            coords.push_back(_coord);
          no_of_match++;
    }
   }
   free(neighbours);
  }
  end = clock();
  cout << float(end-start) / CLOCKS_PER_SEC << " seconds to find the first matched fingerprint" << endl
<< endl;
  for(i = 0; i < total_feat2; i++){
   feat = feat2+i;
   pt5 = cvPoint(feat->x-2 + img1->width,feat->y-2);
   pt6 = cvPoint(feat->x+2 + img1->width,feat->y+2);
   cvRectangle(merged,pt5,pt6,CV_RGB(0,0,255),1,8,0);
```

```
  }

  printf("Total Keypoints matched: %d.\n", coords.size());
  display_merged_img( merged, str );

  draw_features(img1,feat1,total_feat1);
  cvNamedWindow("img1_feature", 1);
  cvShowImage("img1_feature",img1);

  draw_features(img2,feat2,total_feat2);
  cvNamedWindow("img2_feature", 1);
  cvShowImage("img2_feature",img2);

  cvWaitKey( 0 );
  cvReleaseImage( &merged );
  cvReleaseImage( &img1 );
  cvReleaseImage( &img2 );
  release_tree( root );
  return 0;
}



// int main(int argc, char** argv){
//   vector<string> files = vector<string>();
//   vector<coord> coords = vector<coord>();
//   struct coord _coord;
//   int total_feat1, total_feat2;
//   IplImage* img1, *img2, * merged;
```

```
//   struct feature* feat1, * feat2, * feat;
//   struct feature** neighbours;
//   struct kd_node* root;
//   CvPoint pt1, pt2, pt3,pt4,pt5,pt6;
//   double d0,d1;
//   char response;
//   int n1,n2,k,i,no_of_match = 0, j,t;
//   char str[80] = "Fingerprint Recognition";
//
//
//   if(argc != 3) {
//     printf("program file initialImg DBDirectory.\n");
//     exit(1);
//   }
//
//   getdir(argv[2],files);
//
//   if( (img1 = cvLoadImage(argv[1],1)) == 0) {
//     printf("Failed to load Image %s\n", argv[1]);
//     return -1;
//   }
//
//   total_feat1 = SIFT_feature(img1, &feat1,&t);
//   printf("\nOriginal image has %d SIFT features\n",t);
//   printf("\nStart searching\n\n");
//   for(j = 0; j < files.size();j++){
//     if(files[j][0] != '.' && files[j][0] != ' '){
//       //printf("filename = %s\n", files[j].c_str());
```

```
//      ostringstream name;
//      name << argv[2] << "/" << files[j];
//      if( (img2 = cvLoadImage(name.str().c_str(),1)) == 0) {
//          printf("Failed to load Image %s\n", name.str().c_str());
//          return -1;
//      }
//    }else{
//      continue;
//    }
//    merged = merge_imgs(img1,img2);
//    total_feat2 = SIFT_feature(img2, &feat2,&t);
// //     printf("Image %s has %d SIFT featues\n",files[j].c_str(),t);
//    root = kd_build(feat2,total_feat2);
//    for(i = 0; i < total_feat1; i++){
//      feat = feat1 + i;
//      k = nearest_neighbour(root,feat,2, &neighbours,200);
//
//      if(k == 2){
//          d0 = descriptor_distance_squard(feat,neighbours[0]);
//
//          d1 = descriptor_distance_squard(feat,neighbours[1]);
//          pt3 = cvPoint(feat->x-2,feat->y-2);
//          pt4 = cvPoint(feat->x+2,feat->y+2);
//          cvRectangle(merged,pt3,pt4,CV_RGB(0,0,255),1,8,0);
//          cvLine(merged,pt5,pt6,CV_RGB(0,0,255),1,8,0);
//          if (d0/d1 < 0.8){
//            pt1 = cvPoint(cvRound(feat->x),cvRound(feat->y));
//            pt2 = cvPoint(cvRound(neighbours[0]->x),cvRound(neighbours[0]->y));
```

```
//            pt2.x += img1->width;
//            cvLine(merged,pt1,pt2,CV_RGB(0,255,0),1,8,0);
//            _coord.x = feat->x;
//            _coord.y = feat->y;
//            if(!checkExist(coords,feat->x,feat->y))
//              coords.push_back(_coord);
//            no_of_match++;
//          }
//      }
//      free(neighbours);
//    }
//
//    if(no_of_match > KEYPOINT_MATCHED_THRESHOLD && coords.size() >
KEYPOINT_MATCHED_THRESHOLD){
//
//      printf("\nTotal Keypoints: %d. Fingerprint image file: %s\n", coords.size(),files[j].c_str() );
//
//      cout << "Do you want to go further? ";
//      cin >> response;
//
//      if( response == 'n' || response == 'N'){
//          for(i = 0; i < total_feat2; i++){
//            feat = feat2+i;
//            pt5 = cvPoint(feat->x-2 + img1->width,feat->y-2);
//            pt6 = cvPoint(feat->x+2 + img1->width,feat->y+2);
//            cvRectangle(merged,pt5,pt6,CV_RGB(0,0,255),1,8,0);
//          }
//          display_merged_img( merged, str );
```

```
//          draw_features(img1,feat1,total_feat1);
//          cvNamedWindow("img1_feature", 1);
//          cvShowImage("img1_feature",img1);
//
//          draw_features(img2,feat2,total_feat2);
//          cvNamedWindow("img2_feature", 1);
//          cvShowImage("img2_feature",img2);
//
//          cvWaitKey( 0 );
//          cvReleaseImage( &merged );
//          cvReleaseImage( &img1 );
//          cvReleaseImage( &img2 );
//          release_tree( root );
//          break;
//      }else{
//          no_of_match = 0;
//          cvReleaseImage( &img2 );
//          cvReleaseImage( &merged );
//          free(feat2);
//          release_tree( root );
//          coords.clear();
//      }
//    }else{
//      no_of_match=0;
//      cvReleaseImage( &img2 );
//      cvReleaseImage( &merged );
//      free(feat2);
//      release_tree( root );
```

```
//      coords.clear();
//    }
//  }
//  printf("searching finished\n");
//  return 0;
// }
```

Appendix B – sift.c

```
#include "cv.h"

#include "highgui.h"

#include "cxcore.h"

#include <stdio.h>

#include <iostream>

#include <math.h>

#include <cstdlib>

#include <vector>


/* absolute value*/

#ifndef ABS

#define ABS(x) ((x < 0)? -x : x)

#endif


//Interpolates a histogram peak from left, center, and right values

#ifndef interp_hist_peak

#define interp_hist_peak( l, c, r ) ( 0.5 * ((l)-(r)) / ((l) - 2.0*(c) + (r)) )

#endif


#define pixel(image,x,y) ((uchar*)(image->imageData + (y) * image->widthStep))[x]


/* default sigma for gaussian filtering. */

#define SIFT_SIGMA 1.8
```

```
/* number of sampled intervals per octave*/
#define SIFT_INTERVALS 2

/* number of octvs in pyramid */
#define SIFT_OCTVS 4

/* number of scales in each octv */
#define SIFT_SCALES (SIFT_INTERVALS + 3)

/* number of DOG scales in each octv */
#define SIFT_DOG_SCALES (SIFT_INTERVALS + 2)

/* maximum steps of keypoint interpolation before failure */
#define SIFT_MAX_INTERPOLATE_STEPS 5

/* threshold on keypoint contrast |D(x)| */
#define SIFT_CONTRAST_THRESHOLD 0.04

/* threshold on keypoint ration of principle curvatures*/
#define SIFT_CURVATURE_THRESHOLD 10

/* width of border in which to ingore keypoints*/
#define SIFT_IMG_BORDER 4

/* the width of descriptor histogram array */
```

```
#define SIFT_DESCRIPTOR_WIDTH 16

/* number of bins for each histogram in descriptor array */
#define SIFT_DESCRIPTOR_HISTOGRAM_BINS 8

/* total number of elements in feature vector */
#define SIFT_TOTAL_FEATURE_VECTOR_ELEMENT (SIFT_DESCRIPTOR_WIDTH *
SIFT_DESCRIPTOR_WIDTH * SIFT_DESCRIPTOR_HISTOGRAM_BINS)

/* the size of a single descriptor orientation histogram */
#define SIFT_DESCRIPTOR_SCALE_FACTOR 3.0

/* threshold for magnitude of elements of descriptor vector */
#define SIFT_DESCRIPTOR_MAGNITUDE_THRESHOLD 0.2

