SIFT

(Scale-Invariant Feature Transform)

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

Major advantages of SIFT are :

- **Locality:** features are local, so robust to occlusion and clutter (no prior segmentation)
- **Distinctiveness:** individual features can be matched to a large database of objects
- **Quantity:** many features can be generated for even small objects
- **Efficiency:** close to real-time performance
- **Extensibility:** can easily be extended to a wide range of different feature types, with each adding robustness

---

The Major Steps :

- **Scale-space peak selection:** Potential location for finding features.
- **Keypoint Localization:** Accurately locating the feature keypoints.
- **Orientation Assignment:** Assigning orientation to keypoints.
- **Keypoint descriptor:** Describing the keypoints as a high dimensional vector.

1- Scale-space peak Selection

This process is done for different octaves of the image in Gaussian Pyramid.

(4 different Octaves => each octave has 5 different Gaussian blur)

DOG (Difference of Gaussian kernel)

These DoG images are great for finding out interesting keypoints in the image. The difference of Gaussian is obtained as the difference of Gaussian blurring of an image with two different σ.

One pixel in an image is compared with its 8 neighbours as well as 9 pixels in the next scale and 9 pixels in previous scales. This way, a total of 26 checks are made.

2- Keypoint Localization

Keypoints generated in the previous step produce a lot of keypoints. So they made (Eliminate edges and low contrast regions).

- Reject points with bad contrast(Threshold < 0.03)

- Reject edges. (Harris Corner Detector for removing edge features)

3- Orientation Assignment

A neighbourhood is taken around the keypoint location depending on the scale, and the gradient magnitude and direction is calculated in that region.

4- Keypoint descriptor

At this point, each keypoint has a location, scale, orientation. Next is to compute a descriptor for the local image region about each keypoint**.**

To do this, a 16x16 window around the keypoint is taken. It is divided into 16 sub-blocks of 4x4 size.

For each sub-block, 8 bin orientation histogram is created.
So 4 X 4 descriptors over 16 X 16 sample array were used in practice. 4 X 4 X 8 directions give 128 bin values. It is represented as a feature vector to form keypoint descriptor.

CODING

sift =cv.xfeatures2d.SIFT_create (nfeatures =0, nOctaveLayers=3, contrastThreshold=0.04 , edgeThreshold=10 , sigma =1.6)

nfeatures The number of best features to retain.

nOctaveLayers The number of layers in each octave.

contrastThreshold The contrast threshold used to filter out weak features

edgeThreshold The threshold used to filter out edge-like features

sigma The sigma of the Gaussian applied to the input image

---

keypoints, descriptors=cv.Feature2D.detectAndCompute(image, mask, descriptors, useProvidedKeypoints)

Here kp will be a list of keypoints and descriptors is a numpy array of shape $Number\_of\_Keypoints \times 128$

OpenCV provides **cv.drawKeyPoints()** function which draws the small circles on the locations of keypoints.

```
outImage=cv.drawKeypoints(image,kp,outImage,color,flags)
```

image Source image.

keypoints Keypoints from the source image.

outImage Output image.

color Color of keypoints.

flags Flags setting drawing features.

flags = `DrawMatchesFlags::DEFAULT`)

If you pass a flag, **cv.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS** to it, it will draw a circle with size of keypoint and it will even show its orientation. See below

---

FLANN based Matcher

FLANN is a library for performing fast approximate nearest neighbor searches in high dimensional spaces. It contains a collection of algorithms we found to work best for nearest neighbor search and a system for automatically choosing the best algorithm and optimum parameters depending on the dataset. It works faster than `BFMatcher` for large datasets.

