matching.py

1. knnMatch (queryDescriptors, trainDescriptors, matches, k, mask, bool)

Notes:

**queryDescriptors:**

What do the attributes queryIdx and trainIdx represent? How can we use these attributes to keep track of our matched points?

1. cv2.polylines is not drawing properly.

cv2.polylines (img, pts, isClosed, color, thickness, lineType, shift)

Notes:

**pts:** Array of polygonal curves.

**isClosed (flag):** If true, then the drawn entity will be a closed loop.

**lineType:** Type of line segments.

**shift:** Number of fractional bits vertex coordinates (?)

1. Incorrect matches?
2. Handling of matched points are wrong?

Get the reference points from earlier frames using train index. Get the matched point from the current frame using query index. Draw polylines between the train and query index.

Do I need to do some preprocessing on the points to ensure that data formats are compatible?

1. Source codes tend to separate the feature matching method from the feature tracking.
2. Based on pyslam by LuigiFreda,

**Notes:**

Only one descriptor computation is done. Consider computing 2 different sets of descriptors?

Store each frame descriptors into a list.

Frame 1: 10 key points, 10 descriptors with 32 values each.

Frame 2: Another 10 key points, another 10 descriptors with 32 values each.

Will only do matching for second frame onwards. Earlier frame will be the reference frame (train descriptors) while later frame will be the query frame (query descriptors from this frame).

The way that I am comparing the descriptors between the frames is by comparing the entire list of numbers in the current frame with that of the previous frame. Perhaps it would be better to compare individual key points?

How to compare and filter the indices between the 2 lists? Access each element in the list or can you compare the entire lists?

Method 1: Comparing entire list of descriptor values between the 2 frames

Duplicate matches. Create a list storing the index of the matching points, scan through and remove any duplicate indices?

2 index lists, should I scan through both and only keep the indices that match between the 2 lists? Removing the duplicated indices between the train and query sets would make sense since you are getting the values from the same key points list.

\*\*\* Getting the common indices between sets 1 and 2 seem to produce very bad results…

Method 2: Comparing individual descriptor values between the frames

When I draw the points provided by keypoints[m.queryIdx] and keypoints[m.trainIdx], they seem to be giving the same set of points in a fixed pattern. In other words, the query and train points are behaving independently from the detected keypoints.

* Wrong match or no match at all?

\*\*\* Am I failing at the matching or indexing part?

matching\_test2.py

Try a more basic approach (possibly inefficient).

1. Separate keypoints and descriptors lists into 2: One for previous frame and one for current frame.
2. After doing the comparison, we clear the lists to make space for subsequent frames.

THIS TECHNIQUE HAS PRODUCED MUCH BETTER RESULTS!!!

**Feature detection and tracking:**

**Implementation notes:**

trainIdx: Reference keypoint the matcher uses to train.

queryIdx: Query keypoint the matcher will compare to train keypoints.

First method:

Keypoints and descriptors each have a single list to store their information, which will be accessed again later for feature matching. This method produced very bad matches.