/* factor used to convert floating-point descriptor to unsigned char */
#define SIFT_INT_DESCRIPTOR_FACTOR 512.0


/* number of bins in histogram for orientation assignment */
#define SIFT_ORIENTATION_HISTOGRAM_BINS 36

/* gaussian sigma for orientation assignment */
#define SIFT_ORIENTATION_SIGMA_FACTOR 1.5
```

```
/* define the radius of the region for orientation assignment */
#define SIFT_ORIENTATION_RADIUS 3.0 * SIFT_ORIENTATION_SIGMA_FACTOR

/* number of passes of orientation histogram smoothing */
#define SIFT_ORIENTATION_SMOOTH_PASSES 2



/* returns a feature's detection data */
#define feature_detection_data(feature) ( (struct data_detection*)(feature->feature_data) )

using namespace std;

struct feature{
        double x;                   // coordinates
        double y;
        double scale;               // feature scale.
        double orientation;         // feature orientation.
        void* feature_data;         // data structure
        double descriptor[SIFT_TOTAL_FEATURE_VECTOR_ELEMENT];   // descriptors
with max lenth widthxwidthxNoOfBins
        int descriptor_length;     // descriptor length
        CvPoint2D64f image_point;       // feature location
        struct feature* fwd_match;     /**< matching feature from forward image */
        struct feature* bck_match;     /**< matching feature from backmward image */
        struct feature* mdl_match;
```

```
};

struct data_detection{
        int row;
        int col;
        int octv;
        int interval;
        double subinterval;
        double scale_octv;
};

/************************ Function Prototypes ******************************/

// Image Initialization
IplImage* initialize_image(IplImage* img, double sigma);
IplImage* cvt_to_grayscale32(IplImage* img);

// access a particular pixel
float pixel32(IplImage* img, int r, int c);

// descrease the size of image - half of input image.
IplImage* downsample(IplImage *img);

/** Stage 1 - Scale-space Detection, this includes keypoint elimination.  */
```

```
// create Gaussian - pyramid
IplImage*** create_gaussian_pyd(IplImage* base,int intvls,int scales, int octvs, double sigma);

// create Difference-of-Gaussian Pyramid based on Gaussian-pyramid.
IplImage*** create_dog_pyd(IplImage*** gauss_pyd,int dog_scales,int octvs);


CvSeq* detect_scale_space_extrema(IplImage*** dog_pyd,int octvs,int intvls,double
contrast_threshold,int curvature_threshold,CvMemStorage* storage);

// test a pixel with its 26 neighbours at current, above and below scale at the same octv.
int is_extremum(IplImage*** dog_pyd, int octv, int intvl, int row, int col);

/**                          stage 2 keypoint localization                          */
// eliminating keypoints with low contrast.
struct feature* interporlate_extremum(IplImage*** dog_pyd, int octv, int intvl, int row, int col,
int intvls, double contrast_threshold);

// performs extremum interpolation. Equation (3)
void interpolate_step(IplImage*** dog_pyd, int octv, int intvl, int r, int c, double *xi, double *xr,
double *xc);

// interpolated pixel contrast.
double interpolate_contrast( IplImage*** dog_pyd, int octv, int intvl, int row,int col, double xi,
double xr, double xc );
```

```
// computes the scale of a pixel in DoG and the partial derivatives in x, y.
CvMat* deriv_3D( IplImage*** dog_pyd, int octv, int intvl, int row, int col );
//create and initialize a new feature
struct feature* create_feature();

// check if a feature is located along edge.
int is_along_edges(IplImage* img, int row, int col, int curvature_threshold);

// computes scale characteristic scale for each feature.
void calculate_feature_scales(CvSeq* features, double sigma, int intervals);

/**                    stage 3 Orientation Assignment                    */

// computes orientation for each image features in an array.
void calculate_feature_orientation(CvSeq* features, IplImage*** gaussian_pyd);

// computes gradient orientatioin histogram at a pixel.
double* orientation_histogram( IplImage* img, int row, int col, int no_of_bins, int radius, double
sigma);

// computes the gradient and orientatioin.
int calculate_gradient_magnitude_orientation( IplImage* img, int row, int col, double* magnitude,
double* orientation );
```

```
// smooths the orientation histogram by using Gaussian smooth.
void smooth_orientation_histogram( double* histogram, int no_of_bins );

// search for the magnitude of the dominant orientation.
double dominant_orientation( double* histogram, int no_of_bins );

// add featuers to an array for each orientation in a histogram according
// to a given threshold.
void add_good_orientation_features( CvSeq* features, double* histogram, int no_of_bins, double
magnitude_threshold, struct feature* _feature );

// copy a feature to another.
struct feature* copy_feature(struct feature* _feature);


/**                stege 4 keypoint descriptor                                    */

void compute_descriptors( CvSeq* features, IplImage*** gaussian_pyd, int width, int
no_of_bins);

// computes orientation of histogram.
double*** descriptor_histogram( IplImage* img, int row, int col, double orientation, double scale,
int width, int no_of_bins );

// interpolate histogram entry
```

```
// interplates an entry into orientation histogram array.
void interpolate_histogram_entry( double*** histogram, double sub_row_bin, double
sub_col_bin,double sub_ori_bin, double magnitude, int width, int no_of_bins );

// convert orientation histogram array into feature descriptor vector.
void histogram_to_descriptor(double*** histogram, int width, int no_of_bins, struct feature*
_feature);

// normalizes  the feature descriptor vector.
void normalize_descriptor( struct feature* _feature );


// compare features for cvSeqsort
int feature_compare(struct feature* feature1, struct feature* feature2, void* param);

// free memory.
void release_descriptor_histogram( double**** histogram, int width );
void release_pyd( IplImage**** pyd, int octvs, int no_img_per_octv );
int SIFT_feature(IplImage *img, struct feature** feat);
/*************************************************************************/

/* initialize the input image, check the specification of the image and then
   convert it to 32-bit grayscale and apply Gaussian-smooths */
IplImage* initialize_image(IplImage* img, double sigma){
  IplImage* grayscale;
```

```
  double sig, sig_ini = 0.5; // used for gaussian blur for original image.

  grayscale = cvt_to_grayscale32(img);

  sig = sqrt(sigma*sigma - sig_ini*sig_ini);
  cvSmooth(grayscale,grayscale,CV_GAUSSIAN,3,3,sig,sig);

  return grayscale;
}
/* convert the input image to 32-bit grayscale */
IplImage* cvt_to_grayscale32(IplImage* img){
  IplImage* grayscale8, *grayscale32;
  int x,y;
  grayscale32 = cvCreateImage(cvGetSize(img),IPL_DEPTH_32F,1);
  if (img->nChannels == 1)
    grayscale8 = cvCloneImage(img);
  else{
    grayscale8 = cvCreateImage(cvGetSize(img),IPL_DEPTH_8U,1);
    cvCvtColor(img,grayscale8,CV_BGR2GRAY);
  }

  cvConvertScale(grayscale8,grayscale32,1.0/255.0,0);
  cvReleaseImage(&grayscale8);
  return grayscale32;
}
```

```
/* get the pixel value */
float pixel32(IplImage* img, int row, int col){
  return ( (float*)(img->imageData + img->widthStep*row))[col];
}

/* downsample the image for each scales in each octv */
IplImage* downsample(IplImage *img){
  IplImage *half = cvCreateImage(cvSize(img->width/2,img->height/2),img->depth,img-
>nChannels);
  cvResize(img,half,CV_INTER_LINEAR);
  return half;
}

/** Stage 1 - Scale-space Detection, this includes keypoint elimination.  */
/* create Gaussian pyramid
   the intervals is 2 by default, that means there are 5 scales (3+2)
   in each octv. and then apply Gaussian blur for each scale in each octv.
*/
IplImage*** create_gaussian_pyd(IplImage* base,int intervals, int scales, int octvs, double
sigma){
  IplImage*** gauss_pyd;
  const int intvls = intervals;
  int i,j;
  double sig[scales], k = pow(2.0,1.0/intervals),sigtotal,sigpre;
```

```
//   precompute Gaussian sigma
  sig[0] = sigma;
  for(i = 1; i < scales; i++){
    sigpre = pow(k,i-1)*sigma;
    sigtotal = sigpre * k;
    sig[i] = sqrt(sigtotal*sigtotal - sigpre*sigpre);
  }

