Track an Object in 3D Space – Project Readme

Instructions to run the program:

- Please pass appropriate command line arguments for detectorType and descriptorType.
- Command to run the program: './3D_object_tracking detectorType descriptorType'.
- Available Keypoint detection and descriptor algorithms are:
- detectorType = SHITOMASI, HARRIS, FAST, BRISK, ORB, AKAZE, SIFT.
- descriptorType = BRISK, BRIEF, ORB, FREAK, AKAZE, SIFT.
- Note: AKAZE descriptors work only with AKAZE keypoints.
- Note: Avoid 'detectorType = SIFT' + 'descriptorType = ORB' combination.
- Note: for descriptorType = SIFT → use - string descriptorType = 'DES_HOG'.
- Variables such as 'descriptorType, matcherType, selectorType' must be changed manually in the code.

Introduction:

- Initial steps completed were integration of logic and functions for data buffer (ring buffer), keypoint detection, descriptors and keypoint descriptor matching, which were already implemented in Mid Term project.
- Based on Mid Term project review, command line arguments functionality was implemented which would let us choose any combination of keypoint detector, descriptor algorithms from the available list while running the project from terminal.
- Command line arguments functionality was implemented in FinalProject_Camera.cpp in lines 28-40, 183 and 221.

FP.1 Match 3D Objects

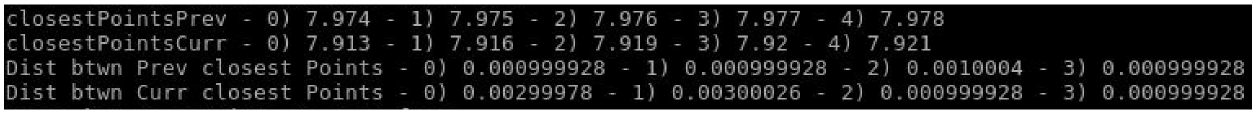
- This involves implementation of the function 'matchBoundingBoxes' in camFusion_Student.cpp file. (Line - 315)
- Inputs available to the function are matches, previous and current DataFrames.
- Target: to return a map container with IDs of bounding boxes that are matched between previous and current frames.
- Order of matched bounding boxIDs returned → map<boxID of prevBB, boxID of currBB>.
- Basic idea was, for every keypoint match we might get a bounding box association, hence we take an outer loop of Keypoint matches. With keypoint matches, we can find matched keypoints in previous and current frames. Then checking which bouding box in previous and current frames enclose these keypoints. This will result in multiple potential matches which are stored in multimap associative container. Later, correct match candidates are found based on highest number of occurrences.
- Detailed explanation:
- With for loop over Keypoint matches → for every keypoint match, keypoint in current (KptCurr) and previous (KptPrev) frames are determined.
- Then checking which bounding box in previous and current frames enclose these keypoints. Due to overlapping of some bounding boxes in the highway scene, there is a possibility that a keypoint be enclosed by more than 1 bounding box. This can happen in both current and

previous frames. Hence in both current and previous frames, if a Keypoint is enclosed by more than 1 bounding box or no bounding box. Then that keypoint match was not considered for 3D object tracking.

- Keypoints enclosed by only 1 bounding box are then considered to fill matched bounding boxIDs in multimap named ‘potentialBBMatches’.
- As the key value of any element in multimap ‘potentialBBMatches’ is bounding boxId of previous frame → ‘potentialBBMatches.equal_range(i)’ function was used to separately deal with matches of each bounding box of previous frame with bounding boxes of current frame. The fact that “Bounding box IDs in any DataFrame starts from 0” helps with this step.
- Then finding matches with highest number of occurrences and inserting them into map container - ‘bbBestMatches’ were done.
- Each individual line numbers in code is not mentioned here as necessary comments were mentioned in code at the respective places.

