DEPARTMENT OF COMPUTER SCIENCE

A SIFT Tool

Xiaosong Wang

A dissertation submitted to the University of Bristol in accordance with the requirements of the degree of Master of Science in the Faculty of Engineering

---

September 2006 | CSMSC-06

Abstract

The Aim of this project is to produce a fully comprehensive software library which implements previous work by David G. Lowe on the subject of the Scale Invariant Feature Transform (SIFT). As background knowledge, this dissertation presents a brief introduction of image matching and a thorough review of image feature detectors and descriptors. Furthermore, documents produced in every stage of software development life circle are also included, such as requirement specification, design specification, testing, user guide and so on. Eventually, the SIFT library is critically evaluated and demonstrated by using images under different transformations and an object recognition system. The results of evaluation prove the distinctive characteristics of SIFT and the achievements of our implementation.

Acknowledgements

I would like to express my deep gratitude to Dr Walterio Mayol-Cuevas and Dr Majid Mirmehdi for their continuing support, helpful advice and constant encouragement through all stages of the project.

I also wish to appreciate my parents. Without their finical support and encouragement, I could not have finished this course and the project.

Declaration

A dissertation submitted to the University of Bristol in accordance with the requirements of the degree of Master of Science in the Faculty of Engineering. It has not been submitted for any other degree or diploma of any examining body. Except where specifically acknowledged, it is all the work of the Author.

Xiaosong Wang, September 2006

Table of Contents

1 Introduction and context

1.1 What is image matching?

Image matching, also referred to as image correspondence, plays an important role in many aspects of computer vision, for instance, objects recognition, image retrieval, stereo correspondence, building panoramas and so on. Since Hobrough presented the first solutions in 1959 [35], image-matching problems have attracted a great number of people to work on this area. However, even now the problem cannot be solved thoroughly.

Two main problems prevent us from making progress. First, the number of elementary primitives in an image is large. There could be thousands, even millions of pixels in one single image. Although nowadays the capacity of computers is expending exponentially, the processing time is not satisfactory, particularly for real time propose. The other problem is the possible variance between matching image pairs. Translations, rotations, scales and luminance changes can cause the difference of two pictures. It is virtually impossible to compare two images using traditional methods such as a direct comparison between grey values.

Two methods enabling a more reliable comparison have been developed: correlation-based methods and feature-based methods [37]. The correlation-based methods still involve all the pixels in images but all pixels will be grouped as windows with certain sizes. On the contrary, feature-based methods just focus on sparse sets of features.

1.1.1 Correlation-based methods

As implied above, correlation-based methods are based on measuring the correlation of windows from two images. The size of the window and the search region should be determined before the comparison and they all influence the efficiency and effectiveness of algorithms. Generally, windows are extracted one by one as template windows from one image. Then each template window will be compared against the entire search region in the other image and the similarity between two windows will be assessed.

Cross correlation [36] is the simplest example of correlation-based methods. The similarity between images is measured by cross correlation coefficients P:

$$P = \frac{\sum_{r=1}^{R}\sum_{c=1}^{C}(g_1(r,c)-\mu_1)(g_2(r,c)-\mu_2)}{\sqrt{\sum_{r=1}^{R}\sum_{c=1}^{C}(g_1(r,c)-\mu_1)^2\sum_{r=1}^{R}\sum_{c=1}^{C}(g_2(r,c)-\mu_2)^2}}; \qquad -1 \le P \le 1 \qquad (1)$$

g1(r, c) grey values of pixels in template window
u1 average of grey values in template window
g2(r, c) grey values of pixels in corresponding window from search region
u2 average of grey values in corresponding window from search region
R, C row and column numbers of template window

As the search region expends, the algorithm may suffer from over-computation and more ambiguous solutions may arise. More reliable methods have been introduced and are discussed in the following part.

1.1.2 Feature-based methods

Feature-based methods narrow down the search for correspondence to some sparse sets of features. Local features are extracted from images by feature detectors before the matching process. Features could be points, edges, lines, and corners. Unlike correlation-based methods, local descriptors such as SIFT descriptor [4] are created to represent these features instead of image windows when we measure the similarity of features. In section 3, we will discuss feature detectors and feature descriptors in detail.

1.2 What is SIFT?

Features such as points and edges may change dramatically after image transformations, for example, after scaling and rotation. As a result, recent works have concentrated on detecting and describing image features that are invariant to these transformations. Scale Invariant Feature Transform (SIFT) [4] is a feature-based image matching approach, which lessens the damaging effects of image transformations to a certain extent. Features extracted by SIFT are invariant to image scaling and rotation, and partially invariant to photometric changes. SIFT mainly covers 4 stages throughout the computation procedure as follows:

1. Local extremum detection: first, use difference-of-Gaussian (DOG) to approximate Laplacian-of-Gaussian and build the image pyramid in scale space. Determine the keypoint candidates by local extremum detection.
2. Strip unstable keypoints: use the Taylor expansion of the scale-space function to reject those points that are not distinctive enough or are unsatisfactorily located near the edge.
3. Feature description: Local image gradients and orientations are computed around keypoints. A set of orientation, scale and location for each keypoint is used to represent it, which is significantly invariant to image transformations and luminance changes.
4. Feature matching: compute the feature descriptors in the target image in advance and store all the features in a shape-indexing feature database. To initiate the matching process for the new image, repeat steps 1-3 above and search for the most similar features in the database.

Each stage throughout the procedure will be presented in detail in section 3, including normalizing the orientation of a descriptor.

1.3 Why implement SIFT?

Feature-based methods have demonstrated tremendous success in a wide range of

applications, such as object recognition [6, 38, 39, 40], mobile robot localization [42, 43], building panoramas [44], image watermarking [41], and so on. SIFT has been shown to be a valuable tool in all these areas of application. Mikolajczyk and Schmid`s evaluation [15] has pointed out that the performance of SIFT has exceeded other local descriptors.

There are two distinctive characteristics that make SIFT increasingly popular in terms of quality and efficiency. First, keypoints extracted by SIFT are highly distinctive so that they are invariant to image transformation and partially invariant to illumination and camera viewpoint changes. Second, in using difference-of-Gaussian, which approximates Laplacian-of-Gaussian, the speed of algorithmic computations is enhanced, at the meantime producing great number of features. These two outstanding characteristics make SIFT popular and it has proved to be a success in various applications. Figure 1 shows an example result of object recognition system by using SIFT. Since rarely comprehensive implementation of SIFT software library is available now, this tool will be extremely useful and it ought to be reused in all kinds of applications, especially in the projects from the vision group in Computer Science department.

Figure 1: Two target objects are shown on the left. The photo in the middle is an image with cluttered objects in it, including the three target objects. The recognition result is illustrated on the right. Even though the target objects have been replaced in different poses and some parts of them have been hidden, the system still can recognize the target objects.

1.4 Organization of this dissertation

This dissertation includes all essential documents in the entire life circle of the project and an evaluation on the performance of the SIFT library.

Section 2 analyses and specifies the requirements of both the SIFT library and object recognition system.

Section 3 is a detailed description of SIFT and a review on feature-based image matching methods. It will help the reader with a better understanding the algorithm of SIFT and other methods in image matching field. In the end, previous implementations of SIFT library are also introduced.

In Section 4, 5 and 6, the methodology used in the project, design and implementation of the software are discussed individually.

Section 7 describes the testing of the software carried out not only on every module but on the whole object recognition system as well.

Section 8 is one of the key parts of this thesis, a comprehensive evaluation of the SIFT library and the object recognition system is presented. Experiment results are categorised by the transformations applied on the testing image sequences.

The last part, section 9, is a thoughtful user guide that will illustrate the usage of the SIFT library and contain a few simple instructions on detection system.

2 Requirement specification

2.1 Software prospective

The main purpose of the project is to implement a comprehensive SIFT library, including a keypoint detector, a descriptor and a keypoint matching module. SIFT library is a software library which implements David G. Lowe`s Scale Invariant Feature Transformation approach [4]. It allows the user to call or reuse the functions in the library to extract and represent the SIFT keypoints in an image. The user only needs to specify the filename of an image and then a list of scale invariant keypoints and their descriptors, 128-D vectors, will be presented. Furthermore, the library also provides a simple image-matching function, which measures the similarity of two images by comparing the keypoint descriptors. The SIFT library is required to be well designed and documented so that it could be subsequently reused in all kinds of applications and research work without hindrance.

The second aim of this project is to develop an object recognition system based on SIFT library in order to fully demonstrate the distinctive characteristics of SIFT. The system realizes a simple object recognition algorithm, which is able to handle detecting multiple objects in a scene image.

2.2 Functional requirement

1. Keypoint detection

Detect keypoint location that is partially invariant to rotation, translation and scaling. Given an image, the keypoint detector ought to output a list of points with x and y coordinates, orientation of the keypoint and scale of the image on which the keypoint is detected.

2. Keypoint description

Use local image information to build the feature vector of the keypoint from keypoint detector, which is invariant to rotation, linear and non-linear illumination change. The descriptor is supposed to output a list of keypoints with their feature vectors in a structured form, which can be printed out or used by application programs.

3. Matching module

Compute the Euclidean distance of two keypoints` feature vector to measure the similarity of them. Given two sets of keypoints from different images, the matching module should output a list of possible matching pairs according to different thresholds.

4. Object detection

Detect target objects (given as object images) in a scene (given as a scene image) and outline the objects over the scene image. Given the file names of object images and scene image, the detector ought to output an image with detection result on it.

2.3 Non-functional requirement

1. Image input / output

SIFT is designed for grey-scale images so that all input images should be grey-scale ones in PGM form. Other type of images, including colour images are acceptable once a conversion step is added before a PGM image is processed.

All output images from matching module or object recognition system will be in the form of PGM.

2. Software environment

The SIFT library will be developed under the Intel OpenCV library [46], calling basic functions from it such as reading and creating images and following its coding and documentation style [45]. Since OpenCV library is mainly programmed in C under Linux, the SIFT library will mainly be developed under Linux. However, once the source code of library has been finished, a DLL library also can be produced under Windows operation system.

Once the library file has been provided in the form of .a file (under Linux) or .dll file (under Windows), it`s not necessary for application program to include OpenCV library when being compiled.

3. Programming language

Because OpenCV library is mainly programmed in C/C++, the SIFT library and objec recognition system will be implemented according to ANSI C standard. Furthermore, since the speed is one of the key points to judge the implementation quality of the SIFT library and C is considerably fast and suitable for all kinds of operation systems, it proved to be the best choice for this project.

4. Library compilation

All the source code should be compiled into a software library file as an output. This library file can be used by applications under any software environments, namely .a file under Linux and .dll file under windows.

5. Hardware environment and development tools

Although the SIFT library will be developed on a PC with Linux operation system, the library is supposed to be reused under both Linux and Windows system. So Eclipse, a java based integrated development environment, will be used to develop SIFT library and the demonstration system. Furthermore, Eclipse also includes the CVS that will be really helpful for us to manage the source code and documents.

2.4 Summary

In this part, requirements are listed and specified in two catalogues, functional and non-functional requirements. Details of the project will be stated in the design part.

3 Background

3.1 Feature detection

Feature-based methods have played an important role in image matching as well as in other application areas as they are invariant to scale and affine transformations. In the past few years, quite a few region detecting approaches that are covariant to a class of transformations have been developed. First, Harris [2] developed a derivative based detector for edge and corner detection by measuring the trace and determinant of the gradient distribution matrix around interest points. Mikolajczyk and Schmid [3, 5, 8] used Harris function and Hessian Matrix to locate points in 2D and then obtain maxima as keypoints by Laplacian operator selection in multi-scale space. Regions detected by these methods were named *Harris-Laplace* and *Hessian-Laplace* regions. A similar idea was explored by Lowe [4, 6] who used Difference-of-Gaussian to approximate Laplacian-of-Gaussian. Lindeberg [1] designed a blob detector by using Laplacian-of-Gaussian and several other derivative based operators and then, working together with Garding [7], made the blob detector invariant to affine transformations by using *affine adaptation* process. Mikolajczyk and Schmid [5, 8] applied *affine adaptation* process to their *Harris-Laplace* and *Hessian-Laplace* detectors and created *Harris-Affine* and *Hessian-Affine* detectors that are affine-invariant. Similar detectors were developed by Baumberg [9] as well as Schaffalitzky and Zisserman [10]. In spite of using Gaussian derivatives, other detectors were developed by Tuytelaars and Van Gool [11] based on edge and intensity-extrema, by Kadir et al. [13] based on entropy of pixel intensity and by Metas et al. [12] based on stable extremal regions.

3.1.1 Gaussian derivative based detectors

The generation of a scale-space requires a Gaussian Kernel, represented by $G(x, y, \sigma)$. Different levels of resolutions are generally created by convolution with Gaussian Kernel:

$$L(x, y, \sigma) = G(x, y, \sigma) * I(x, y), \tag{2}$$

where * is the convolution operator and $I(x, y)$ is the intensity of input image, and

$$G(x, y, \sigma) = \frac{1}{2\pi\sigma^2} \exp(-\frac{x^2 + y^2}{2\sigma^2}), \tag{3}$$

Lindeberg [1] pointed out that Gaussian Kernel is the only possible kernel for scale-space representation.

1. Decide Differential Expressions in 2D:

The first step of feature detection is to locate points in different scales that are invariant to rotation. Several derivative based differential expressions have been created as follows. Note that all expressions are scale normalized:

Harris function $\det(M) - \alpha trace^2(M)$ (4)

Hessian matrix $\det(H) - \alpha trace^2(H)$ (5)

$$M = \mu(x, y, \sigma_1, \sigma_D) = \begin{bmatrix} \mu_{11} & \mu_{12} \\ \mu_{21} & \mu_{22} \end{bmatrix} = \sigma_D^2 G(x, y, \sigma_1) * \begin{bmatrix} I_x^2(x, y, \sigma_D) & I_x I_y(x, y, \sigma_D) \\ I_x I_y(x, y, \sigma_D) & I_y^2(x, y, \sigma_D) \end{bmatrix} \quad (6)$$

$$H = H(x, y, \sigma_D) = \begin{bmatrix} h_{11} & h_{12} \\ h_{21} & h_{22} \end{bmatrix} = \begin{bmatrix} I_{xx}(x, y, \sigma_D) & I_{xy}(x, y, \sigma_D) \\ I_{xy}(x, y, \sigma_D) & I_{yy}(x, y, \sigma_D) \end{bmatrix} \quad (7)$$

Laplacian $\left|\sigma^2 (L_{xx}(x, y, \sigma) + L_{yy}(x, y, \sigma))\right|$ (8)

Difference $\left|L(x, y, k\sigma) - L(x, y, \sigma)\right|$ (9)

Harris function was firstly developed by Harris [2], and then it was used in *Harris-Laplace* and *Hessian-Laplace* [3, 5, 8]. *Laplacian* was used by Linderberg [1] in blob detector. Lowe [4, 6] applied Difference-of-Gaussian in SIFT.

2. Determine Extrema in scale-space:

The most common way to detect local extrema is to compare the interest point with its eight neighbors in the current images and 9 neighbors in each adjacent scale: 26 neighbors in total. Linderberg [1] and Lowe [4, 6] used this method in blob detector and SIFT. Figure 2 [4] illustrates such a detection with reference to a sample point.

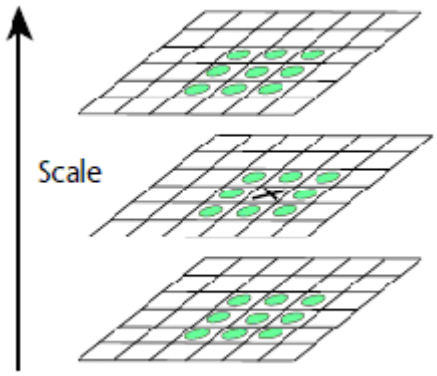

Figure 2: Local extrema are determined by comparing the point with its 26 neighbors marked with circles in the figure. (Image taken from [4])

A similar method has been used in *Harris-Laplace* and *Hessian-Laplace* detectors, which locate the interest points by using the trace and determinant of Harris function or Hessian Matrix and then select the local extrema by using Laplacian differential expressions. That is because Laplacian expression is proved to be able to extract more candidate points correctly by experiments [3].

All the method introduced so far is just scale-invariant and none of them is robust against affine transformation. So Lindeberg and Garding [7] developed a process named *affine*

adaptation that is based on the second moment matrix. Mikolajczyk and Schmid [5, 8] applied this method to their *Harris-Laplace* and *Hessian-Laplace* detectors and created *Harris-Affine* and *Hessian-Affine* detectors. Baumberg [9] as well as Schaffalitzky and Zisserman [10] presented some similar detectors in the meantime. The *affine adaptation* process will be introduced in the following part.