  // allocate memory for images across entire pyramid. number of image is based on the number of
intervals.
  gauss_pyd = (IplImage***)malloc(octvs * sizeof(IplImage**));
  for(i = 0; i < octvs;i++){
    gauss_pyd[i] = (IplImage**)malloc(octvs * sizeof(IplImage*));
  }
  // create Gaussian images for each octaves.
  for(j = 0; j < octvs; j++){
    for(i = 0; i < scales; i++){
      if (j == i && i == 0)
            gauss_pyd[j][i] = cvCloneImage(base);

      else if(i == 0)
            gauss_pyd[j][i] = downsample(gauss_pyd[j-1][intvls]);
      else{
            gauss_pyd[j][i] = cvCreateImage(cvGetSize(gauss_pyd[j][i-1]),IPL_DEPTH_32F,1);
//          printf("before smooth\n");
```

```
            cvSmooth(gauss_pyd[j][i-1],gauss_pyd[j][i],CV_GAUSSIAN,0,0,sig[i],sig[i]);
//          printf("before smooth\n");
      }
    }
  }
  return gauss_pyd;
}

/* create Difference of Gaussian images
   in this case, there are 4 DoG images(dog scales) in each octv.
   it is done by substracting the adjacent scales in the Gaussian pyramid.
*/
IplImage*** create_dog_pyd(IplImage*** gauss_pyd,int dog_scales,int octvs){
  IplImage*** dog_pyd;
  int i,j;

  /* allcate memory for DoG images*/
  dog_pyd = (IplImage***)malloc(octvs * sizeof(IplImage**));
  for(i = 0; i < octvs;i++){
    dog_pyd[i] = (IplImage**)malloc(octvs * sizeof(IplImage*));
  }

  // DoG images are created by subtracting two adjacent Gaussian image.
  for(j = 0; j < octvs; j++){
    for(i = 0; i < dog_scales; i++){
```

```
      dog_pyd[j][i] = cvCreateImage(cvGetSize(gauss_pyd[j][i]),IPL_DEPTH_32F,1);
      cvSub(gauss_pyd[j][i+1],gauss_pyd[j][i],dog_pyd[j][i],NULL);
    }
  }
  return dog_pyd;
}

/* section 4. scale space extrema detection
  This stage finds the local extrema from each dog scales.
  The top and bottom scale is only used for comparing.
  the extrema is selected only from two middle scales as
  the number of intervals is set to 2 by default.
  The extream is selected as a candidate keypoint.
  and then low-contrast keypoints and keypoints are along edges
  are going to be eliminated.
 */
CvSeq* detect_scale_space_extrema(IplImage*** dog_pyd,int octvs,int intvls,double
contrast_threshold,int curvature_threshold,CvMemStorage* storage){
  CvSeq* features;
  double prelim_contrast_threshold = 0.5 * contrast_threshold/intvls;
//   double prelim_contrast_threshold = contrast_threshold;
  struct feature* _feature;
  struct data_detection* detectdata;
  int i,j,k,g, c = 0;
  double pixVal;
```

```
 features = cvCreateSeq(0,sizeof(CvSeq),sizeof(struct feature),storage);


 for(i = 0; i < octvs; i++){
  for(j = 1; j <= intvls; j++){
   for(k = SIFT_IMG_BORDER; k < dog_pyd[i][0]->height-SIFT_IMG_BORDER;k++){
        for(g = SIFT_IMG_BORDER; g < dog_pyd[i][0]->width-SIFT_IMG_BORDER;g++){
          if(ABS((pixVal = pixel32(dog_pyd[i][j],k,g))) > prelim_contrast_threshold){
            if (is_extremum(dog_pyd,i,j,k,g)){          // check if a specified pixel is minimum
or maximum
                    // if extrema, check if it's a keypoint with low-contrast.
                    _feature = interporlate_extremum(dog_pyd,i,j,k,g,intvls,contrast_threshold);
                    if (_feature){
                      detectdata = feature_detection_data(_feature);
                      // check if a keypoint is located along an edge.
                      if(!is_along_edges(dog_pyd[detectdata->octv][detectdata->interval],
detectdata->row, detectdata->col,curvature_threshold) ){
                        // if a keypoint is not located along edge and is not low contrast, then it is
selected as a candidate keypoint.
                        cvSeqPush(features,_feature);
                      }else
                        free(detectdata);
                        free(_feature);
                    }
            }
```

```
            }
          }
      }
    }
  }
  return features;
}

/* compare a pixel with its 26 neighbours which are located at the current, above and below scale
*/
int is_extremum(IplImage*** dog_pyd, int octv, int intvl, int row, int col){
  double pixVal = pixel32(dog_pyd[octv][intvl],row,col);
  int i,j,k;

/* check pixel value with its 26 neighbours in adjacent scales*/
  /* check for maximum */
  if (pixVal > 0) {
    for(i = -1; i <= 1; i++){
      for(j = -1; j <= 1; j++){
          for(k = -1; k <= 1; k++){
            if (pixVal < pixel32(dog_pyd[octv][intvl+i], row + j, col + k))
              return 0;
          }
      }
    }
```

```
  }else{ /* check for minimum */
    for(i = -1; i <= 1; i++){
      for(j = -1; j <= 1; j++){
            for(k = -1; k <= 1; k++){
              if (pixVal > pixel32(dog_pyd[octv][intvl+i], row + j, col + k))
                return 0;
            }
      }
    }
  }
  return 1;
}

/**                          stage 2 keypoint localization                          */
/* interpolate scale space extrema, eliminate the featuers with low contrast
   contrast threshold is set to 0.03 based on section 4 of Lowe's paper.
*/
struct feature* interporlate_extremum(IplImage*** dog_pyd, int octv, int intvl, int row, int col,
int intvls, double contrast_threshold){
  struct feature* _feature;
  struct data_detection* detectdata;
  double xi,xr,xc,contrast;
  int i = 0;

  while(i < SIFT_MAX_INTERPOLATE_STEPS){
```

```
    interpolate_step(dog_pyd,octv,intvl,row,col,&xi,&xr,&xc);
    if(ABS(xi) < 0.5 && ABS(xr) < 0.5 && ABS(xc) < 0.5)
      break;
    col += cvRound(xc);
    row += cvRound(xr);
    intvl += cvRound(xi);
    if(intvl < 1 || intvl > intvls || col < SIFT_IMG_BORDER || row < SIFT_IMG_BORDER ||
col >= dog_pyd[octv][0]->width - SIFT_IMG_BORDER || row >= dog_pyd[octv][0] -> height -
SIFT_IMG_BORDER){
            return NULL;
    }
      i++;
  }

  if (i >= SIFT_MAX_INTERPOLATE_STEPS)
    return NULL;
  contrast = interpolate_contrast(dog_pyd,octv,intvl, row,col,xi,xr,xc);
  if(ABS(contrast) < contrast_threshold / intvls)
    return NULL;

  _feature = create_feature();
  detectdata = feature_detection_data(_feature);
  _feature->image_point.x = _feature->x = (col+xc) * pow(2.0,octv);
  _feature->image_point.y = _feature->y = (row+xr) * pow(2.0,octv);
  detectdata->row = row;
```

```
  detectdata->col = col;
  detectdata->octv = octv;
  detectdata->interval = intvl;
  detectdata->subinterval = xi;

   return _feature;
}

/* extremum interpolatioin accroding to the equation (3)*/
void interpolate_step(IplImage*** dog_pyd, int octv, int intvl, int row, int col, double *xi, double
*xr, double *xc){
  CvMat* dD, *hessian, *hessian_inv,X;
  double x[3] ={0}, dx,dy,ds;
  double v, dxx, dyy, dss, dxy, dxs, dys;  // for hessian 3D

  dD = deriv_3D( dog_pyd, octv, intvl, row, col );

  // compute the 3D hessian matrix for a pixel by using differences of neighboring sample points.
  v = pixel32( dog_pyd[octv][intvl], row, col );
  dxx = ( pixel32( dog_pyd[octv][intvl], row, col+1 ) +
            pixel32( dog_pyd[octv][intvl], row, col-1 ) - 2 * v );
  dyy = ( pixel32( dog_pyd[octv][intvl], row+1, col ) +
            pixel32( dog_pyd[octv][intvl], row-1, col ) - 2 * v );
  dss = ( pixel32( dog_pyd[octv][intvl+1], row, col ) +
            pixel32( dog_pyd[octv][intvl-1], row, col ) - 2 * v );
```

