

# Lesson 14 Image Processing---Feature Matching

## 1.Brute-Force Matching

The feature descriptor is to describe the key point with a set of vectors after the key point is calculated. It includes not only the key point, but also the pixels around the key point that has made a contribution. It serves as the basis for target matching, and enables the key points to have more invariant characteristics, such as illumination changes, 3D viewpoint changes, etc.

Each feature descriptor in one set of features is matched with the nearest feature descriptor of the other set, then the obtained distances are sorted, and finally the feature with the shortest distance is selected as the matching point for the two.

Next, use brute force matching to match the features of the two images.

### 1.1Operation Steps

---

#### Note:


1) Before operation, please copy the routine “bf\_demo.py” and sample picture “test.jpg” in “4.OpenCV->Lesson 14 Image Processing --- Feature Matching->Routine Code” to the shared folder.

2) For how to configure the shared folder, please refer to the file in “2. Linux Basic Lesson->Lesson 3 Linux Installation and Source Replacement”.

3) The input command should be case sensitive and the keywords can be complemented by “Tab” key.

---

1) Open virtual machine and start the system. Click “”, and then

“” or press “**Ctrl+Alt+T**” to open command line terminal.

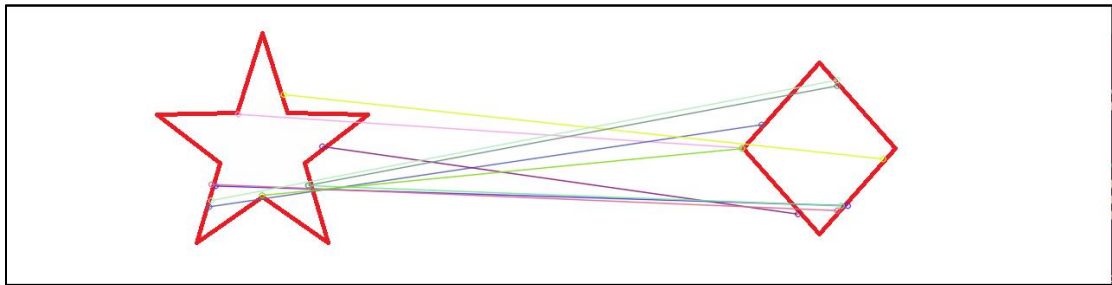
2) Input command “**cd /mnt/hgfs/share/**” and press Enter to enter the shared folder.

```
hiwonder@ubuntu:~$ cd /mnt/hgfs/Share/
```

3) Input command “**python3 bf\_demo.py**” and press Enter to run the routine.

```
ubuntu@ubuntu-virtual-machine:/mnt/hgfs/share$ python3 bf_demo.py
```

## 1.2 Program Outcome



Take out one part of the original image, and then match features between these two images.

## 1.3 Code Analysis

The routine “**bf\_demo.py**” can be found in “**4.OpenCV->Lesson 14 Image Processing --- Feature Matching->Routine Code**”.

```
1  import cv2
2
3  img1 = cv2.imread('test.jpg')
4  img2=cv2.imread('test1.jpg')
5
6  # initialize ORB feature detector
7  orb = cv2.ORB_create()
8
9  # detect feature and descriptor
10 kp1, des1 = orb.detectAndCompute(img1,None)
11 kp2, des2 = orb.detectAndCompute(img2,None)
12
13 # Create a brute force (BF) matcher
14 bf = cv2.BFMatcher_create(cv2.NORM_HAMMING, crossCheck=True)
15
16 # match descriptor
17 matches = bf.match(des1,des2)
18
19 # draw ten matched descriptors
20 img3 = cv2.drawMatches(img1, kp1, img2, kp2, matches[:10], None, flags=2)
21
22 cv2.imshow("show",img3)
23 cv2.waitKey()
24 cv2.destroyAllWindows()
```

**Import Module:** Import cv2 module

**Read Picture:** Call imread function to read picture and the parameter stands for the name of the picture.

**Initialized detector:** use ORB\_create constructor for initialization.

```
orb = cv2.ORB_create()
```

**Detect feature and descriptor:** detectAndCompute function will be used to detect. And the specific format and parameter are as follow.

detectAndCompute(src,mask)

- 1) The first parameter “**src**” is the image to process the threshold.
- 2) The second parameter “**mask**” is the image in which the object is black and the rest is white. If the mask is not required, set the parameter as None.

```
kp1, des1 = orb.detectAndCompute(img1,None)
kp2, des2 = orb.detectAndCompute(img2,None)
```

**Create BFMatcher object:** BFMatcher belongs to features2d module and inherits from DescriptorMatcher. The function format is as follow.