3. Affine adaptation:

The main idea of *affine adaptation* is to execute an affine transformation from the affine scale space to the uniform scale space over the second moment matrix that is suitable to estimate the anisotropic shape of a local image structure [7, 9]. Then corresponding points and regions are only differentiable by an arbitrary rotation in 2D.

The second moment matrix is defined in the affine scale space in the following equation:

$$\mu(x, \Sigma_I, \Sigma_D) = \det(\Sigma_D) g(\Sigma_I) * ((\nabla L)(x, \Sigma_D)(\nabla L)(x, \Sigma_D)^T) \quad (10)$$

where x=(x, y) and Σ_I, Σ_D are the covariance matrices of integration and differentiation Gaussian operators. With little loss of generality, the number of degrees of freedom is narrowed down by setting $\Sigma_I = s\Sigma_D$, where s is a scalar.

Two points x_L, x_R are extracted from two images L and R, each point having a corresponding μ_R, μ_L. This relationship is shown in the following equation with an affine transformation matrix A, represented by $x_R = Ax_L$:

$$\mu(x_L, \Sigma_{I,L}, \Sigma_{D,L}) = A^T \mu(x_R, \Sigma_{I,R}, \Sigma_{D,R}) A = A^T \mu(Ax_L, A\Sigma_{I,L}A^T, A\Sigma_{D,L}A^T) A \quad (11)$$

Lindeberg and Garding [7] presented that if μ_L verifies

$$M_L = \mu(x_L, \Sigma_{I,L}, \Sigma_{D,L}), \quad \Sigma_{I,L} = \sigma_I M_L^{-1}, \quad \Sigma_{D,L} = \sigma_D M_L^{-1} \quad (12)$$

and the counterpart of point x_R verifies

$$M_R = \mu(x_R, \Sigma_{I,R}, \Sigma_{D,R}), \quad \Sigma_{I,R} = \sigma_I M_R^{-1}, \quad \Sigma_{D,R} = \sigma_D M_R^{-1} \quad (13)$$

then M_L, M_R are related by

$$M_L = A^T M_R A \quad A = M_R^{-1/2} R M_L^{1/2} \quad \Sigma_R = A\Sigma_L A^T \quad (14)$$

where R is an arbitrary rotation (see Figure 3 [14]).

Mikolajczyk and Schmid [5, 8] applied the above process to *Harris-Laplace* and *Hessian-Laplace* (respectively named *Harris-Affine* and *Hessian-Affine* detectors) during which an iterative algorithm was developed to locate affine-invariant points and regions accurately.

3.1.2 Other techniques based detectors

Besides Gaussian derivatives, there are many other techniques used to detect scale or affine invariant points and regions.

Matas et al. [12] applied a watershed like segmentation algorithm on region detection and created Maximally Stable Extremal Regions, which are connected components after appropriate

thresholding. Those regions are invariant to photometric changes and all the points in the region will remain in its original region after continuous geometry transformations.

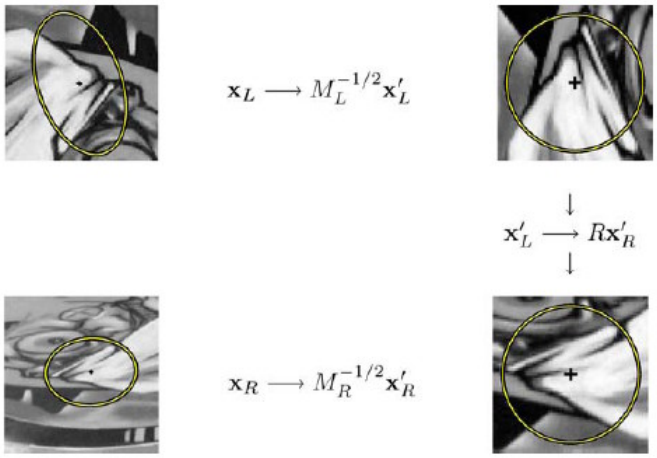

Figure 3: the top row shows the affine normalization of x_L, and the bottom row shows the affine normalization of x_R. These two representations x'_L, x'_R in the uniform scale space were related by an arbitrary rotation. (Image taken from [14])

Tuytelaars and Van Gool [11] constructed two affine invariant detectors that are based on edges and intensity-extrema respectively. The edge based region detectors starts with a Harris corner point and includes another two points that describe the shape of this corner. The other one is based on the image intensities. Tuytelaars created an intensity function and computed all the values of this function along the rays from a local extremum. The regions will be closed by the maxima points, which exhibit intensity changes in their grey values, located along the rays.

Kadir et al. [13] developed a region detection method based on calculating the entropy of probability in an ellipse region centred on the interest point and selecting the regions by ranking the strength of the derivative of probability.

In section 3.1, we introduced some important existing scale or affine invariant detectors categorized by the techniques used. SIFT is one of the simple detectors available yet invariant to translation, rotation and scale and is extremely useful in practice [15].

3.2 Feature representation

After pin-pointing the interested points and regions, descriptors are needed to represent and characterize these regions and take advantage of these regions` distinctive features. Current researches have focused on developing descriptors invariant to a class of image transformations. Numerous descriptors have been constructed which are based on a wide range of techniques, such as Gaussian derivatives [4], Gabor wavelet [22], moment invariants [25] and so on. The following sections will further explore these descriptors.

3.2.1 Gradient distribution based descriptors

One of the typical examples of gradient distribution based descriptors is SIFT which is a 128 dimension feature vector composed of image gradient magnitude and orientation of the points around the keypoints. The main procedure [4] is illustrated in Figure 4 and is as follows:

1. Compute the magnitude and orientation of image gradient of points in a 16*16 pixel region near the keypoint whose level of Gaussian blur is most similar to the scale of the keypoint (after which the descriptor will be invariant to scale). At the same time, the points are weighed by a Gaussian kernel showed as a circle on the left side of Figure 4.
2. Group the elements into 4*4 sub regions, each of which contains 8 arrows pointing towards different directions. The length of an arrow is the sum of the gradient magnitudes of points whose gradient orientation is near that arrow. The descriptor is shown on the right side of Figure 4.
3. Rotate the coordinates of the descriptor according to the orientation of keypoints (after which the descriptor will be invariant to rotation).
4. Normalize the feature vector to a unit vector (after which the descriptor will be invariant to image contrast changes).
5. Threshold all the value larger than 0.2 in the feature vector and normalize the new vector again (after which the descriptor will be partially invariant to photometric changes since the scale change will influence the magnitude of gradients more than the orientation of gradients).

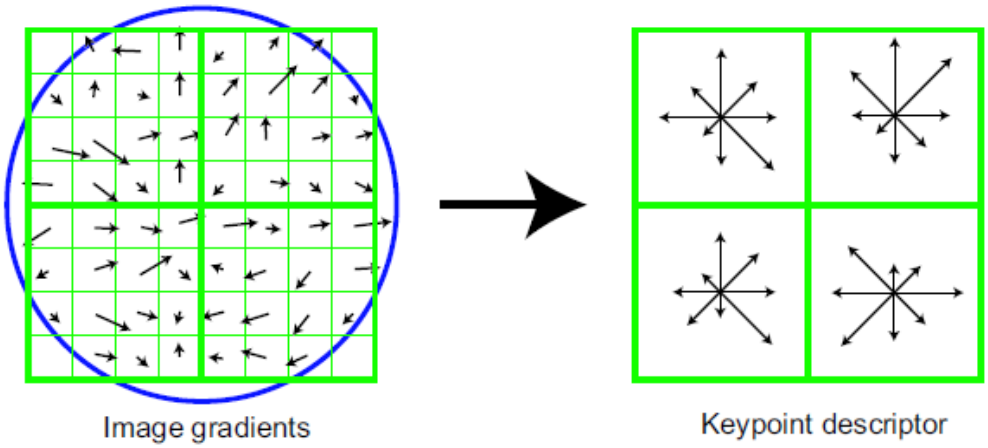

Figure 4: A sample region is shown on the left where magnitude and orientation of gradient need to be computed. The circle represents the range of the Gaussian Window. Part of a sample descriptor is extracted and shown on the right. The original one will contain 8*8 sets of arrows. Each set of arrows is composed of 8 directions and the length of each arrow is related to the sum of magnitudes corresponding to that direction. (Image taken from [4])

Pairwise Geometric Histogram (PGH) [20] and Shape Context [21] take a similar approach but using the edge information instead of a point`s gradients. PGH compiled a pair of histograms (scene histogram and model histogram) as the descriptor, based on line

segmentation, and uses the similarity matrix to measure the scale change of images. Shape Context computes the descriptor on a log-polar coordination system. The descriptor has 5 bins on 12 different directions and counts the number of points on the shape contour, which fall into the bins. The reason why log-polar coordinates are used is to make points near the interest point weigh more than the points far away. Both methods are scale invariant.

Symmetry descriptor [28] is based on the distribution of symmetry points and it computes the symmetry measure function to identify pairs of symmetry points by calculating the magnitude and orientation of the intensity gradient for each pair of points. Since symmetry regions will remain ellipses after affine transformation, the symmetry region can be determined by fitting ellipses to the symmetry points.

3.2.2 Intensity based descriptors

If two matching images are similar enough, the simplest and fastest descriptor will be a histogram of all pixel intensities. However, when the images grow bigger, the simplest will be no longer be the best in terms of quality after a class of transformations. As a result, more sensible descriptors have been introduced, such as spin image.

Spin image was first presented by Johnson and Hebert [17], and later Lazebnik and Schmid [18] adapted it to texture representation. The basic idea of spin map is to project 3-D points on surfaces of an object into a 2-D space with a certain basis, which produce the spin image. Spin images are scale invariant since the projection keeps the global information of the surface. In Intensity Domain Spin Image [18], the spin image is used as a 2-D histogram descriptor. The descriptor of a certain point consists of two elements, the distance from the point to the shape centre and the intensity level of that point. This also works well on texture representation [27].

Another texture representation technique is non-parametric local transform [19] developed by Zabih and Woodfill. Non-parametric transform is not relying on the intensity value but the distribution of intensities. Two non-parametric transformations were introduced in [19]. The first one, called rank transform, counted the number of points whose intensities are smaller than the interest point in the neighborhood. The second one, called census transform, is a string summarizing the structure of points in the neighborhood. The combination of the resulting information from these two transformations constitutes the descriptor.

Andrew created a new descriptor, called Channel [26]. The Channel is the function to measure the intensity change along certain directions from the interest point. A set of channels in different directions makes up the description of the point. But this method is just partially invariant to scale changes because the more significant the scale is, the more intensity information will be lost along the channel.

3.2.3 Derivative based descriptors

Koenderink [30] derived local jet first and inspected its features. It is a set of Gaussian derivatives and often used to represent local image features in the interest region. Since

Gaussian derivatives are dependent on the changes in the image`s orientation, several methods have been developed to remove the rotation dependency.

The straightforward approach is a combination of local jet components. The vector of components will then be made invariant to rotation as well as scale changes [23, 29]. An example vector [29] is given as equation 15.

$$v[0\ldots8] = \begin{bmatrix} D \\ D_i D_i \\ D_i D_{ij} D_j \\ D_{ii} \\ D_{ij} D_{ji} \\ \varepsilon_{ij}(D_{jkl} D_i D_k D_l - D_{jkl} D_i D_l D_l) \\ D_{iij} D_j D_k D_k - D_{ijk} D_i D_j D_k \\ -\varepsilon_{ij} D_{jkl} D_i D_k D_l \\ D_{ijk} D_i D_k D_l \end{bmatrix} \tag{15}$$

where $\varepsilon_{ij} = -\varepsilon_{ji} = 1$, $\varepsilon_{ii} = \varepsilon_{jj} = 0$ and D is the Gaussian Derivative.

A more complex method is named Steerable filter [24], which steers filters to obtain same components of local jet from a rotated image so as to make the descriptor invariant to rotation.

3.2.4 Other technique based descriptors

There are many other techniques used to represent the image feature, for instance, Gabor Wavelet [22], complex filters [9, 10] and moment invariants [25]. The major function is to convert the problem of representing image feature to another field or domain, such as frequency domain [22] and complex field [9, 10] and extract distinctive characteristics to represent image regions.

Moment invariant is a simple example. Two types of moments, shape moments and intensity moments, have been combined together in [25] to form some affine and photometric invariant representations. Shape moment and intensity moment on point (p, q) are presented as follows:

$$MS_{pq} = \iint_{\Omega} x^p y^q dxdy \qquad \text{and} \qquad MI_{pq} = \iint_{\Omega} i(x,y) x^p y^q dxdy \tag{16}$$

where Ω is the region, $i(x, y)$ is the intensity level of point (p, q). A 2nd order affine and photometric invariant sample is given as follows:

2nd order intensity moments + 0th order shape moment

$$\frac{(MI_{20}MI_{00} - MI_{10}^2)(MI_{02}MI_{00} - MI_{01}^2) - (MI_{11}MI_{00} - MI_{10}MI_{01})^2}{MI_{00}^4 MS_{00}^2} \tag{17}$$

One advantage of the combination of different type of moments is that a mixture of moments keeps the order of expression low so that the computing consumption will be low. Furthermore, the combination makes the representation more robust against both affine transformation and photometric changes, especially for the 2nd order moment invariants.

In section 3.2, most existing important representation techniques are categorized and in each category, a typical descriptor was discussed in detail. Most of them are not affine invariant, as it depends on which kind of detector they will use and the descriptors` dimension. There is a conflict between the complexity of descriptor (robustness against image clutter and distortion) and the computing time. SIFT including both the detector and descriptor as a whole system provides better results compared to other systems under Mikolajczyk`s evaluation framework [15]. As was mentioned in section 2, SIFT still has some shortcomings, and as a result some improvements have been introduced and these are discussed in section 3.3, mainly reducing the accuracy as a trade-off for processing speed.

3.3 Improvement on SIFT

SIFT is well designed and derives distinctive image feature descriptors for image matching but it still suffers from its high cost of computation not only for the descriptor but also for keypoint detection. Even though SIFT exhibits outstanding performances in term of quality, quite a few improvements have been made on it to reduce the processing time.

PCA-SIFT [16] is one of the most successful extensions, which applies Principal Components Analysis (PCA) [34], a standard technique for dimensional reduction, on the target image patch and produces a similar descriptor as the standard SIFT one, so that the nearest matching algorithm can still be used in PCA-SIFT.

The PCA-SIFT descriptor [16] is based on the interest region detected by the standard SIFT, in which a 41*41 gradient image patch around the interest point is extracted and PCA is used to project the high-dimension feature vector to the eigenspace which is computed in advance according to the local image. The dimension generally will be around 20 so that it will reduce the computation cost significantly.

Based on the PCA-SIFT, another algorithm, called i-SIFT, has been presented by Fritz et al. [32], which applied Information Theoretic Selection to cut down the number of interest points so as to reduce the cost for computing descriptors.

For the detector part, Grabner et al. [33] presented Difference-of-mean (DOM) instead of DOG. The SIFT algorithm has speeded up and meanwhile the performance still remains good.

3.4 Previous work

There have been several implementation of SIFT, two of which are quite famous for their good quality of keypoints extracted. One is presented by Andrea Vedaldi from University of California. His SIFT implementation is under Matlab and a short version in C++ is also available on the website. Sebastian Nowozin from technology university of Berlin implements another version. He used C# to make the library object oriented. However, the speed of it is not satisfactory. One big problem is that both of them do not supply helpful documents about the implementation, which make their program difficult to use and change for particular research purpose.

3.5 Summary

Since Schmid and Mohr [31] first addressed the method of matching an image feature against a set of features, a lot of work has been done in image matching involving all kinds of feature detectors and descriptors. In section 3, we introduced most of the existing important feature detection and description methods. SIFT`s characteristics are so distinctive that it is worthwhile implementing and it has the potential to be improved and extended. In following parts, documents in every stage of software development will be presented.

4 Development methodology

The processing speed and the location and representation of keypoints are two key factors for judging the quality of the implementation of SIFT library, but it`s impossible to achieve a high standard for the first version of the library. So the prototype model is used here, which is illustrated in Figure 5. A primary version is supposed to be finished in early stage of the project and plenty of time ought to be spent on improving it. I am planning an iterative development method. The iteration will involve adjust some details of implementation such as the choice of factor σ for Gaussian blur, the frequency of sampling in octaves and so on. After every adjustment, the results will be evaluated by comparing it with ones from David Lowe`s demo. In addition, a regression testing will follow every modification and improvement.