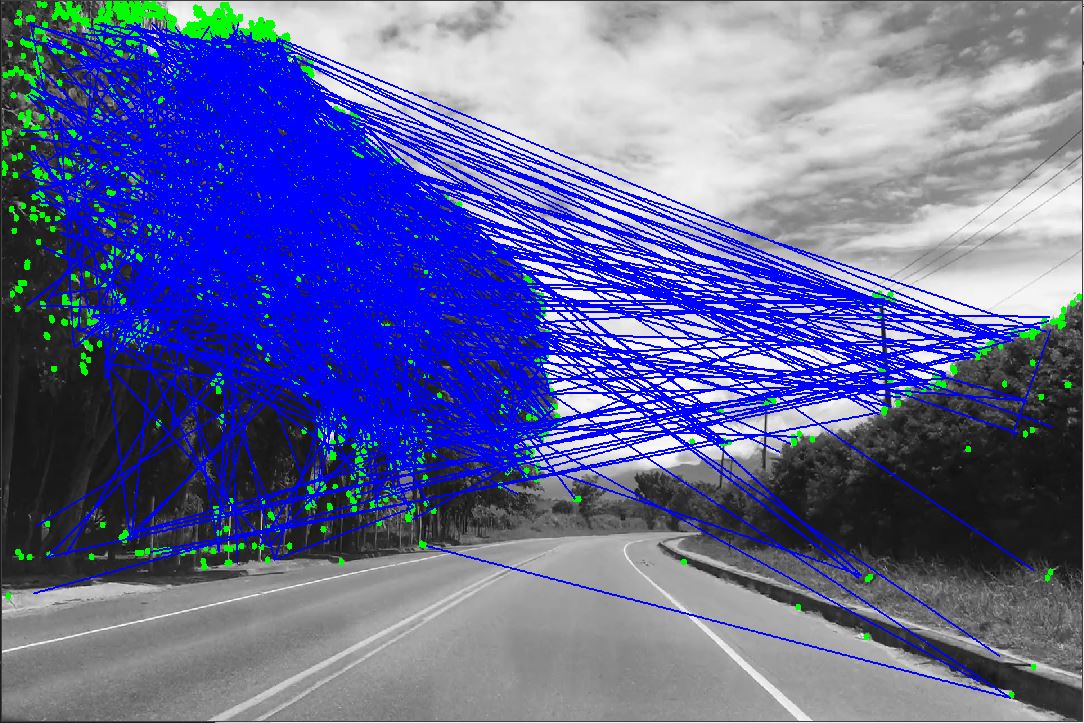


Fig 1.1 Bad matches in second frame (similar to other frames)

descriptors[counter] and descriptors[counter-1] are taken from a single descriptor list. Hard to tell if the matcher is even comparing the correct descriptor values since matcher seems to keep returning similar values for most of the frames, so this gives me 2 possible issues.

Moreover, all I did was to append new keypoints to the original list, without removing the keypoints from the older frames. Since the train and query indices returned by the matcher seems to repeat, accessing the keypoints list with these indices (keypoints[trainIdx] and keypoints[queryIdx]) mean that I am just accessing the same few values from the earlier frames, disregarding those from the later frames. This explains why blue lines are being drawn even on areas that have no keypoints being detected in the current frame.

