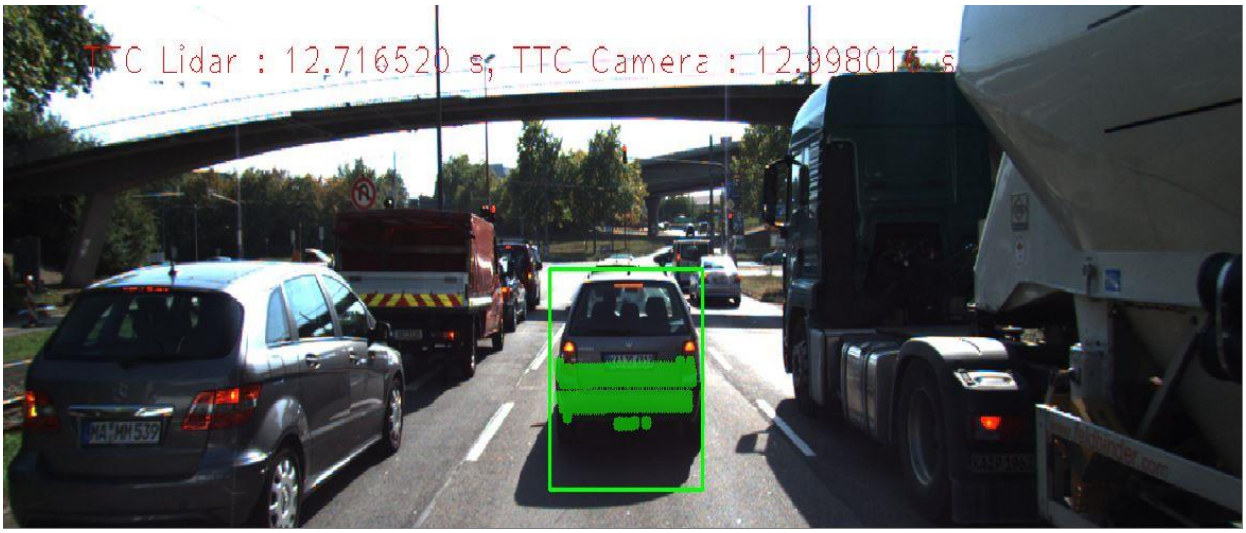
3D Model - Tracking



In this project, we are tracking an object in environment with the help of combination of feature detection in consecutives images and lidar point cloud. So, we can calculate TTC with vehicle in front of autonomous system.

1. Match 3D Objects

With the help of key point matching in previous and current image with can be able to detect the 3D object in array of images. As we know vector of DMatch data structure hold the all-matched key points in both images, we the help of looping through all the matched key points we can find all the objects with bounding box associated with it.

If the key point associate with query index present in previous image boxes and point associate with train index present in current image boxes, then that match represents the same box in both the images, hence pertain to same object in both the images.

A key point present in box or not and which box ID, can be calculated with the help of loop of the all boxes present in particular frame.

Now if the match present in both the frames so the BOX ID of the particular frame (both frames) can be saved as the map of box IDs (paired together) and saved in a vector of matched bounding boxes.

