FP.1 Match 3D Objects

The method `matchBoundingBoxes` consists of two steps. In the first step, we examine which boxID the previous and current keypoints for each matched pairs are belong to, and store the result to `MapVoteAll`.

[camFusion_Student.cpp]

```
  std::vector<BoundingBox> boundingBoxes_prev = prevFrame.boundingBoxes;
  std::vector<BoundingBox> boundingBoxes_curr = currFrame.boundingBoxes;

  int maxid_prev = boundingBoxes_prev.size();
  int maxid_curr = boundingBoxes_curr.size();

  std::vector<MapVote> MapVoteAll;

  for (auto it1 = matches.begin(); it1 != matches.end(); ++it1)
  {
    int kpindex_prev = it1->queryIdx;
    int kpindex_curr = it1->trainIdx;

    cv::KeyPoint prev_key = prevFrame.keypoints[kpindex_prev];
    float x_prev = prev_key.pt.x;
    float y_prev = prev_key.pt.y;

    cv::KeyPoint curr_key = currFrame.keypoints[kpindex_curr];
    float x_curr = curr_key.pt.x;
    float y_curr = curr_key.pt.y;

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

         MapVote MapVote_temp;
         MapVote_temp.id_prev = -1;
         MapVote_temp.id_curr = -1;

         cv::Rect roip = it2->roi;
         if(x_prev >= roip.x && x_prev <= roip.x+roip.width && y_prev>= roip.y &&
y_prev <= roip.y+roip.height)
          {

              for (auto it3 = boundingBoxes_curr.begin(); it3 !=
boundingBoxes_curr.end(); ++it3)
              {
                cv::Rect roic = it3->roi;
                if(x_curr >= roic.x && x_curr <= roic.x+roic.width && y_curr>=
roic.y && y_curr <=roic.y+roic.height)
                {
                  MapVote_temp.id_prev = it2->boxID;
```

```
                    MapVote_temp.id_curr = it3->boxID;
                MapVoteAll.push_back(MapVote_temp);
              }

            }

        }
    }
}
```

In the second step, we examine all the member of `MapVoteAll` prepared in the first step and determine the previous and current boxID correspodance. For each previous boxID, vote the correspoinding current boxID and register the combination with the highest number of votes to `bbBestMatches`.

```
for (int i=0; i<=maxid_prev; i++){
  int vote_cnt[65535]={}; // initialize with all 0

    for (int j=0; j<=maxid_curr; j++){

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

        if( it->id_prev == i && it->id_curr == j){
          vote_cnt[j]= vote_cnt[j] + 1;
        }
      }
    }

  int maxval = 0;
  int j_maxval = -1;
  for (int j=0; j<=maxid_curr; j++){
      if( vote_cnt[j] > maxval){
        maxval = vote_cnt[j];
        j_maxval = j;
      }
  }

  bbBestMatches[i] = j_maxval;

}
```

FP.2 Compute Lidar-based TTC

The method `computeTTCLidar` computes TTC using the distance the closest lidar point from subject vehicle. In order to remove outliers of the x coordinate of lidar point, points that do not fall within 3*SD from the mean of the x coordinate are removed, and then the closest point is determined.

First we calculate the mean `mean` and standard deviation(SD) `stddev` of the lidar points.

```
// auxiliary variables
double dT = 1/frameRate; // time between two measurements in seconds
double laneWidth = 4.0; // assumed width of the ego lane

// find closest distance to Lidar points within ego lane
double minXPrev = 1e9, minXCurr = 1e9;

// Calculate mean and standard deviation
double sum=0;
double sq_sum=0;
double mean=0;
double stddev=0;

for (auto it = lidarPointsPrev.begin(); it != lidarPointsPrev.end(); ++it)
{
   sum = sum + it->x;
 }
 mean = sum / lidarPointsPrev.size();
for (auto it = lidarPointsPrev.begin(); it != lidarPointsPrev.end(); ++it)
{
   sq_sum = sq_sum + (it->x - mean)*(it->x - mean);
 }
 stddev = sq_sum / lidarPointsPrev.size();
 stddev = sqrt(stddev);

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

    //if (abs(it->y) <= laneWidth / 2.0 )
    if (abs(it->y) <= laneWidth / 2.0 && abs(it->x - mean)<= 3*stddev)
    { // 3D point within ego lane?
        minXPrev = minXPrev > it->x ? it->x : minXPrev;
    }
}

for (auto it = lidarPointsCurr.begin(); it != lidarPointsCurr.end(); ++it)
{
   sum = sum + it->x;
 }
 mean = sum / lidarPointsCurr.size();
for (auto it = lidarPointsCurr.begin(); it != lidarPointsCurr.end(); ++it)
{
   sq_sum = sq_sum + (it->x - mean)*(it->x - mean);
 }
 stddev = sq_sum / lidarPointsCurr.size();
 stddev = sqrt(stddev);
```

Next, we find the closest lidar point after removing outliers, and calculate TTC.