For `FlannBasedMatcher`, it accepts two sets of options which specifies the algorithm to be used, its related parameters etc. First one is `Index`. For various algorithms,

indexParams(algorithm,tree)

flann_algorithm { FLANN_INDEX_LINEAR = 0,
FLANN_INDEX_KDTREE = 1,
FLANN_INDEX_KMEANS = 2,
FLANN_INDEX_COMPOSITE = 3,
FLANN_INDEX_KDTREE_SINGLE = 3,
FLANN_INDEX_SAVED = 254,
FLANN_INDEX_AUTOTUNED = 255 }

flann_centers_init_t { FLANN_CENTERS_RANDOM = 0,
FLANN_CENTERS_GONZALES = 1,
FLANN_CENTERS_KMEANSPP = 2 }

SearchParams (int checks=32, float eps=0, bool sorted=true)

Checks : specifies the maximum leafs to visit when searching for neighbours. A higher value for this parameter would give better search precision, but also take more time.

eps :Search for eps-approximate neighbors (only used by KDTreeSingleIndex and KDTreeCuda3dIndex).

Sorted : Used only by radius search, specifies if the neighbors returned should be sorted by distance.

Flann=cv.FlannBasedMatcher(indexParams, searchParams)

matches = flann.knnMatch(descriptors1, descriptors2, k)

```
cv.drawMatchesKnn(img1, kp1, img2, kp2, good,
                  outImg=img_match, matchColor=None, singlePointColor=(255, 255,255),
flags=2)
```

HOMOGRAPHY

As for the resulting matrix. It is called a homography matrix, or [H]matrix and it represents the transformation of one point in an image plane to the same point in another image plane.

It returns a mask which specifies the inlier and outlier points.

cv.**FindHomography**(srcPoints, dstPoints, method=0, ransacReprojThreshold=3.0)

- **srcPoints** – Coordinates of the points in the original plane.(kp1)
- **dstPoints** – Coordinates of the points in the target plane (kp2)

- method

0 - a regular method using all the points

•CV_RANSAC - RANSAC-based robust method
•CV_LMEDS - Least-Median robust method

ransacReprojThreshold –

Maximum allowed reprojection error to treat a point pair as an inlier (used in the RANSAC method only).

The method `RANSAC` can handle practically any ratio of outliers but it needs a threshold to distinguish inliers from outliers.

The method `LmeDS` does not need any threshold but it works correctly only when there are more than 50% of inliers.

Finally, if there are no outliers and the noise is rather small, use the default method (`method=0`).

The out put of (cv2.findHomography) is :

- **H is the matrix of homography 3x3**
- **mask**– Optional output mask set by a robust method (`CV_RANSAC` or `CV_LMEDS`). Note that the input mask values are ignored

Resources ..

1- https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_feature2d/py_sift_intro/py_sift_intro.html#sift-intro

2- https://docs.opencv.org/3.4/d5/d3c/classcv_1_1xfeatures2d_1_1SIFT.html

3-https://docs.opencv.org/trunk/d4/d5d/group__features2d__draw.html#ga5d2bafe8c1c45289bc3403a40fb88920

4- https://docs.opencv.org/trunk/da/df5/tutorial_py_sift_intro.html

5- https://docs.opencv.org/3.4/d0/d13/classcv_1_1Feature2D.html

6- https://docs.opencv.org/3.4/dc/de2/classcv_1_1FlannBasedMatcher.html

7- https://docs.opencv.org/3.2.0/d5/d03/structcv_1_1flann_1_1SearchParams.html

8- https://docs.opencv.org/3.2.0/db/df4/structcv_1_1flann_1_1IndexParams.html#a85b622bb758a971ab7881d66b5b6123c

9- http://amroamroamro.github.io/mexopencv/opencv_contrib/SURF_descriptor.html#3

10- https://docs.opencv.org/2.4/modules/flann/doc/flann_fast_approximate_nearest_neighbor_search.html#

11- https://docs.opencv.org/3.4/db/d39/classcv_1_1DescriptorMatcher.html

12 - http://amroamroamro.github.io/mexopencv/opencv/feature_homography_demo.html

13- https://docs.opencv.org/2.4/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html?highlight=findhomography

14- https://www.cs.ubc.ca/research/flann/uploads/FLANN/flann_manual-1.8.4.pdf