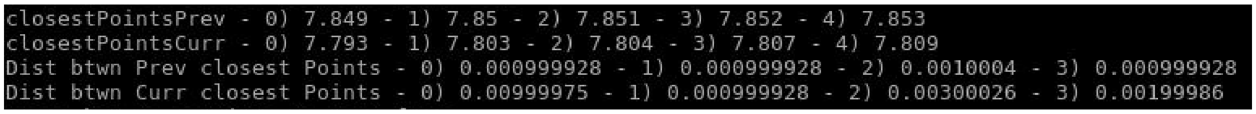
FP.2 Compute Lidar-based TTC

- This task involves implementation of the function ‘computeTTCLidar’ in camFusion_Student.cpp at line 152.
- Assumption: Here we have single randomly popping LiDAR points as outliers. In the case of outliers existing as small cluster of LiDAR points, these approaches must be improvised. (Mentioned later).
- My main idea to identify outlier is to check distance between successive closest LiDAR points and find a threshold distance value. If distance between 2 successive closest LiDAR point is greater than distance threshold then the closest point is an outlier. (explained below)
- Below are 2 example cases of distance to 5 closest LiDAR points (**<u>without an outlier</u>**) and the distance between these points.
- Example 1:



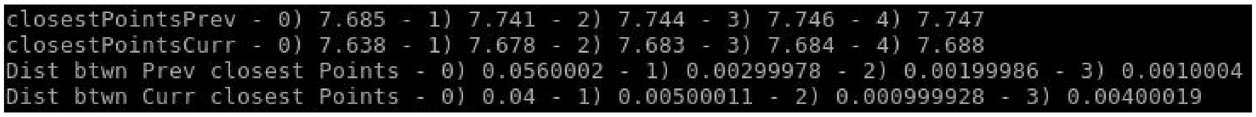
```
closestPointsPrev - 0) 7.974 - 1) 7.975 - 2) 7.976 - 3) 7.977 - 4) 7.978
closestPointsCurr - 0) 7.913 - 1) 7.916 - 2) 7.919 - 3) 7.92 - 4) 7.921
Dist btwn Prev closest Points - 0) 0.000999928 - 1) 0.000999928 - 2) 0.0010004 - 3) 0.000999928
Dist btwn Curr closest Points - 0) 0.00299978 - 1) 0.00300026 - 2) 0.000999928 - 3) 0.000999928
```

- Example 2:



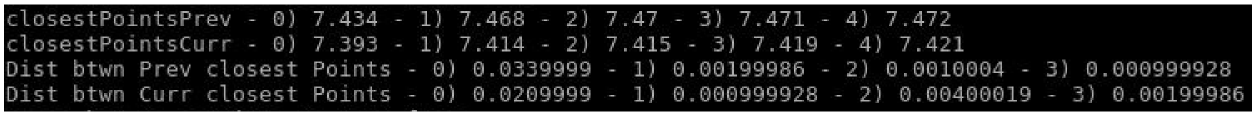
```
closestPointsPrev - 0) 7.849 - 1) 7.85 - 2) 7.851 - 3) 7.852 - 4) 7.853
closestPointsCurr - 0) 7.793 - 1) 7.803 - 2) 7.804 - 3) 7.807 - 4) 7.809
Dist btwn Prev closest Points - 0) 0.000999928 - 1) 0.000999928 - 2) 0.0010004 - 3) 0.000999928
Dist btwn Curr closest Points - 0) 0.00999975 - 1) 0.000999928 - 2) 0.00300026 - 3) 0.00199986
```

- We can see that the distance between successive closest points without an outlier is very small.
- Code lines from 199 – 225 were used to print these results.
- Below are 3 example cases of distance to 5 closest LiDAR points (**<u>with an outlier</u>**) and the distance between these points.
- Example 1:

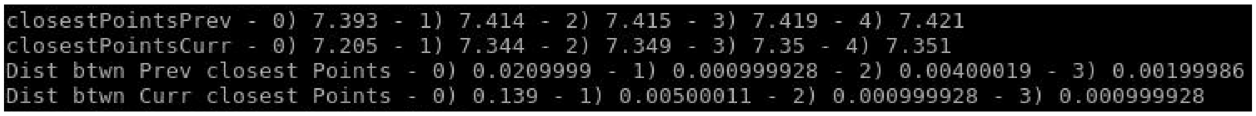


```
closestPointsPrev - 0) 7.685 - 1) 7.741 - 2) 7.744 - 3) 7.746 - 4) 7.747
closestPointsCurr - 0) 7.638 - 1) 7.678 - 2) 7.683 - 3) 7.684 - 4) 7.688
Dist btwn Prev closest Points - 0) 0.0560002 - 1) 0.00299978 - 2) 0.00199986 - 3) 0.0010004
Dist btwn Curr closest Points - 0) 0.04 - 1) 0.00500011 - 2) 0.000999928 - 3) 0.00400019
```