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

        if (abs(it->y) <= laneWidth / 2.0 && abs(it->x - mean)<= 3*stddev)
        { // 3D point within ego lane?
            minXCurr = minXCurr > it->x ? it->x : minXCurr;
        }
    }

    // compute TTC from both measurements
    TTC = minXCurr * dT / (minXPrev - minXCurr);
```

FP.3 Associate Keypoint Correspondences with Bounding Boxes

The method `clusterKptMatchesWithROI` is to prepare the TTC computation in FP.4. We examin all the matched keypoints. If the current keypoint is within the designated bounding box, it is stored to `kptMatches_tmp`.

```
  std::vector<cv::DMatch> kptMatches_tmp;
  cv::Rect roic = boundingBox.roi;

  for (auto it1 = kptMatches.begin(); it1 != kptMatches.end(); ++it1)
  {
    int kpindex_prev = it1->queryIdx;
    int kpindex_curr = it1->trainIdx;

    cv::KeyPoint curr_key = kptsCurr[kpindex_curr];
    float x_curr = curr_key.pt.x;
    float y_curr = curr_key.pt.y;

    if(x_curr >= roic.x && x_curr <= roic.x+roic.width && y_curr >= roic.y &&
y_curr <= roic.y+roic.height)
    {
       kptMatches_tmp.push_back(*it1);
    }
  }
```

Next we calculate the mean `mean` and standart deviation (SD) `stddev` of the euclidian distance between the previous and current keypoints. We use the results to remove outlier in the succeeding part.

```
  double sum=0;
  double sq_sum=0;
  double mean=0;
```

```
 double stddev=0;

 for (auto it1 = kptMatches_tmp.begin(); it1 != kptMatches_tmp.end(); ++it1)
 {
   int kpindex_prev = it1->queryIdx;
   int kpindex_curr = it1->trainIdx;

   cv::KeyPoint prev_key = kptsPrev[kpindex_prev];
   float x_prev = prev_key.pt.x;
   float y_prev = prev_key.pt.y;

   cv::KeyPoint curr_key = kptsCurr[kpindex_curr];
   float x_curr = curr_key.pt.x;
   float y_curr = curr_key.pt.y;

   sum = sum + sqrt((x_curr-x_prev)*(x_curr-x_prev)+(y_curr-y_prev)*(y_curr-
y_prev));
 }

 mean = sum / kptMatches_tmp.size();

 for (auto it1 = kptMatches_tmp.begin(); it1 != kptMatches_tmp.end(); ++it1)
 {
   int kpindex_prev = it1->queryIdx;
   int kpindex_curr = it1->trainIdx;

   cv::KeyPoint prev_key = kptsPrev[kpindex_prev];
   float x_prev = prev_key.pt.x;
   float y_prev = prev_key.pt.y;

   cv::KeyPoint curr_key = kptsCurr[kpindex_curr];
   float x_curr = curr_key.pt.x;
   float y_curr = curr_key.pt.y;

   double diff = sqrt((x_curr-x_prev)*(x_curr-x_prev)+(y_curr-y_prev)*(y_curr-
y_prev)) - mean;
   sq_sum = sq_sum + diff*diff;
 }
 stddev = sq_sum / kptMatches_tmp.size();
 stddev = sqrt(stddev);
```

Finally we remove the matched keypoints whose distance is larger than the mean + 3 * SD or smaller than the mean - 3 * SD.

```
for (auto it1 = kptMatches_tmp.begin(); it1 != kptMatches_tmp.end(); ++it1)
{
  int kpindex_prev = it1->queryIdx;
  int kpindex_curr = it1->trainIdx;

  cv::KeyPoint prev_key = kptsPrev[kpindex_prev];
  float x_prev = prev_key.pt.x;
```

```
    float y_prev = prev_key.pt.y;

    cv::KeyPoint curr_key = kptsCurr[kpindex_curr];
    float x_curr = curr_key.pt.x;
    float y_curr = curr_key.pt.y;

    double distance = sqrt((x_curr-x_prev)*(x_curr-x_prev)+(y_curr-y_prev)*(y_curr-
y_prev)) ;
    if(distance <= mean + 3*stddev && distance >= mean - 3*stddev){
      boundingBox.kptMatches.push_back(*it1);
      //cout << "[OK] distance = " << distance << endl;
    }else{
      // do nothing
      //cout << "[Removed!] distance = " << distance << endl;
    }

  }
```

FP.4 Compute Camera-based TTC

The method `computeTTCCamera` computes the time-to-collision using key point correspodances selected in FP.3. To calculate TTC, current/previous distance ratio between keypoints is needed. Here we use the median of the distance ratio instead of the mean to avoid the affect of outliers.

First, we compute distance ratios between all matched keypoints between curr. and prev. frame and store them to `distRatios`.