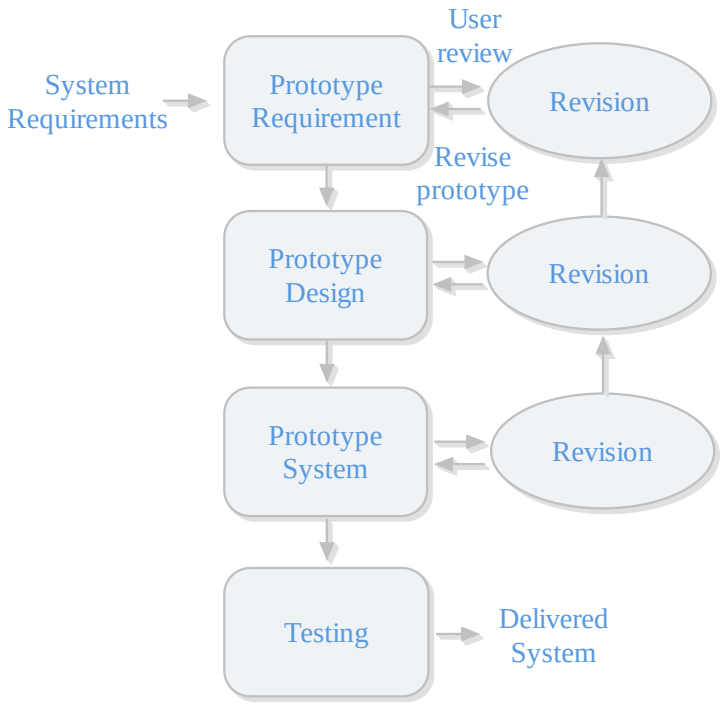

Figure 5: The diagram illustrates the prototype used in the project.

The demonstration system will be developed by using classical waterfall model. In case the SIFT library is not efficient enough for real time use, the object recognition system will be simplified as detection of object images in a scene image. As before, all modification and improvement will be tested carefully.

5 Design specification

5.1 SIFT library

The SIFT library consists of three main modules, namely, keypoint detector, keypoint descriptor and a simple image matching module. Figure 6 illustrates the structure of the library at a high level. Notice that the keypoint detection and keypoint description are separated as single modules. There are several reasons for this split. First, the keypoint detector module itself can be reused in other program. Second, it is useful to have an idea of keypoint location and orientation before computing the descriptor for them. Last but not least, the split will simplify the extension later. As mentioned in section 2, we may apply some other algorithms in the demonstration system such as PCA-SIFT [16] which uses a different keypoint detector but the standard SIFT descriptor can still be used in this approach.

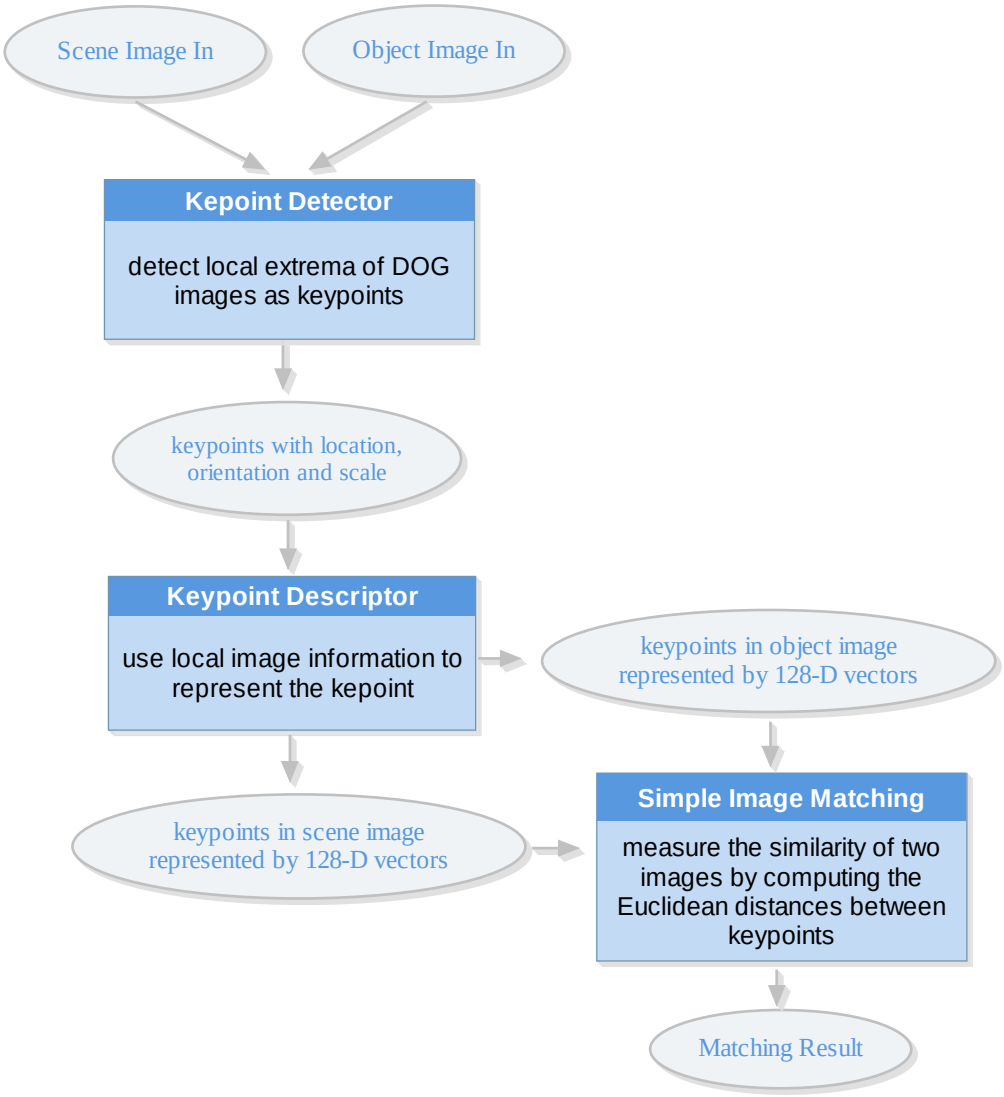

Figure 6: Overview of the SIFT library

Keypoint detector, keypoint descriptor and image matching module will be discussed respectively in the following sections. Details of design decisions for each module will be presented as well.

5.1.1 Keypoint detection

Before processing the images, we need to read in images first. OpenCV provides a powerful image read-in function that supports all common image types. Initially, we only accept PGM grey-value images and we assume that the image type can be identified by its extension filename.

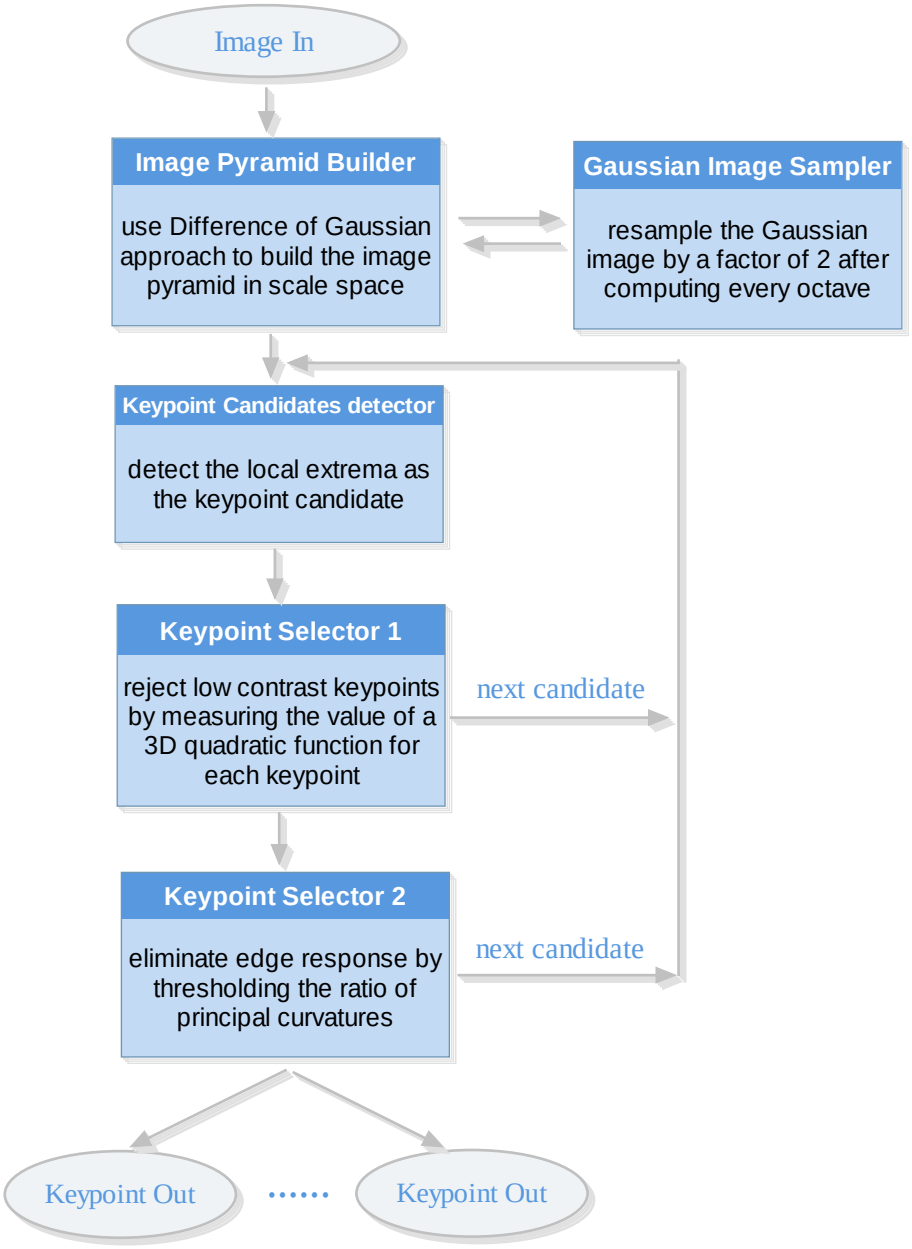

Figure 7: Dataflow diagram of keypoint detection module

The whole procedure of keypoint detection is shown in Figure 7. The first step will be building the image pyramid in scale space by computing the DOG of the image. The first level of the pyramid will be built on an image whose size doubles the one of the original image. Other scales in every octave will be produced by Gaussian blurring the image on the first level continuously. For each level (octave) of the pyramid, local extrema will be detected as the keypoint candidates. We decide to reject low contrast and unstable keypoint candidates by two keypoint selectors right after obtaining each candidate. The detection procedure will be repeated until all points have been checked. As a result, it comes out a list of keypoints with their locations, orientations and scales.

5.1.2 Keypoint representation

Given the list of keypoints, we need compute the orientation histogram to describe the region around the keypoint. First of all, the gradient magnitude and orientation of sample points around the keypoint will be calculated. Moreover, the orientation and an orientation histogram will be computed for each keypoint after that. One thing that does not been clarified in Lowe`s paper is how this 16*16 region will be located around the keypoint. We decide to vary the size of regions according to the scale of the image on which keypoints were detected. So when the image is scaled up or down, the region will become larger or smaller, which increase the matching possibility of the same keypoint in different scales. Furthermore, in order to make the descriptor invariant to rotation, illumination and contrast changes, we rotate the image to keypoint`s orientation and apply a set of operations on the vector, including two normalizations and one thresholding. Figure 8 illustrates the detail of these operations and also shows the whole procedure of keypoint description ended by outputting a list of 128-D image feature vectors.

5.1.3 Matching module

The SIFT library includes a simple image matching module which reads in two sets of keypoint descriptors from the scene image and the object image and measures the similarity of two features by computing the Euclidean Distance between two feature vectors. Given $X(x_1, x_2, \ldots, x_n)$ and $Y(y_1, y_2, \ldots, y_n)$, the Euclidean distance can be represented as following equation:

$$D = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2 + \cdots + (x_n - y_n)^2} \tag{18}$$

For every target keypoint, two keypoints will remain after comparison, which have the most nearest distances to the target keypoint. If the nearest keypoint is 0.6 times closer to the target keypoint than the second nearest one, the nearest keypoint and the target keypoint will be outputed as a matching pair.

The matching approach presented above just aims at giving a simple image-matching example. It`s quite time-consuming since every feature vector will be compared with the target

one. In Lowe`s paper [4], a more practical method, called best-bin-first, provides a more efficient way to search the matching pairs. But in this project, the matching module and the recognition system are only produced to demonstrate the performance of SIFT library. So the efficiency of matching procedure is not our purpose.

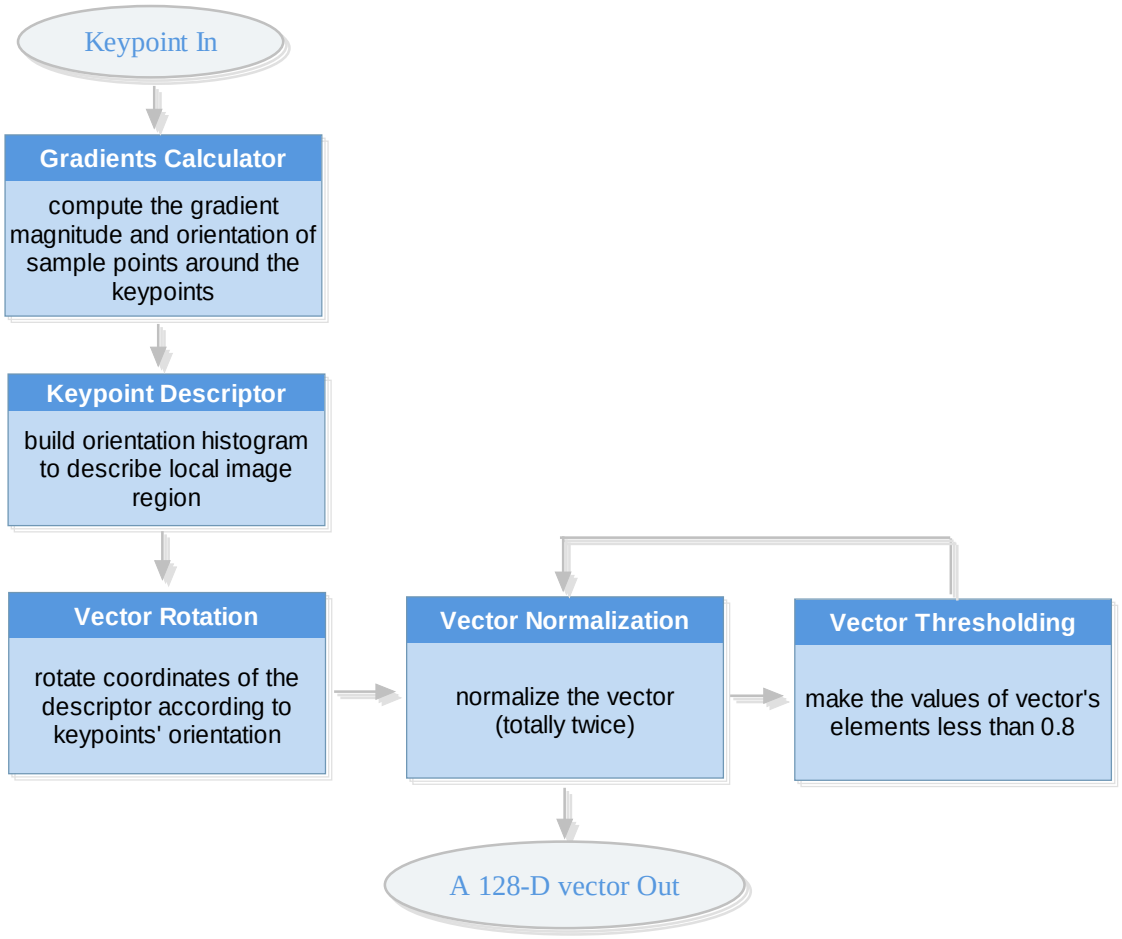

Figure 8: Dataflow diagram of keypoint representation module

5.2 Object recognition system

Because the demonstration system will be treated as an extension and it turns out that it`s quite difficult to obtain the same output from the SIFT library as the one from Lowe`s demo, not too much time was left to develop the demonstration system when SIFT library had been completed. We decide to simplify the system to an image object detection system. In other words, the system is able to detect image objects in a scene image. All images will be inputted together and the system will output the detection result that is drawn over the scene image. One of the sample results was shown as Figure 1 in section 1.3.

