SUN YAT-SEN UNIVERSITY, SYSU-CMU JOINT INSTITUTION OF ENGINEERING

Final Project Code Manual

Shitao Weng(sweng@andrew.cmu.edu)
Shushan Chen(shushanc@andrew.cmu.edu)

November 30, 2014

Contents

1 Introduction

This document is a code manual for our final project. Our project is basically a ground-up implementation of Scale Invariant Feature Matching with application to image classification.

In order to test the accuray of SIFT feature, two public face recognition database are used here. One is **AT&T** face database [1], which containing 400 images for 40 person. Another one is **Yale** face dataset [2], which contains 165 images for 15 subjects. In these two databases, our SIFT feature matching gains more than 95% accuracy. Several other demo programs, such as real time recognizing person in camera video, are also written to show how our codes work. For detailed information about demo, please refer to "Installation-Usage.pdf". For an overview of this document, please refer to table 1.1, which shows the document structure.

Section	**Content**	**Related Source Files**
Section 2	SIFT	src/include/SiftExtractor.h src/lib/SiftExtractor.cpp
Section 3.1	SIFT Match	src/include/SiftMatcher.h src/lib/SiftMatcher.cpp
Section 3.2	KD TREE	src/include/kdTree.h src/lib/kdTree.cpp
Section 4.1	Demo	src/kd_demo.cpp src/match2img.cpp src/match2db.cpp src/accuracyTest.cpp src/camera.cpp
Section 4.2 & 4.3 & 4.4	Other Source codes	src/include/feature.h src/lib/feature.cpp src/include/ImageSet.h src/lib/ImageSet.cpp src/include/imgSuffix.def src/include/ImageFileName.h src/lib/ImageFileName.cpp src/include/configure.h src/include/configureFrontEnd.h src/include/siftConfigure.def

Table 1.1: Document structure

2 SIFT

2.1 Introduction

SIFT (Scale Invariant Feature Transform) is developed by Lowe [3] [4] for distinctive image feature generation in object recognition applications. These features have been shown to be invariant to image rotation and scale, and robust across a substantial range of affine distortion, addition of noise, and change in illumination [5]. It is able to perform reliable matching between different views of an object or scene.

2.2 Key Stages

To generate the set of image SIFT features, there are four major stages (detailed description and related code is provided in next part, "*Code Detailed Description*"):

Stage 1. Scale-space extrema detection

The first stage is to detect the potential interest points that are invariant under different scales. This can be achieved by searching across all possible scales (known as scale space) to find the stable features. In SIFT, location of keypoints are defined as maxima and minima of the result of *different of Gaussians (DoG)* function applied in scale space to a series of smoothed and resampled images.

Stage 2. Accurate Keypoint localization

To gain more accurate locations of keypoints, quadratic function fitting is applied to determine the interpolated location of keypoints. Also, low contrast candidate points and edge response points are removed in this stage.

Stage 3. Orientation assignment

Based on local image gradient directions, dominant orientations (one or more) are assigned to each keypoint location. So far, each feature obtains orientation, scale, and location. Before performing operations, image data can be transformed relative to such information, which ensures the invariance.

Stage 4. Keypoint descriptor representation

Descriptor is a vector representation computed for the local image region that is as distinctive as possible at each candidate keypoint. It is obtained by considering pixels around keypoint location, blurring and resampling of local image orientation planes.

2.3 Code Detailed Description

Relevant source files:

1. src/include/SiftExtractor.h
2. src/lib/SiftExtractor.cpp

Note that here only the important functions are described to help users make sense about the process of SIFT, some small functions are not described here.

2.3.1 Creating the Difference of Gaussian Pyramid

This step is to construct "*Gaussian Scale Space*" for input image, which is performed by convolution of the original image with gaussian functions of different widths, and to calculate the *difference of Gaussian (DoG)* as the difference between two filtered images.

```
void generatePyramid(Mat * img, vector< Octave > &octaves);
{
    void generateBlurLayers(int layers, double *sigmas);
    void generateDOGPyramid(vector< Octave > & octaves);
}
```

Note: By adding brace around some functions, it means that these functions are called by the outer function. Take above functions for example, generateBlurLayers and generateDOGPyramid are called by generatePyramid. This representation is also used in the rest of this document.