**static Ptr<BFMatcher> create( int normType , bool crossCheck )**

1) The first parameter “**normType**” can be set as NORM\_L1, NORM\_L2, NORM\_HAMMING or NORM\_HAMMING2. The HOG descriptors of SIFT and SURF correspond to the Euclidean distances L1 and L2; the BRIEF descriptors of ORB and BRISK correspond to the Hamming distance HAMMING; HAMMING2 corresponds to the ORB algorithm when WTA\_K = 3 or 4.

2) **Euclidean distance:** it is defined as the distance between two points in n-dimensional space.

$$L1 = \sum_I |\text{src1}(I) - \text{src2}|$$
$$L2 = \sqrt{\sum_I (\text{src1}(I) - \text{src2}(I))^2}$$

Hamming distance: It is computer's XOR operation suitable for binary string descriptors, such as BRIEF descriptors. Its definition is as follow.

$$\text{Hamming}(a, b) = \sum_{i=0}^{n-1} (a_i \oplus b_i)$$

3) The second parameter “**crossCheck**” is set as “**FALSE**” by default. If set as TRUE, the matching is valid only when the features in the two groups match with each other. In other words, only when the x point descriptor in group A and the y point in group B are the best matching points for each other, the matching is effective.

```
bf = cv2.BFMatcher_create(cv2.NORM_HAMMING, crossCheck=True)
```

**Match descriptor:** **detectAndCompute** function will be adopted, and its specific format and parameters are as follow.

**match(queryDescriptors,trainDescriptors)**

- 1) The first parameter “**queryDescriptors**” is the image feature vector to be matched.
- 2) The second parameter “**trainDescriptors**” is the image feature vector that needs to be matched.

```
matches = bf.match(des1,des2)
```

**Draw matches:** **drawMatches** function will be used. The specific format and parameters are as follow.

**drawMatches(src1,kp1,src2,kp2,match,matchesMask,flags)**

- 1) The first parameter “**src1**” is the matching image 1.
- 2) The second parameter “**kp1**” is the feature of image 1.
- 3) The third parameter “**src2**” is the matching image 2.
- 4) The fourth parameter “**kp2**” is the feature of the image 2.
- 5) The fifth parameter “**match**” is the set of matching points to be drawn.
- 6) The sixth parameter “**matchesMask**” determines which images to draw. If it is set as None, all the images will be drawn.
- 7) The seventh parameter “**flags**” represents the drawing flag. 0 indicates that all the features will be drawn, and 2 indicates that only the matched feature will be drawn. 4 stands for the drawing styles.

```
img3 = cv2.drawMatches(img1, kp1, img2, kp2, matches[:10], None, flags=2)
```

**Display image:** Call **imshow** function to display image and the parameter in bracket refers to the title of the window and displayed image.

**Close window:** waitKey function will wait until the keyboard is pressed, and then execute destroyAllWindows function to close the window.

```
cv2.imshow("show",img3)
cv2.waitKey()
cv2.destroyAllWindows()
```

## 2.Nearest Neighbor Matching

FLANN (Fast Library for Approximate Nearest Neighbors) is a FLANN is a open-source library for performing fast approximate nearest neighbor searches in high dimensional spaces.

The nearest neighbor matching operator FlannBasedMatcher based on the FLANN library is much more efficient than BFMatcher in the field of large feature datasets or some real-time processing.

Next, adopt nearest neighbor matching to match the features of the two images.

### 2.1Operation Steps

---



**Note:**

1) Before operation, please copy the routine “flann\_demo.py” and sample pictures “test.jpg” and “test1.jpg” in “4.OpenCV->Lesson 14 Image Processing --- Feature Matching->Routine Code” to the shared folder.

2) For how to configure the shared folder, please refer to the file in “2. Linux Basic Lesson->Lesson 3 Linux Installation and Source Replacement”.