```
    vector<double> distRatios;

    for (auto it1 = kptMatches.begin(); it1 != kptMatches.end() - 1; ++it1)
    { // outer kpt. loop

        // get current keypoint and its matched partner in the prev. frame
        cv::KeyPoint kpOuterCurr = kptsCurr.at(it1->trainIdx);
        cv::KeyPoint kpOuterPrev = kptsPrev.at(it1->queryIdx);

        for (auto it2 = kptMatches.begin() + 1; it2 != kptMatches.end(); ++it2)
        { // inner kpt.-loop

            double minDist = 100.0; // min. required distance

            // get next keypoint and its matched partner in the prev. frame
            cv::KeyPoint kpInnerCurr = kptsCurr.at(it2->trainIdx);
            cv::KeyPoint kpInnerPrev = kptsPrev.at(it2->queryIdx);

            // compute distances and distance ratios
            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)
            { // avoid division by zero

                double distRatio = distCurr / distPrev;
                distRatios.push_back(distRatio);
            }
        } // eof inner loop over all matched kpts
    }     // eof outer loop over all matched kpts
```

Next, we sort `distRatios` to find the median of the distance ratio and calculate TTC using the result.

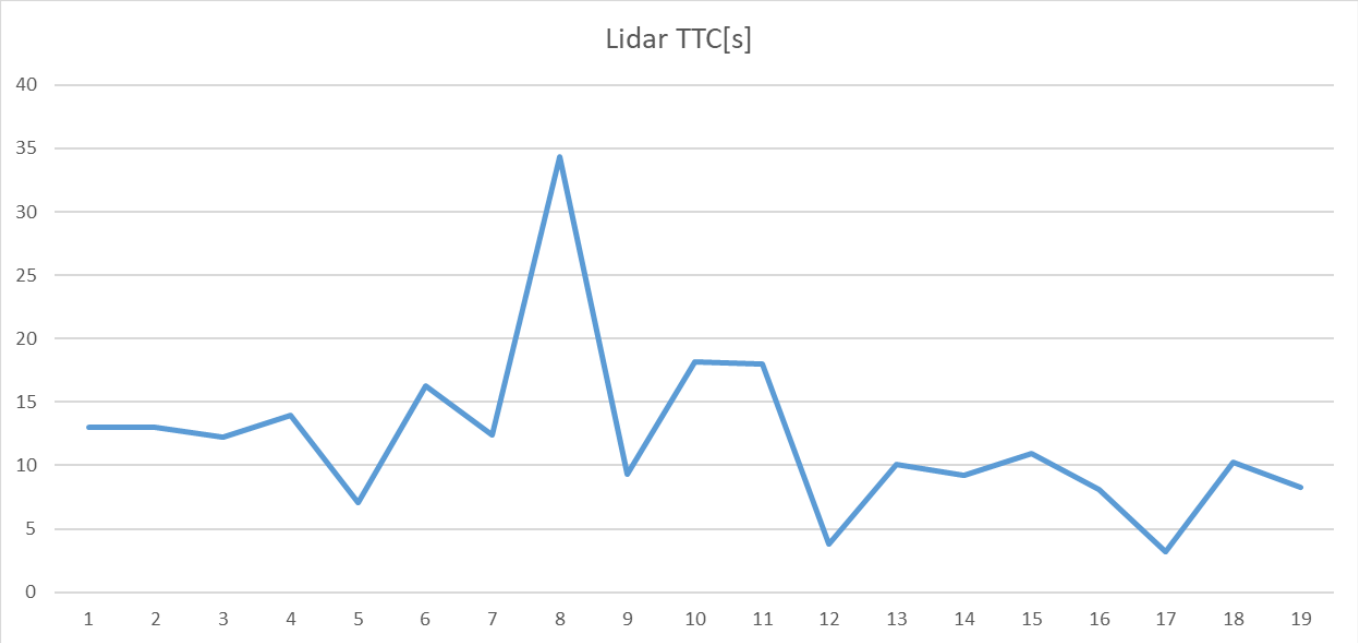
```
    std::sort(distRatios.begin(), distRatios.end());
    long medIndex = floor(distRatios.size() / 2.0);
    double medDistRatio = distRatios.size() % 2 == 0 ? (distRatios[medIndex - 1] +
distRatios[medIndex]) / 2.0 : distRatios[medIndex]; // compute median dist. ratio
to remove outlier influence