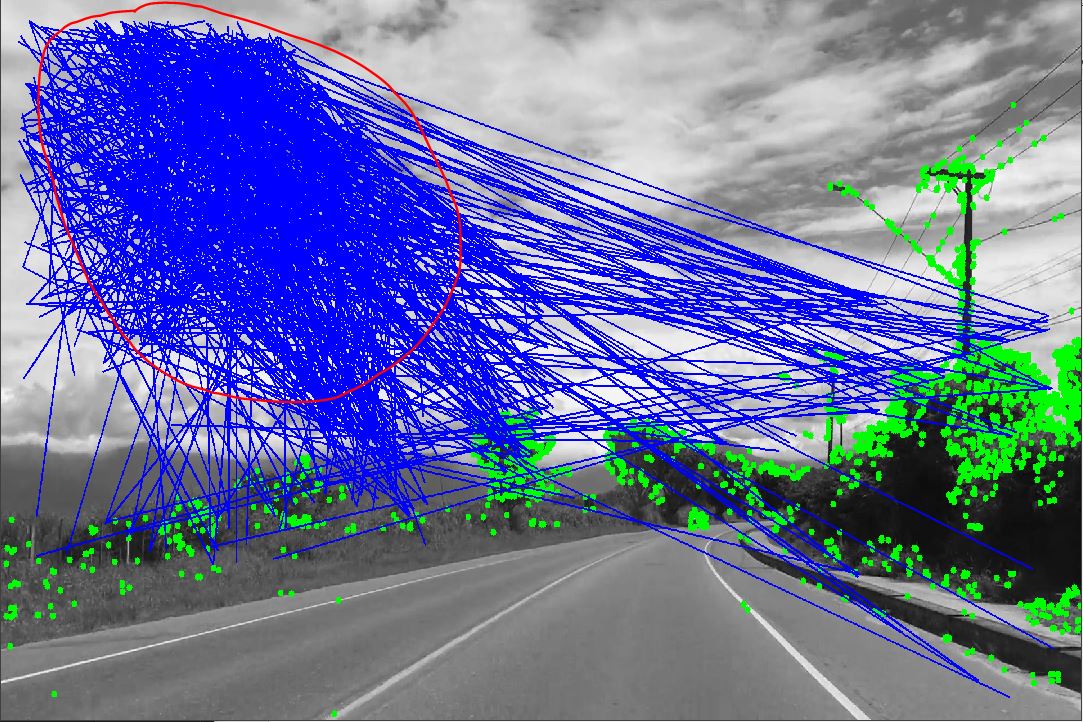


Fig 1.2 Bad matches in empty area

After deliberation, it seems like a good idea to discard values (descriptor and keypoints) from old frames and only keep those in the recent frames. This makes it easier to compare the immediate frames, since I won’t run the risk of comparing keypoint values from ancestor frames (more than 2 frames older). Furthermore, I will be more convinced that the matcher is comparing the correct descriptor values.

Troubleshoot error by adding more lists and variables to better keep track of the relevant information. Also, clear the old frame information when updating new frame information since they are redundant.

Second method:

The second method is a more practical method. Instead of storing keypoints and descriptors for **ALL** frames inside their respective lists, I will only focus on the current and previous (reference) frames. I used 2 lists to store keypoint information and 2 variables to store descriptor information.

ref\_kps will store the keypoint information for the reference frame while cur\_kps will store those for the current frame.

ref\_des will store descriptor information for the reference frame while cur\_des will store those for the current frame.

The lists and values will be reset after every matching frame so that redundant information can be removed.

This will make it easier to distinguish the between the reference frame information and the current frame information. Evidently, this method has produced much better results compared to the first method.



Fig 2.1 Better matches in the second frame



Fig 2.2 No false matches in empty areas

The disadvantage of this second method is that you can only compare once every 2 frames. Would that significantly affect the quality of the SLAM, if we cannot compare every frame?

Closing statement

Now that we can get decent matches between frames, how do we make use of those points to provide us with useful information? Can we calculate our pose using these matched features? What information can we derive from the matched points?

Bad matches inevitably exist, but we filtered them using Lowe’s ratio test (simple method). Is it possible to further filter the bad matches from the good ones? There is still a lot of room for improvements for the feature tracking and matching.

1. Matches seem a bit clustered in some areas. Increase matches to cover more of the image area (can be done by changing the ratio in ratio test).
2. Bad matches still exist, possible to filter more?