Function	Function Description
generatePyramid	the main function to perform the pipeline of generating **DoG** and all other functions are called by it
generateBlurLayers	use specific Gaussian filter to generate corresponding layer of scale space
generateDOGPyramid	use the result of second function, Gaussian scale space, to generate DoG pyramid

Table 2.1: Function Description

Function	Parameter	Type	Parameter Description
generatePyramid	img	[in]	image data
	octaves	[out]	DoG Pyramid for input image
generateBlurLayers	layers	[in]	layer index in Gaussian scale space
	sigmas	[in]	sigma for Gaussian filter
generateDOGPyramid	octaves	[out]	DoG Pyramid for input image

Table 2.2: Parameter Description

2.3.2 Extrema Detection

In this step, extrema (maxima and minima) points in the **DoG** pyramid are detected as the keypoint candidates. In order to do this, the sample point is compared to its eight neighbors in the current layer and other eighteen points in above and below layers.

```
void extremaDetect(Octave & octave, int octIdx, vector<Feature> &
    outFeatures);
{
    bool isExtrema(Octave & octave, int layer, int x, int y,
        EXTREMA_FLAG_TYPE *nxtMinFlags, EXTREMA_FLAG_TYPE* nxtMaxFlags,
        int rollIdx);
}
```

Function	Function Description
extremaDetect	the main function to perform extrema detection
isExtrema	to check whether it is an extrema or not.

Table 2.3: Function Description

Function	Parameter	Type	Parameter Description
extremaDetect	octave	[in]	octave in Gaussian scale space
	octIdx	[in]	layer index
	outFeatures	[out]	extrema points
isExtrema	octave	[in]	octave in Gaussian scale space
	layer	[in]	layer index
	(x,y)	[in]	point position in that layer
	other parameters	[in]	improve efficiency of detecting extrema points
	-	[return]	If it is a extrema or not

Table 2.4: Parameter Description

2.3.3 Accurate keypoint location

This stage is to remove some undesirable candidate points by checking if they are in low contrast or poorly localized on an edge.

```
bool shouldEliminate(Octave &octave, int &layer, int &x, int &y, double *
    _X);
{
    bool poorContrast(Octave & octave, int &layer, int &x, int &y, double
        *_X);
    bool edgePointEliminate(Mat &img, int x, int y);
}
```

Function	Function Description
shouldEliminate	to determine whether the candidate point should be eliminated or not and it depends on the result of other two functions
poorContrast	checks if the candidate point is in low contrast
edgePointEliminate	checks if its location is along edges or not

Table 2.5: Function Description

Function	Parameter	Type	Parameter Description
shouldEliminate	octave	[in]	octave in Gaussian scale space
	layer	[in/out]	layer index in octave, is updated after interpolation
	(x,y)	[in/out]	position in that layer, are updated after interpolation
	_X	[out]	offset(Δ[layer,x,y]), are updated interpolation
	-	[return]	It can be eliminated or not
poorcontrast	octave	[in]	octave in Gaussian scale space
	layer	[in/out]	layer index in octave, is updated after interpolation
	(x,y)	[in/out]	position in that layer, are updated after interpolation
	_X	[out]	offset(Δ[layer,x,y]), are updated interpolation
	-	[return]	It is poor contrast or not
edgePointEliminate	img	[in]	one layer image in octave
	(x,y)	[in]	point position in that layer
	-	[return]	If it has high edge response or not

Table 2.6: Parameter Description

2.3.4 Orientation assignment

The keypoint descriptor can be represented according to the assigned orientation to achieve invariance to image rotation. This stage is to assign consistent orientations to keypoints.

```
void calcFeatureOri(vector< Feature >& features);
{
    void calcOriHist(Feature& feature, vector< double >& hist);
    bool calcMagOri(Mat* img, int x, int y, double& mag, double& ori);
    double getMatValue(Mat* img, int x, int y);
    void smoothOriHist(vector< double >& hist );
    void addOriFeatures(vector<Feature>& features, Feature& feat, vector<
        double >& hist);
}
```

Function	Function Description
calcFeatureOri	the main function to control the pipeline of calculating orientation for each keypoint feature and all other functions are called by it
calcOriHist	to calculate orientation histogram
calcMagOri	to calculate magnitude and orientation of one keypoint
getMatValue	to get the value of the specified layer of octave on the position (x, y)
smoothOriHist	to smooth the orientation histogram
addOriFeatures	to set orientation to each keypoint feature and add new features to some keypoints if some orientations' density of them are over 80 percent of the dominant one.