    dT = 1 / frameRate;
    TTC = -dT / (1 - medDistRatio);
```

FP.5 Performance Evaluation 1

The TTC calculation result is in the below. The calculated TTC largely changes between frames. One of the causes may be outliers. We might not sufficiently remove the affect of outliers. Another cause may be due to the vehicle's egomotion. If the pitching occurs in acceleration or deceleration, lidar sensor's angle will change and the measured distance varies. This may lead to the instantaneous distance error.

frame#	TTC[s]
1	12.9722
2	12.9722
3	12.264
4	13.9161
5	7.11572
6	16.2511
7	12.4213
8	34.3404
9	9.34376
10	18.1318

frame#	TTC[s]
11	18.0318
12	3.83244
13	10.1
14	9.22307
15	10.9678
16	8.09422
17	3.17535
18	10.2926
19	8.30978



FP.6 Performance Evaluation 2

The TTC calculation results are summarized in table and graph below for all valid combinations of detector and descriptor, which are reported in mid-term project.

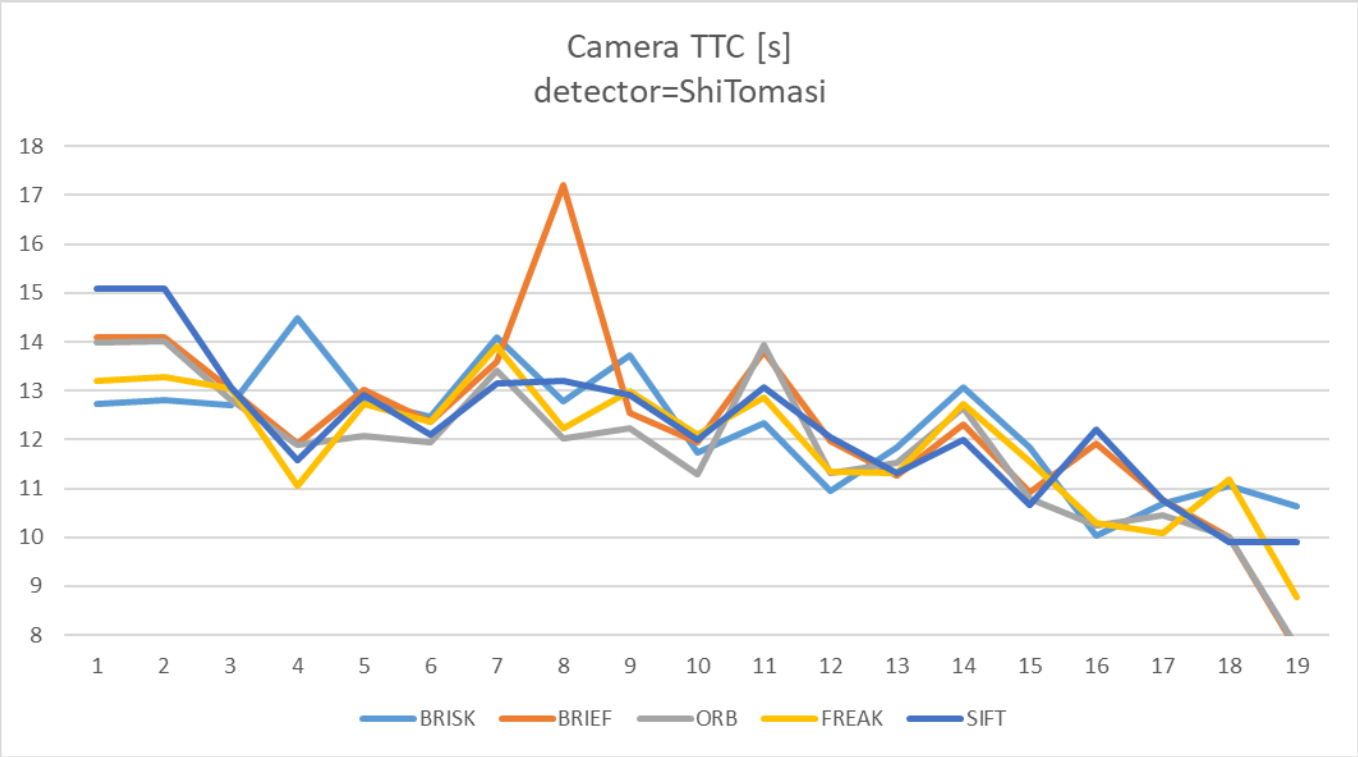
The "AKAZE" detector seems to be the best regardless of descriptors since the calculated TTC is stable between frames (judging from the images, there seems to be no sudden change in TTC or relative velocity). The second best would be "ShiTomasi".

The calculated TTC stability depends more on the detector than on the descriptor. Extracting good keypoints with an appropriate detector would be a key to compute correct TTC.

detector: ShiTomasi

frame#	BRISK	BRIEF	ORB	FREAK	SIFT

frame#	BRISK	BRIEF	ORB	FREAK	SIFT