```
dxy = ( pixel32( dog_pyd[octv][intvl], row+1, col+1 ) -
        pixel32( dog_pyd[octv][intvl], row+1, col-1 ) -
        pixel32( dog_pyd[octv][intvl], row-1, col+1 ) +
        pixel32( dog_pyd[octv][intvl], row-1, col-1 ) ) / 4.0;
dxs = ( pixel32( dog_pyd[octv][intvl+1], row, col+1 ) -
        pixel32( dog_pyd[octv][intvl+1], row, col-1 ) -
        pixel32( dog_pyd[octv][intvl-1], row, col+1 ) +
        pixel32( dog_pyd[octv][intvl-1], row, col-1 ) ) / 4.0;
dys = ( pixel32( dog_pyd[octv][intvl+1], row+1, col ) -
        pixel32( dog_pyd[octv][intvl+1], row-1, col ) -
        pixel32( dog_pyd[octv][intvl-1], row+1, col ) +
        pixel32( dog_pyd[octv][intvl-1], row-1, col ) ) / 4.0;

hessian = cvCreateMat( 3, 3, CV_64FC1 );
cvmSet( hessian, 0, 0, dxx );
cvmSet( hessian, 0, 1, dxy );
cvmSet( hessian, 0, 2, dxs );
cvmSet( hessian, 1, 0, dxy );
cvmSet( hessian, 1, 1, dyy );
cvmSet( hessian, 1, 2, dys );
cvmSet( hessian, 2, 0, dxs );
cvmSet( hessian, 2, 1, dys );
cvmSet( hessian, 2, 2, dss );

hessian_inv = cvCreateMat(3,3,CV_64FC1);
```

```
  cvInvert(hessian,hessian_inv,CV_SVD);
  cvInitMatHeader(&X,3,1,CV_64FC1,x,CV_AUTOSTEP);
  cvGEMM(hessian_inv,dD,-1,NULL,0,&X,0);

  cvReleaseMat(&dD);
  cvReleaseMat(&hessian);
  cvReleaseMat(&hessian_inv);

  *xi = x[2];
  *xr = x[1];
  *xc = x[0];
}

/* computes the partial derivatives in x, y and scale of a pixel in the DoG pyramid.*/
CvMat* deriv_3D( IplImage*** dog_pyd, int octv, int intvl, int row, int col )
{
  CvMat* dI;
  double dx, dy, ds;
  // calculate the partial derivatives.
  dx = ( pixel32( dog_pyd[octv][intvl], row, col+1 ) -
         pixel32( dog_pyd[octv][intvl], row, col-1 ) ) / 2.0;
  dy = ( pixel32( dog_pyd[octv][intvl], row+1, col ) -
         pixel32( dog_pyd[octv][intvl], row-1, col ) ) / 2.0;
  ds = ( pixel32( dog_pyd[octv][intvl+1], row, col ) -
         pixel32( dog_pyd[octv][intvl-1], row, col ) ) / 2.0;
```

```
  // create matrix.
  dI = cvCreateMat( 3, 1, CV_64FC1 );
  // set matrix based on partial derivatives of x,y.
  cvmSet( dI, 0, 0, dx );
  cvmSet( dI, 1, 0, dy );
  cvmSet( dI, 2, 0, ds );

  return dI;
}

/* interpolate pixel contrast remove keypoints with low contrast. */
double interpolate_contrast( IplImage*** dog_pyd, int octv, int intvl, int row, int col, double xi,
double xr, double xc ){
  CvMat* dD, X, T;
  double t[1], x[3] = { xc, xr, xi };

  cvInitMatHeader( &X, 3, 1, CV_64FC1, x, CV_AUTOSTEP );
  cvInitMatHeader( &T, 1, 1, CV_64FC1, t, CV_AUTOSTEP );
  dD = deriv_3D( dog_pyd, octv, intvl, row, col );
  cvGEMM( dD, &X, 1, NULL, 0, &T,  CV_GEMM_A_T );
  cvReleaseMat( &dD );

  return pixel32( dog_pyd[octv][intvl], row, col ) + t[0] * 0.5;
```

```
}

struct feature* create_feature(){
  struct feature* _feature;
  struct data_detection* detectdata;

  _feature = (struct feature*)malloc(sizeof(struct feature));
  memset(_feature,0,sizeof(struct feature));
  detectdata = (struct data_detection*)malloc(sizeof(struct data_detection));
  memset(detectdata,0,sizeof(struct data_detection));
  _feature->feature_data = detectdata;

  return _feature;
}

/* eliminating edge repsponses
   determines wheather a feature is along edges by computing
   the ratio principal curvature across the edge.using 2D Hessian matrix.
   curvature threshold is set to 10 based on the section 4.1 of Lowe's paper.
*/
int is_along_edges(IplImage* dog_img, int row, int col, int curvature_threshold){
  double d, dxx,dyy,dxy,tr,det;

  /* principal curvatures are computed by using 2D hessian */
  // get a specified pixel value.
```

```
  d = pixel32(dog_img, row,col);
  // calculate its derivatives based on the differences of
  // neighbouring sample points.
  dxx = pixel32(dog_img,row,col+1) + pixel32(dog_img,row,col-1) - 2 * d;
  dyy = pixel32(dog_img, row + 1, col) + pixel32(dog_img,row-1,col) - 2 * d;
  dxy = ( pixel32(dog_img, row + 1, col + 1) - pixel32(dog_img,row + 1, col - 1) -
            pixel32(dog_img, row - 1, col + 1) + pixel32(dog_img,row - 1, col - 1) ) / 4.0;
  // Hessian matrx is formed as
  // |Dxx, Dxy|
  // |Dxy, Dyy|
  // calculate the sum of the eigenvalues from the trace(tr) of Hessian Matrix.
  tr = dxx + dyy;
  // calculate the product of the eigenvalues from the determinant;
  det = dxx * dyy - dxy * dxy;


  if(det <= 0)
    return 1;
  // using ratio of curvatures to determine if a keypoint is located along edges.
  if (tr * tr / det < (curvature_threshold + 1.0) * (curvature_threshold + 1.0) / curvature_threshold)
    return 0;
  return 1;
}

/* find the feature characteristic scale */
```

```
void calculate_feature_scales(CvSeq* features, double sigma, int intervals){
  struct feature* _feature;
  struct data_detection* detectdata;
  double interval;
  int i;
  for ( i = 0; i < features -> total; i++){
    _feature = CV_GET_SEQ_ELEM(struct feature, features,i);
    detectdata = feature_detection_data(_feature);
    interval = detectdata -> interval + detectdata -> subinterval;
    _feature->scale = sigma * pow(2.0, detectdata->octv + interval / intervals);
    detectdata->scale_octv = sigma * pow(2.0, interval / intervals);
  }
}

/**                    stage 3 Orientation Assignment                    */
/* section 5. calculate orientation for each image in the array.
   an orientation histogram is formed from the gradient orientations of
   sample points within a region around the keypoint.
   each sample added to the histogram is weighted by its gradient magnitude
   and by gaussian-weight with a sigma 1.5 times the scale of the keypoint.
*/
void calculate_feature_orientation(CvSeq* features, IplImage*** gaussian_pyd){
  struct feature* _feature;
  struct data_detection* detectdata;
  double* histogram;
```

```
  double orientationmax,orientation_peak_ratio = 0.8;      // used to determine good orientation.
  int i,j;
  int sop = 2; // sift orientation passes
  for(i = 0; i < features-> total && i < 2000; i++){
   _feature = (struct feature*)malloc(sizeof(struct feature));
   cvSeqPopFront(features,_feature);
   detectdata = feature_detection_data(_feature);
   histogram = orientation_histogram(gaussian_pyd[detectdata->octv][detectdata->interval],

                                     detectdata->row,detectdata-
>col,SIFT_ORIENTATION_HISTOGRAM_BINS,
                                     cvRound(SIFT_ORIENTATION_RADIUS * detectdata-
>scale_octv),
                                     SIFT_ORIENTATION_SIGMA_FACTOR * detectdata-
>scale_octv);

   for(j = 0; j < sop; j++)
     smooth_orientation_histogram(histogram,SIFT_ORIENTATION_HISTOGRAM_BINS);

   // find the highest peak in the histogram.
   orientationmax
=dominant_orientation(histogram,SIFT_ORIENTATION_HISTOGRAM_BINS);
   // any other local peak that is within 0.8 of the highest peak is used to create
   // a keypoint with that orientation.
```

