FP.1 Match 3D Objects:

The function takes in the previous and current data frames and provides a mapping between identical bounding boxes in the current and previous frame.

Loop All Keypoint Matches and create a multimap with BoundingBox Ids pairs from previous and current data frame.

```
void matchBoundingBoxes(std::vector<cv::DMatch> &matches, std::map<int,
{

    multimap<int, int> bbMatches;
    int prevBoxID = -1;
    int currBoxID = -1;

    for(auto match: matches)
    {
        cv::KeyPoint prevKP = prevFrame.keypoints[match.queryIdx];
        cv::KeyPoint currKP = currFrame.keypoints[match.trainIdx];

        //Finding BoxID for previos frame
        for(auto boundingBox: prevFrame.boundingBoxes)
        {
            if(boundingBox.roi.contains(prevKP.pt))
            {
                prevBoxID = boundingBox.boxID;
                break;
            }
        }

        //Finding BoxID for crrent frame
        for(auto boundingBox: currFrame.boundingBoxes)
        {

            if(boundingBox.roi.contains(currKP.pt))
            {
                currBoxID = boundingBox.boxID;

                break;
            }
        }

        bbMatches.insert(pair<int, int>(currBoxID,prevBoxID));

    }// End of Loop
```

Extract Keypoints from KeyMatches in current and Previous frame.

If the keypoint is in the bounding box. Then save the Box ID and associate with current and previous bounding box

Update a multimap with boundingboxID pairs

Loop through the Multimap and return the Keypair with highest correspondence and ignore the rest.

(The situations when no bounding box was associated to a Keypoint is filtered by using its initial value of –1)

```
//Logic for extracting best matches


for(auto boundingBox: currFrame.boundingBoxes)
{
    auto it =  bbMatches.equal_range(boundingBox.boxID);
    int maxIndex;
    vector<int> count(prevFrame.boundingBoxes.size(),0);
    for(auto itr = it.first;itr !=it.second;itr++)
    {       if(itr->second!= -1) //When no BoxID was updated, defaut value  =-1
                count.at(itr->second) +=1;

    }
    maxIndex = distance(count.begin(), max_element(count.begin(),count.end()));
    bbBestMatches.insert(pair<int,int>(maxIndex, boundingBox.boxID));
}
```

FP.2 Compute Lidar-based TTC:

The time to collision is calculated by using the equations for a constant velocity model. The Lidar points are filtered corresponding to the bounding box in the ego line by using the lane width and the ROI obtained from YOLO object classification model. Some erroneous points are observed very close to the boot of the vehicle in the top view, these are filtered out using the mean of the points. A Euclidean clustering using a KD Tree can be also implemented but since the points are already smoothened using a ROI, taking the mean is sufficient.

```cpp
void computeTTCLidar(std::vector<LidarPoint> &lidarPointsPrev,
                     std::vector<LidarPoint> &lidarPointsCurr, double frameRate, double &TTC)
{
    // auxiliary variables
    double dT = 1/frameRate;        // time between two measurements in seconds
    double laneWidth = 4.0;         // assumed width of the ego lane
    double xMeanPrev = 0;
    double xMeanCurr = 0;
    size_t countCurr = 0;
    size_t countPrev = 0;

    // find closest distance to Lidar points within ego lane
    double minXPrev = 1e9, minXCurr = 1e9;
    for (auto it = lidarPointsPrev.begin(); it != lidarPointsPrev.end(); ++it)
    {

      if(abs(it->y) < laneWidth/2 )
      {
        xMeanPrev += it->x;
        countPrev++;
      }
    }

    xMeanPrev /= countPrev;

    for (auto it = lidarPointsCurr.begin(); it != lidarPointsCurr.end(); ++it)
    {
      if(abs(it->y) < laneWidth/2 )
      {
        xMeanCurr += it->x;
        countCurr++;
      }
    }

    xMeanCurr /= countCurr;

    // compute TTC from both measurements
    TTC = xMeanCurr * dT / (xMeanPrev - xMeanCurr);

}
```