1	12.7196	14.1041	13.9863	13.2095	15.0889
2	12.7975	14.1042	14.0087	13.2738	15.0889
3	12.6968	13.0249	12.8008	13.0387	13.0659
4	14.4793	11.9127	11.8924	11.0399	11.5827
5	12.7779	13.022	12.0792	12.7282	12.9162
6	12.4657	12.3511	11.9549	12.3506	12.1074
7	14.0887	13.6007	13.4001	13.9057	13.1405
8	12.7924	17.2033	12.0142	12.2412	13.1881
9	13.7279	12.5406	12.2435	13.0031	12.9116
10	11.7376	11.9423	11.2759	12.0881	11.9949
11	12.3288	13.8117	13.9372	12.8622	13.0762
12	10.9393	11.9619	11.3267	11.3507	12.0367
13	11.8481	11.2716	11.5356	11.3076	11.322
14	13.0781	12.3202	12.6532	12.7283	12.0037
15	11.8438	10.9335	10.7801	11.5384	10.6469
16	10.0418	11.9156	10.2304	10.297	12.1966
17	10.6857	10.7594	10.4498	10.0955	10.7601
18	11.0598	9.99594	9.99594	11.1811	9.90596
19	10.6426	7.69701	7.74606	8.77152	9.91264



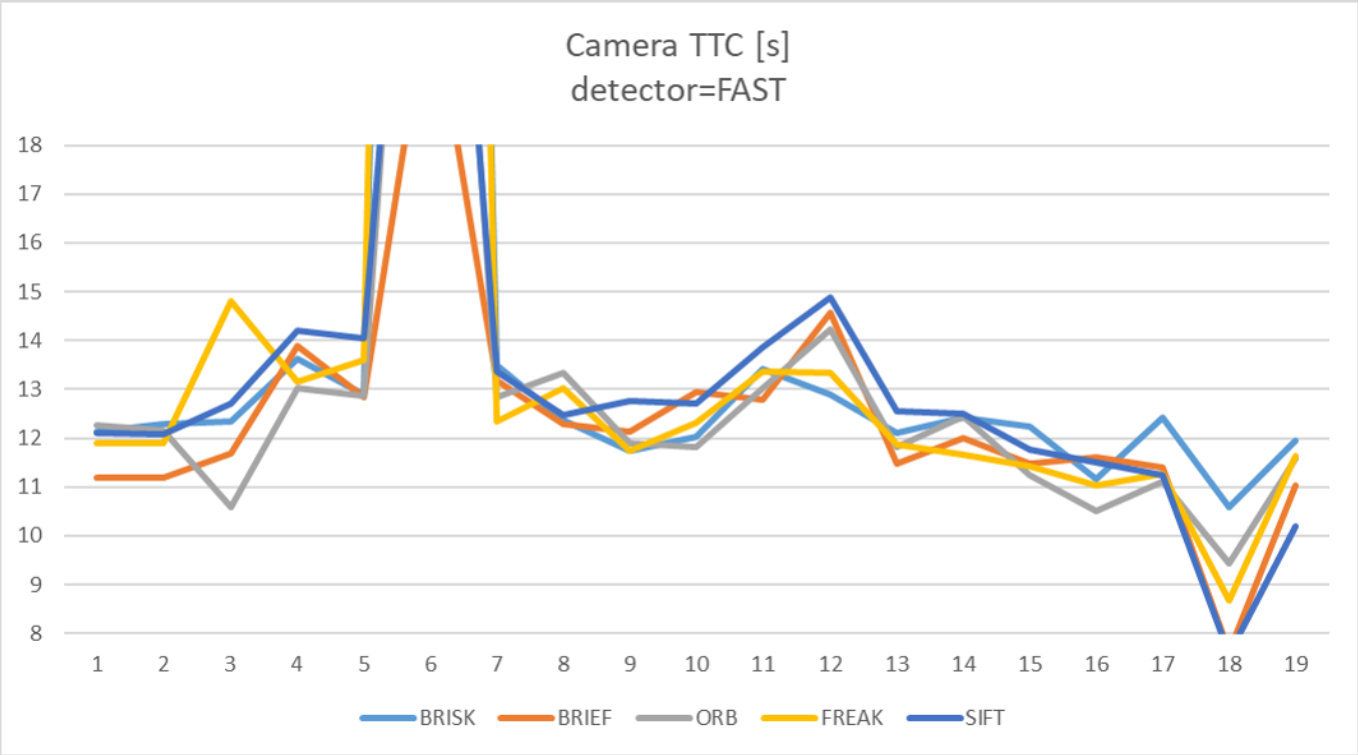
detector: HARRIS

frame#	BRISK	BRIEF	ORB	FREAK	SIFT
1	10.9082	10.9082	10.9082	9.74953	10.9082
2	10.9082	10.9082	10.9082	9.74953	10.9082
3	10.586	11.0081	11.0081	10.586	63.8475
4	26.2905	-11.4731	-11.4731	-10.295	-80.8525
5	11.7693	11.5792	11.5792	11.7693	11.5792
6	-inf	34.7543	19.4967	39.584	13.6432
7	12.9945	15.2483	13.6217	13.5907	27.8744
8	12.2792	14.2744	13.497	12.3379	13.497
9	12.9162	17.6204	17.6204	12.9162	17.6204
10	nan	3.30058	nan	nan	3.93864
11	-inf	-13.4405	-inf	186.357	-13.4405
12	11.2142	11.7414	11.7414	11.7414	11.7414
13	11.6948	11.6948	11.6948	11.6948	11.6948
14	25.6953	44.327	20.645	44.327	20.645
15	5.6061	5.6061	5.66097	12.288	5.66097