The algorithm of the object recognition system is illustrated in Figure 9. In order to demonstrate the performance of SIFT library, we do not consider the efficiency of the system. Every keypoint in the list will be compared with the target one so that the comparison result can

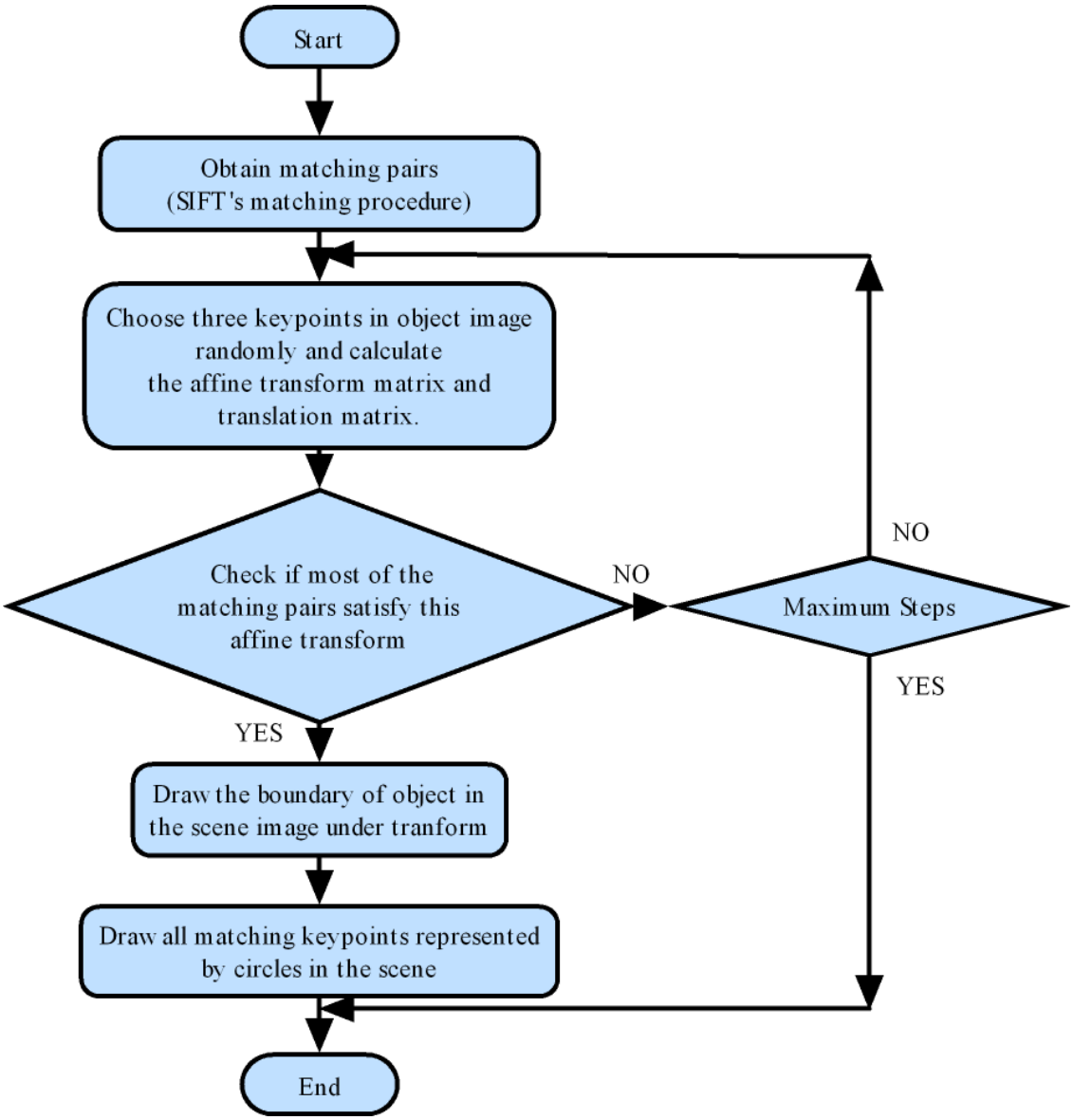

Figure 9: Algorithm of the object detection system

test the library thoroughly. The main idea of the system is extracted from David Lowe`s paper[4] but quite a few places changed according to this application. First, instead of using Hough transform to cluster the keypoints, we simply pick up three keypoints randomly and sort out the possible affine transform matrix and translation matrix between object image and scene image. After that, all matching keypoints will be checked if they satisfy the transform. If majority of them do, we decide the object is in the scene image. Otherwise, another three points will be picked and above procedure will be repeated until reaching the maximum time. It may happen that one of the three points or more are incorrect matching every time the program picks three points randomly. In the end, the object may not be detected but it does exit in the scene image. However, the possibility of that is quite small since the correct ratio of matching keypoints extracted by SIFT library is high enough for this application. For instance, if the correct

matching ratio is over 95% and the maximum time is 10, the possibility of missing detection approximately equals to zero. After the transform has been decided, the boundary and the keypoints will be drawn on the scene image.

5.3 User interface

The SIFT library is designed to be reused in all kinds of applications. So no user interface has been included in the library. Compared with user interface, module interface is more important for programmers to use functions of the library. The detail of it will be introduced in the User Guide, which will help user to master the library.

The user interface of object recognition system is based on command line. The user only needs to specify object images and a scene image.

5.4 Project plan

The main development tasks are listed as follows:

1. Implement all modules in SIFT library: the main task in this period is to finish the first version of SIFT library. The processing speed may not be considered at first. However, the program should be implemented as modularized as possible for speed and quality improvements later. (Duration 4 weeks)
2. Test and refine SIFT library: Although some basic tests for each module have been conducted, the whole library will be integrated and tested in this period. After that, we need to consider the factors that influence the processing speed. They have been discussed in section 3 so we will omit it here. All sorts of grey-value images will be used in case tests and the initial evaluation. (Duration 4 weeks)
3. Demonstration system: link the object recognition system to SIFT library and implement the system according to the algorithm stated above. At the same time, take some pictures for experiments (Duration 1 week)
4. Test and improve the demonstration system: make sure demonstration system work properly and adjust the thresholds to obtain a better detection (Duration 1 week)
5. Final testing and any possible changes: spare time for final testing and any events unscheduled. Another important task is to collect evaluation data for dissertation. (Duration 4 weeks)
6. Finish poster and dissertation: write up dissertation, design poster and prepare for presentation. (Duration 4 weeks)

The detail of schedule is illustrated in Project Gantt Chart (Figure 10).

5.5 Summary

In this section, we specify a number of significant design decisions in a high level. Project Plan is also given in the end to trace the progress of the whole project. Main problems encountered and our solution to them will be detailed in next part.

Figure 10: Project Gantt Chart

6 Implementation

In last section, we have outlined the structure of the SIFT library and the object recognition system. In this part, detailed problems we faced during the implementation period will be stated. Important data structures and methods will also be explained with necessary sample pseudocodes.

6.1 Image representation

Image is one of the major data structures used frequently in the SIFT library, so an efficient and powerful representation method is supposed to be used in order to raise the processing speed and provide a convenient way to read and write. IplImage is our choice, which is part of OpenCV library. User can specify the pixel depth for images, which is quite useful for SIFT library since three different types of images are used in the library, grey scale images (8-bit per pixel), Difference of Gaussian images (32-bit per pixel), gradient magnitude map of the image (32-bit per pixel) and gradient orientation map of the image (32-bit per pixel).

In addition, OpenCV library provides standardized and efficient methods to allocate and release the image structure. Large numbers of operation in computer vision field have also been developed based on IplImage. Some of them are used in this project, for an example, Gaussian filter.

When building the image scale pyramid, the program will produce a list of scales for each octave and later another three lists which contain the information about difference of Gaussian and gradient magnitude and orientation of images. So we need a data structure to store all these images. Since all these images will be used in order, linked lists are chosen to store them. There`s also an advantage when the number of scale in each octave changes.

```
ADT ImageList:
/* structure of every node in the image list */
typedef struct CvImageListNode {
    IplImage        *img;           /* scale in Octave */
    CvImageListNode *next;          /* pointer to the next node */
}
CvImageListNode;

/* store the images in each scale space */
typedef struct CvImageList {
    CvImageListNode *start;     /* pointer to the first node */
    CvImageListNode *current;
    CvImageListNode *end;       /* pointer to the last node */
    int             imageNum;       /* number of images */
}
CvImageList;
```

In ImageList ADT, several special operations on images are also implemented, such as doubling the size of image, halving the size of image and image minus (calculate the difference of two images).

6.2 Building image pyramid

Another basic step before detecting keypoints is building the image scale pyramid. This part is the most difficult one during the implementation. Quite a lot of time has been spent on adjusting the structure of octaves, which make up the pyramid. This module mainly consists of two ADTs, ScalePyramid and ScaleOctave.

In ADT ScalePyramid, detailed structure of the pyramid is stated as follows:

```
/* structure of every octave list node */
typedef struct CvOctaveListNode {
    CvScaleOctave      *octave;    /* pointer to a octave */
    CvOctaveListNode   *next;      /* pointer to the next node */
}
CvOctaveListNode;

/* store all the octaves in the pyramid */
typedef struct CvOctaveList {
    CvOctaveListNode   *start;     /* pointer to the first node */
    CvOctaveListNode   *current;
    CvOctaveListNode   *end;       /* pointer to the last node */
    int                octavesNum; /* number of octaves */
}
CvOctaveList;

/* structure of scale pyramid */
typedef struct CvScalePyramid {
    CvOctaveList   *octaves;       /* pointer to the octave list */
}
CvScalePyramid;
```

Every pyramid includes a linked list that stores every level of Octaves in the pyramid. The data structure of every Octave is specified below:

```
/* structure of octave */
typedef struct CvScaleOctave {
    CvScaleOctave   *up;               /* pointer to the upper octave */
    CvScaleOctave   *down;                 /* pointer to the lower octave */
    CvImageList     *gaussianMapList;      /* Gaussian Map list */
    CvImageList     *diffMapList;      /* difference of Gaussian Map list */
    CvImageList     *magnitudeMapList; /* gradient magnitude of images */
    CvImageList     *orientationMapList;/* gradient orientation of images */
    IplImage        *baseImg;      /* the base image for this scale space */
    float           baseScale;             /* scale of the base image */
}
CvScaleOctave;
```

The method of building image scale pyramid exactly follows the instructions from David Lowe`s paper. Every Octave contains not only a list of images blurred by Gaussian filter one by one but also a list of DOG maps, a list of magnitude maps and a list of orientation maps, which have the same order as the Gaussian maps`. It`s for the convenience of obtaining the gaussian map, magnitude map and orientation map of same scale when the program computer feature

vectors for keypoints. Furthermore, Octave structure contains two handles that point to the octaves above and below. When detection is finished in one octave, the program can go to next level directly without using the next pointer in the octave list.

6.3 Keypoint representation

The keypoint representation is also a key problem during the implementation. The data structure of keypoint is used through all modules in the library. From the SIFT detection module, a list of candidate keypoints will be detected and adjusted. After that, unstable keypoints will be eliminated. When the locations of all keypoints have been fixed, the feature vector of each keypoint will be computed. So the keypoints involve every step of the procedure and the content and the number of keypoint data structures may change greatly. At first, we only have one data structure for keypoints as follows:

```
ADT keypoint list:
/* store each KeyPoint in the list */
typedef struct CvKeyPointListNode {
    float   x;                  /* the x coordinate of keypoint */
    float   y;                  /* the y coordinate of keypoint */
    int level;              /* the level in octave */
    float   imgScale;           /* the scale of image */
    float   baseScale;          /* the scale of the base image of the octave */
    float   orientation;        /* the orientation of the histogram */
    float   *featureVector;     /* feature vector(128 dimension) */
    CvKeyPointListNode *next;
}
CvKeyPointListNode;

/* store all the keypoints detected and described */
typedef struct CvKeyPointList {
    CvKeyPointListNode *start;     /* pointer to the first node */
    CvKeyPointListNode *current;
    CvKeyPointListNode *end;       /* pointer to the last node */
    int num;               /* the number of keypoints */
}
CvKeyPointList;
```

This data structure uses a linked list to store all the keypoints and every node in the list contains all the information about keypoint, including location, level, scale, orientation and feature vector. However, in keypoint detection module, some of the elements such as feature vector will not be used and some of keypoints will be rejected before being processed in descriptor module. Therefore, great numbers of memories will be wasted if we allocate all the memories in the first module. Because of that, we decide to set up another data structure to represent the result of SIFT detector. The interim data structure is called local extrema list and it`s shown below:

```
ADT local extrema list:
 /* every node in local extrema list */
typedef struct CvLocalExtremaListNode {
    float   x;          /* x coordinate of extreme point */
    float   y;          /* y coordinate of extreme point */
    int level;      /* the level in octave */
    float scaleChanged;    /* scale value after adjustment */
    float valueD;       /* |D(x^)| */
    CvLocalExtremaListNode *next;      /* pointer to next node */
}
CvLocalExtremaListNode;


/* store all the local extrema */
typedef struct CvLocalExtremaList {
    CvLocalExtremaListNode *start;     /* pointer to the first node */
    CvLocalExtremaListNode *current;
    CvLocalExtremaListNode *end;       /* pointer to the last node */
    int num;        /* total number of nodes in the list */
}
CvLocalExtremaList;
```

These two ADTs also consist of some significant methods related to the data structures. In ADT localExtremaList, Given an octave, the detection method searches local extrema through every level of scales and after every candidate keypoint is detected, the location of the candidate will be adjusted by interpolation in the DOG image and then a edge response test will be carried on it as well. If the candidate can survive all the tests, it will be added into the local extrema list.

Problems are disclosed when the number of candidate keypoints are huge. The computation time is not satisfactory at all. Keypoints with small difference value also will be detected in low frequency area. So we decide to add a thresholding procedure before checking if the point is a local extrema, which reduces the processing time greatly. However, at the same time, some real keypoints are rejected before being detected and the numbers diversified in different scales. Finally, we apply different thresholding for scales. In another words, the thresholds are related to the scale of the image on which the keypoint is detected.

The thresholds including ones mentioned above, the threshold for D-value and the edge response ratio have been given in Lowe`s paper [4]. However, we find that different sets and combinations of these thresholds will result in different outputs and influence the performance of SIFT library dramatically.

In ADT keypointList, codes are designed to sort out the orientation of keypoints and build the image feature vector for every keypoints from the detector. The procedure of computing the orientation of keypoint is stated clearly in Lowe`s paper but the part of building histogram is too vague to implement a good version at first. In the first version, as illustrated in Figure 4, the gradient magnitude and orientation of a keypoint in the image will only contribute the histogram in which this keypoint is located. It turns out that feature vectors contain quite a few zero bins. In essence, the feature vectors are too sparse to obtain a thoughtful matching for same keypoint from different images.

Our solution to this problem is illustrated in Figure 11. First, for every pixel in the neighbourhood around the keypoint, we scale up the point like point k in the left image of Figure 11, whose distances to four histogram`s centres are all less than 1. As shown in the image, vertically, the weight of contribution of point k to histogram 1 is Wx1 and the weight of contribution to histogram 3 is Wx3. Similarly, we will have Wy1 for histogram 1and Wy2 for histogram 2 horizontally. For another dimension, orientation Ko, the gradient magnitude of point k will contribute 2 bins weighted by the distance from point`s orientation to two bins`, Wo1 for orientation 1 and Wo2 for orientation 2 in Figure 11. The equation is shown below:

$$Contribution_{n,m} = Mag * Wx_n * Wy_n * WO_m * W_{Mag} \qquad n = 1 \cdots 4, m = 1,2 \tag{19}$$

For an example,

$$Contribution_{1,2} = Mag * Wx_1 * Wy_1 * WO_2 * W_{Mag}$$

represents the value is added to histogram 1 on direction bin 2(East North).

After the improvement, the number of matching pairs increases significantly. The feature vector contains more information so that it is more reliable for matching.