Table 2.7: Function Description

Function	Parameter	Type	Parameter Description
calcFeatureOri	features	[in/out]	keypoint features, feature's orientation is updated after executing function
calcOriHist	features	[in]	keypoint features
	hist	[out]	orientation histogram
calMagOri	img	[in]	one specified layer image in octave
	(x,y)	[in]	point position in that layer
	mag	[out]	magnitude of gradient
	ori	[out]	orientation of gradient
	-	[return]	this calculation is successful or not
getMatValue	img	[in]	one specified layer image in octave
	(x,y)	[in]	point position in that layer
	-	[return]	value of the specified layer of octave on the position (x,y)
smoothOriHist	hist	[in/out]	orientation histogram after smooth, is updated after executing the function
addOriFeatures	features	[in/out]	keypoint features, feature orientation is updated after executing the function
	feat	[in]	specified feature to be checked
	hist	[in]	orientation histograms for new features

Table 2.8: Parameter Description

2.3.5 Keypoint descriptor representation

The local image gradients are measured at the selected layer of octave in the region around each keypoint and are transformed into representation that is invariant to significant levels of resampling and blurring.

```
void calcDescriptor(vector<Feature>& features);
{
    void calcDescHist(Feature& feature, vector< vector< vector<double> >
        >& hist);
    void interpHistEntry(vector< vector< vector<double> > >& hist, double
         xIdx, double yIdx, double resultIdx, double weiMag);
    void hist2Desc(vector< vector< vector<double> > >& hist, Feature&
        feature);
    void furtherProcess(Feature& feature);
}
```

Function	Function Description
calcDescriptor	main function to calculate the descriptor representation
calcDescHist	to calculate the descriptor histogram
interpHistEntry	to interpolate histogram entry in order to get a more accurate value
hist2Desc	to transform the descriptor histogram into descriptor
furtherProcess	to truncate and normalize keypoint features

Table 2.9: Function Description

Function	Parameter	Type	Parameter Description
calcDescriptor	features	[in/out]	keypoint features, feature's descriptor is updated after executing the function
calcDescHist	feature	[in]	keypoint feature vector
	hist	[out]	descriptor histogram
interpHistEntry	hist	[in/out]	descriptor histogram
	(xIdx, yIdx)	[in]	the position after rotating the image according to the feature orientation
	resultIdx	[in]	bin index corresponding to new orientation in rotated image
	weiMag	[in]	gradient magnitude after weighting
hist2Desc	hist	[in]	descriptor histogram
	feature	[out]	keypoint feature, feature's descriptor is updated after executing the function
furtherProcess	feature	[in/out]	keypoint is normalized and truncated by threshold

Table 2.10: Parameter Description

3 SIFT MATCH & KDTREE

3.1 SIFT MATCH

SiftMatcher is actually a front-end class or an abstract of KD-TREE. Programmers who want to use this library should use **SiftMatcher** rather than KdTree.

Relevant source files:

1. src/include/SiftMatcher.h
2. src/lib/SiftMatcher.cpp

3.1.1 LOAD TRAINING DATAS

```
void loadDir(const char *dirName);
{
    void loadFile(const char *fileName);
}
void loadFeatures(std::vector<Feature> & inputFeat);
```

Function	Function Description
loadDir	load image features from one directory, it will call function **loadFile**
loadFile	load features from an image, it stores results to *.sift* file to avoid repeated SIFT feature calculation for same image
loadFeatures	load image features from a set of features

Table 3.1: Function Description

Function	Parameter	Type	Parameter Description
loadDir	dirName	[in]	directory name
loadFile	fileName	[in]	file name
loadFeatures	features	[in]	a set of features

Table 3.2: Parameter Description

3.1.2 BUILD KD-TREE

```
void setup();
```

Function	Function Description
setup	should be called after you load all the training image into this class object. It will build a **KD-TREE** on existed template feature points

Table 3.3: Function Description

3.1.3 Match

```
std::pair<Feature *, Feature *> match(Feature & input);
{
    unsigned long match(vector<Feature> &inputFeats);
}
```

Function	Function Description
match(Feature &)	match feature, return two nearest features from the kd-tree
match(vector<Feature > &)	match a set of feature, is called by the above function, and return a tag.