Below code shows the implementations -

```cpp
void matchBoundingBoxes(std::vector<cv::DMatch> &matches, std::map<int, int> &bbBestMatches, DataFrame &prevFrame, DataFrame &currFrame)
{
    vector<cv::Point2f> matched_points1, matched_points2;
    vector<int> prevframeIDX, preframeboxIDs, currframeIDX, currframeboxIDs;
    map<int, int> idx,preframemap, currframemap, allmaps;
    bool preflag = false, currflag = false;
    for (int i = 0; i < matches.size(); i++)
    {

        int idx1 = matches[i].trainIdx;
        int idx2 = matches[i].queryIdx;

        cv::Point P(prevFrame.keypoints[idx2].pt.x, prevFrame.keypoints[idx2].pt.y);
        for (auto prevframeboxes = 0; prevframeboxes < prevFrame.boundingBoxes.size(); ++prevframeboxes) {
            if (prevFrame.boundingBoxes[prevframeboxes].roi.contains(P)) {
                prevframeIDX.push_back(idx2);
                preflag = true;
                preframeboxIDs.push_back(prevFrame.boundingBoxes[prevframeboxes].boxID);
                preframemap.insert(pair<int, int>(idx2, prevFrame.boundingBoxes[prevframeboxes].boxID));
            }
        }

        cv::Point C(currFrame.keypoints[idx1].pt.x, currFrame.keypoints[idx1].pt.y);
        for (auto currframeboxes = 0; currframeboxes < currFrame.boundingBoxes.size(); ++currframeboxes) {
            if (currFrame.boundingBoxes[currframeboxes].roi.contains(C)) {
                currframeIDX.push_back(idx1);
                currflag = true;
                currframeboxIDs.push_back(currFrame.boundingBoxes[currframeboxes].boxID);
                currframemap.insert(pair<int, int>(idx1, currFrame.boundingBoxes[currframeboxes].boxID));
            }
        }

        if ((preflag) && (currflag)) {
            bbBestMatches.insert(pair<int, int>(preframemap[idx2], currframemap[idx1]));
            idx.insert(pair<int, int>(idx2, idx1));
        }
        preflag = false;
        currflag = false;
        matched_points1.push_back(prevFrame.keypoints[idx2].pt);
        matched_points2.push_back(currFrame.keypoints[idx1].pt);
    }

}
```

2. Compute Lidar-based TTC

Computing the time-to-collision in seconds for all matched 3D objects using only Lidar measurements from the matched bounding boxes between current and previous frame with the of distance, velocity and time relation. For dealing with outliers – average of all points has been taken. So, the error in calculation reduces.

```
void computeTTCLidar(std::vector<LidarPoint> &lidarPointsPrev,
                     std::vector<LidarPoint> &lidarPointsCurr, double frameRate, double &TTC)
{
    double dT = 1/frameRate;
    double laneWidth = 4.0;
    vector<double> preX, currX;
    double totalpreX{ 0 }, totalcurrX{0};

    double minXPrev = 1e9, minXCurr = 1e9;
    for (auto it = lidarPointsPrev.begin(); it != lidarPointsPrev.end(); ++it)
    {

        if (abs(it->y) <= laneWidth / 2.0)
        {
            minXPrev = minXPrev > it->x ? it->x : minXPrev;
            preX.push_back(minXPrev);
        }
    }

    for (auto it = lidarPointsCurr.begin(); it != lidarPointsCurr.end(); ++it)
    {

        if (abs(it->y) <= laneWidth / 2.0)
        {
            minXCurr = minXCurr > it->x ? it->x : minXCurr;
            currX.push_back(minXCurr);
        }
    }
    for (auto i : preX) {
        totalpreX += i;
    }
    for (auto k : currX) {
        totalcurrX += k;
    }

    // compute TTC from both measurements
    TTC = (totalcurrX/currX.size()) * dT / ((totalpreX/preX.size()) - (totalcurrX / currX.size()));
}
```

