Submission for Internship Assignment Evaluation

Feature Detection and Matching

Tanuj Vishnoi,
Former Intern, Nvidia Labs, New Delhi
Thapar Institute of Engineering and Technology

1. Problem statement

The problem was to find the correct set of features and match the cropped image with the full ones, in addition to finding the box-coordinates of the cropped image in the query image (full-sized image).

Fig 1: From cropped image (training) and Query image, the aim was to find box coordinates in query image.

2. Methodology Adopted

Object detection is the ability to recognize particular objects in images and being able to determine the location of those objects within the images. Preferentially since, we have a single image, feature matching is the best suited approach for the problem. We try to find features in both the training and query image and try to map their features and get the desired coordinates.

Major requirements of the feature matching algorithm:

1. Scale Invariance
2. Rotational Invariance
3. Illumination Invariance
4. Noise Invariance

Taking all the points in consideration, **Oriented FAST and Rotated BRIEF (ORB)** seems to be a handy and most suitable algorithm taking in consideration the requirements of feature matcher. ORB is basically a fusion of FAST keypoint detector and BRIEF descriptor with many

modifications to enhance the performance. First it use FAST to find keypoints, then apply Harris corner measure to find top N points among them. It also use pyramid to produce multiscale-features. I will be using opencv-python for the implementation of the assignment and algorithm.

Steps for ORB

1. Locating Keypoints: The first step in the ORB algorithm is to locate all the keypoints in the training image. After the keypoints have been located, ORB creates their corresponding binary feature vectors and groups them together in the ORB descriptor. For this we use the code `ORB_create()`. Although there are other parameters in ORB, I have used a few user defined and all rest are default. The values of each parameter have majorly been obtained by brute force approach.

```
orb = cv2.ORB_create(nfeatures=1000, scaleFactor=1.5, edgeThreshold = 31, WTA_K = 4, patchSize = 40)
```

A. **nfeatures** (int) : It determines the maximum number of features (keypoints) to locate.
B. **scaleFactor** (float) : It is the pyramid decimation ratio,which must be greater than 1. ORB uses an image pyramid to find features. For eg. a scaleFactor = 2 means the classical pyramid, where each next level has 4x less pixels than the previous. A big scale factor will diminish the number of features found.
C. **edgeThreshold** (int) : It determines the size of the border where features are not detected. Since the keypoints have a specific pixel size, the edges of images must be excluded from the search. The size of the edgeThreshold should be equal to or greater than the patchSize parameter.
D. **WTA_K** (int) : It is the number of random pixels used to produce each element of the oriented BRIEF descriptor. The possible values are 2, 3, and 4, with 2 being the default value. For example, a value of 4 means four random pixels are chosen at a time to compare their brightness. The index of the brightest pixel is returned. Since there are 3 pixels, the returned index will be either 0, 1, or 2. The max value that can be assigned is four.
E. **patchSize** (int) : It's the size of the patch used by the oriented BRIEF descriptor. Of course, on smaller pyramid layers the perceived image area covered by a feature will be larger.

I have then used the cv2.detectAndCompute (image) method to locate the keypoints in the given training image and to compute their corresponding ORB descriptor.