16	-13.6263	-14.7808	-14.7808	-25.2781	-14.7808

frame#	BRISK	BRIEF	ORB	FREAK	SIFT
17	6.19345	7.29175	6.71705	6.98094	6.71705
18	12.5848	12.5848	12.5848	nan	12.5848
19	-inf	-inf	-inf	nan	-inf

detector: FAST

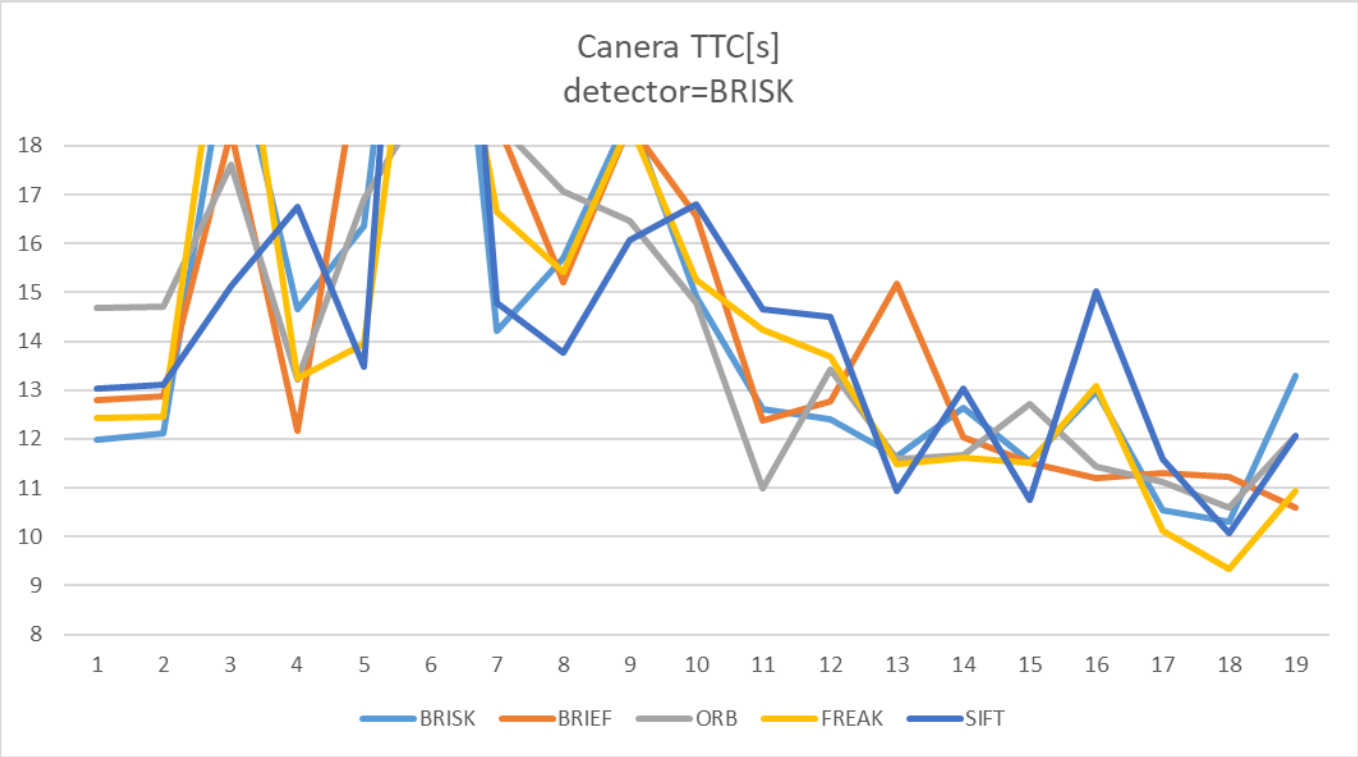
frame#	BRISK	BRIEF	ORB	FREAK	SIFT
1	12.1352	11.1897	12.2505	11.9072	12.1008
2	12.3	11.1897	12.1538	11.9004	12.0672
3	12.3453	11.682	10.5966	14.815	12.7155
4	13.6189	13.8858	13.0225	13.1475	14.2092
5	12.8853	12.8498	12.8595	13.6117	14.0316
6	68.6242	21.3581	32.168	74.6061	30.6117
7	13.4935	13.1897	12.8447	12.3292	13.3691
8	12.3693	12.3001	13.327	13.0141	12.4642
9	11.7285	12.1448	11.9077	11.7351	12.753
10	12.0205	12.9544	11.8145	12.319	12.7138
11	13.4169	12.7943	13.0149	13.3607	13.8688
12	12.8854	14.558	14.2159	13.3333	14.8887
13	12.1142	11.4828	11.8219	11.862	12.5506
14	12.4129	12.0086	12.4364	11.6599	12.5109
15	12.239	11.488	11.2534	11.42	11.7777
16	11.1574	11.6077	10.5046	11.0296	11.5035
17	12.4091	11.3897	11.1033	11.2643	11.2304
18	10.5959	7.63552	9.42491	8.66857	7.57429
19	11.9602	11.0196	11.5897	11.6293	10.1921



detector: BRISK

frame#	BRISK	BRIEF	ORB	FREAK	SIFT
1	11.9882	12.7898	14.6839	12.4266	13.0279
2	12.1189	12.8798	14.7184	12.4573	13.1022
3	20.4706	18.2904	17.6032	22.3054	15.138
4	14.6503	12.1663	13.22	13.2383	16.7583
5	16.3706	20.3251	16.8995	13.9445	13.487
6	27.6679	19.911	19.1438	24.0315	31.8721
7	14.2061	18.5156	18.4688	16.637	14.7952
8	15.7091	15.1935	17.0574	15.4191	13.7548
9	18.656	18.4226	16.4516	18.4506	16.0673