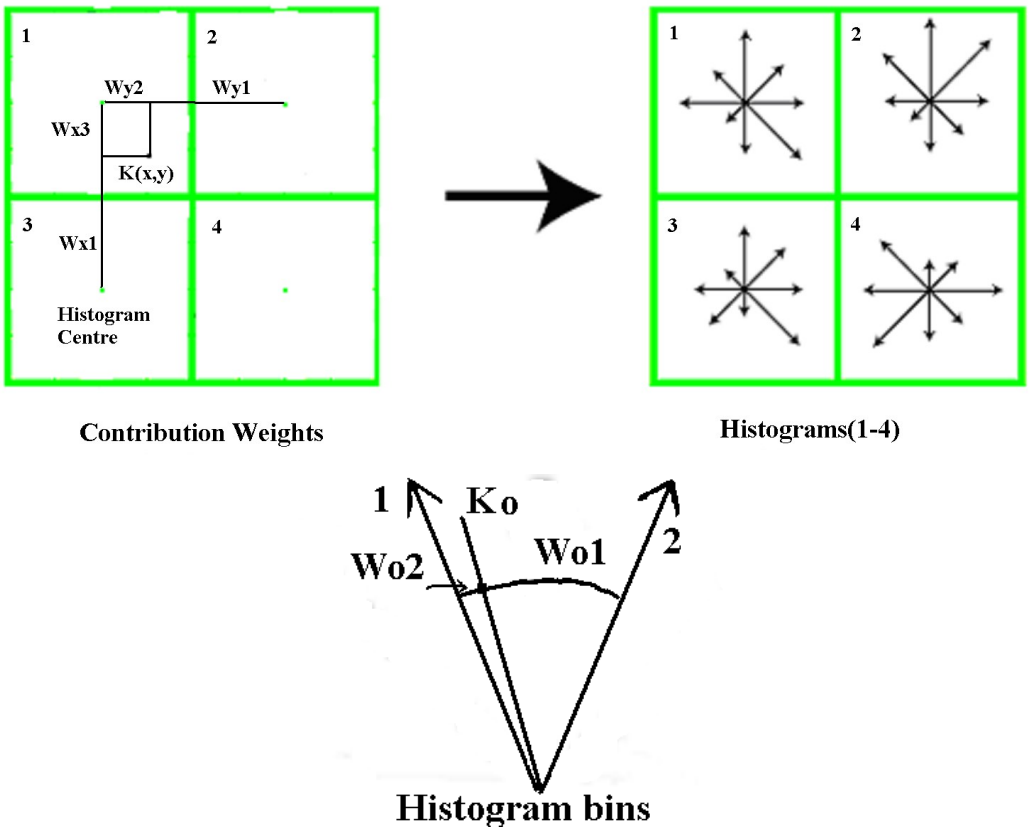

Figure 11: This figure shows how each keypoint`s contributions to four histograms are weighted.

6.4 Other modules in SIFT library

Most of the applications of SIFT library may only expect the SIFT library to supply a simply method which outputs a list of keypoints with their feature vectors according to the image provided. DetectKeypoint module provides a function like this, which only requires an image in the form of IplImage as input and will output keypoints stored in a keypoint list. So this module is the first and maybe the only module that an application programmer needs to know. In addition, the module also contains a method for printing out the keypoint list to a txt file in the format of:

```
1007 128                  // number of keypoints     dimension of feature vector
53.18 485.07 1.85 2.251           // x, y coordinates    scale    orientation
2 3 1 24 14 0 1 0 1 2 0 57 48 0 1 0 0 0 0 76       // 128-D feature Vector
76 0 0 0 0 0 2 76 47 0 0 0 1 2 5 10 6 2 2 0
1 1 1 21 16 0 0 0 0 0 0 76 75 0 0 0 0 0 1 44
28 0 0 0 2 1 2 5 2 3 5 1 0 0 0 4 55 3 0 0
0 0 0 26 76 1 0 0 0 0 0 20 36 1 0 0 1 0 1 5
13 8 6 0 0 0 0 3 76 4 0 0 0 0 0 3 76 2 0 0
0 0 0 1 22 5 0 0
........
```

6.5 Image Matching

No matter what kind of applications have been implemented upon SIFT library, the matching procedure cannot be skipped. And the ratio of correct matching is also a key point to judge the performance of SIFT. Therefore, the matching module has been included in the SIFT library.

Given two sets of keypoints, the matching method will return a list of matching pairs in the form of following data structure:

```
/* structure of match list node */
typedef struct MatchListNode {
    int xO;     /* x coordinate in object image */
    int yO;     /* y coordinate in object image */
    int xS;     /* x coordinate in scene image */
    int yS;     /* y coordinate in scene image */
    float scaleS;
    int correctSign;   /* if satisfy the affine transform */
    MatchListNode *next;
}
MatchListNode;

/* structure of match list */
typedef struct MatchList {
    MatchListNode   *points[MATCHLISTMAX];
    int             num;
}
MatchList;
```

The programmer can simply use the matching list to draw an image with corresponding points connected by lines between object image and scene image. Sample result is given as Figure 12. Complex application like object recognition can use the match list to figure out the relationship between two images so as to detect the object in a scene. Every node of the matching list also contains a variable which illustrate if the matching is correct or not, which will be introduced in detail later.

Figure 12: Sample result of matching pairs. Two pictures are taken from different angles but a large number of keypoints can still be matched correctly.

6.6 Object recognition

One difficult problem in object recognition system is how to sort out the affine transform matrix (A) and the translation matrix (T). As the general approach given in [6], we represent affine transforms from object point $O[x, y]$ to scene point $S[u, v]$ as following equation:

$$S^T = MO^T + T, \quad M = \begin{bmatrix} m_1 & m_2 \\ m_3 & m_4 \end{bmatrix}, \quad T = \begin{bmatrix} t_x \\ t_y \end{bmatrix} \tag{20}$$

Since there are 6 unknown variables in the equation above, at least three matching pairs are required to solve this linear system. The detail of this linear system is list below:

$$\begin{bmatrix} x_1 & y_1 & 0 & 0 & 1 & 0 \\ 0 & 0 & x_1 & y_1 & 0 & 1 \\ x_2 & y_2 & 0 & 0 & 1 & 0 \\ 0 & 0 & x_2 & y_2 & 0 & 1 \\ x_3 & y_3 & 0 & 0 & 1 & 0 \\ 0 & 0 & x_3 & y_3 & 0 & 1 \end{bmatrix} \begin{bmatrix} m_1 \\ m_2 \\ m_3 \\ m_4 \\ t_x \\ t_y \end{bmatrix} = \begin{bmatrix} u_1 \\ v_1 \\ u_2 \\ v_2 \\ u_3 \\ v_3 \end{bmatrix} \tag{21}$$

Eventually, use the method provided by OpenCV to solve this system of linear equations.

After recognising the object in the scene, the result is supposed to be shown on the scene image. First, the method of drawing the boundaries of the object is quite straightforward. We apply the affine transform (sorted out above) to four corner point of the object image, (0, 0), (0, width), (height, 0) and (height, width). Provided the coordinates of these points in the scene image, we simply connect the points by using Bresenham algorithm. In addition, other results should be drawn on the scene image are correct matching pairs, which are represented by circles with radii related to the scales of keypoints. One of the sample results has been shown in Section 1.

6.7 Summary

This section includes detailed decisions and solutions to main problem occurred during the implementation period. Data structures used in the project are also listed and explained in the form of pseudocodes.

7 Software testing

The testing of this project mainly concerns two aspects: Bug testing and Performance testing. Bug testing is an important step both during the implementation and after the implementation. Several testing techniques have been used in this project, namely black / white box testing, randomised testing and regression testing, they will be discussed separately on each software level. The testing applied here is supposed to find out most of the bugs that cause the system crashes or output of error results. So no performance testing will be shown here. Performance testing will be introduced in the Evaluation part in detail.

7.1 Method testing

After coding every function, we design particular testing harnesses to check the running status, input and output of the program. The testing harness is supposed to test every line of the codes. Once a fault has been found, we will identify the fault as the following steps:

1. The program code may contain an error, such as typing errors, fault calling of other functions and so on. This kind of errors can be removed easily with careful investigation of codes.

2. The program design may be not foresighted, such as capacity faults (using an integer to represent a float), missing one condition for if statement and so on. White box testing is perfect for detecting this kind of faults.

3. The algorithm design may be wrong, which is pretty serious and may need to review the system design and specification. Randomised cases always can cause this kind of problems and output unexpected results that will help the programmer locate the bug.

Method testing is the foundation of module testing and system testing. Before going to the next stage, we would make sure every function works as expected.

7.2 Module testing

Even though every function in the module has been tested thoroughly, crashes may still happen when we do module testing. Conflicts between functions can cause incoherence within the module, for example, function interface conflict. In this case, most of the module also contains one or two definition of data structures. They can be tested with their operations separately.

Another purpose of module testing is to check the input and output of the module as a whole. Special designed cases and randomised cases are used here to cover all possible inputs and results.

Besides module testing as a whole, integration testing of several functions are always useful to trace the problem when it occurs on the module level but all functions proves to be correct. Scaling down the range of functions one by one is a popular way to find the incoherent components.

7.3 System testing

One difficulty we face during the system testing is to trace the status of the system. So we use assertion testing to make sure the status of the system is correct. In addition to the assertion testing, debugging tools such as DDD is quite useful to display the value of variables and return values of functions. Well-designed cases give us the chance to go through every line of the program and make the bug efficiently.

Since we use prototype model to develop this project, after the first version, there are lots of changes based on it. Regression testing is used to validate every modification to the program. Moreover, after every modification, CVS will record the changes automatically and also will manage the version of testing program.

This part of system testing only guarantees the correctness of the result but the fineness. The performance of the system will be discussed as a key point in the evaluation part.

7.4 Summary

In this section, all testing methods used in the project have been listed and explained and the program has proved to be a robust program from the bottom to the up level. Next part will illustrate a complete system evaluation.

8 Critical evaluation

This section will introduce the experiments conducted on the SIFT library in order to present a critical evaluation on the software. First, the image set used in the experiment will be listed and the production method of them will be explained as well. Furthermore, an evaluation standard and the experiment environment will be stated and fixed. Eventually, the result of experiments will be catalogued by different transformation applied on the images, such as rotation, scaling, blurring, affine transform and illumination change. Analyses of the result will also be included in every heading.

8.1 Image set

The evaluation is carried on real images under different transformations, including rotation, scaling, illumination change, viewpoint change and image blur. For every catalogue, a sequence of images are taken in the range from small image transformations to large ones and the transformations are significant enough to illustrate the features of SIFT. The details about how these images are taken will be introduced below and sample images are illustrated in Figure 13.

1. Rotation: rotated images are taken by rotating the camera according to camera`s optic axis (the line between image centre and object geometric centre). The angles range from 0 to 180 degrees since the rest angles $(\pi, 2\pi]$ will provide similar results as the chosen ones. Images are taken every 15 degrees.

2. Scaling: because of the shortage of photograph equipments, not only the scaling of the images but also image blurring and illumination change are completed by using image processing software, Photoshop. The images are scaled according to original image`s scale in the range from 1 to 1/8. A more decent experiment may vary the camera`s zoom to obtain different scales of the object.

3. Image blur: as the same reason of Scaling, the blurring of images is obtained by using Gaussian blurring tool in Photoshop. Gaussian filter`s radius changes from 0.5 pixels to 5 pixels.

4. Illumination change: we modify the intensity values of images to obtain the effect of illumination change. The intensity values range from 35 to 95 and SIFT is not capable to extract any keypoints from images with intensity value under 35.

5. Affine transformation: This transformation is the most difficult one to be quantised. We solve the problem by rotating the object twice according to different axes. In this experiment, we rotate the object around X-axis for the first time and the angles for every image are the same. However, for the second time, the object will be rotated according to the Z-axis with angles ranged from 0 to 45 degrees. The coordinate system is defined as follows:

I. Object`s geometric centre is the origin

II. XY plane is parallel to the ground.

III. X-axis is parallel to camera`s optic axis.

(1)

(2)

(3)

(4)

(5)

Figure 13: Sample image sequences used in the evaluation: (1) Rotation; (2) Scaling; (3) Image blur; (4) Illumination change; (5) View point change;

8.2 Evaluation criterion

Three aspects have been highlighted as criteria among the experiment results to evaluate the performance of SIFT library. The results are extracted from the matching between two images. One of these two images is always the original image and the other is selected from the image sequences with different transformations. The matching procedure that has been introduced in the design part can provide the number of matching pairs between these two images as one criterion. The second criterion is the number of correct matches, which can be obtained from the results of object recognition system. The third criterion is the ratio of the number of correct matches with respect to the number of matching pairs between two images:

$$Correct\ Ratio = \frac{number\ of\ correct\ matches}{number\ of\ matching\ pairs} \tag{22}$$

SIFT library may detect different numbers of keypoints under diversified conditions (scenes under different transformations). At the same time, the number of correct matches will change according to the number of matching pairs. However, perfectly implemented SIFT library is supposed to obtain the same Correct Ratio under any conditions, which should always be nearly 1. As a matter of fact, the perfect ratio is really hard to obtain though we have tried out best to make the experiment environment as sound as possible and implementation as comprehensive as possible. Although the perfect ratio is unattainable, the ratios are supposed to higher than a certain value close to 1. As you can see in the following part, our implementation is basically invariant to rotation, scaling and other transformations. The correct ratios maintain on a high level even though the number of matching pairs reduces significantly.

8.3 Experiment results

8.3.1 Image rotation

Images are taken by rotating the camera every 15 degrees from 0 to 180 degrees. The results of matching with the original image are shown in the table and diagram below:

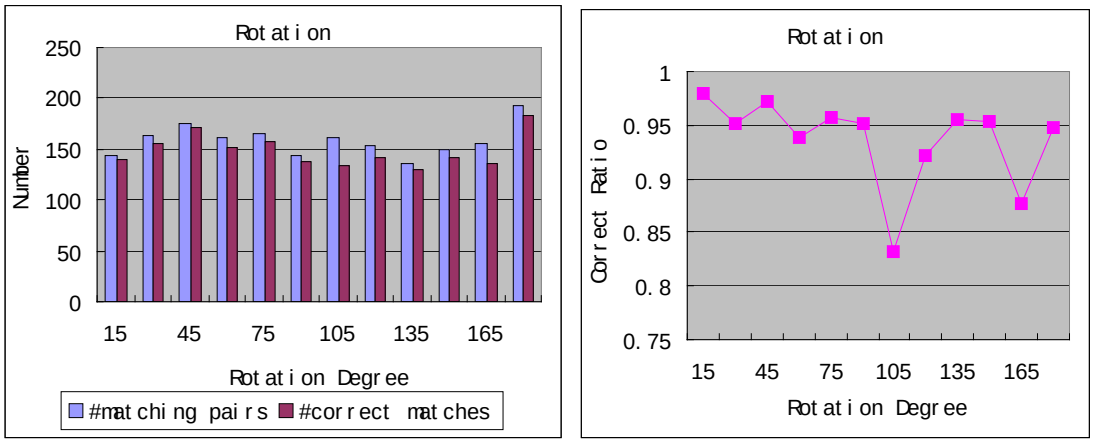

Figure 14: Experiment results of rotation image sequence.

From the left diagram in Figure 14, we can see the numbers of matching pairs and correct matches fluctuate slightly around 150. In addition, the correct ratios are all above 90% besides two exceptions, which may be influenced by the quality of images like illumination change or focus failure. As a whole, we can say that the rotation does not influence the performance of SIFT library greatly.

8.3.2 scale change

The sequence of scaled images is produced by scaling down the original image one by one. If the size of original image is too small, keypoints cannot be extracted from the smallest scaled image. So we use an original image with a resolution of 1024 * 840. However, the scaling still influences the performance of SIFT library greatly. As we can see from the diagram (Figure 15), when the scale of image is 1/8 of the original one`s, only a few keypoints can be detected, which is not enough for object recognition system. Furthermore, except a sudden fall on scale 0.375, the correct ratio keep on a high level, which may be caused by SIFT detector. Although SIFT detector extracts keypoints on scales all over the scale space, the gap between the scales still exists. When the scale of the target object is right in the gap, the performance of SIFT library will be affected significantly, leading to a drop of correct ratio and the number of matching pairs. On the contrary, if the scale of object is similar to the scale of image on which keypoints have been detected, the number of matching pairs will increase greatly, for an example, scale 0.5 in the left diagram. In essence, the scaling may influence the number of matching pairs significantly, but the correct ratio can still be high as expected.