```
add_good_orientation_features(features,histogram,SIFT_ORIENTATION_HISTOGRAM_BINS,
orientationmax * orientation_peak_ratio,_feature);

    free(_feature);
    free(detectdata);
    free(histogram);
  }
}

/* calculates gradient orientation histogram at a specified pixel.
   sigma is used for gaussian weighting of histogram entries.
   returns the array 'histogram' which contains the orientation histogram
   between 0 and 2 pi.
   Gaussian-weighted circular window with a sigma taht is 1.5 times that of the
   scale of the keypoint.
*/
double* orientation_histogram( IplImage* img, int row, int col, int no_of_bins, int radius, double
sigma){
  double* histogram;
  double magnitude, orientation, w, exp_denom, PI2 = CV_PI * 2.0;
  int bin, i, j;

  histogram = (double*)calloc( no_of_bins, sizeof( double ) );
  exp_denom = 2.0 * sigma * sigma;
```

```
  for( i = -radius; i <= radius; i++ ){
    for( j = -radius; j <= radius; j++ ){
      // if valid set magnitude and orientation.
      if( calculate_gradient_magnitude_orientation( img, row + i, col + j, &magnitude,
&orientation ) ){
            w = exp( -( i*i + j*j ) / exp_denom );
            bin = cvRound( no_of_bins * ( orientation + CV_PI ) / PI2 );
            bin = ( bin < no_of_bins )? bin : 0;
            histogram[bin] += w * magnitude;
      }
    }
  }
  return histogram;
}

/* calculate gradient magnitude and orientation at a specific pixel
   based on the equation in section 5 in Lowe's paper
*/

int calculate_gradient_magnitude_orientation( IplImage* img, int row, int col, double* magnitude,
double* orientation ){
  double dx, dy;

  if( row > 0  &&  row < img->height - 1  &&  col > 0  &&  col < img->width - 1 ){
    dx = pixel32( img, row, col+1 ) - pixel32( img, row, col-1 );
```

```
    dy = pixel32( img, row-1, col ) - pixel32( img, row+1, col );
    *magnitude = sqrt( dx*dx + dy*dy );
    *orientation = atan2( dy, dx );
      return 1;                  //if pixel is valid otherwise return 0;
  }else
      return 0;
}

/* use gaussian blur to smooth orientation histogram */
void smooth_orientation_histogram( double* histogram, int no_of_bins ){
  double prev, tmp, hist0 = histogram[0];
  int i;

  prev = histogram[no_of_bins-1];
  for( i = 0; i < no_of_bins; i++ ){
    tmp = histogram[i];
    histogram[i] = 0.25 * prev + 0.5 * histogram[i] + 0.25 * ( ( i+1 == no_of_bins )? hist0 :
histogram[i+1] );
    prev = tmp;
  }
}

/* finds the magnitude of the dominant orientation in a histogram and
   returns the largest bin value in histogram
   note that peaks in the orientation histogram corresponds to
```

```
   dominant directions of local gradients.
*/
double dominant_orientation( double* histogram, int no_of_bins ){
  double orientationmax;
  int maxbin, i;

  orientationmax = histogram[0];
  maxbin = 0;
  for( i = 1; i < no_of_bins; i++ ){
    if( histogram[i] > orientationmax ){
      orientationmax = histogram[i];
      maxbin = i;
    }
  }
  return orientationmax; // return the largest bin value in the histogram.
}

/* if orientation in histogram is greater than the magnitude threshold then
   add this feature to the array.otherwise ignore.
   finally, the highest peak in the histogram is detected, any other
   local peak that is within 0.8 of the highest peak is used to
   create a keypoint with that orientation.
*/
void add_good_orientation_features( CvSeq* features, double* histogram, int no_of_bins, double
magnitude_threshold, struct feature* _feature ){
```

```
  struct feature* new_feature;
  double bin, PI2 = CV_PI * 2.0;
  int l, r, i;                    // for left center and right.

  for( i = 0; i < no_of_bins; i++ ){
    if (i == 0){
      l = no_of_bins - 1;
    }else
      l = i - 1;
    r = ( i + 1 ) % no_of_bins;

    // magnitude threshold is used for measure if new feature is added.
    if( histogram[i] > histogram[l]  &&  histogram[i] > histogram[r]  &&  histogram[i] >=
magnitude_threshold ){
      bin = i + interp_hist_peak( histogram[l], histogram[i], histogram[r] );
      bin = (bin<0)?no_of_bins+bin : (bin>=no_of_bins)?bin-no_of_bins : bin;
      new_feature = copy_feature(_feature);          // copy feature with different orientations.
      new_feature->orientation = ( ( PI2 * bin ) / no_of_bins ) - CV_PI;
      cvSeqPush( features, new_feature );
      free( new_feature );
    }
  }
}

/* copy a feature */
```

```
struct feature* copy_feature(struct feature* _feature){
  struct feature* _new_feature;
  struct data_detection* detectdata;

  _new_feature = create_feature();
  detectdata = feature_detection_data(_new_feature);
  memcpy( _new_feature, _feature, sizeof(struct feature));
  memcpy( detectdata, feature_detection_data(_feature), sizeof(struct data_detection));
  _new_feature->feature_data = detectdata;

  return _new_feature;
}

/**                    stege 4 keypoint descriptor                                    */
/* Section 6 Computes features descriptors. */
void compute_descriptors( CvSeq* features, IplImage*** gaussian_pyd, int width, int
no_of_bins){
  // parameter width is the width of orientation histograms.
  struct feature* _feature;
  struct data_detection* detectdata;
  double*** histogram;
  int i;

  for( i = 0; i < features->total && i < 3000; i++ ){
    _feature = CV_GET_SEQ_ELEM( struct feature, features, i );
```

```
    detectdata = feature_detection_data( _feature );
    // width is 16x16 no_of_bins is 36 by default.
    histogram = descriptor_histogram(gaussian_pyd[detectdata->octv][detectdata->interval],
detectdata->row,detectdata->col, _feature->orientation, detectdata->scale_octv, width,
no_of_bins);

    histogram_to_descriptor(histogram, width, no_of_bins, _feature);
//     printf("here, i = %d\n", i);
    release_descriptor_histogram(&histogram, width);

  }

}

/* section 6.1 Descriptor representation
   Computes the orientation histograms and keypoint descriptor.
   takes image for descriptor computation
   row and col are represented the coordinates of center of the orientation histogram array
   rather than a pixel position.
   returns a array of orientation histograms.
*/
double*** descriptor_histogram( IplImage* img, int row, int col, double orientation, double scale,
int width, int no_of_bins ){
  double*** histogram;
```

```
  double cos_t, sin_t, histogram_width, exp_denom, r_rot, c_rot,
gradient_magnitude,gradient_orientation, w, sub_row_bin, sub_col_bin, sub_ori_bin,
bins_per_radius, PI2 = 2.0 * CV_PI;

  int radius, i, j;
  // widthxwidthxno_of_bins 16x16x8
  histogram = (double***)calloc( width, sizeof( double** ) );
  for( i = 0; i < width; i++ ){
    histogram[i] = (double**)calloc( width, sizeof( double* ) );
    for( j = 0; j < width; j++ )
      histogram[i][j] = (double*)calloc( no_of_bins, sizeof( double ) );
  }

  cos_t = cos(orientation);
  sin_t = sin(orientation);
  bins_per_radius = no_of_bins / PI2;
  exp_denom = width * width * 0.5;
  histogram_width = SIFT_DESCRIPTOR_SCALE_FACTOR * scale;
  radius = histogram_width * sqrt(2) * ( width + 1.0 ) * 0.5 + 0.5;

  for( i = -radius; i <= radius; i++ ){
//     printf("i = %d ", i);
    for( j = -radius; j <= radius; j++ ){
/*Calculate histogram array coordinates rotated relative to orientation.
  and subtract 0.5 so samples that fall */
```

```
    c_rot = ( j * cos_t - i * sin_t ) / histogram_width;
    r_rot = ( j * sin_t + i * cos_t ) / histogram_width;
    sub_row_bin = r_rot + width / 2 - 0.5;
    sub_col_bin = c_rot + width / 2 - 0.5;

    if( sub_row_bin > -1.0  &&  sub_row_bin < width  &&  sub_col_bin > -1.0  &&  sub_col_bin
< width )
    if( calculate_gradient_magnitude_orientation( img, row + i, col+ j, &gradient_magnitude,
&gradient_orientation)){
          gradient_orientation -= orientation;
          while( gradient_orientation < 0.0 )
             gradient_orientation += PI2;

          while( gradient_orientation >= PI2 )
             gradient_orientation -= PI2;

          sub_ori_bin = gradient_orientation * bins_per_radius;
          w = exp( -(c_rot * c_rot + r_rot * r_rot) / exp_denom );
          interpolate_histogram_entry( histogram, sub_row_bin, sub_col_bin, sub_ori_bin,
gradient_magnitude * w, width, no_of_bins );
     }
   }
  }
  return histogram;
```