3. Associate Key point Correspondences with Bounding Boxes

In this task all the matches associate with particular bounding box in both previous frame and current frame saved in bounding box key points matches. A loop has been performed through all the matches points in both frames, if the point key point present in both the frame, that particular match should be save as match key point associated with that box. The problem with these all matches it that, it contains some outliers. So, to remove the outliers, standard deviation is used. 95% percent within two standard deviations is used to remove the far matches in the vector of matches. Below code shows the implementation.

```cpp
// associate a given bounding box with the keypoints it contains
void clusterKptMatchesWithROI(BoundingBox &boundingBox, std::vector<cv::KeyPoint> &kptsPrev, std::vector<cv::KeyPoint> &kptsCurr, std::vector<cv::DMatch> &kptMatches)
{
    std::vector<cv::DMatch> matches;
    double dist{0.0}, meandist, variance = 0.0, stdDeviation;

    for (auto i = 0; i < kptMatches.size(); ++i) {
        int idx1 = kptMatches[i].trainIdx;
        int idx2 = kptMatches[i].queryIdx;
        if (boundingBox.roi.contains(kptsPrev[idx2].pt)) {
            if (boundingBox.roi.contains(kptsCurr[idx1].pt)) {
                matches.push_back(kptMatches[i]);
            }
        }
    }

    for (auto i = 0; i < matches.size(); ++i) {
        dist += (double)matches[i].distance;
        //cout << matches[i].distance << ",";
    }
    //cout << endl;

    meandist = dist / matches.size();

    for (auto i = 0; i < matches.size(); ++i) {
        variance += pow(matches[i].distance - meandist, 2);
    }
    stdDeviation = sqrt(variance / matches.size());

    for (auto i = 0; i < matches.size(); ++i) {
        if (((matches[i].distance) < (meandist + (2 * stdDeviation))) && ((matches[i].distance) > (meandist - (2 * stdDeviation)))) {
                boundingBox.kptMatches.push_back(matches[i]);
        }
    }
}
```