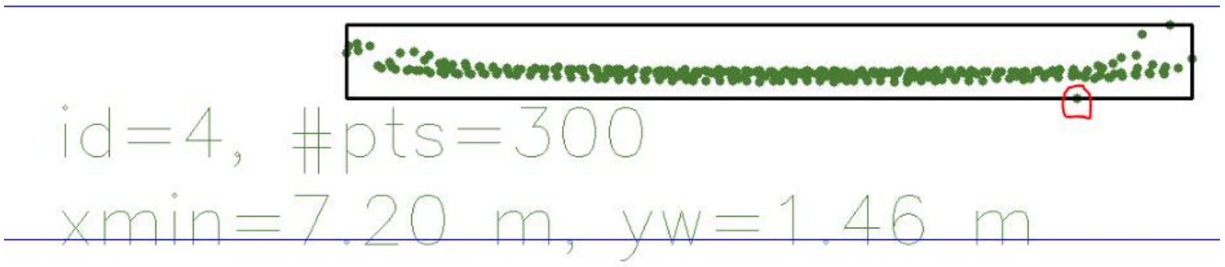
- Example 2:

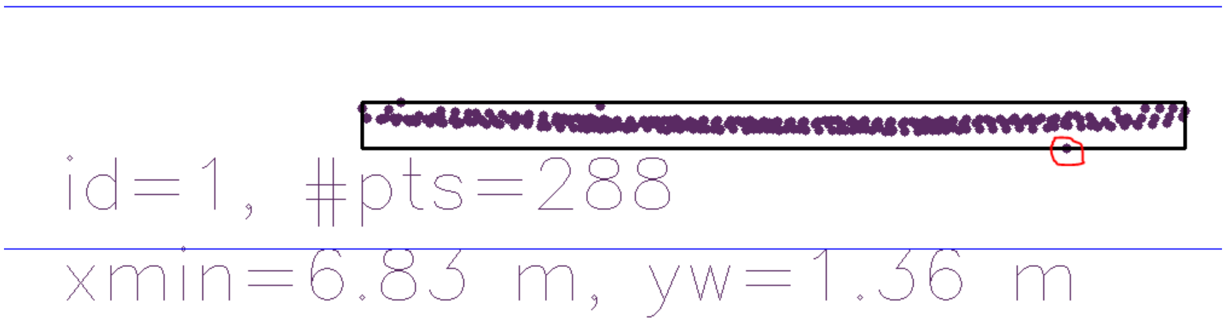


```
closestPointsPrev - 0) 7.434 - 1) 7.468 - 2) 7.47 - 3) 7.471 - 4) 7.472
closestPointsCurr - 0) 7.393 - 1) 7.414 - 2) 7.415 - 3) 7.419 - 4) 7.421
Dist btwn Prev closest Points - 0) 0.0339999 - 1) 0.00199986 - 2) 0.0010004 - 3) 0.000999928
Dist btwn Curr closest Points - 0) 0.0209999 - 1) 0.000999928 - 2) 0.00400019 - 3) 0.00199986
```

- Example 3:
- 

```
closestPointsPrev - 0) 7.393 - 1) 7.414 - 2) 7.415 - 3) 7.419 - 4) 7.421
closestPointsCurr - 0) 7.205 - 1) 7.344 - 2) 7.349 - 3) 7.35 - 4) 7.351
Dist btwn Prev closest Points - 0) 0.0209999 - 1) 0.000999928 - 2) 0.00400019 - 3) 0.00199986
Dist btwn Curr closest Points - 0) 0.139 - 1) 0.00500011 - 2) 0.000999928 - 3) 0.000999928
```

- We can observe that, with outliers, distance between successive closest points is ranging from around 0.02 to 0.15. Hence distance threshold is taken as 0.01.
- If the distance between 2 successive closest LiDAR points is greater than "distance threshold = 0.01" then the closest point is an outlier.
- Note – Tests were conducted with both "distance threshold" = 0.01 and 0.03. But chose 0.01 as final value.
- Implemented this task with 2 different approaches.
- Approach 1 – from Line 161 – 178.
- Approach 1 – as we have single outlier points as shown below. I collected 5 closest distances from point cloud into vectors "closestPointsPrev" and "closestPointsCurr". Function with name "nextClosestLidarPoints" helps in achieving this. We can collect as many closest distances as required by just changing the last parameter of the function. I chose 4 here as we have only 1 outlier point.
- 