10	14.9058	16.5574	14.7812	15.2526	16.7991
11	12.6185	12.385	10.9855	14.2428	14.6452
12	12.3895	12.7576	13.4277	13.6828	14.4889
13	11.6513	15.1675	11.6015	11.4963	10.9353
14	12.6469	12.0461	11.6599	11.6151	13.0322
15	11.5469	11.5107	12.7097	11.512	10.7575
16	12.9792	11.1943	11.4331	13.0798	15.0221

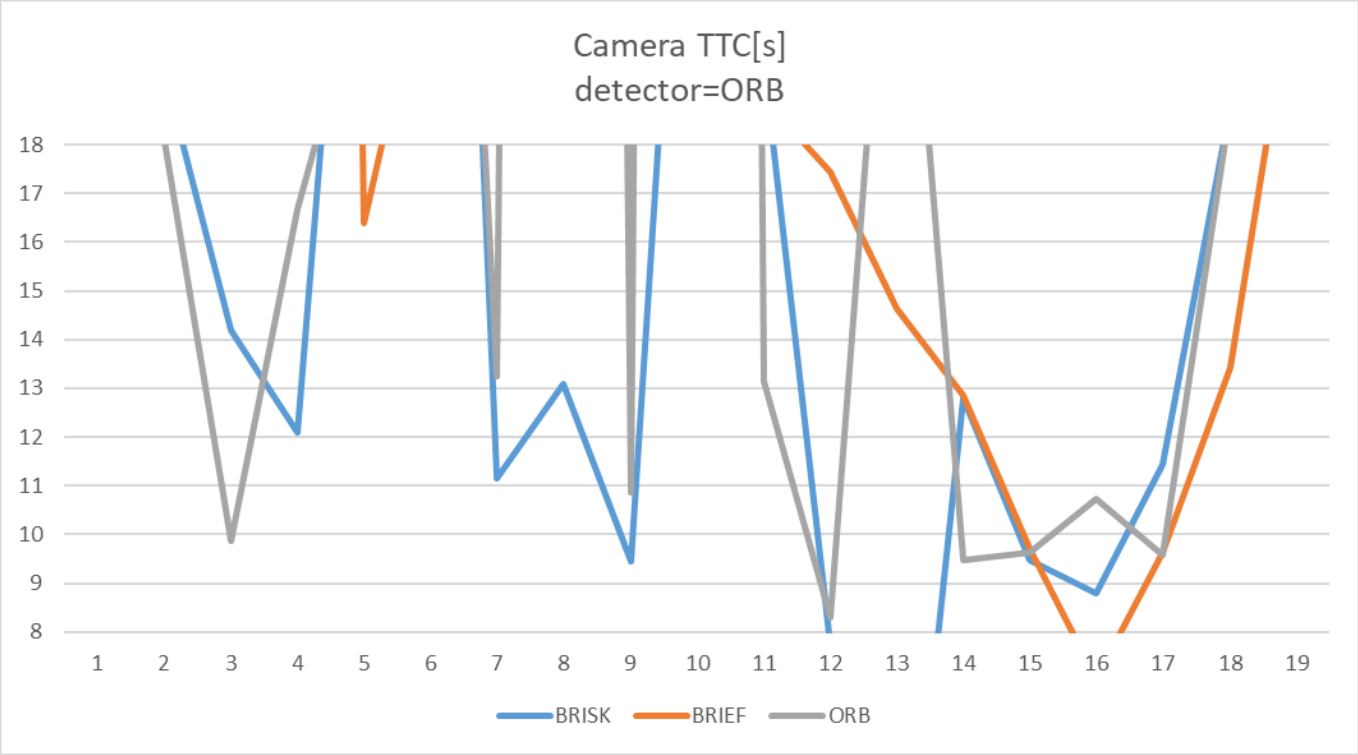
frame#	BRISK	BRIEF	ORB	FREAK	SIFT
17	10.5548	11.3149	11.123	10.1356	11.583
18	10.2987	11.2201	10.5872	9.32818	10.0638
19	13.2817	10.6055	12.0654	10.9372	12.0584



detector: ORB

frame#	BRISK	BRIEF	ORB	FREAK	SIFT
1	19.4375	21.3418	18.0656	12.1791	13.3163
2	19.4626	22.07	18.0656	11.9371	13.3282
3	14.1936	40.9596	9.87733	20.1206	11.8853
4	12.0855	103.665	16.6755	12.2284	12.3578
5	28.8362	16.395	21.591	10.8706	47.5977
6	43.147	22.3407	39.391	743.244	965.042
7	11.1609	18.3567	13.2591	11.5438	20.7949
8	13.0845	37.3205	145.778	-inf	-inf
9	9.45873	34.6298	10.8546	9.02151	10.4477
10	30.3425	188.579	179.41	13.3112	21.2278
11	19.6544	18.9258	13.1476	-545.044	12.7501
12	7.71005	17.4453	8.28796	7.24512	9.2983

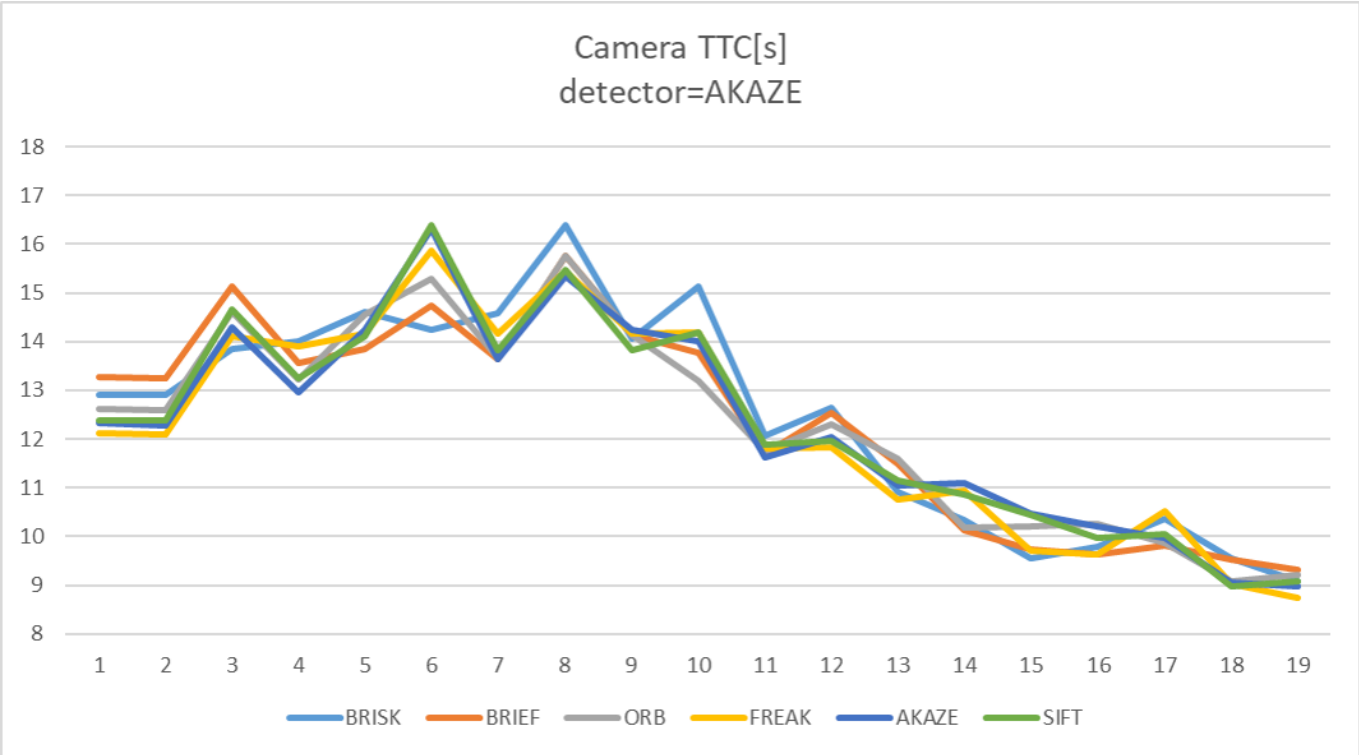
frame#	BRISK	BRIEF	ORB	FREAK	SIFT
13	-inf	14.6478	26.0409	40.8841	-inf
14	12.8	12.8535	9.46076	6.57691	8.33753
15	9.48525	9.71308	9.63066	60.2671	13.7643
16	8.79792	7.04396	10.724	8.44355	21.5257
17	11.4315	9.64141	9.5731	7.71824	8.62491
18	18.7733	13.442	18.7821	16.7528	15.7683
19	33.15	21.9022	43.2437	8.73235	7.01314



detector: AKAZE

frame#	BRISK	BRIEF	ORB	FREAK	SIFT
1	19.4375	21.3418	18.0656	12.1791	13.3163