```
}



/* Interpolates an entry into orientation histogram array that form
   the feature descriptor.
   sub_row_bin row coordinate of entry
   sub_col_bin column coordinate of entry
   sub_ori_bin orientation coordinate of entry.
   magnitude, size of entry.
   width is 16x16x8 SIFT_DESCRIPTOR_WIDTH ^2 *
SIFT_DESCRIPTOR_HISTOGRAM_BINS
   no_of bins 8
*/
void interpolate_histogram_entry( double*** histogram, double sub_row_bin, double
sub_col_bin,double sub_ori_bin, double magnitude, int width, int no_of_bins ){
  double d_r, d_c, d_o, v_r, v_c, v_o;
  double **row, *h;
  int r0, c0, o0, rb, cb, ob, r, c, o;


  r0 = cvFloor( sub_row_bin );
  c0 = cvFloor( sub_col_bin );
  o0 = cvFloor( sub_ori_bin );
  d_r = sub_row_bin - r0;
  d_c = sub_col_bin - c0;
  d_o = sub_ori_bin - o0;
```

```
/* there are upto 8 bins used to distribute the entry,and each entry is multiple
   by a weight of 1-d for each dimension in a bin, where d is the distance
   from the center value of the bin measured in bin units. */
 for( r = 0; r <= 1; r++ ){
  rb = r0 + r;
  if( rb >= 0  &&  rb < width ){
    v_r = magnitude * ( ( r == 0 )? 1.0 - d_r : d_r );
    row = histogram[rb];
    for( c = 0; c <= 1; c++ ){
          cb = c0 + c;
          if( cb >= 0  &&  cb < width ){
            v_c = v_r * ( ( c == 0 )? 1.0 - d_c : d_c );
            h = row[cb];
            for( o = 0; o <= 1; o++ ){
              ob = ( o0 + o ) % no_of_bins;
              v_o = v_c * ( ( o == 0 )? 1.0 - d_o : d_o );
              h[ob] += v_o;
             }
          }
    }
   }
  }
}
```

```
/* Converts orientation histograms into a feature's descriptor vector.
   normalizes the vector firstly, and then for large change in relative
   magnitude, we reduces the influence of large gradient magnitudes by
   thresholding the values in the unit feature vector to each be no
   larger than SIFT_DESCRIPTOR_MAGNITUDE_THRESHOLD which is 0.2 from
   Lowe's paper and finally, renormalizing to unit length.
*/
void histogram_to_descriptor(double*** histogram, int width, int no_of_bins, struct feature*
_feature){
  int int_val, i, x, y, j, k = 0;

  for( x = 0; x < width; x++ )
    for( y = 0; y < width; y++ )
      for( j = 0; j < no_of_bins; j++ )
            _feature->descriptor[k++] = histogram[x][y][j];

  _feature->descriptor_length = k;
  normalize_descriptor( _feature );
  for( i = 0; i < k; i++ )
    if( _feature->descriptor[i] > SIFT_DESCRIPTOR_MAGNITUDE_THRESHOLD )
      _feature->descriptor[i] = SIFT_DESCRIPTOR_MAGNITUDE_THRESHOLD;
  normalize_descriptor( _feature );

  /* converts floating-point descriptor to integer valued descriptor */
  for( i = 0; i < k; i++ ){
```

```
    int_val = SIFT_INT_DESCRIPTOR_FACTOR * _feature->descriptor[i];
    _feature->descriptor[i] = MIN( 255, int_val );
  }
}


/* Normalizes a feature's descriptor vector
   as the feature vector is modified to reduce
   the effects of illumination change.
*/
void normalize_descriptor( struct feature* _feature ){
  double cur, length_inv, lenth_sq = 0.0;
  int i, dl = _feature->descriptor_length;

  for( i = 0; i < dl; i++ ){
    cur = _feature->descriptor[i];
    lenth_sq += cur*cur;
  }
  length_inv = 1.0 / sqrt( lenth_sq );
  for( i = 0; i < dl; i++ )
    _feature->descriptor[i] *= length_inv;
}
```

```
/* compare the features and will be later used by CvSeqSort (decreasing-scale ordering)
   return -1 if vice versa and 0 if their scales are equal. */
int feature_compare(struct feature* feature1, struct feature* feature2, void* param){
  struct feature* f1 = (struct feature*) feature1;
  struct feature* f2 = (struct feature*) feature2;

  if( f1->scale < f2->scale )
    return 1;
  if( f1->scale > f2->scale )
    return -1;
  return 0;
}



/* Free memory held by descriptor histogram */
void release_descriptor_histogram( double**** histogram, int width ){
  int i, j;
  // width of the histogram
  for( i = 0; i < width; i++){
    for( j = 0; j < width; j++ )
      free( (*histogram)[i][j] );
    free( (*histogram)[i] );
  }
  free( *histogram );
  *histogram = NULL;
```

```
}



/* free memory held by a scale space pyramid */
void release_pyd( IplImage**** pyd, int octvs, int no_img_per_octv ){
  int i, j;
  for( i = 0; i < octvs; i++ ){
    for( j = 0; j < no_img_per_octv; j++ )
      cvReleaseImage( &(*pyd)[i][j] );
    free( (*pyd)[i] );
    }
  free( *pyd );
  *pyd = NULL;
}

/* find the sift features and return the number of
   keypoints. */
int SIFT_feature(IplImage *img, struct feature** _feature, int *t){
  IplImage *initial_image;
  IplImage*** gaussian_pyd, *** dog_pyd;
  CvMemStorage *memstorage = cvCreateMemStorage(0);
  CvSeq* features;
  int i, total;
  initial_image = initialize_image(img,SIFT_SIGMA);
```

```
  // create gaussian pyramid.
  gaussian_pyd =
create_gaussian_pyd(initial_image,SIFT_INTERVALS,SIFT_SCALES,SIFT_OCTVS,SIFT_SI
GMA);

  // create Difference-of-Gaussian pyramid.
  dog_pyd = create_dog_pyd(gaussian_pyd,SIFT_DOG_SCALES,SIFT_OCTVS);

  // scale space detection.
  features =
detect_scale_space_extrema(dog_pyd,SIFT_OCTVS,SIFT_INTERVALS,SIFT_CONTRAST_T
HRESHOLD,SIFT_CURVATURE_THRESHOLD,memstorage);

  // total of keypoints before calculate different orientations for the same location.
  *t = features->total;
  // find the feature scale
  calculate_feature_scales(features, SIFT_SIGMA, SIFT_INTERVALS);

  // orientation assignmnet.
  calculate_feature_orientation(features,gaussian_pyd);
  // calculate descriptors
  compute_descriptors( features, gaussian_pyd, SIFT_DESCRIPTOR_WIDTH,
SIFT_DESCRIPTOR_HISTOGRAM_BINS );

  /* sort features in decreasing-scale oriding and move from CvSeq to array */
```

```
  cvSeqSort( features, (CvCmpFunc)feature_compare, NULL );
  total = features->total;
  *_feature = (struct feature*)calloc(total, sizeof(struct feature));
  *_feature = (struct feature*)cvCvtSeqToArray(features, *_feature, CV_WHOLE_SEQ );
  for( i = 0; i < total; i++ ){
    free( (*_feature)[i].feature_data );
    (*_feature)[i].feature_data = NULL;
  }

  cvReleaseMemStorage( &memstorage );
  cvReleaseImage( &initial_image );
  release_pyd( &gaussian_pyd, SIFT_OCTVS, SIFT_SCALES );
  release_pyd( &dog_pyd, SIFT_OCTVS, SIFT_DOG_SCALES );
  return total;
}
```

Appendix C – Kdstruct.c

/** kdimensional-tree is used for organizing points in a k-dimensional space. it's useful for range searches and nearest neibgbour searches which is the techniques for sift keypoints matching. Section 7 of Lowe's Paper.*/

```
#include <stdio.h>
#include <cxcore.h>
#include <limits.h>
#include "sift.c"

#ifndef parent
#define parent(i) (i-1)/2
#endif
#ifndef left
#define left(i) 2*i+1
#endif
#ifndef right
#define right(i) 2*i+2
#endif

struct kd_node{
  int key_index;
  double key_value;
  int isLeaf;
  struct feature *features;
```