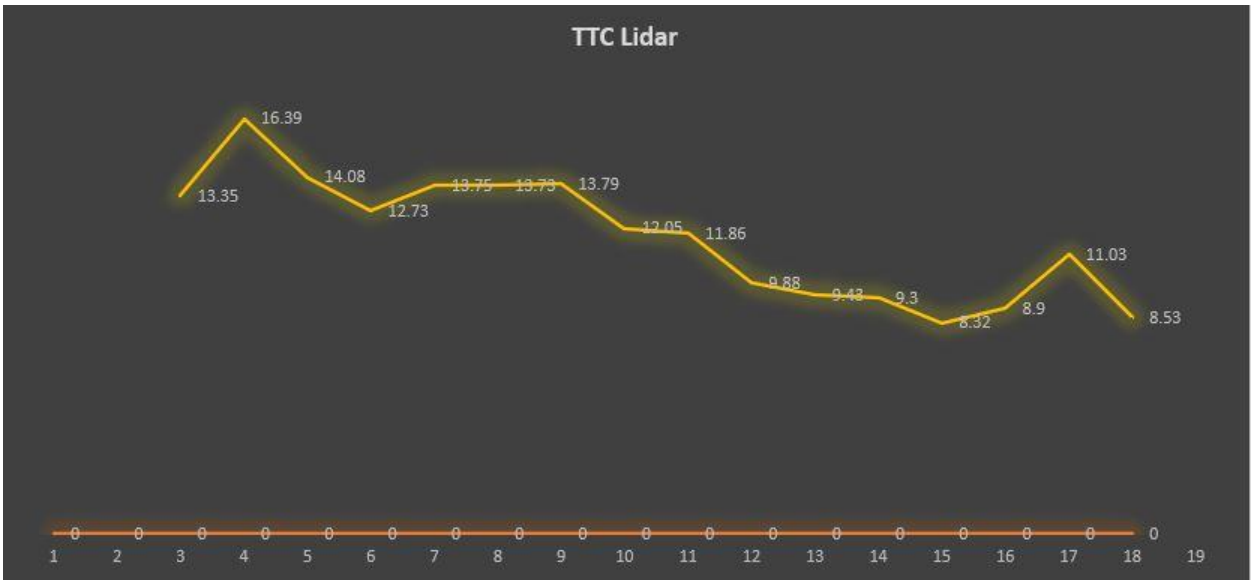
4. Compute Camera-based TTC

Two successive images collected from MONO camera setup is used to calculated TTC with the common points of interests. MedianDistRation is used to deal with the outliers, below code shows the implementation.

```cpp
void computeTTCCamera(std::vector<cv::KeyPoint> &kptsPrev, std::vector<cv::KeyPoint> &kptsCurr,
                      std::vector<cv::DMatch> kptMatches, double frameRate, double &TTC, cv::Mat *visImg)
{
    vector<double> distRatios;
    for (auto it1 = kptMatches.begin(); it1 != kptMatches.end() - 1; ++it1)
    {

        cv::KeyPoint kpOuterCurr = kptsCurr.at(it1->trainIdx);
        cv::KeyPoint kpOuterPrev = kptsPrev.at(it1->queryIdx);

        for (auto it2 = kptMatches.begin() + 1; it2 != kptMatches.end(); ++it2)
        {

            double minDist = 100.0;

            cv::KeyPoint kpInnerCurr = kptsCurr.at(it2->trainIdx);
            cv::KeyPoint kpInnerPrev = kptsPrev.at(it2->queryIdx);


            double distCurr = cv::norm(kpOuterCurr.pt - kpInnerCurr.pt);
            double distPrev = cv::norm(kpOuterPrev.pt - kpInnerPrev.pt);

            if (distPrev > std::numeric_limits<double>::epsilon() && distCurr >= minDist)
            {

                double distRatio = distCurr / distPrev;
                distRatios.push_back(distRatio);
            }
        }
    }

    if (distRatios.size() == 0)
    {
        TTC = NAN;
        return;
    }

    std::sort(distRatios.begin(), distRatios.end());
    long medIndex = floor(distRatios.size() / 2.0);
    double medDistRatio = distRatios.size() % 2 == 0 ? (distRatios[medIndex - 1] + distRatios[medIndex]) / 2.0 : distRatios[medIndex];
    double dT = double(1) / frameRate;
    TTC = -dT / (1 - medDistRatio);
}
```