2	19.4626	22.07	18.0656	11.9371	13.3282
3	14.1936	40.9596	9.87733	20.1206	11.8853
4	12.0855	103.665	16.6755	12.2284	12.3578
5	28.8362	16.395	21.591	10.8706	47.5977
6	43.147	22.3407	39.391	743.244	965.042
7	11.1609	18.3567	13.2591	11.5438	20.7949
8	13.0845	37.3205	145.778	-inf	-inf

frame#	BRISK	BRIEF	ORB	FREAK	SIFT
9	9.45873	34.6298	10.8546	9.02151	10.4477
10	30.3425	188.579	179.41	13.3112	21.2278
11	19.6544	18.9258	13.1476	-545.044	12.7501
12	7.71005	17.4453	8.28796	7.24512	9.2983
13	-inf	14.6478	26.0409	40.8841	-inf
14	12.8	12.8535	9.46076	6.57691	8.33753
15	9.48525	9.71308	9.63066	60.2671	13.7643
16	8.79792	7.04396	10.724	8.44355	21.5257
17	11.4315	9.64141	9.5731	7.71824	8.62491
18	18.7733	13.442	18.7821	16.7528	15.7683
19	33.15	21.9022	43.2437	8.73235	7.01314



detector: SIFT

frame#	BRISK	BRIEF	FREAK	SIFT
1	11.7937	12.2492	11.2639	11.5486
2	11.7937	12.1161	11.2996	11.638
3	13.3913	12.8707	13.3701	12.6771
4	14.32	15.0877	13.5417	12.6772

frame#	BRISK	BRIEF	FREAK	SIFT
5	20.5542	20.8978	19.932	19.0674
6	15.734	13.8872	13.7049	12.7759
7	10.7762	11.9657	12.0424	11.2969
8	15.6798	14.6563	14.7184	13.4442
9	15.1504	15.4846	12.5378	13.8641
10	13.4009	12.2307	13.8269	13.2061
11	11.9965	10.3096	11.3857	10.838
12	12.9654	12.991	12.0832	11.0478
13	10.1869	10.014	10.0939	11.3228
14	10.0041	10.282	10.0359	9.29176
15	9.15168	9.68692	9.83924	10.2517
16	9.53867	10.3044	9.26398	10.2197
17	9.49771	8.67121	9.70694	9.22835
18	9.49963	8.8478	9.21557	8.68319
19	9.37958	8.74495	9.4102	8.79626