Table 3.4: Function Description

Function	Parameter	Type	Parameter Description
match(Feature &)	input	[in]	input feature
	-	[return]	Nearest & second nearest feature from different objects
match(vector<Feature &>)	inputFeats	[in]	A set of features
	-	[return]	A tag if existed, be used to find a matched object.

Table 3.5: Parameter Description

```
bool isGoodMatch(std::pair<Feature *, Feature *> matchs, Feature &
    inputFeat) {
    ...
    ...
    return (bestVal / secBestVal < matchRatio);
}
```

Function	Function Description
isGoodMatch	judge if a match is good or not

Table 3.6: Function Description

Function	Parameter	Type	Parameter Description
isGoodMatch	matchs	[in]	the first and second nearest of input feature in kd-tree
	-	[return]	whether it is a good match or not

Table 3.7: Parameter Description

3.2 KD TREE

In this section, we give an function-level introduction of our implementation for **KD-TREE**. You don't need to read it if you only want to use the front-end functions.

Relevant source files:

1. src/include/kdTree.h
2. src/lib/kdTree.cpp

3.2.1 BUILD KD TREE

```
void buildTree(std::vector<Feature> & features);
```

Function	Function Description
buildTree	Build a kd-tree on the input features

Table 3.8: Function Description

Function	Parameter	Type	Parameter Description
buildTree	features	[in]	A set of template features

Table 3.9: Parameter Description

```
void split( KDNode * parent );
```

Function	Function Description
split	Recursive split the nodes, At every split process, it call **selectDimension** and **findMedian** to split the kd-tree node and split based on the median value from the dimension with larest variance

Table 3.10: Function Description

Function	Parameter	Type	Parameter Description
split	parent	[in/out]	split this specific node

Table 3.11: Parameter Description

```
int selectDimension( KDNode * node );
double findMedian( KDNode * node, int k );
```

Function	Function Description
selectDimension	Select the dimension with largest variance
findMedian	Find the median value at k-th dimension

Table 3.12: Function Description

Function	Parameter	Type	Parameter Description
selectDimension	node	[in]	a KDNode that contains several feature points
	-	[return]	the dimension with larest variance
findMedian	node	[in]	a KDNode that contains several feature points
	k	[in]	dimension to be calculated
	-	[return]	the median of this dimension

Table 3.13: Parameter Description

3.2.2 BBF SEARCH

After generating a balanced kd-tree, the remaining of this class is searching process.

```
std::pair<Feature *,Feature *> bbfNearest( Feature & input );
```

Function	Function Description
bbfNearest	This is basically a **dfs** search process with support of prioriry ququе, this search strategy is introduced by Dr. Lowe[3]. The basical idea is searching the closer branch firstly, and drop the very bad branches.

Table 3.14: Function Description

Function	**Parameter**	**Type**	**Parameter Description**
bbfNearest	input	[in]	input feature
	-	[return]	The nearest and second nearest features on the kd-tree

Table 3.15: Parameter Description

4 Other source codes

This section give a short description for other source codes, such as the demo codes.

4.1 DEMO

This section give detailed description for how the demo work. To see the expected output, please refer to another document **Installation-Usage.pdf**.

Relevant source files:

1. src/kd_demo.cpp
2. src/match2img.cpp
3. src/match2db.cpp
4. src/accuracyTest.cpp
5. src/camera.cpp

Because the process of demo codes are similar, here we only take **camera.cpp** as a example to help readers understand how our codes work.

4.1.1 CAMERA

This is the only demo that we use algorithm-level support from opencv. Our project is focus on implementation of SIFT and KD-TREE. So we use opencv to do face detection job in vedio camera.

4.1.1.1 Init camera

```
cap.set( CV_CAP_PROP_FRAME_WIDTH,CAP_WIDTH);
cap.set( CV_CAP_PROP_FRAME_HEIGHT,CAP_HEIGHT);
```

4.1.1.2 Build template kd-tree

```
matcher.loadDir( templateDir );
matcher.setMatchRatio(matchRatio);
matcher.setup();
```

As it is mentioned in section 3.1, it will build a kd-tree on the image files from template directory and set the match ratio.