Fig 3.1 Ratio = 0.5

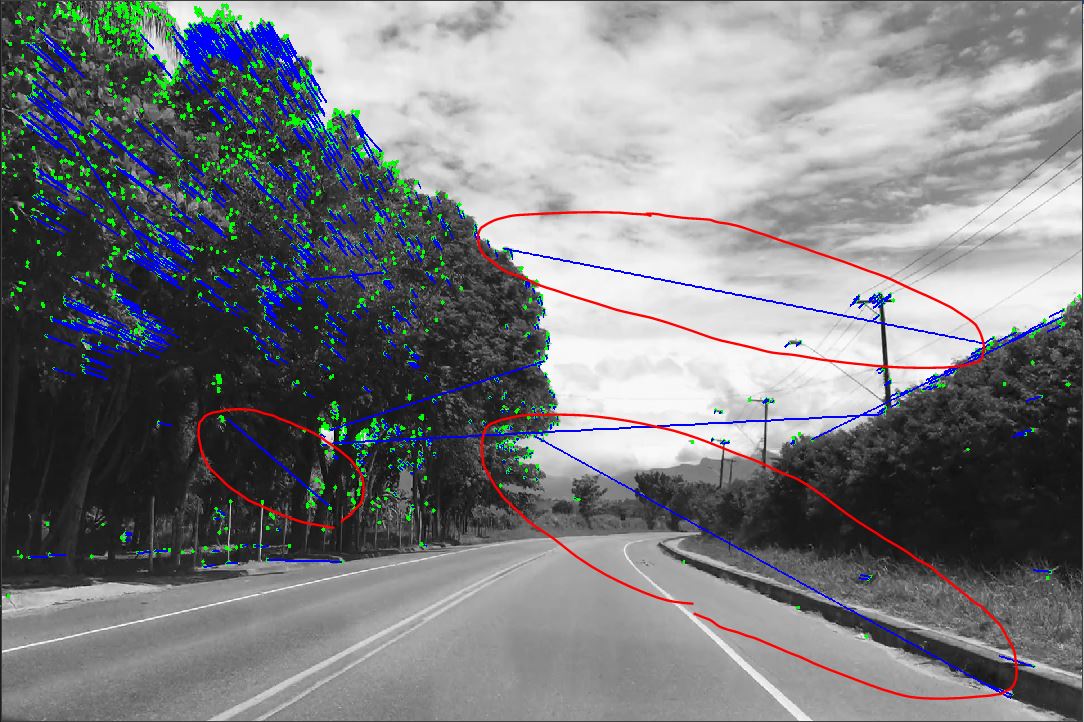


Fig 3.2 Ratio = 0.75

As observed from Fig 3.1 and Fig 3.2, increasing the ratio will produce more matches in the frame. However, it also introduces more bad matches.

**Update 09/04/2021:**

Can successfully compare descriptors for every frame (originally only once every 2 frames).

* Tested real-time tracking with webcam: Bad results obtained. Motion blurring caused by webcam’s low framerate. Maybe can improve with USB camera that has higher frame rate (to test at home with Logitech webcam).
* Decided to stick with feature tracking rather than feature matching.

Problems with feature tracking:

* Keypoints keep ‘flickering’ so the tracking is not that robust.
* Find a way to stop the flickering and “force detect” the same points in every frame.

**SLAM: Getting the pose of the camera based on detected keypoints**

**Preparation**

To get the pose of the camera based on the detected keypoints, we probably need to calculate some distances. What kinds of distances are there? What techniques or libraries can we use to calculate the distance?

Types of distances

1. Euclidean distance

Euclidean distance is the standard Cartesian coordinate type of distance. Euclidean distance can be obtained from the detected keypoints using some techniques such as triangulation.

1. Hamming distance

Hamming distance is used to calculate distance between two binary data strings. This distance was used in the BFMatcher method to compare the distances between 2 potentially matching points. This is probably **not useful** (enquire) in determining camera pose based on the keypoints detected.

What is RANSAC and how is it relevant to camera pose estimation?

Random sample consensus (RANSAC) can also be interpreted as an outlier detection method. Outliers are usually present when we do any form of data collection. These outliers can affect the results of our calculations when they are considered as part of our data set. RANSAC is an iterative technique that guesses the parameters so that we can disregard the effects that outliers have on our calculations. RANSAC generally performs better with more iterations.

Feature matching will inevitably produce some bad matches. If we consider these points when performing pose estimates, they will skew the pose calculations.

Forums:

<https://stackoverflow.com/questions/13557972/opencv-camera-pose-estimation>

1) Steps to determine camera pose:

1. Find keypoints.
2. Fundamental matrix.
3. Essential matrix.
4. Break essential matrix into rotation and translation vectors.

Fundamental matrix is related to epipolar geometry (enquire epipolar geometry).

Essential matrix relates corresponding points in stereo images, assuming the camera satisfies the pinhole camera model. Could it be something to do with the intrinsic and extrinsic parameters of the camera?