```
  int no_of_features;
  struct kd_node *kd_left;
  struct kd_node *kd_right;
};

// Binary Bin First search.

struct bbfs{
  double d;
  struct kd_node* old_data;
};

// an element in a minimizing priority queue
struct priq_node{
  struct kd_node* data;
  int key;
};



// minimizing priority queue
struct min_priq{
  struct priq_node* pq_array;    /* array containing priority queue */
  int no_of_element;              /* number of elements allocated */
  int no;                    /* number of elements in priq */
};
```

```
struct feature;
/************************ function prototypes ***************************/
struct kd_node* initialize_node(struct feature* features, int no_of_features);
struct kd_node* kd_build(struct feature* features, int no_of_features);
int nearest_neighbour(struct kd_node* root, struct feature* _feature, int k, struct feature
***neighbours, int maxnnbr);
void build_subtree(struct kd_node* node);
void key_assignment( struct kd_node* node );
void partition_features( struct kd_node* node );
int partition_array( double* array, int n, double pivot );
void release_tree(struct kd_node* root);

struct min_priq* initialize();
int minpq_insert( struct min_priq* minpq, struct kd_node* data, int key );


struct kd_node* explore_to_leaf( struct kd_node* node, struct feature* feature,struct min_priq*
min_pq );
int insert_into_nbr_array( struct feature* feature, struct feature** neighbours, int no, int maxe );
double descriptor_distance_squard( struct feature* feat1, struct feature* feat2 );
double median_select( double* array, int n );
double rank_select( double* array, int n, int r );
void insertion_sort( double* array, int n );

struct min_priq* initialize_minpq();
```

```
int double_array(struct priq_node** array, int no, int size);
void decrease_pq_key(struct priq_node* pq_array, int i, int key);
struct kd_node* minpq_extract_min( struct min_priq* min_pq );
void restore_minpq_order( struct priq_node* pq_array, int i, int n );
void minpq_release( struct min_priq** min_pq );
/*****************************************************************/

// initialize a k dimensional tree node with a number of features
struct kd_node* initialize_node(struct feature* features, int no_of_features){
  struct kd_node* node;
  node = (struct kd_node*)malloc(sizeof(struct kd_node));
  memset(node, 0, sizeof(struct kd_node));
  node->key_index = -1;
  node->features = features;
  node->no_of_features = no_of_features;
  return node;
}
// build k-dimensional tree database from the array of keypoints.
struct kd_node* kd_build(struct feature* features, int no_of_features){
  struct kd_node* root;
  // create root and set properties.
  root = initialize_node(features,no_of_features);
//   printf("after initil node\n");
  build_subtree(root);
//   printf("after spread_subtree\n");
```

```
  return root;
}
//
void build_subtree(struct kd_node* node){
  if (node->no_of_features == 1 || node->no_of_features == 0){
    node->isLeaf = 1;
    return;
  }
  key_assignment(node);
  partition_features( node );
//   printf("partition_features finished\n");
  if (node->kd_left)
    build_subtree(node->kd_left);
  if (node->kd_right)
    build_subtree(node->kd_right);
}

// detect the value at the descriptor index to partition a k dimensional tree nodes' features.
void key_assignment( struct kd_node* node ){
  struct feature* features;
  double keyvalue, x, mean, var, maxvar = 0;
  int descriptor_len, no_of_features, i, j, keyindex = 0;
  double* tmp;

  features = node->features;
```

```
  no_of_features = node->no_of_features;
  descriptor_len = features[0].descriptor_length;

  /* partition key index */
  for( j = 0; j < descriptor_len; j++ ){
    mean = var = 0;
    for( i = 0; i < no_of_features; i++ )
      mean += features[i].descriptor[j];
    mean /= no_of_features;
    for( i = 0; i < no_of_features; i++ ){
      x = features[i].descriptor[j] - mean;
      var += x * x;
    }
    var /= no_of_features;
    if( var > maxvar ){
      keyindex = j;
      maxvar = var;
    }
  }

  /* partition key value */
  tmp = (double*)calloc(no_of_features,sizeof(double));
  for( i = 0; i < no_of_features; i++ )
    tmp[i] = features[i].descriptor[keyindex];
//   printf("last loop finished\n");
```

```
  keyvalue = median_select( tmp, no_of_features );
//   printf("median_select finished\n");
  free( tmp );
//   printf("free finished\n");
  node->key_index = keyindex;
  node->key_value = keyvalue;
}

//Finds the median value of an array.  The array's elements are re-ordered
//  by this function.

double median_select( double* array, int n ){
  return rank_select( array, n, (n - 1) / 2 );
}



//Finds the element of a specified rank in an array using the linear time median-of-medians
algorithm


double rank_select( double* array, int n, int r ){
  double* tmp, med;
  int gr_5, gr_tot, rem_elts, i, j;
```

```
/* base case */
if( n == 1 )
  return array[0];

/* divide array into groups of 5 and sort them */
gr_5 = n / 5;
gr_tot = cvCeil( n / 5.0 );
rem_elts = n % 5;
tmp = array;
for( i = 0; i < gr_5; i++ )
  {
    insertion_sort( tmp, 5 );
    tmp += 5;
  }
insertion_sort( tmp, rem_elts );

/* recursively find the median of the medians of the groups of 5 */
tmp = (double*)calloc( gr_tot, sizeof( double ) );
for( i = 0, j = 2; i < gr_5; i++, j += 5 )
  tmp[i] = array[j];
if( rem_elts )
  tmp[i++] = array[n - 1 - rem_elts/2];
med = rank_select( tmp, i, ( i - 1 ) / 2 );
free( tmp );
```

```
  /* partition around median of medians and recursively select if necessary */
  j = partition_array( array, n, med );
  if( r == j )
    return med;
  else if( r < j )
    return rank_select( array, j, r );
  else
    {
      array += j+1;
      return rank_select( array, ( n - j - 1 ), ( r - j - 1 ) );
    }
}

// Partitions an array around a specified value.

int partition_array( double* array, int n, double pivot )
{
  double tmp;
  int p, i, j;

  i = -1;
  for( j = 0; j < n; j++ )
    if( array[j] <= pivot )
      {
            tmp = array[++i];
```

```
            array[i] = array[j];
            array[j] = tmp;
            if( array[i] == pivot )
              p = i;
        }
  array[p] = array[i];
  array[i] = pivot;

  return i;
}
/* create two children for a specified tree node by
   partitioning the features*/

void partition_features( struct kd_node* node ){
  struct feature* features, tmp;
  double keyvalue;
  int no_of_features, keyindex, p, i, j = -1;

  features = node->features;
  no_of_features = node->no_of_features;
  keyindex = node->key_index;
  keyvalue = node->key_value;
  for( i = 0; i < no_of_features; i++ ){
    if(features[i].descriptor[keyindex] <= keyvalue ){
      tmp = features[++j];
```

```
    features[j] = features[i];
    features[i] = tmp;
    if( features[j].descriptor[keyindex] == keyvalue )
    p = j;
   }
  }
  tmp = features[p];
  features[p] = features[j];
  features[j] = tmp;

  /* if all records fall on same side of partition, make node a leaf */
  if( j == no_of_features - 1 ){
   node->isLeaf = 1;
   return;
  }

  node->kd_left = initialize_node( features, j + 1 );
  node->kd_right = initialize_node( features + ( j + 1 ), ( no_of_features - j - 1 ) );
}
```

//Finds an image feature's approximate k nearest neighbors in a kd tree using Best Bin First search.