4.1.1.3 Get signal from camera

```
while(true) {
    cap >> frame;
    Mat showFrame = frame;

    if(cnt --) {
    }
    else {
        /* Face Detection and recognition thread */

        cnt = FRAME_INTERVAL;
    }
    /* Draw Processing codes */

    imshow( "Capture", showFrame);
    if( waitKey(30)>=0 ) break;
}
```

This loop will periodically get frame image from camera. It will process face detection the recognition every **FRAME_INTERVAL** times.

4.1.1.4 Face Detection

```
void detectFace(Mat frame, vector<Rect> &faces) {
Mat frame_gray;
cvtColor(frame, frame_gray, CV_BGR2GRAY);
equalizeHist(frame_gray, frame_gray);

face_cascade.detectMultiScale( frame_gray, faces, 1.1, 2, 0|
    CV_HAAR_SCALE_IMAGE, Size(30, 30) );
}
```

This process is supported by opencv. It returns several **Rectangles** denoting faces on an input image.

4.1.1.5 Face Recognition

```
void reconition(Mat frame, vector<Rect> &faces, vector<string > &results)
     {
    int faceIdx;
    vector<Feature> feats;
    results.clear();

    SiftExtractor extractor;

```

```
    for(faceIdx = 0; faceIdx < faces.size(); faceIdx ++) {
        Mat face(frame, faces[faceIdx]);

        feats.clear();

        extractor.sift(&face, feats);

        unsigned long matchTag = matcher.match(feats);

        results.push_back(ImgFileName::descriptor(matchTag));
    }
}
```

This recongtion process apply **sift** on each rectangle denoting a face on an image. And apply **match** on the template kd-tree to get a tag. Which can be used to get matched object (Typically a name for face detection);

Please refer to **Installation-Usage.pdf** to see the expected result.

4.1.2 KD_DEMO

Please refer to **Installation-Usage.pdf**.

4.1.3 MATCH2IMG

Please refer to **Installation-Usage.pdf**.

4.1.4 MATCH2DB

Please refer to **Installation-Usage.pdf**.

4.1.5 ACCURACYTEST

Please refer to **Installation-Usage.pdf**.

4.2 FEATURE

Relevant source files:

1. src/include/feature.h
2. src/lib/feature.cpp

This class is used to store a feature, it also buffer datas used during the sift feature extraction process.

4.3 IMAGESET & IMGFILENAME

Relevant source files:

1. src/include/ImageSet.h
2. src/include/ImageFileName.h
3. src/include/imgSuffix.def
4. src/lib/ImageSet.cpp
5. src/lib/ImageFileName.cpp

These class are used to process image files. For example load all the image files from a directory, judge if it is a image file based on its suffix.

imgSuffix.def shows the file type that we support for processing sift:

```
IMG_SUFFIX("png")
IMG_SUFFIX("jpg")
IMG_SUFFIX("jpeg")
IMG_SUFFIX("pgm")
...
```

4.4 CONFIGURES

1. src/include/configure.h
2. src/include/configureFrontEnd.h
3. src/include/siftConfigure.def

These files give configurations for our implementation of SIFT, KD-TREE and front-end demo codes.

References

[1] F. S. Samaria, F. S. S. *t, A. Harter, and O. A. Site, "Parameterisation of a stochastic model for human face identification," 1994.

[2] Y. face database, "http://vision.ucsd.edu/content/yale-face-database."

[3] D. G. Lowe, "Distinctive image features from scale-invariant keypoints," *Int. J. Comput. Vision*, vol. 60, pp. 91–110, Nov. 2004.

[4] D. G. Lowe, "Object recognition from local scale-invariant features," in *Proceedings of the International Conference on Computer Vision-Volume 2 - Volume 2*, ICCV '99, (Washington, DC, USA), pp. 1150–, IEEE Computer Society, 1999.

[5] S. Se, D. Lowe, and J. Little, “Vision-based mobile robot localization and mapping using scale-invariant features,” in *In Proceedings of the IEEE International Conference on Robotics and Automation (ICRA*, pp. 2051–2058, 2001.