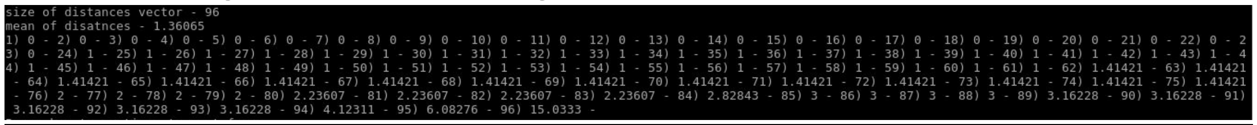
- 

- Then from code lines 276 – 303. Distance between 3 closest successive points was checked and if all these are less than distance threshold = 0.01, then the closest point is considered as a stable and reliable point. If that is not the case, then the closest point is an outlier, hence ignored and this process is continued for next 3 closest points.
- This would work only because here we have single random outlier points, but if we have a small cluster of outliers, then this distance threshold checking logic must be adopted to those requirements.
- But this certainly is not a generalized approach as we are considering only first 5 closest points, we can consider more points by just changing parameters of "nextClosestLidarPoints" function,

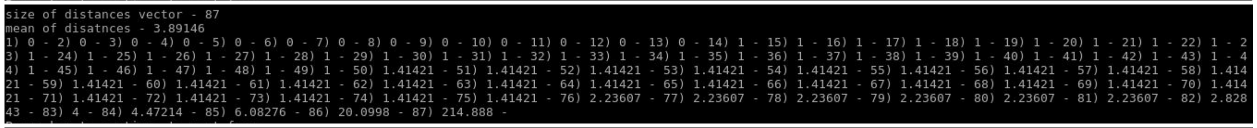
but that has to be done manually and involves multiple loops through Lidar point clouds (increasing time consumption).

- Hence Approach 2 – code lines 183 – 193.
- In approach 2, we take all Lidar point distances into a vector, then sort them in ascending order and repeat distance threshold checking logic from lines 276 – 303.
- Approach 2 is generalized and will work in all cases, as long as we have single random outliers.
- Code lines from 232 – 303, involves improvisation of same distance threshold checking logic.
- Approach 1 and 2 deliver same results.
- Based on parameters like time complexity, memory consumption and generalized solution:
- Approach 1 – not generalized solution, slow and less memory consumption.
- Approach 2 – generalized solution, fast and more memory consumption.
- I request for your opinion on which one can be considered superior and would like to know if there is a better approach in solving "Lidar based TTC" problem.

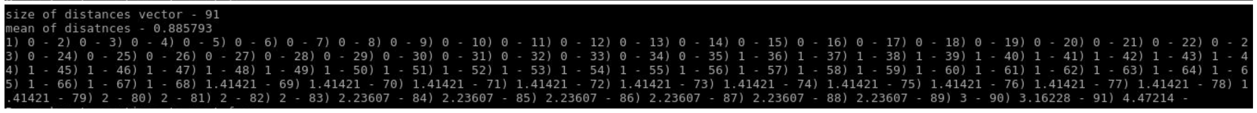
FP.3: Associate Keypoint Correspondences with Bounding Boxes