```
int nearest_neighbour( struct kd_node* root, struct feature* _feature, int k, struct feature***
neighbours, int maxnnbr ){
  struct kd_node* expl;
  struct min_priq* minpq;
  struct feature* tree_feature, ** _neighbours;
  struct bbfs* _bbfs;
  int i, t = 0, n = 0;

  if( ! neighbours  ||  ! _feature  ||  !root ){
    return -1;
  }

  _neighbours = (struct feature**)calloc( k, sizeof( struct feature* ) );
  minpq = initialize_minpq();
  minpq_insert( minpq, root, 0 );
  while( minpq->no > 0  &&  t < maxnnbr ){
    expl = (struct kd_node*)minpq_extract_min( minpq );
    if( ! expl ){
      minpq_release( &minpq );
      for( i = 0; i < n; i++ ){
          _bbfs = (struct bbfs*)_neighbours[i]->feature_data;
          _neighbours[i]->feature_data = _bbfs->old_data;
          free( _bbfs );
      }
      free( _neighbours );
```

```
  *neighbours = NULL;
  return -1;
}

expl = explore_to_leaf( expl, _feature, minpq );
if(!expl){
  minpq_release( &minpq );
  for( i = 0; i < n; i++ ){
        _bbfs = (struct bbfs*)_neighbours[i]->feature_data;
        _neighbours[i]->feature_data = _bbfs->old_data;
        free( _bbfs );
  }
  free( _neighbours );
  *neighbours = NULL;
  return -1;
}

for( i = 0; i < expl->no_of_features; i++ ){
  tree_feature = &expl->features[i];
  _bbfs = (struct bbfs*)malloc( sizeof( struct bbfs ) );
  if( !_bbfs ){
        minpq_release( &minpq );
        for( i = 0; i < n; i++ ){
          _bbfs = (struct bbfs*)_neighbours[i]->feature_data;
          _neighbours[i]->feature_data = _bbfs->old_data;
```

```
            free( _bbfs );
          }
          free( _neighbours );
          *neighbours = NULL;
          return -1;
        }
        _bbfs->old_data = (struct kd_node*)tree_feature->feature_data;
        _bbfs->d = descriptor_distance_squard(_feature, tree_feature);
        tree_feature->feature_data = _bbfs;
        n += insert_into_nbr_array( tree_feature, _neighbours, n, k );
      }
      t++;
    }

  minpq_release( &minpq );
  for( i = 0; i < n; i++ ){
    _bbfs = (struct bbfs*) _neighbours[i]->feature_data;
    _neighbours[i]->feature_data = _bbfs->old_data;
    free( _bbfs );
  }
  *neighbours = _neighbours;
  return n;
}

// explores a k dimensional tree from a given node to a leaf.
```

```
struct kd_node* explore_to_leaf( struct kd_node* node, struct feature* feature,struct min_priq*
min_pq ){
  struct kd_node* unexpl, * expl = node;
  double keyvalue;
  int keyindex;

  while( expl  &&  ! expl->isLeaf ){
    keyindex = expl->key_index;
    keyvalue = expl->key_value;
    if( keyindex >= feature->descriptor_length ){
      return NULL;
    }
    if( feature->descriptor[keyindex] <= keyvalue ){
      unexpl = expl->kd_right;
      expl = expl->kd_left;
    }else{
      unexpl = expl->kd_left;
      expl = expl->kd_right;
    }
    if( minpq_insert( min_pq, unexpl, ABS( keyvalue - feature->descriptor[keyindex] ) ) ){
      return NULL;
    }
  }
  return expl;
}
```

```
// inserts a feature into the nearest-neighbour array.
int insert_into_nbr_array( struct feature* feature, struct feature** neighbours, int no, int maxe ){
  struct bbfs* fdata, * ndata;
  double dn, df;
  int i, ret = 0;

  if( no == 0 ){
    neighbours[0] = feature;
    return 1;
  }

  /* check at end of array */
  fdata = (struct bbfs*)feature->feature_data;
  df = fdata->d;
  ndata = (struct bbfs*)neighbours[no-1]->feature_data;
  dn = ndata->d;
  if( df >= dn ){
    if( no == maxe ){
      feature->feature_data = fdata->old_data;
      free( fdata );
      return 0;
    }
    neighbours[no] = feature;
    return 1;
```

```
  }

  /* find the right place in the array */
  if( no < maxe ){
    neighbours[no] = neighbours[no-1];
    ret = 1;
  }else{
    neighbours[no-1]->feature_data = ndata->old_data;
    free( ndata );
  }
  i = no-2;
  while( i >= 0 ){
    ndata = (struct bbfs*)neighbours[i]->feature_data;
    dn = ndata->d;
    if( dn <= df ) break;
    neighbours[i+1] = neighbours[i];
    i--;
  }
  i++;
  neighbours[i] = feature;

  return ret;
}

void release_tree(struct kd_node* root){
```

```
  if(!root) return;
  release_tree(root->kd_left);
  release_tree(root->kd_right);
  free(root);
}
//Calculates the squared Euclidian distance between two feature descriptors.

double descriptor_distance_squard( struct feature* feat1, struct feature* feat2 ){
  double diff, dsq = 0.0;
  double* descr1, * descr2;
  int i, d;

  d = feat1->descriptor_length;
  if( feat2->descriptor_length != d )
    return DBL_MAX;
  descr1 = feat1->descriptor;
  descr2 = feat2->descriptor;

  for( i = 0; i < d; i++ ){
    diff = feat1->descriptor[i] - feat2->descriptor[i];
    dsq += diff*diff;
//     if(descr1[i]!= 0.0)
//       printf("feat1->descr = %f, feat2->descr = %f\n",descr1[i],descr2[i]);
  }
```

```
//   printf("dis = %f\n",dsq);
  return dsq;
}



//Sorts an array in place into increasing order by using insertion sort.


void insertion_sort( double* array, int n ){
  double k;
  int i, j;


  for( i = 1; i < n; i++ )
    {
     k = array[i];
     j = i-1;
     while( j >= 0  &&  array[j] > k )
            {
             array[j+1] = array[j];
             j -= 1;
            }
     array[j+1] = k;
    }
}


/******************** minimising priority queue *************************/
```

```
// initialize priority queue.
struct min_priq* initialize_minpq(){
  struct min_priq* minpq;
  minpq = (struct min_priq*)malloc(sizeof(struct min_priq));
  minpq->pq_array = (struct priq_node*)calloc(512,sizeof(struct priq_node));
  minpq->no_of_element = 512;
  minpq->no = 0;
  return minpq;
}

// inserts an element into a minimizing priority queue.
int minpq_insert( struct min_priq* minpq, struct kd_node* data, int key ){
  int no = minpq->no;

  /* double array allocation if necessary */
  if( minpq->no_of_element == no ){
     minpq->no_of_element = double_array( &minpq->pq_array, minpq-
>no_of_element,sizeof( struct priq_node ) );
     if(!minpq->no_of_element){
          return 1;
     }
  }

  minpq->pq_array[no].data = data;
```

```
  minpq->pq_array[no].key = INT_MAX;
  decrease_pq_key( minpq->pq_array, minpq->no, key );
  minpq->no++;

  return 0;
}
// double the array size.

int double_array(struct priq_node** array, int no, int size){
  struct priq_node* temp;
  temp = (struct priq_node*)realloc(*array, 2 * no * size);
  if(!temp){
    if(*array) free(*array);
    *array = NULL;
    return 0;
  }
  *array = temp;
  return no*2;
}



/* decrease a min priority queue element key
   i is element index at which key need to be decreased.
*/
```

```
void decrease_pq_key(struct priq_node* pq_array, int i, int key){
  struct priq_node temp;

  if( key > pq_array[i].key )
    return;

  pq_array[i].key = key;
  while( i > 0  &&  pq_array[i].key < pq_array[parent(i)].key ){
    temp = pq_array[parent(i)];
    pq_array[parent(i)] = pq_array[i];
    pq_array[i] = temp;
    i = parent(i);
  }
}

// removes and returns the element of a mini pq with smllest key.
struct kd_node* minpq_extract_min( struct min_priq* min_pq ){
  struct kd_node* data;

  if( min_pq->no < 1 ){
    return NULL;
  }
  data = min_pq->pq_array[0].data;
  min_pq->no--;
```

```
  min_pq->pq_array[0] = min_pq->pq_array[min_pq->no];
  restore_minpq_order( min_pq->pq_array, 0, min_pq->no );

  return data;
}

// restores correct pq order recursively to a mini pq array.
void restore_minpq_order( struct priq_node* pq_array, int i, int n ){
  struct priq_node tmp;
  int l, r, min = i;

  l = left( i );
  r = right( i );
  if( l < n )
    if( pq_array[l].key < pq_array[i].key )
      min = l;
  if( r < n )
    if( pq_array[r].key < pq_array[min].key )
      min = r;

  if( min != i ){
    tmp = pq_array[min];
    pq_array[min] = pq_array[i];
    pq_array[i] = tmp;
    restore_minpq_order( pq_array, min, n );
```

```
  }
}

// free memory.
void minpq_release( struct min_priq** min_pq ){
  if( ! min_pq ){
    return;
  }
  if(*min_pq && (*min_pq)->pq_array ){
    free( (*min_pq)->pq_array );
    free( *min_pq );
    *min_pq = NULL;
  }
}
/*****************************************************************************/
```