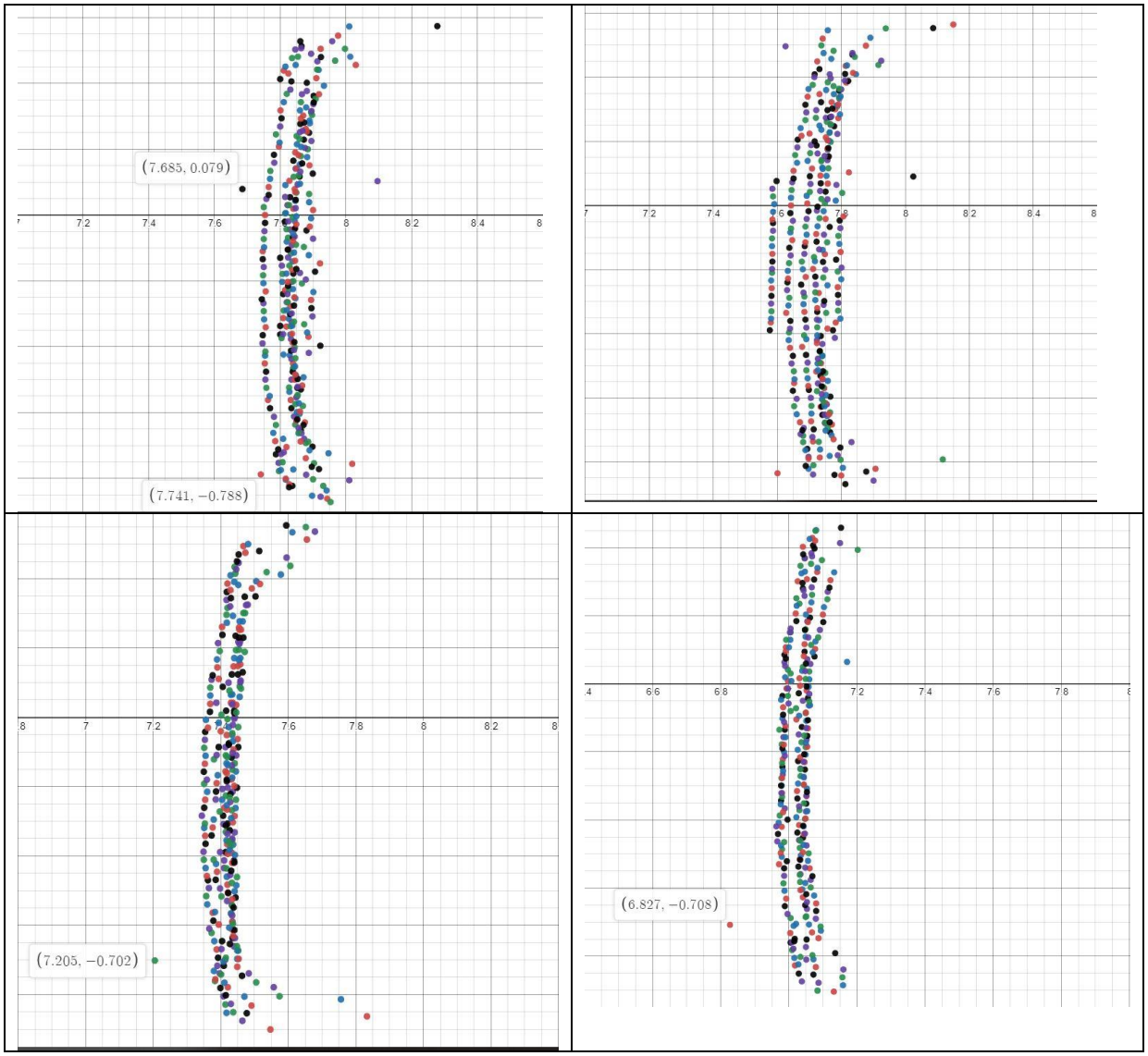
5. Performance Evaluation 1

Images	Time(s)
1st	
2nd	
3rd	13.35
4th	16.39
5th	14.08
6th	12.73
7th	13.75
8th	13.73
9th	13.79
10th	12.05
11th	11.86
12th	9.88
13th	9.43
14th	9.3
15th	8.32
16th	8.9
17th	11.03
18th	8.53



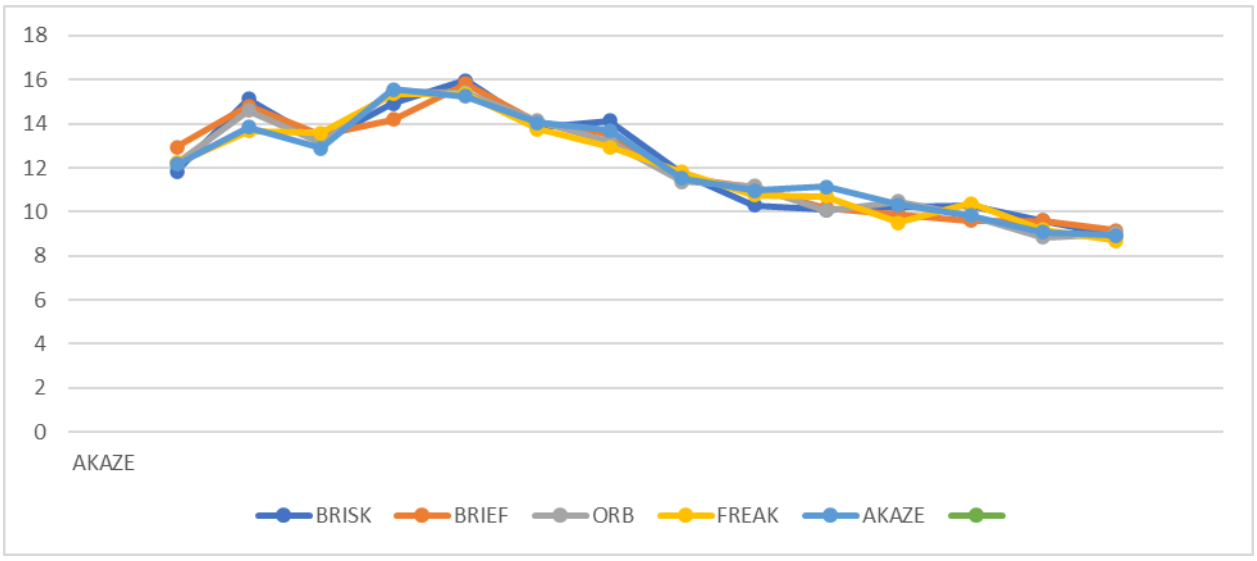
As it is observable some of the TTC calculation is way out, even though reduction of outlier's points has performed. Some of TTC is not correct because the calculation performed is not 100% accurate to remove the outliers in the data set of lidar point.