3) The input command should be case sensitive and the keywords can be complemented by “Tab” key.

---

1) Open virtual machine and start the system. Click “”, and then “” or press “**Ctrl+Alt+T**” to open command line terminal.

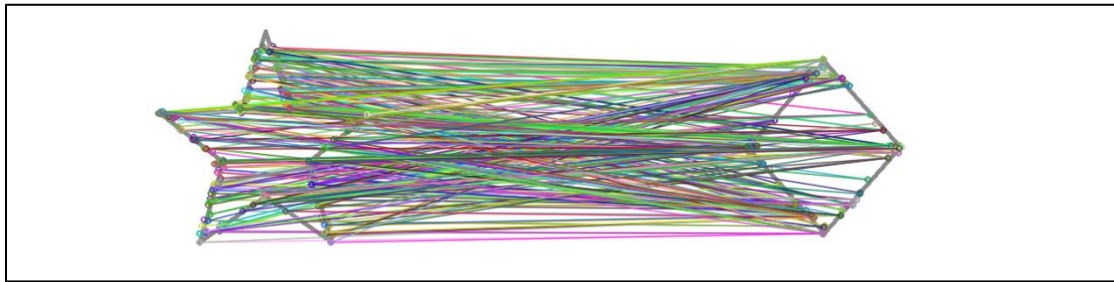
2) Input command “**cd /mnt/hgfs/share/**” and press Enter to enter the shared folder.

```
hiwonder@ubuntu:~$ cd /mnt/hgfs/Share/
```

3) Input command “**python3 flann\_demo.py**” and press Enter to run the routine.

```
ubuntu@ubuntu-virtual-machine:/mnt/hgfs/share$ python3 flann_demo.py
```

## 2.2 Program Outcome



## 2.3 Code Analysis

The routine “**flann\_demo.py**” can be found in “**4. OpenCV Computer Vision Lesson->Lesson 14 Image Processing---Feature Matching->Routine Code**”.



```

1  import numpy as np
2  import cv2 as cv
3
4  img1 = cv.imread('test.jpg',cv.IMREAD_GRAYSCALE)    # index image
5  img2 = cv.imread('test1.jpg',cv.IMREAD_GRAYSCALE)    # training image
6
7  # initialize ORB descriptor
8  orb = cv.ORB_create()
9
10 # search keypoint and descriptor based on ORB
11 kp1, des1 = orb.detectAndCompute(img1,None)
12 kp2, des2 = orb.detectAndCompute(img2,None)
13
14 # parameters of FLANN
15 FLANN_INDEX_LSH = 6
16 index_params= dict(algorithm = FLANN_INDEX_LSH,
17                    table_number = 6, # 12
18                    key_size = 12,    # 20
19                    multi_probe_level = 1) #2
20 search_params = dict(checks=50)    # transfer a empty dictionary
21 flann = cv.FlannBasedMatcher(index_params,search_params)
22
23 matches = flann.knnMatch(des1,des2,k=2)
24
25 img3 = cv.drawMatchesKnn(img1,kp1,img2,kp2,matches,None)
26 cv.imshow("show",img3)
27 cv.waitKey()
28 cv.destroyAllWindows()

```

**Set FLANN parameter:** FlannBasedMatcher function will be adopted. And its format and parameters are as follow.

FlannBasedMatcher (IndexParams,SearchParams) (The two parameters in the bracket refers to the type of the dictionary)

- 1) The first parameter "**IndexParams**" the algorithm designated to use.
- 2) The second parameter "**SearchParams**" is the number of times the tree in the specified index should be traversed recursively. Higher values provide better accuracy, but also take more time.

```

16 index_params= dict(algorithm = FLANN_INDEX_LSH,
17                    table_number = 6, # 12
18                    key_size = 12,    # 20
19                    multi_probe_level = 1) #2
20 search_params = dict(checks=50)    # transfer a empty dictionary
21 flann = cv.FlannBasedMatcher(index_params,search_params)

```

Nearest neighbor matching: knnMathch function will be used, And the function format and parameters are as follow.



`knnMatch(queryDescriptors,trainDescriptors,k)`

The first two parameters are consistent with match, and "k" is the best number of matches to be returned.)

```
matches = flann.knnMatch(des1,des2,k=2)
```