FP.3 Associate Keypoint Correspondences with Bounding Boxes

The keypoint matches need to be assigned to each bounding box before TTC calculation using the camera. As in the case of Lidar, incorrected Keypoint matches will result in erroneous results. Hence all such points are removed whose Euclidean distance between the KeyPoint match in the current and the previous frame is greater than the average of distance between all keypoint matches.

```
// associate a given bounding box with the keypoints it contains
void clusterKptMatchesWithROI(BoundingBox &boundingBox, std::vector<cv::KeyPoint> &kptsPrev, std::vector<cv::KeyPoint> &kptsCurr,
{
space/SFND_3D_Object_Tracking/build
    double avgDistance = 0;
    std::vector<cv::DMatch> matchesROI; //Temporary buffer to avoid iterating whole points again.

    for(auto match: kptMatches)
    {
        if(boundingBox.roi.contains(kptsCurr[match.trainIdx].pt))
        {
            avgDistance += match.distance;
            numMatchesROI++;
            matchesROI.push_back(match);
        }
    }

    avgDistance /= numMatchesROI;

    for(auto match: matchesROI)
    {
        if(boundingBox.roi.contains(kptsCurr[match.trainIdx].pt) && match.distance < avgDistance)
            boundingBox.kptMatches.push_back(match);
    }

}
```

FP.4 Compute Camera-based TTC:

The images are already transformed using the Intrinsic and Extrinsic matrices provided from KITI setup. For this the median of the distance ratios between similar KeyPoint's in the current and previous frame is used.

```
// Compute time-to-collision (TTC) based on keypoint correspondences in successive images
void computeTTCCamera(std::vector<cv::KeyPoint> &kptsPrev, std::vector<cv::KeyPoint> &kptsCurr,
                      std::vector<cv::DMatch> kptMatches, double frameRate, double &TTC, cv::Mat *visImg)
{
    // compute distance ratios between all matched keypoints
    vector<double> distRatios; // stores the distance ratios for all keypoints between curr. and prev. frame
    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 = 50.0; // min. required distance
            double maxDist = 150.0;

            // 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 && distCurr <= maxDist)
            { // 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

    // only continue if list of distance ratios is not empty
    if (distRatios.size() == 0)
    {
        TTC = NAN;
        return;
    }
```