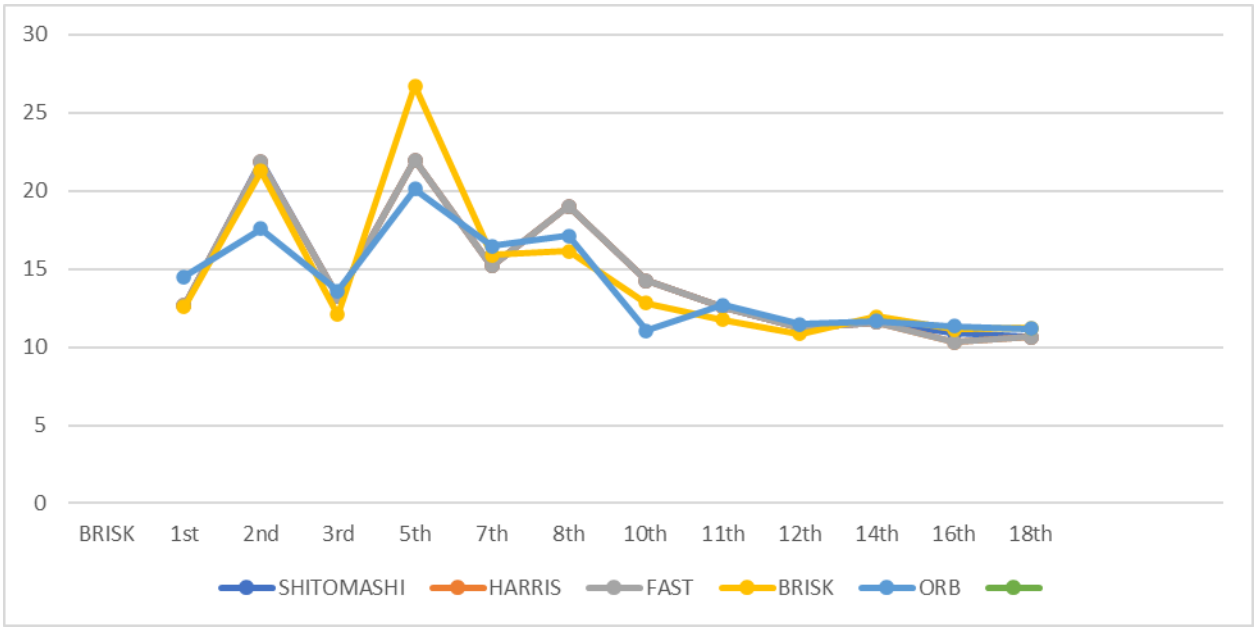
Below are some example of the outliers lies in the point cloud (while calculating TTC at images 4th, 5th, 17th etc.)