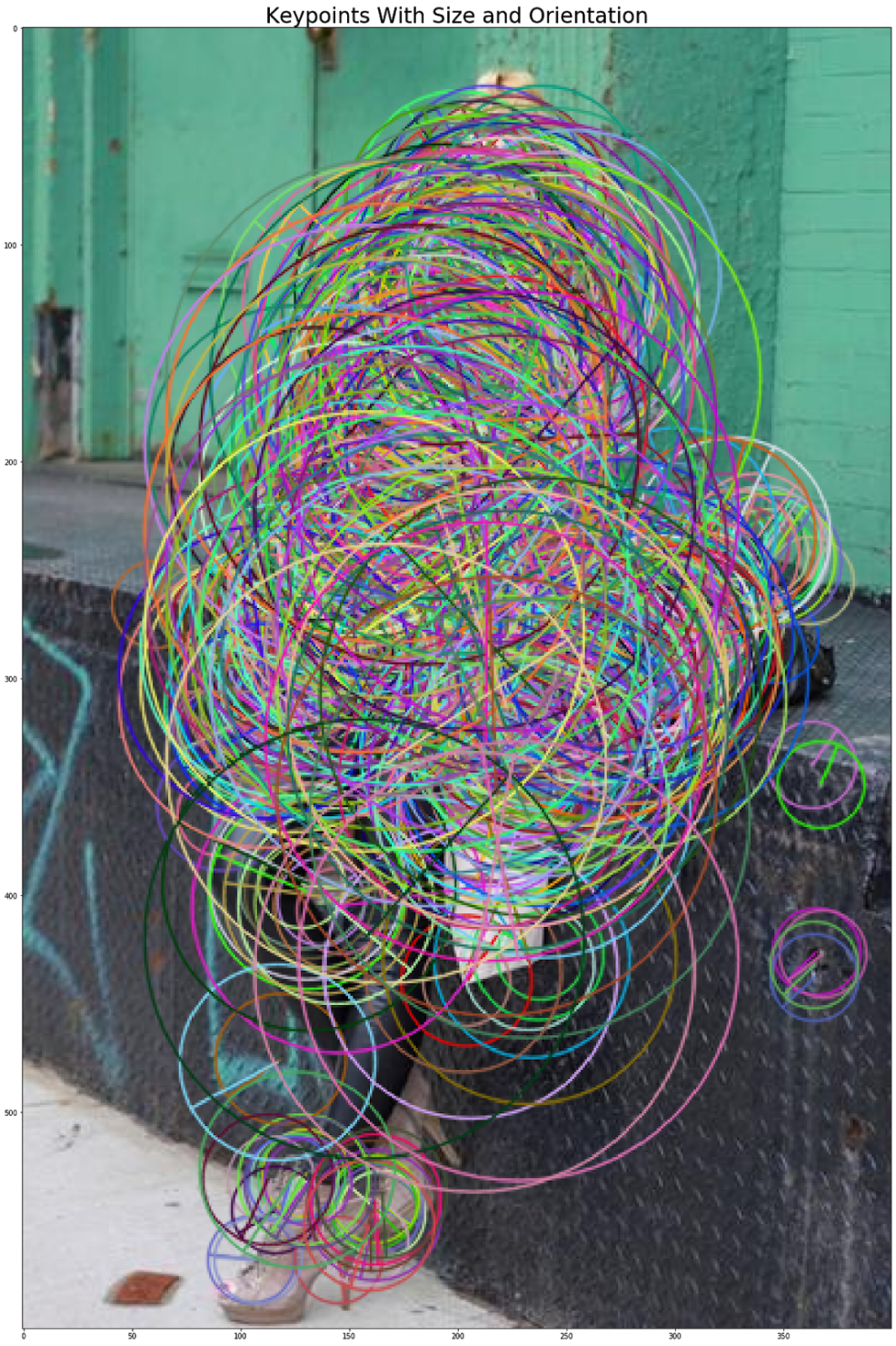

Fig: Query image with Keypoints

As visible above, every keypoint has a center, a size, and an angle. The center determines the location of each keypoint in the image; the size of each keypoint is determined by the patch size

used by BRIEF to create its feature vector; and the angle tells us the orientation of the keypoint as determined by rBRIEF.

Once the keypoints for the training image have been found and their corresponding ORB descriptor has been calculated, the same thing can be done for the query image.