- With a for loop through keypoint matches, all those matches whose keypoint is enclosed by region of interest of bounding box are added to vector – 'enclosedMatches'.
- As per the suggestion in question, I calculated Euclidian distance between current and previous keypoints of a match and added the distance to a vector – 'distances'.
- Later mean of the distances was calculated. To understand how to filter keypoint correspondences based on Euclidian distances, their values along with the mean distance were printed and observed like show below.
- Below, I provided 3 scenarios.
- Please zoom in to get a clear view of the images below.
- 

```
size of distances vector - 96
mean of disatnces - 1.36065
1) 0 - 2) 0 - 3) 0 - 4) 0 - 5) 0 - 6) 0 - 7) 0 - 8) 0 - 9) 0 - 10) 0 - 11) 0 - 12) 0 - 13) 0 - 14) 0 - 15) 0 - 16) 0 - 17) 0 - 18) 0 - 19) 0 - 20) 0 - 21) 0 - 22) 0 - 2
3) 0 - 24) 1 - 25) 1 - 26) 1 - 27) 1 - 28) 1 - 29) 1 - 30) 1 - 31) 1 - 32) 1 - 33) 1 - 34) 1 - 35) 1 - 36) 1 - 37) 1 - 38) 1 - 39) 1 - 40) 1 - 41) 1 - 42) 1 - 43) 1 - 4
4) 1 - 45) 1 - 46) 1 - 47) 1 - 48) 1 - 49) 1 - 50) 1 - 51) 1 - 52) 1 - 53) 1 - 54) 1 - 55) 1 - 56) 1 - 57) 1 - 58) 1 - 59) 1 - 60) 1 - 61) 1 - 62) 1.41421 - 63) 1.41421
 - 64) 1.41421 - 65) 1.41421 - 66) 1.41421 - 67) 1.41421 - 68) 1.41421 - 69) 1.41421 - 70) 1.41421 - 71) 1.41421 - 72) 1.41421 - 73) 1.41421 - 74) 1.41421 - 75) 1.41421
 - 76) 2 - 77) 2 - 78) 2 - 79) 2 - 80) 2.23607 - 81) 2.23607 - 82) 2.23607 - 83) 2.23607 - 84) 2.82843 - 85) 3 - 86) 3 - 87) 3 - 88) 3 - 89) 3.16228 - 90) 3.16228 - 91)
 3.16228 - 92) 3.16228 - 93) 3.16228 - 94) 4.12311 - 95) 6.08276 - 96) 15.0333 -
```

- 

```
size of distances vector - 87
mean of disatnces - 3.89146
1) 0 - 2) 0 - 3) 0 - 4) 0 - 5) 0 - 6) 0 - 7) 0 - 8) 0 - 9) 0 - 10) 0 - 11) 0 - 12) 0 - 13) 0 - 14) 1 - 15) 1 - 16) 1 - 17) 1 - 18) 1 - 19) 1 - 20) 1 - 21) 1 - 22) 1 - 2
3) 1 - 24) 1 - 25) 1 - 26) 1 - 27) 1 - 28) 1 - 29) 1 - 30) 1 - 31) 1 - 32) 1 - 33) 1 - 34) 1 - 35) 1 - 36) 1 - 37) 1 - 38) 1 - 39) 1 - 40) 1 - 41) 1 - 42) 1 - 43) 1 - 4
4) 1 - 45) 1 - 46) 1 - 47) 1 - 48) 1 - 49) 1 - 50) 1.41421 - 51) 1.41421 - 52) 1.41421 - 53) 1.41421 - 54) 1.41421 - 55) 1.41421 - 56) 1.41421 - 57) 1.41421 - 58) 1.414
21 - 59) 1.41421 - 60) 1.41421 - 61) 1.41421 - 62) 1.41421 - 63) 1.41421 - 64) 1.41421 - 65) 1.41421 - 66) 1.41421 - 67) 1.41421 - 68) 1.41421 - 69) 1.41421 - 70) 1.414
21 - 71) 1.41421 - 72) 1.41421 - 73) 1.41421 - 74) 1.41421 - 75) 1.41421 - 76) 2.23607 - 77) 2.23607 - 78) 2.23607 - 79) 2.23607 - 80) 2.23607 - 81) 2.23607 - 82) 2.828
43 - 83) 4 - 84) 4.47214 - 85) 6.08276 - 86) 20.0998 - 87) 214.888 -
```

- 

```
size of distances vector - 91
mean of disatnces - 0.885793
1) 0 - 2) 0 - 3) 0 - 4) 0 - 5) 0 - 6) 0 - 7) 0 - 8) 0 - 9) 0 - 10) 0 - 11) 0 - 12) 0 - 13) 0 - 14) 0 - 15) 0 - 16) 0 - 17) 0 - 18) 0 - 19) 0 - 20) 0 - 21) 0 - 22) 0 - 2
3) 0 - 24) 0 - 25) 0 - 26) 0 - 27) 0 - 28) 0 - 29) 0 - 30) 0 - 31) 0 - 32) 0 - 33) 0 - 34) 0 - 35) 1 - 36) 1 - 37) 1 - 38) 1 - 39) 1 - 40) 1 - 41) 1 - 42) 1 - 43) 1 - 4
4) 1 - 45) 1 - 46) 1 - 47) 1 - 48) 1 - 49) 1 - 50) 1 - 51) 1 - 52) 1 - 53) 1 - 54) 1 - 55) 1 - 56) 1 - 57) 1 - 58) 1 - 59) 1 - 60) 1 - 61) 1 - 62) 1 - 63) 1 - 64) 1 - 6
5) 1 - 66) 1 - 67) 1 - 68) 1.41421 - 69) 1.41421 - 70) 1.41421 - 71) 1.41421 - 72) 1.41421 - 73) 1.41421 - 74) 1.41421 - 75) 1.41421 - 76) 1.41421 - 77) 1.41421 - 78) 1
.41421 - 79) 2 - 80) 2 - 81) 2 - 82) 2 - 83) 2.23607 - 84) 2.23607 - 85) 2.23607 - 86) 2.23607 - 87) 2.23607 - 88) 2.23607 - 89) 3 - 90) 3.16228 - 91) 4.47214 -
```