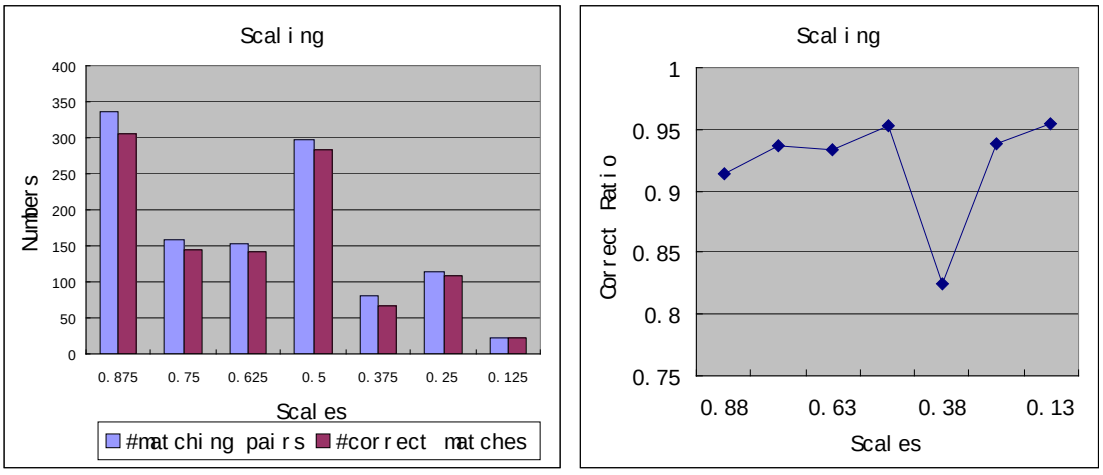

Figure 15: Experiment results of scale change image sequence.

8.3.3 Affine transform

This transform is the most difficult to be quantised and to be evaluated thoroughly. We use two rotations to stimulate the affine transform and the results are shown in Figure 16:

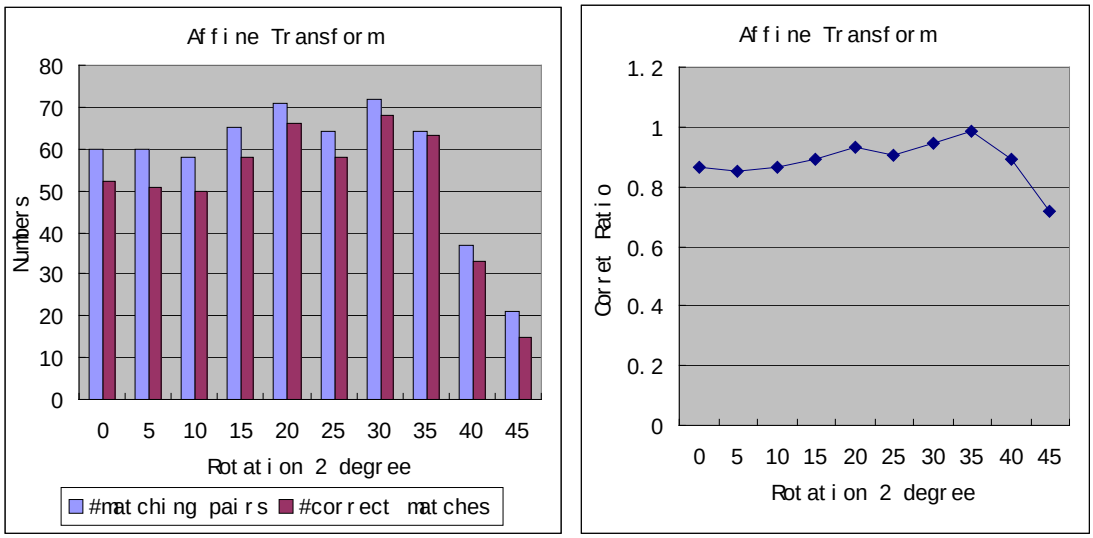

Figure 16: Experiment results of viewpoint change image sequence.

As shown in Figure 16, the numbers of matching pairs and correct matches change smoothly before the sudden drop on degree 40. Rotation bigger than that will cause a significant fall of the performance of SIFT library. From the last example of viewpoint change image sequence in Figure 13, we can see that the viewable area of target object is almost half of the original one. Because of that, a great amount of the pixel information has been lost so that the loss of keypoints is reasonable. Even though the number of matching pairs drops, the correct ratio is still around 0.8.

8.3.4 Image blurring

In this experiment, we use Gaussian filter to test the image blur`s influence on SIFT library. While the radius of filter increases, the number of matching pairs may be decreased gradually but the correct ratio should keep on a high level.

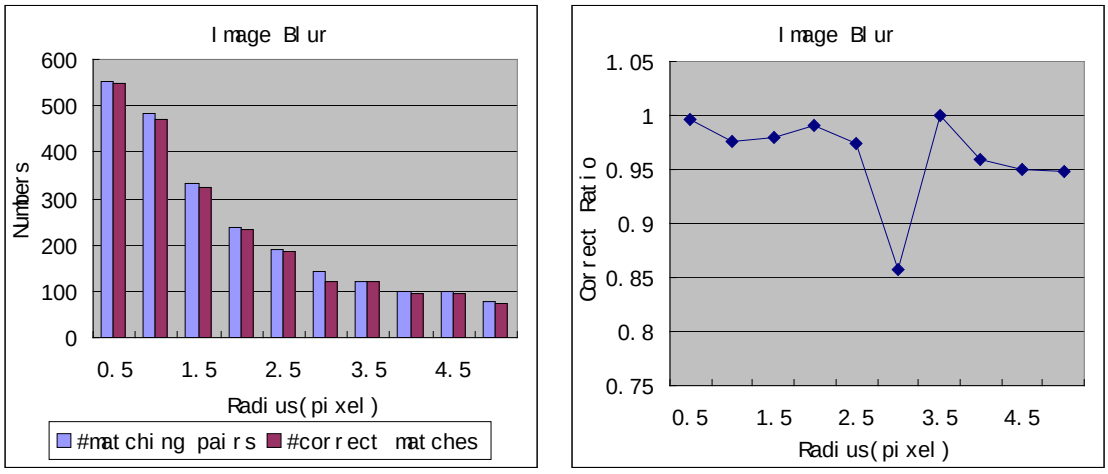

Figure 17: Experiment results of blurring image sequence.

As shown in Figure 17, the results of experiment prove the presumption. Most of the correct ratio is above 95% and the only exception is still over 0.85.

8.3.5 Illumination change

Like the last experiment, the result of illumination changes is also predictable. In order to make SIFT invariant to illumination, David required normalizing the feature vector twice and thresholding the gradient magnitude to eliminate the effect of illumination change, which is really effective. The result illustrated in Figure 18 proves both the outstanding character of SIFT and the success of our implementation.

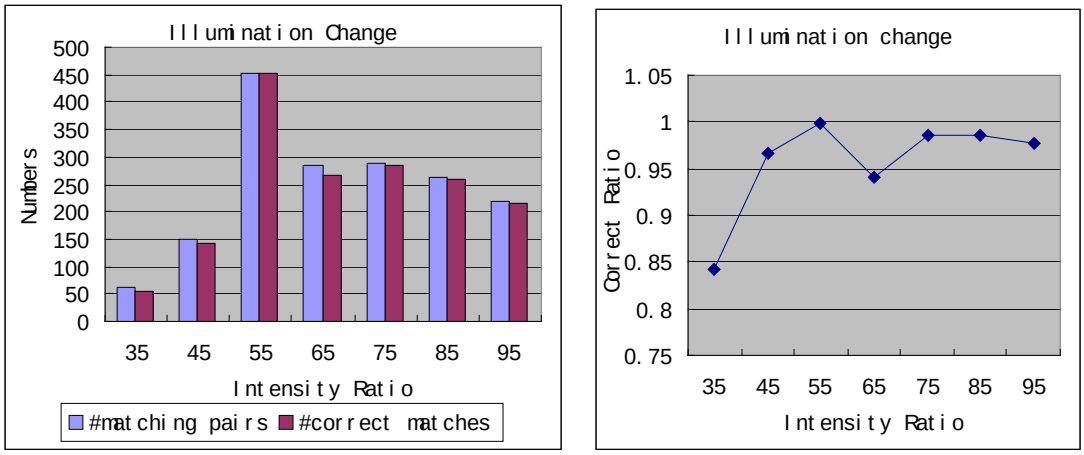

Figure 18: Experiment results of illumination change image sequence.

Since SIFT only thresholds the high side of the gradient magnitude, the performance of detecting keypoints in low intensity images are not thoughtful. As we can see in Figure 18, the correct ratio is below 0.85 when intensity ratio is 35%. In fact, the SIFT library is not able to extract any keypoints when the intensity ratio is below 30%. So the low correct ratio might be caused by the design of SIFT itself but the implementation.

8.4 Demonstration system

In addition to the evaluation introduced above, we also develop an object recognition system to demonstrate the distinctive characteristics of SIFT in applications. As mentioned before, the detection system is able to detect objects in a scene image and output the result with the boundaries of objects drawn on the scene image. Sample result is shown in Figure 19. As we can see from the image, the outlines of the objects illustrate that objects have been applied complex transformations, especially on the mask can in the centre of the image. However, there are still quite a few keypoints detected and matched in this clustered evironment.

Figure 19: Sample result from object recognition system.

8.5 Summary

In this section, we evaluated the performance of SIFT library with images under 5 different geometric and photometric transformations. The evaluation result illustrates the success of this implementation of SIFT library, which realizes SIFT`s distinctive character, basically invariant to rotation, scaling and affine transform. As long as the transform is not too large, this implementation can provide a high correct matching ratio between two images, which is one of the key requirements of an object recognition system and other applications. However, the effect of transformation to the number of matching pairs is great. The number will reduce when the transformations become complex. This may be solved by redesigning this implementation and adjusting the variables in this SIFT library.

9 User guide

9.1 SIFT library

Introduction

SIFT library is a software library which implements David Lowe`s previous work, Scaled Invariant Feature Transformation [4]. The library is able to output the location, scale, orientation and a 128-D feature vector of the keypoints detected from the given image. For high-level users, certain variables and thresholds can be adapted to different requirements for research purposes or applications. The whole library is written under the standard of ANSI C and Open CV library.

What is in the library?

After unzipping the library, we can find several directories and files under the library directory as follows:

```
/app        // directory containing the source code of applications
/exec       // directory containing all executable files after compilation
/images     // directory containing some images for testing
/include    // directory containing all include files
/source     // directory containing all source code files
/test       // directory containing source code files of several testing program
makefile    // makefile for compilation
.cdtproject // Eclipse project file
.project    // Eclipse project file
```

Once the SIFT library has been compiled, another directory named "`/lib`" will be generated under the library directory, which will contains the library file "`libSIFT.a`".

How to Compile the library

Under Linux system, it is quite simple for the user to compile the SIFT library by typing:

```
[xw5133@localhost SIFTlib]$  make sift
ar   r   ./lib/SIFTlib.a   cvImageList.o   cvScalePyramid.o   cvScaleOctave.o
cvLocalExtrema.o cvKeyPoint.o cvDetectKeyPoint.o
ar: creating ./lib/SIFTlib.a
ranlib ./lib/SIFTlib.a
```

How to use the library in an application

When using the SIFT library in an application, the user need to link the library file as follows:

```
[xw5133@localhost SIFTlib]$  g++ `pkg-config --cflags opencv` -g -o app
application.o ./lib/SIFTlib.a -lm `pkg-config --libs opencv`
```

What to input

All images used in the library are grey scale images in the format of IplImage. When you calling the function in the library, please make sure your images are grey scale ones. If not, please convert the input image to a grey scale one. OpenCV supplies powerful function to read in an image as a grey scale image.

Library thresholds

```
#define CV_START_SCALE 1.0
```

CV_START_SCALE defines the scale of input image.

```
#define CV_PRE_GAUSSIAN_SIGMA  1.0
```

CV_PRE_GAUSSIAN_SIGMA defines the sigma for blurring the input image for the first time, which influence the performance of detector. Smaller sigma provides greater number of detailed keypoints. After thresholded, keypoints detected in following scale may be omitted. Please make sure it`s bigger than 0.5, which is also highly recommended in Lowe`s paper.

```
#define CV_OCTAVE_GAUSSIAN_SIGMA   1.6
```

CV_OCTAVE_GAUSSIAN_SIGMA defines the sigma for blurring the image when building the image scale pyramid, which combined with the number of scales in each octave, is one of the key points that influence the speed of the detector. Larger sigma consumes more computation resources. 1.6 is used throughout Lowe`s experiments and it guarantees good performance and reasonable computation time.

```
#define CV_SCALE_OCTAVE_LEVELS 3
```

CV_SCALE_OCTAVE_LEVELS defines the number of scale in each octave. In Lowe`s paper, 3 is proved to produce the best repeatability when matching the same object in different scenes and enough keypoints in each difference of Gaussian maps. However, for speed`s sake, fewer levels may be good for some applications that may not require large number of keypoints matching.

```
#define CV_MIN_IMAGE_SIZE  15
```

CV_MIN_IMAGE_SIZE defines the minimum size of image to process after downscaling the image from lower level in the pyramid. Smaller images may not produce keypoints at all even though it`s really fast to process these images.

```
#define CV_DOG_THRESHOLD   18.0
```

CV_DOG_THRESHOLD defines the value of Difference of Gaussian that will be used to detect keypoint on it when it`s bigger than this threshold. It will change greatly according to other thresholds. So it`s better to adjust this value whenever you change others. And it plays an important role in balancing the computation time and the number of keypoints.

```
#define CV_EIGENVALUE_RATIO20.0
```

CV_EIGENVALUE_RATIO eliminates the edge responses and remove unstable keypoint along edges. 10 is recommended in Lowe`s paper but in my case it`s related to other thresholds.
valueR = (RATIO + 1)^2 / RATIO;
if Tr(H)^2 / Det(H) is bigger than valueR, the keypoint will be omitted.
Tr and Det are the trace and determinant of hessian matrix(H).

```
#define CV_DVALUE_THRESHOLD5.0
```

CV_DVALUE_THRESHOLD is the threshold for accurate keypoint localization.
Value D is computed by Taylor expression of the scale-space function. Details please see Lowe`s paper.
If value D is less than this threshold, the keypoint will be eliminated.

```
#define CV_BIN_DIMENSION   36
```

CV_BIN_DIMENSION defines the dimension of orientation histogram.

```
#define CV_DESCRIPTOR_DIMENSION    4
#define CV_HISTOGRAM_DIMENSION 8
#define CV_FEATURE_VECTOR_DIMENSION    128
```

CV_DESCRIPTOR_DIMENSION defines the dimension of descriptor. The horizontal and the vertical dimension should be the same. Combined with CV_HISTOGRAM_DIMENSION, it decides the dimension of feature vector which equals to CV_DESCRIPTOR_DIMENSION ^ 2 * CV_HISTOGRAM_DIMENSION.

Library functions

```
void cvBuildKeyPointFeature(CvScaleOctave *octave,
                            CvLocalExtremaList *extremaList,
                            CvKeyPointList *keyPointList)
```

Build histogram for every local extrema

```
CvScaleOctave *cvBuildOctave(IplImage *base, float startScale)
```

Build a single octave, including the production of gaussian maps and DOGs.

```
CvScalePyramid *cvBuildPyramid(IplImage *base, float startScale)
```

Build the scale pyramid with the base image.

```
CvKeyPointList *cvDetectKeyPoint(IplImage *source)
```

David Lowe`s Scaled Image Feature Transform (SIFT). The main function of SIFT library. By calling this function, the application program will obtain the keypoint list according to the input image.

```
CvLocalExtremaList *cvDetectLocalExtrema(CvScaleOctave *octave)
```

Detect local Extrema in the given octave.

```
IplImage *cvDoubleImageScale(IplImage *source)
```

Double the size of the image by using linear interpolation. If (x, y) is the size of original image, (2x-2, 2y-2) will be the one of new image.

```
IplImage *cvHalfImageScale(IplImage *source)
```

Halve the size of the image by using linear interpolation. If (x, y) is the size of original image, (x/2, y/2) will be the one of new image.

```
IplImage *cvImageMinus(IplImage *first, IplImage *second)
```

Image operator `-`: first image - second image

```
void cvLocateKeyPoint(CvScaleOctave *octave,
                      CvLocalExtremaListNode *extrema,
                      float baseScale,
                      CvKeyPointList *keyPointList)
```

Locate the keypoint accurately and compute the orientation assignment

```
void cvPrintKeyPoint(CvKeyPointList *keyPointList)
```

Print out the keypoint list.

A sample program

```
#include "cv.h"
#include "highgui.h"
#include <stdlib.h>
#include <math.h>
#include <stdio.h>
#include "../include/cvSIFT.h"

int main( int argc, char** argv )
{
    IplImage *img;
    CvKeyPointList *keyPointList;
    int height, width, channels;

    if(argc != 2){
    printf("Usage: detectorTest <image-file-name>\n\7");
    exit(0);
    }

    /* load the image */
    img=cvLoadImage(argv[1], 0); /* force image to grey scale one*/
    if(!img){
    printf("Could not load image file: %s\n",argv[1]);
    exit(0);
    }

    keyPointList = cvDetectKeyPoint(img); /* obtain the keypoint list */
    cvPrintKeyPoint(keyPointList);   /* print out the keypoints */
    cvReleaseKeyPointList(keyPointList);
    cvReleaseImage(&img);

    return 0;
}
```

9.2 Object recognition system

Introduction

The Object recognition application is implemented to demonstrate the SIFT library. The system can detect several objects in a scene. All objects and the scene will be inputted as image files. In the end, the system will draw objects` boundaries and keypoints over the scene image as an output.