2. Feature Matching: Once the ORB descriptors for both the training and query images are obtained, the final step is to perform keypoint matching between the two images using their corresponding ORB descriptors. This matching is usually performed by a matching function. One of the most commonly used matching functions is called Brute-Force.
   In the code below I have used OpenCV's BFMatcher class to compare the keypoints in the training and query images.. The parameters of the Brute-Force matcher are setup using the cv2.BFMatcher()function.

```
cv2.BFMatcher(normType = cv2.NORM_L2,
              crossCheck = false)
```

Parameters:

A. **normType**: It specifies the metric used to determine the quality of the match. By default, normType = cv2.NORM_L2, which measures the distance between two descriptors. However, for binary descriptors like the ones created by ORB, the Hamming metric is more suitable. The Hamming metric determines the distance by counting the number of dissimilar bits between the binary descriptors. When the ORB descriptor is created using WTA_K = 4, four random pixels are chosen and compared in brightness. The index of the brightest pixel is returned as either 0 or 1. Such output only occupies 2 bit, and therefore the cv2.NORM_HAMMING metric should be used. The index of the brightest pixel is returned as either 0, 1, 2, or 3. Such output will occupy 2 bits, and therefore a special variant of the Hamming distance, known as the cv2.NORM_HAMMING2 (the 2 stands for 2 bits), should be used instead. Then, for any metric chosen, when comparing the keypoints in the training and query images, the pair with the smaller metric (distance between them) is considered the best match.
B. **crossCheck** (bool): Cross-checking is very useful for eliminating false matches. Cross-checking works by performing the matching procedure two times. The first time the keypoints in the training image are compared to those in the query image; the second time, however, the keypoints in the query image are compared to those in the training image (i.e. the comparison is done backwards). When cross-checking is enabled a match is considered valid only if keypoint A* in the training image is the best match of keypoint *B in the query image and vice-versa (that is, if keypoint B* in the query image is the best match of point *A in the training image).
   Once the parameters of the BFMatcher have been set, I have used `cv2.match(descriptors_train, descriptors_query)` method to find the matching keypoints between the training and query images using their ORB descriptors.

Although, orb is scale and rotation invariant, in finding keypoints, still there is a chance of negative matching, if the cropped image is 3d oriented for which a function from calib3d module, ie `cv2.findHomography()` is used. If we pass the set of points from both the cropped and full images, it will find the perspective transformation of that object. Then we can use `cv2.perspectiveTransform()` to find the cropped image. However, according to opencv documentation, there can be some possible errors while matching which may affect the result. To solve this problem, algorithm uses RANSAC or LEAST_MEDIAN (which can be decided by the flags).
Another brute force method applied is eliminating through area of cropped box. If the difference of area between cropped box from query image and training image is more than 500, they are eliminated. It is a simple brute force approach based in observation. Also if the matches is less than 15 the match is discarded.
In this many outliers have been eliminated.

3. Json-building with coordinates: The task was to take each full image and then iterate it through each cropped image to get a match. Therefore, first the keypoints of both the cropped and full images is found and stored using `def get_features_all_crops()` and `def get_features_all_fulls()` functions and stored as two lists. Then a dictionary of full_data is made which stores all the details in required format.

The final json file is final_data.json

Fig: Feature matched query image with training image

3. Comparison with SIFT and SURF

ORB algorithm was brought up by Ethan Rublee, Vincent Rabaud, Kurt Konolige and Gary R. Bradski in their paper ORB: An efficient alternative to SIFT or SURF in 2011. As the title says, it is a good alternative to SIFT and SURF in computation cost, matching performance and mainly the patents. Yes, SIFT and SURF are patented and you are supposed to pay them for its use. But ORB is not !!! Thus I have implemented the ORB algorithm for feature matching.

4. Improvements Required

1. As visible, there is a small sample with some coordinates have got negative values, which is majorly due to perspective transform. I have tried working on the issue using cv2.wrapperspective opencv method, but it eliminated many positive results. If we remove all negative coordinates, which have some false samples also due to 3d rotation, the json would be final2.json.
2. The perspective transformation error can be removed by manually separating the 3d oriented crops and scheduling the process for them individually.
3. The parameters of orb can further be improvised using more testing.