- We are not for keypoint matches with Euclidian distance = 0.
- Based on observing all scenarios, to not deviate too much from mean distance and not to lose useful information, I would consider those keypoint matches whose Euclidian distance is greater than or equal to 1 and less than 3 times mean distance.
- Loop through keypoint matches vector. Remove those matches that are outside set target.
- Later, filtered keypoint matches are added to the bounding boxes.

FP.4: Compute Camera-based TTC

- Here, the same implementation that was taught in 'camera based TTC estimation' was used to estimate TTC.
- To make the estimation robust, 'median Distance ratio' was considered instead of mean.

FP.5 : Performance Evaluation 1

- All the below results are as per distance threshold = 0.01m (explained above). Also in the middle row, which is distance to preceding vehicle, the values are distances to reliable closest LiDAR points considered for TTC estimation. These values are obtained after identifying and ignoring outlier LiDAR points based on distance threshold logic. To print these results in terminal → uncomment 199 – 225 lines in camFusion_Student.cpp.

Image number in sequence	Distance to preceding vehicle based on distance threshold logic (m)	LiDAR based TTC in seconds
1	7.974	
2	7.913	12.972
3	7.849	12.264
4	7.793	13.916
5	7.741	14.886
6	7.678	12.187
7	7.577	7.501
8	7.555	34.34
9	7.515	18.787
10	7.468	15.889
11	7.414	13.729
12	7.344	10.491
13	7.272	10.099
14	7.194	9.223
15	7.129	10.9677
16	7.042	8.094
17	6.963	8.813
18	6.896	10.292
19	6.814	8.309

- The above highlighted cases (in yellow) are 2 situations where I think LiDAR based TTC estimation is way off.
- Reasons for why TTC estimation in these cases is way off depend on:
    - How the distance to closest LiDAR point is changing in successive frames.
    - Usage of constant Velocity Model.
- For example, moving from frames 7 to 8 in above table, we can observe the distance to closest LiDAR point decreases from 7.577 to 7.555 meters, which is less than how the distance to closest LiDAR point is changing in successive frames in other scenarios like moving from frame 1 to 2. Hence, this suggests that constant velocity model does not deliver very promising results.

FP.6: Performance Evaluation 2

- Results of comparing all detector / descriptor combinations implemented in previous chapters with regard to the TTC estimate on a frame-by-frame basis are present in excel sheet submitted. Excel sheet is present in the 'SFND_3D_OBJECT_TRACKING' project folder itself.
- TTC values that are way off are highlighted in yellow in excel sheet. The main aspect that we are calculating while estimating TTC is distance ratio. In code I took median of the distance ratios to

make it a robust estimation. The value of TTC heavily depends on distance ratio, hence the factors causing TTC values to go way off are distance ratio and considering median of the distance ratios.

- Excel sheet contains both numeric values and graphs.
- After observing the trends of all available detector / descriptor combinations, multiple combinations appeared to be performing at more or less similar level, but
    - SHITOMASI (detector) / ORB (descriptor)
    - SHITOMASI (detector) / FREAK (descriptor)
    - SHITOMASI (detector) / SIFT (descriptor)
      combinations appeared to perform the best as they showed a gradual decrease in TTC without drastic deviations as we moved further through frames.

Question:

- With LiDAR based TTC values and camera based TTC values with different detector / descriptor combinations, with these values varying from each other for every frame, how (what method) can we make use of these to come up with a final and reliable TTC value for each frame?