While calculating the median value the size of the vector if it is odd or even needs to be considered.

```
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 t

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

FP.5 Performance Evaluation 1

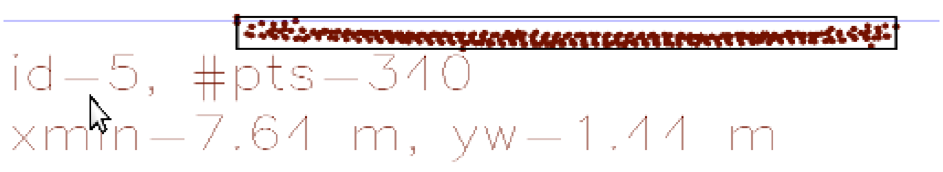
	distance Top View	TTC MANUAL	TTC Lidar	TTC Camera
1	7.97			
2	7.91	13.18333333	12.289	13.666
3	7.85	13.08333333	13.354	10.604
4	7.79	12.98333333	16.384	15.494
5	7.68	6.981818182	14.076	12.075
6	7.61	10.87142857	12.729	13.553
7	7.58	25.26666667	13.751	18.933
8	7.55	25.16666667	13.731	12.21
9	7.47	9.3375	13.79	13.052
10	7.13	2.097058824	12.058	10.366

Many a time's the TTC calculated has slight deviations. These can be attributed to multiple factors.

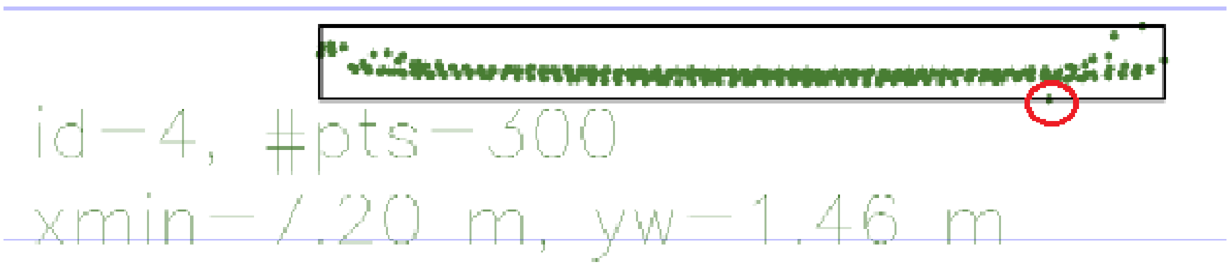
1. Lidar detects some points that are not part of the object we are tracking. This can be due to fact that it can influenced by environment factors and ghost points because of multiple reflection from other surface.
2. Sensor Noise.
3. Our assumption that the vehicle is a constant velocity model.

Instances:

1. Some noise in measurement leading to the lidar points being more scattered.



2. Some ghost points detected 10-20 cm ahead of the object we are tracking



FP.6 Performance Evaluation 2

SHITOMASI-BRISK and FAST-ORB and AKAZE BRIEF pairs give good results in term of accuracy.

TTC calculation for different detector descriptor combination is shown below.

Color code GREEN(10 – 15 seconds), YELLOW (15-20 and 5-10), RED (>20 and <5 seconds)

TTC Lidar	SHITOMASI BRISK (BF and I	SHITOMASI BRISK (FLANN and KNN)
12.289	13.666	12.949
13.354	10.604	12.998
16.384	15.494	12.818
14.076	12.075	X
12.729	13.553	12.355
13.751	18.933	13.209
13.731	12.21	12.461
13.79	13.052	10.43
12.058	10.366	10.01

DETECTOR	DESCRIPTORS	IMG1	IMG2	IMG3	IMG4	IMG5	IMG6	IMG7	IMG8	IMG9	IMG10
LIDAR			12.289	13.354	16.384	14.076	12.729	13.751	13.731	13.79	12.058
	BRIEF		8.813	20.39	X	X	12.879	13.871	12.337	X	7.159
	ORB		X	11.008	22.946	12.485	13.369	6.969	12.279	9.024	9.272
	FREAK		X	13.828	X	X	X	5.515	X	X	7.791
	AKAZE										
HARRIS	SIFT		3.695	11.008	58.284	X	37.38	X	14.274	12.916	8.39
	BRIEF		11.198	9.257	13.981	50.855	17.231	X	X	10.937	13.294
	ORB		12.085	12.115	11.374	13.27	X	13.33	14.174	10.715	24.889
	FREAK		11.074	X	12.803	X	8.199	12.148	10.223	11.398	X
	AKAZE										
FAST	SIFT		12.559	12.397	13.803	X	879.6	14.23	11.276	10.276	13.027
	BRIEF		15.827	12.168	14.387	15.642	13.011	12.109	18.745	16.557	12.157
	ORB		11.526	18.93	X	X	X	18.505	13.981	15.535	16.471
	FREAK		13.169	20.125	11.197	15.9	25.077	13.109	16.139	17.299	16.785
	AKAZE										
BRISK	SIFT		17.865	16.679	16.399	X	37.627	16.049	15.603	16.322	16.444
	BRIEF		9.296	16.128	12.992	X	13.035	X	20.225	73.18	14.315
	ORB		29.075	X	12.751	X	10.2	X	X	X	X
	FREAK		9.335	52.875	12.155	41.879	X	X	11.14	18.057	9.35
	AKAZE										
ORB	SIFT		9.963	17.796	10.765	X	25.508	X	X	X	X
	BRIEF		12.289	13.354	12.413	X	13.549	14.077	15.388	15.906	13.805
	ORB		10.931	14.339	12.97	12.733	13.379	12.665	17.36	14.282	11.807
	FREAK		11.866	13.618	13.385	14.508	14.726	16.252	17.336	12.107	14.826
	AKAZE		12.453	13.915	13.038	X	15.969	14.087	16.244	13.292	14.604
AKAZE	SIFT		12.447	13.837	12.757	X	14.969	13.773	16.057	13.711	14.964
	BRIEF		11.263	15.141	12.362	X	16.528	X	13.117	15.562	13.089
	ORB										
	FREAK		10.479	13.476	13.108	16.831	12.954	X	18.749	14.532	18.579
	AKAZE										
SIFT	SIFT		10.558	13.169	12.819	X	14.212	12.194	14.337	13.926	13.2