How to Compile

Under Linux system, the user only needs to type:

```
[xw5133@localhost SIFTlib]$ make objectDetector
```

How to use

The user only need to specify the file name of object images and scene image after the command as following format:

```
./objectDetector -S scene.pgm -O object1.pgm [object2.pgm ... objectn.pgm]
```

The number of object images is flexible but there should be only one scene image.

What to input

All input images ought to be PGM image files. Otherwise, the system will require you to name the correct image file.

10 Conclusion

10.1 Main achievements

This project has completed the main aims and objectives set at the very beginning. After experiencing the entire life circle of software development, we present this thesis as our final output that has main achievements as follows:

1. Documents produced in every stage of software development life circle, including requirement specification, design specification, implementation, development methodology, testing and user guide.

2. A brief introduction of image matching and a comprehensive review of image feature detectors and descriptors.

3. Implementing the SIFT library and an object recognition system to demonstrate it.

4. A critical evaluation of the SIFT library, which prove the good performance of this implementation.

10.2 Future Development

Even though the evaluation has prove that this implementation fulfil the requirement of this project, the SIFT library still cannot work as good as Lowe`s demo, especially the detector. More time should be spent on improving the detector to obtain better locations of the keypoints, which is also influence the performance of descriptor significantly.

A better design of implementing the descriptor ought to be figure out so that the keypoints extracted by SIFT library are more robust to the transformations such as affine transform and non-linear illumination change.

The performance of SIFT library, particularly its processing time, is the determinate factor of the viability of real time object recognition. According to Grabner`s experiments [33] for SIFT carried out on a computer with 3.2 GHz Pentium 4 processor, the processing time for an 800x600 image is 4.24s and 1.34s for 400x320. Several factors should be considered in the future, which may influence the performance of the SIFT library and object recognition system:

1. Quantity of features (keypoints): several techniques have been developed in Lowe`s paper [4, 6] to reduce the number of keypoints and localize the features accurately. However, the number of features that will be described is still colossal. Methods such as Information Theoretic Selection have been used to eliminate the number of candidate keypoints [32]. The result is fairly impressive though the accuracy of recognition has slightly declined.
2. Dimension of feature descriptor: the dimension of standard SIFT is 128, and given its size, it is time-consuming when building the descriptor and matching between the descriptors concurrently. Another descriptor, PCA-SIFT, has been created by Ke and Sukthankar [16] by the application of Principal Components Analysis to the normalized image gradients and reducing the dimension of the descriptor.
3. Size of the video images: This element directly influences the processing speed. A

reasonable size should be deduced by striking a balance between the processing speed and the quality of the output images.

4. Organization of feature database: indexing high-dimensional data is still a problem, which actually decides the speed of matching. As a result, an optimal storage of keypoints descriptor should be considered.

SIFT brings a lot of advantages but also suffers from the above drawbacks. In essence, the problem is a trade-off between the quality of the matching process and the processing time. A real time object recognition system is the final objective of this project.

10.3 Summary

The final section discussed the main achievements and possible future works of this project. Possible aspects that may influence the performance of SIFT library and object recognition system are listed and ought to be considered in the future development.

Bibliography

[1] T. Lindeberg. Feature Detection with Automatic Scale Selection. *International Journal of Computer Vision*, 30(2): 79-116, 1998.

[2] C. Harris and M. Stephens. A Combined Corner and Edge Detector. In *Alvey Vision Conference*, pages 147-151, 1988.

[3] K. Mikolajczyk, C. Schmid. Indexing Based on Scale Invariant Interest Points. In *Proceedings of the 8th International Conference of Computer Vision, Vancouver, Canada*, pages 525-531, 2001.

[4] D. Lowe. Distinctive Image Features from Scale-Invariant Keypoints. *International Journal of Computer Vision*, 2(60): 91-110, 2004.

[5] K. Mikolajczyk and C. Schmid. Scale and Affine Invariant Interest Point Detectors. *International Journal of Computer Vision*, 1(60): 63-86, 2004.

[6] D.G. Lowe. Object Recognition from Local Scale-Invariant Features. In *Proceedings of the 7th International Conference of Computer Vision, Kerkyra, Greece*, pages 1150-1157, 1999.

[7] T. Lindeberg and J. Garding. Shape-Adapted Smoothing in Estimation of 3-D Shape Cues from Affine Deformations of Local 2-D Brightness Structure. *Image and Vision Computing*, 15(6): 415-434, 1997.

[8] K. Mikolajczyk and C. Schmid. An Affine Invariant Interest Point Detector. In *Proceedings of 7th European Conference of Computer Vision, Copenhagen, Denmark*, volume I, pages 128-142, 2002.

[9] A. Baumberg. Reliable Feature Matching across Widely Separated Views. In *Proceedings of Conference on Computer Vision and Pattern Recognition, Hilton Head Island, South Carolina, USA*, pages 774-781, 2000.

[10] F. Schaffalitzky and A. Zisserman. Multi-View Matching for Unordered Image Sets. In *Proceedings of the 7th European Conference on Computer Vision, Copenhagen, Denmark*, pages 414-431, 2002.s

[11] T. Tuytelaars and L. Van Gool. Matching Widely Separated Views Based on Affine Invariant Regions. *International Journal of Computer Vision*. 1(59): 61-85, 2004.

[12] J. Matas, O. Chum, M. Urban, and T. Pajdla. Robust Wide Baseline Stereo from Maximally Stable Extremal Regions. In *Proceedings of the 13th British Machine Vision Conference, Cardiff, UK*. pages 384-393, 2002.

[13] T. Kadir, M. Brady, and A. Zisserman. An Affine Invariant Method for Selecting Salient Regions in Images. In *Proceedings of the 8th European Conference on Computer Vision, Prague, Czech Republic*, pages 345-457, 2004.

[14] K. Mikolajczyk, T. Tuytelaars, C. Schmid, A. Zisserman, J. Matas, F. Schaffalitzky, T. Kadir, and L.V. Gool. A Comparison of Affine Region Detectors. *International Journal of Computer Vision*, 65(1/2), 43-72, 2005

[15] K. Mikolajczyk and C. Schmid. A performance evaluation of local descriptors. IEEE *Transactions on Pattern Analysis and Machine Learning*, 27(10): 1615-1630, 2005.

[16] Y. Ke and R. Sukthankar. PCA-SIFT: A More Distinctive Representation for Local Image Descriptors. In *Proceedings of Conference on Computer Vision and Pattern Recognition, Washington, USA*, pages 511-517, 2004.

[17] A. Johnson and M. Hebert. Object Recognition by Matching Oriented Points. In *Proceedings of Conference on Computer Vision and Pattern Recognition, Puerto Rico, USA*, pages 684-689, 1997.

[18] S. Lazebnik, C. Schmid, and J. Ponce. Sparse Texture Representation Using Affine-Invariant Neighborhoods. In *Proceedings of Conference on Computer Vision and Pattern Recognition, Madison, Wisconsin, USA*, pages 319-324, 2003.

[19] R. Zabih and J. Woodfill. Non-Parametric Local Transforms for Computing Visual Correspondence. In *Proceedings of the 3rd European Conference on Computer Vision, Stockholm, Sweden*, pages 151-158, 1994.

[20] A. Ashbrook, N. Thacker, P. Rockett, and C. Brown. Robust Recognition of Scaled Shapes Using Pairwise Geometric Histograms. In *Proceedings of the 6th British Machine Vision Conference, Birmingham, England*, pages 503-512, 1995.

[21] S. Belongie, J. Malik, and J. Puzicha. Shape Matching and Object Recognition Using Shape Contexts. IEEE *Transactions on Pattern Analysis and Machine Intelligence*, 2(4): 509-522, 2002.

[22] X. Wu and B. Bhanu. Gabor Wavelet Representation for 3-D Object Recognition. IEEE *Transactions on Image Processing*, 6(1): 47-64, 1997.

[23] L. Florack, B. ter Haar Romeny, J. Koenderink, and M. Viergever. General Intensity Transformations and Second Order Invariants. In *Proceedings of the 7th Scandinavian Conference on Image Analysis, Aalborg, Denmark*, pages 338-345, 1991.

[24] W. Freeman and E. Adelson. The Design and Use of Steerable Filters. IEEE *Transactions on Pattern Analysis and Machine Intelligence*, 13(9): 891-906, 1991.

[25] L. Van Gool, T. Moons, and D. Ungureanu. Affine/Photometric Invariants for Planar Intensity Patterns. In *Proceedings of the 4th European Conference on Computer Vision, Cambridge, England*, pages 642-651, 1996.

[26] A. Vardy and F. Oppacher. A scale invariant local image descriptor for visual homing. *Lecture notes in Artificial Intelligence*, 3575: 362-381, 2005.

[27] S. Lazebnik, C. Schmid and J. Ponce. A sparse texture representation using local affine regions. IEEE *Transaction on Pattern Analysis and Machine Intelligence*, 27(8): 1265-1278, 2005.

[28] A. Anjulan and N. Canagarajah. Affine invariant feature extraction using symmetry. *Lecture Notes in Computer Science*, 3708: 332-339, 2005.

[29] K. Terasawa, T. Nagasaki and T. Kawashima. Robust matching method for scale and rotation invariant local descriptors and its application to image indexing. *Lecture Notes in Computer Science*, 3689: 601-615, 2005.

[30] J. Koenderink and A. van Doorn. Representation of Local Geometry in the Visual System. *Biological Cybernetics*, 55: 367-375, 1987.

[31] C. Schmid and R. Mohr. Local grayvalue invariants for image retrieval. IEEE *Transactions on Pattern Analysis and Machine Intelligence*, 19(5): 530-535, 1997.

[32] G. Fritz, C. Seifert and M. Kumar. Building detection from mobile imagery using informative SIFT descriptors. *Lecture Notes in Computer Science*, 3540: 629-638, 2005.

[33] M. Grabner, H. Grabner and H. Bischof. Fast approximated SIFT. *Lecture Notes in Computer Science*, 3851: 918-927, 2006.

[34] I. T. Joliffe. Principal Component Analysis. Springer-Verlag, 1986.

[35] G.L. Hobrough. Automatic stereo plotting, Photogrammetric Eng. & Remote Sensing, (25) 5, 763-769, 1959.

[36] C. HEIPKE. Overview of Image Matching Techniques. OEEPE - Workshop on the application of digital photogrammetric workstations, Lausanne, Switzerland, March 4-7, 1996. (http://phot.epfl.ch/workshop/wks96/art_3_1.html)

[37] E. Trucco and A. Verri. Introductory techniques for 3-D computer vision. Prentice Hall, 1998.

[38] A. Agarwal and B. Triggs. A local basis representation for estimating human pose from cluttered images. Lecture Notes in Computer Science, 3851: 50-59, 2006.

[39] S. Jeong, J. Chung and S. Lee. Design of a simultaneous mobile robot localization and spatial context recognition system. Lecture Notes in Artificial Intelligence, 3683: 945-952, 2005.

[40] S. I. Olsen. Exemplar based recognition of visual shapes. Lecture Notes in Computer Science, 3540: 852-861, 2005.

[41] H. Y. Lee, C. H. Lee and H. K. Lee. Feature-based image watermarking method using scale-invariant keypoints. Lecture Notes in Computer Science 3768: 312-324, 2005.

[42] J. Kosecka, F. Y. Li and X. L. Yang. Global localization and relative positioning based on scale-invariant keypoints. Robotics and Autonomous System, 52(1): 27-38, 2005.

[43] P. Loncomilla and J. Ruiz-Del-Solar. Improving SIFT-based object recognition for robot applications. Lecture Notes in Computer Science, 3617: 1084-1092, 2005.

[44] M. Brown and D. G. Lowe. Recognising panoramas. International Conference on Computer Vision, Nice, France, pages 1218-25, 2003.

[45] OpenCV Coding Style Guide. (http://www.intel.com/technology/computing/opencv/coding_style/)

[46] Official Wiki for OpenCV. (http://opencvlibrary.sourceforge.net/)

[47] J. Beis and D. G. Lowe. Shape indexing using approximate nearest-neighbour search in high-dimensional spaces. In Conference on Computer Vision and Pattern Recognition, Puerto Rico, pages: 1000-1006, 1997.

Appendix A: Sample Source Code

CvDetectKeyPoint.c:

```
#include <stdio.h>
#include "cv.h"
#include "cvDetectKeyPoint.h"


/* David Lowe`s Scaled Image Feature Transform (SIFT). The main function
      of SIFT library.
 * By calling this function, the application program will obtain the keypoint
      list
 * according to the input image.
 */
CvKeyPointList *cvDetectKeyPoint(IplImage *source)
{
      IplImage  *base, *blurred;
      float startScale;
      int level = 1;
      CvScalePyramid      *scalePyramid;
      CvOctaveListNode *octaveTemp;
      CvLocalExtremaList *extremaList;
      CvKeyPointList *keyPointList;

      /* preprocess Gaussian blur on the doubled image */
      blurred = cvCloneImage(source);
      cvSmooth(source, blurred, CV_GAUSSIAN, 0, 0,
      CV_PRE_GAUSSIAN_SIGMA);

/* double the image size as the base of the pyramid and the scale will be
halved */
base = cvDoubleImageScale(blurred);
if (base == NULL){
      printf("The size of image is too small.\n");
      exit(1);
}
startScale = CV_START_SCALE / 2.0;

/* build the image scale pyramid */
scalePyramid = cvBuildPyramid(base, startScale);

/* detect local extrema and compute discriptor for each keypoint */
keyPointList = (CvKeyPointList *)calloc(1, sizeof(CvKeyPointList));
octaveTemp = scalePyramid->octaves->start;
while (octaveTemp != NULL) {
      /* detect keypoint location and its scale in current octave*/

      extremaList = cvDetectLocalExtrema(octaveTemp->octave);

      /* compute the gradient magnitude and orientation of scaled images
in this octave */
      cvComputeMagnOri(octaveTemp->octave);

      /* build histogram for every localExtrema */
      cvBuildKeyPointFeature(octaveTemp->octave, extremaList,
keyPointList);
```

```
            /* free space */
            cvReleaseExtremaList(extremaList);

            /* set to next octave */
            octaveTemp = octaveTemp->next;
        }

        cvReleasePyramid(scalePyramid);
        return keyPointList;
}

/* print out keypoint list */
void cvPrintKeyPoint(CvKeyPointList *keyPointList){
        CvKeyPointListNode *tempNode;
        int i;

        /* print out the number of keypoints and the dimension of feature vector
        */
        printf("%d %d\n", keyPointList->num,
        CV_FEATURE_VECTOR_DIMENSION);
        tempNode = keyPointList->start;
        while(tempNode != NULL){
            /* print out the location, scale and orientation of keypoint */
            printf("%.2f %.2f %.2f %.3f\n", tempNode->x, tempNode->y,
        tempNode->imgScale, tempNode->orientation);

            /* print out the feature vector */
            for (i=0; i<CV_FEATURE_VECTOR_DIMENSION; i++){
                if (i%20 == 0 && i != 0){
                    printf("\n");
                }
                printf("%.0f ", tempNode->featureVector[i]);
            }
            printf("\n");

            /* set to next keypoint */
            tempNode = tempNode->next;
        }
}
```