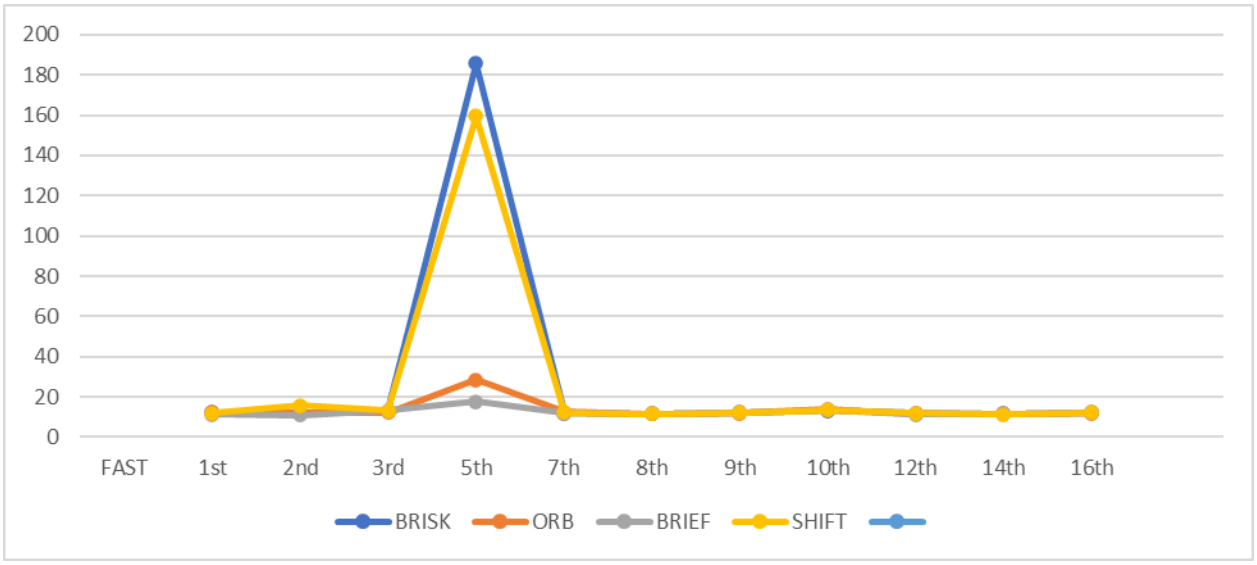


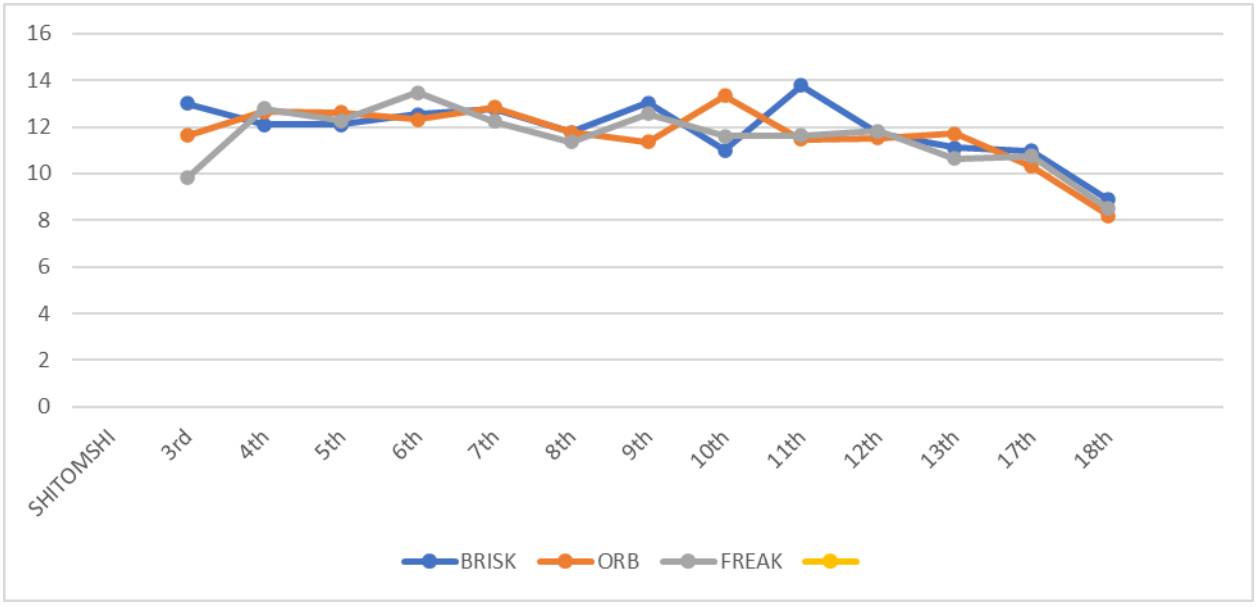
6. Performance Evaluation 2

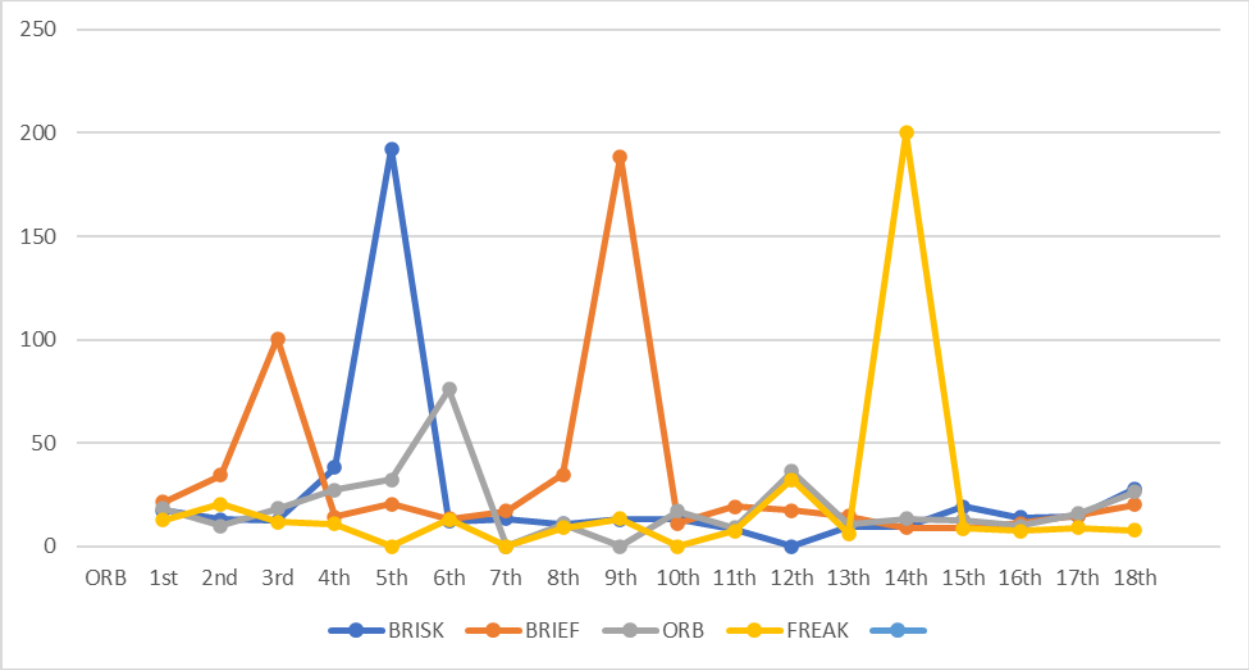
All detector / descriptor combinations implementation is shown below and Excel is attached uploaded with files.


18
16
14
12
10
8
6
4
2
0
AKAZE
BRISK
BRIEF
ORB
FREAK
AKAZE


30
25
20
15
10
5
0
BRISK
1st
2nd
3rd
5th
7th
8th
10th
11th
12th
14th
16th
18th
SHITOMASHI
HARRIS
FAST
BRISK
ORB


200
180
160
140
120
100
80
60
40
20
0
FAST
1st
2nd
3rd
5th
7th
8th
9th
10th
12th
14th
16th
BRISK
ORB
BRIEF
SHIFT


16
14
12
10
8
6
4
2
0
SHITOMSHI
3rd
4th
5th
6th
7th
8th
9th
10th
11th
12th
13th
17th
18th
BRISK
ORB
FREAK



Note – Some unaccepted result are observed via ORB keypoint detection with combination of BRISK/BRIEF/FREAK descriptor.