CvKeyPoint.c:

```
#include <stdio.h>
#include "cvKeyPoint.h"

/* build histogram for every local extrema */
void cvBuildKeyPointFeature(CvScaleOctave *octave,
                            CvLocalExtremaList *extremaList,
                            CvKeyPointList *keyPointList)
{
        CvLocalExtremaListNode *tempExtrema;
        CvKeyPointListNode *tempKeyPoint;
        CvKeyPointList *keyPointCandidateList;

        tempExtrema = extremaList->start;
        while (tempExtrema != NULL){
            /* obtain the accurate location and scale */
            cvLocateKeyPoint(octave, tempExtrema, octave->baseScale,
        keyPointList);
```

```
            tempExtrema = tempExtrema->next;
        }
}

/* locate the keypoint and compute the orientation assignment */
void cvLocateKeyPoint(CvScaleOctave *octave,
                                CvLocalExtremaListNode *extrema,
                                float baseScale,
                                CvKeyPointList *keyPointList)
{
        float keyPointScale, circleSigma, gaussianWeight;
        IplImage *magnitude, *orientation;
        int i, j;
        int circleRadius, xStart, xEnd, yStart, yEnd, binIdx, *peakLocation,
        descriptorRadius;
        float *bin, binMax, binPrev, binNew, tempOri, xR, yR, *degreeAdjusted,
        *valueAdjusted;
        CvKeyPointListNode *tempNode;

        /* compute the scale of keypoint */
        keyPointScale = CV_OCTAVE_GAUSSIAN_SIGMA * powf(2.0,
        ((float)extrema->level)/(float)(CV_SCALE_OCTAVE_LEVELS+2.0))
                                + extrema->scaleChanged;

        /* compute the variable for gaussian weight circle */
        circleSigma = keyPointScale * 3.0;    /* changed from 3.0 to 1.5 */
        circleRadius = (int)(circleSigma * 1.5 + 0.5);

        /* get the gradient magnitude and orientation of the image with the
keypoing scale */
octave->magnitudeMapList->current                                =
octave->magnitudeMapList->start;
octave->orientationMapList->current                              =
octave->orientationMapList->start;
for (i=1; i<extrema->level; i++){
        octave->magnitudeMapList->current                        =
octave->magnitudeMapList->current->next;
        octave->orientationMapList->current                      =
octave->orientationMapList->current->next;
}
magnitude = octave->magnitudeMapList->current->img;
orientation = octave->orientationMapList->current->img;

/* set the border of the neighborhood */
xStart = (int)((extrema->x - circleRadius) > 1 ? (extrema->x -
circleRadius) : 1);
xEnd = (int)((extrema->x + circleRadius) < (magnitude->height - 1) ?
(extrema->x + circleRadius) : (magnitude->height - 1));
yStart = (int)((extrema->y - circleRadius) > 1 ? (extrema->y -
circleRadius) : 1);
yEnd = (int)((extrema->y + circleRadius) < (magnitude->width - 1) ?
(extrema->y + circleRadius) : (magnitude->width - 1));

/*locate space for bin array */
bin = (float *)calloc(1, CV_BIN_DIMENSION * sizeof(float));
degreeAdjusted = (float *)calloc(1, sizeof(float));
valueAdjusted = (float *)calloc(1, sizeof(float));
peakLocation = (int *)calloc(1, CV_BIN_DIMENSION * sizeof(int));
```

```
/* compute the bin */
for (j=xStart; j<xEnd; j++) {
      for (i=yStart; i<yEnd; i++) {
            xR = j - extrema->x;
            yR = i - extrema->y;

            /* check if the point is in the gaussian weight circle */
            if ((xR*xR + yR*yR) >= circleRadius * circleRadius){
                  continue;
            }

            /*compute the gaussian weight */
            gaussianWeight = expf(- ((xR * xR + yR * yR) / (2.0 *
circleSigma * circleSigma)));

            /* fill in the bin*/
            tempOri                    =                    ((float
*)(orientation->imageData))[j*(orientation->widthStep / sizeof(float)) +
i];
            binIdx = (int)((tempOri + CV_PI) / (2.0 * CV_PI) *
(float)CV_BIN_DIMENSION);
            bin[binIdx]                +=                   ((float
*)(magnitude->imageData))[j*(magnitude->widthStep / sizeof(float)) + i]
* gaussianWeight;
      }
}

/* smooth the orientation bin */
binPrev = bin[CV_BIN_DIMENSION-1];
for (i=0; i<CV_BIN_DIMENSION; i++){
     binNew = (binPrev + bin[i] + bin [(i+1) % CV_BIN_DIMENSION])
/ 3.0;
      binPrev = bin[i];
      bin[i] = binNew;
}

/* find the maximum and its index of the bin */
binMax = 0;
binIdx = 0;
for (i=0; i<CV_BIN_DIMENSION; i++){
      if (bin[i] > binMax){
            binMax = bin[i];
            binIdx = i;
      }
}

/* orientation adjustment */
cvParabolaFit(bin[(i + CV_BIN_DIMENSION - 1) %
CV_BIN_DIMENSION], bin[i],
                  bin[(i + 1) % CV_BIN_DIMENSION], degreeAdjusted,
valueAdjusted);

/* detect local peaks other than the maximum */
for (i=0; i<CV_BIN_DIMENSION; i++){
      if (bin[i] == binMax){
            peakLocation[i] = CV_BOOLEAN_TRUE;
            continue;
```

```
            }
//          else {
//                peakLocation[i] = CV_BOOLEAN_FALSE;
//          }
            if (bin[i] < (0.8 * (binMax + *valueAdjusted))){
                continue;
            }
            if (bin[i] < bin[(i+1) % CV_BIN_DIMENSION] || bin[i] < bin[(i-1+CV_BIN_DIMENSION) % CV_BIN_DIMENSION]){
                continue;
            }
            peakLocation[i] = CV_BOOLEAN_TRUE;
        }

        /* compute descriptor and add keypoints to the keyPointList */
        descriptorRadius = (int)(((CV_DESCRIPTOR_DIMENSION + 1.0) / 2.0) * sqrt(2.0) * 2.0 * keyPointScale + 0.5);
//      descriptorRadius = CV_DESCRIPTOR_DIMENSION * 2;
        for (i=0; i<CV_BIN_DIMENSION; i++){
            if (peakLocation[i] == CV_BOOLEAN_TRUE){
                /* calloc space for the candidate key point list */
                tempNode = (CvKeyPointListNode *)calloc(1, sizeof(CvKeyPointListNode));
                cvParabolaFit(bin[(i + CV_BIN_DIMENSION - 1) % CV_BIN_DIMENSION], bin[i],
                        bin[(i + 1) % CV_BIN_DIMENSION], degreeAdjusted, valueAdjusted);
                tempNode->orientation = ((float)i + *degreeAdjusted) * 2.0 / ((float)CV_BIN_DIMENSION) * CV_PI - CV_PI;
                tempNode->x = extrema->x;
                tempNode->y = extrema->y;
                tempNode->level = extrema->level;
                tempNode->imgScale = keyPointScale;
                tempNode->baseScale = octave->baseScale;
                /* compute the image feature vector for every keypoint*/
                if (cvComputeFeatureVector(tempNode, octave, magnitude, orientation, descriptorRadius) == CV_RETURN_SUCCESS){
                    tempNode->x = tempNode->x * tempNode->baseScale;
                    tempNode->y = tempNode->y * tempNode->baseScale;
                    tempNode->imgScale = tempNode->imgScale * tempNode->baseScale;
                    if(cvAddKeyPointListNode(keyPointList, tempNode) == CV_RETURN_FAILURE){
                        printf("ERROR: Can`t add node to the keypoint list.\n");
                        exit(2);
                    }
                }
                else {
                    free(tempNode);
                }
            }
        }
    }

    /* add node to the keypoint list */
    int cvAddKeyPointListNode(CvKeyPointList *list,
                              CvKeyPointListNode *node)
```

```
{
    if (node == NULL){
        return CV_RETURN_FAILURE;
    }

    if (list->start == NULL) {
        list->start = node;
        list->current = list->start;
        list->end = list->start;
        list->num++;
    }
    else{
        list->end->next = node;
        list->end = list->end->next;
        list->num++;
    }
    return CV_RETURN_SUCCESS;
}

/* compute the image feature vector for every keypoint in the list */
int cvComputeFeatureVector(CvKeyPointListNode *node,
                           CvScaleOctave *octave,
                           IplImage *magnitude,
                           IplImage *orientation,
                           int radius)
{
    float ori, xR, yR, sigmaSq, magWeight, oriDimTemp, gramFactor,
    halfDesDim, xWeight[2], yWeight[2], oriWeight[2];
    int i, j, xDim[2], yDim[2], oriDim[2], vectorIdx, xAB, yAB, l, m, n;

ori = - node->orientation;
sigmaSq = radius * radius / 4.0;
gramFactor = 2.0 * node->imgScale;      /* scale down to histogram
coordinate */
halfDesDim = ((float)CV_DESCRIPTOR_DIMENSION) / 2.0;


/* calloc space for the feature vector */
node->featureVector = (float *)calloc(1,
CV_FEATURE_VECTOR_DIMENSION * sizeof(float));


for(j=-radius; j<radius; j++){
    for(i=-radius; i<radius; i++){
        /* compute the relative coordinate of x and y according to
keypoint`s orientation*/
        xR = cosf(ori) * j - sinf(ori) * i;
        yR = sinf(ori) * j + cosf(ori) * i;

        /* compute the absolute coordinate of x and y */
        xAB = (int)(j + node->x + 0.5);
        yAB = (int)(i + node->y + 0.5);

        /* check if it`s out of image */
        if (xAB<0 || xAB>(magnitude->height - 1) || yAB<0 ||
yAB>(magnitude->width - 1)){
            continue;
        }
```

```
            /* compute the relative coordinate of x and y in orientation histogram according to keypoint`s scale */
            xR = xR / gramFactor;
            yR = yR / gramFactor;

            /* check if it`s out of range */
            if (xR<=-(halfDesDim + 0.5) || xR>=(halfDesDim + 0.5) || yR<=-(halfDesDim + 0.5) || yR>=(halfDesDim + 0.5)){
                continue;
            }



            /* compute the magnitude weight according to the distance to the centre */
            magWeight = exp(-(xR * xR + yR * yR) / sigmaSq * 2.0) *
                        (((float *)(magnitude->imageData + xAB * magnitude->widthStep))[yAB]);
//          magWeight = (1.0 - sqrtf(xR * xR + yR * yR) / (halfDesDim + 0.5) / sqrtf(2.0)) *
//                      (((float *)(magnitude->imageData))[xAB * (magnitude->widthStep/sizeof(float)) + yAB]);



           /* rerange xR and yR to (0, CV_DESCRIPTOR_DIMENSION) */
            xR += halfDesDim - 0.5;
            yR += halfDesDim - 0.5;

            /* compute the index and weight in the feature vector */
            xDim[0] = (int) xR;
            xWeight[0] = (1.0 - (xR - xDim[0]));
            yDim[0] = (int) yR;
            yWeight[0] = (1.0 - (yR - yDim[0]));
            xDim[1] = (int) (xR + 1.0);
            xWeight[1] = xR - xDim[1] + 1.0;
            yDim[1] = (int) (yR + 1.0);
            yWeight[1] = yR - yDim[1] + 1.0;

            oriDimTemp = ((float *)(orientation->imageData))[xAB*(orientation->widthStep/sizeof(float))+yAB];
            oriDimTemp = oriDimTemp + CV_PI - node->orientation;
            /* make sure that the value of oriDimTemp is in the range [0, 2*CV_PI] */
            if (oriDimTemp > 2 * CV_PI){
                oriDimTemp = oriDimTemp - 2 * CV_PI;
            }
            else if (oriDimTemp < 0){
                oriDimTemp = oriDimTemp + 2 * CV_PI;
            }

            oriDimTemp = oriDimTemp / 2.0 / CV_PI * (CV_HISTOGRAM_DIMENSION-1);  /* need to rotate? */
            oriDim[0] = (int) oriDimTemp;
            oriDim[1] = (oriDim[0] + 1) % CV_HISTOGRAM_DIMENSION;
            oriWeight[0] = 1.0 - (oriDimTemp - oriDim[0]);
```

```
                oriWeight[1] = oriDimTemp - oriDim[0];

                for(l=0; l<2; l++){
                    for(m=0; m<2; m++){
                        for(n=0; n<2; n++){
                            vectorIdx = (yDim[m] * CV_DESCRIPTOR_DIMENSION + xDim[l]) * CV_HISTOGRAM_DIMENSION + oriDim[n];
                            if (vectorIdx >= 128 || vectorIdx < 0){
//                              if (xDim[l] >= 4 || xDim[l] < 0){
//                                  printf("x=%d\n", xDim[l]);
//                              }
//                              if (yDim[m] >= 4 || yDim[m] < 0){
//                                  printf("y=%d\n", yDim[m]);
//                              }
//                              if (oriDim[n] >= 7 || oriDim[n] < 0){
//                                  printf("ori=%d\n", oriDim[n]);
//                              }
//                              printf("ERROR: the index is out of range.\n");
                                continue;
                            }
                            /* add the contribution to the vector */
                            node->featureVector[vectorIdx] = node->featureVector[vectorIdx]
                                + xWeight[l] * yWeight[m] * oriWeight[n] * magWeight;
                        }
                    }
                }
            }
        }

        /* normalize the vector twice, one after magnitudes thresholding */
        if (normAndThesh(node->featureVector) == CV_RETURN_FAILURE){
            return CV_RETURN_FAILURE;
        }

        return CV_RETURN_SUCCESS;
    }

    /* normalize, theshold for reducing the influence of large image gradient
     * magnitudes and then normalize the feature vector again
     */
    int normAndThesh(float *vector)
    {
        int i;
        float norm=0.0, range, newVector[CV_FEATURE_VECTOR_DIMENSION];

        /* normalize the feature for the first time */
        for(i=0; i<CV_FEATURE_VECTOR_DIMENSION; i++){
            norm = norm + vector[i] * vector[i];
        }
        norm = sqrt(norm);
        if (norm == 0){
```

```
            return CV_RETURN_FAILURE;
        }

        /* thresholding to get rid of the influence of large gradient magnitude */
        for(i=0; i<CV_FEATURE_VECTOR_DIMENSION; i++){
            newVector[i] = vector[i] / norm;
            if (newVector[i] > CV_MAX_MAGNITUDE_THRESHOLD){
                newVector[i] = CV_MAX_MAGNITUDE_THRESHOLD;
            }
        }

        norm = 0.0;
        /* normalize the feature for the second time */
        for(i=0; i<CV_FEATURE_VECTOR_DIMENSION; i++){
            norm = norm + newVector[i] * newVector[i];
        }
        norm = sqrt(norm);
        if (norm == 0){
            return CV_RETURN_FAILURE;
        }

        for(i=0; i<CV_FEATURE_VECTOR_DIMENSION; i++){
            vector[i] = fabsf(newVector[i] * 255.0 / norm);
        }
        return CV_RETURN_SUCCESS;
}

/* fit three points to a parabola for obtaining extremium
  * f(x) = a (x - m)^2 + d
  */
int cvParabolaFit(float left,
                  float centre,
                  float right,
                  float *degreeAdjusted,
                  float *valueAdjusted)
{
        float a, m, d;

        *degreeAdjusted = 0.0;
        *valueAdjusted = 0.0;

        a = ((left + right) - 2.0 * centre) / 2.0;

        if (a == 0.0){
            return CV_RETURN_FAILURE;
        }

        m = (((left - centre) / a) - 1.0) / 2.0;
        d = centre - m * m * a;

        if (m < -0.5 || m > 0.5){
            return CV_RETURN_FAILURE;
        }

        *degreeAdjusted = m;
        *valueAdjusted = d - centre;
        return CV_RETURN_SUCCESS;
}
```

CvScalePyramid.c:

```
#include <stdio.h>
#include "cvScalePyramid.h"
#include "cvScaleOctave.h"
#include "cvImageList.h"

/* Build the scale pyramid with the base image */
CvScalePyramid *cvBuildPyramid(IplImage *base,
                               float startScale)
{
    CvScalePyramid  *scalePyramid;
    IplImage  *imageTemp;
    CvScaleOctave *octaveTemp;
    CvScaleOctave *octave;

    /* locate memory to the scale pyramid and octave list */
    scalePyramid = (CvScalePyramid *)calloc(1, sizeof(CvScalePyramid));
    scalePyramid->octaves = (CvOctaveList *)calloc(1,
    sizeof(CvOctaveList));

    /* process the base image and build the pyramid */
    imageTemp = base;
    octaveTemp = NULL;
    while (imageTemp->width >= CV_MIN_IMAGE_SIZE &&
    imageTemp->height >= CV_MIN_IMAGE_SIZE) {
        octave = cvBuildOctave(imageTemp, startScale);

        /* add current octave to the octavelist */
        if (cvAddOctaveListNode(scalePyramid->octaves, octave) !=
CV_RETURN_SUCCESS){
            printf("ERROR: fail to add a octave node to the list.\n");
            exit(2);
        }

        /* scale down the image for next level of pyramid */
        imageTemp =
cvHalfImageScale(octave->gaussianMapList->end->img);

        /* set the neighbor levels in pyramid */
        if (octaveTemp != NULL){
            octave->up = octave;
        }
        octave->down = octaveTemp;
        octaveTemp = octave;
        startScale = startScale * 2.0;
    }
    return scalePyramid;
}
```