TASK FP.1 Match 3D Objects

The matchBoudningBoxes function implemented through all the input matches and checking for each previous and current bounding box region of interest, the matched keypoints that are enclosed by that region of interest and then store the bounding boxes Ids pair. Then check the highest number of occurrences for each pair and add it to the bbBestMatches map.
highest number of keypoint correspondences.

This implementation can be shown as follows:

```
void matchBoundingBoxes(std::vector<cv::DMatch> &matches, std::map<int,
int> &bbBestMatches, DataFrame &prevFrame, DataFrame &currFrame)
{
   int prevbbSize = prevFrame.boundingBoxes.size();
   int currbbSize = currFrame.boundingBoxes.size();
   int mapbbIds[prevbbSize][currbbSize] = {};

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

       // get current keypoint and its matched partner in the prev. frame
       cv::KeyPoint kpCurr = currFrame.keypoints[it1->trainIdx];
       cv::KeyPoint kpPrev = prevFrame.keypoints[it1->queryIdx];

       vector<int> prevbbMatchIds, currbbMatchIds;

       // For each prev, curr boudning box check it the keypoints are
enclosed by the bounding box and store the box id

       for(int i= 0;i<currbbSize;i++)
       {
           if(currFrame.boundingBoxes[i].roi.contains(kpCurr.pt))
               currbbMatchIds.push_back(currFrame.boundingBoxes[i].boxID);
       }

       for(int j= 0;j<prevbbSize;j++)
       {
           if(prevFrame.boundingBoxes[j].roi.contains(kpPrev.pt))
```

```
                prevbbMatchIds.push_back(prevFrame.boundingBoxes[j].boxID);
        }

        // count the matched prev and curr boxIds
        for(int prevbbId=0;prevbbId<prevbbMatchIds.size(); prevbbId++)
        {
            for(int currbbId=0;currbbId<currbbMatchIds.size(); currbbId++)
            {

mapbbIds[prevbbMatchIds[prevbbId]][currbbMatchIds[currbbId]] += 1;
            }
        }
    }

    // Fill the bbBsetMatches map with the highest number of occurrences
between the prev. and curr. boudning box
    for(int i=0;i<prevbbSize;i++)
    {
        int maxMatch=0;
        int bestId=0;
        for(int j=0;j<currbbSize;j++)
        {
            if(mapbbIds[i][j] > maxMatch)
            {
                maxMatch = mapbbIds[i][j];
                bestId = j;
            }
        }
        bbBestMatches[i]=bestId;
    }

}
```

TASK FP.2 Compute Lidar-based TTC

Compute the time-to-collision based on the Lidar measurements from the matched bounding boxes between current and previous frame. The TTC measurements are based on the model of a constant-velocity.
To deal with outlier Lidar points in a way to avoid severe estimation errors, I tried both mean and median calculation to get the values of minXPrev and minXCurr. The median calculation shows slightly better performance than the mean one.
The implementation of the function can be shown as follows:

```
void computeTTCLidar(std::vector<LidarPoint> &lidarPointsPrev,
                     std::vector<LidarPoint> &lidarPointsCurr, double
frameRate, double &TTC)
{
   // auxiliary variables
   double dT = 0.1;        // time between two measurements in seconds

   // find closest distance to Lidar points within ego lane
   double minXPrev = 1e9, minXCurr = 1e9;
   vector<double> prevXPoints, currXPoints;
   for (auto it = lidarPointsPrev.begin(); it != lidarPointsPrev.end();
++it)
   {
       prevXPoints.push_back(it->x);
   }

   for (auto it = lidarPointsCurr.begin(); it != lidarPointsCurr.end();
++it)
   {
       currXPoints.push_back(it->x);
   }

   // Calculate the minXPrev and minXCurr using the median calculation of
the prevXPoints and currXPoints
  /*  std::sort(prevXPoints.begin(), prevXPoints.end());
   long medIndex = floor(prevXPoints.size() / 2.0);
```

```
    minXPrev = prevXPoints.size() % 2 == 0 ? (prevXPoints[medIndex - 1] +
prevXPoints[medIndex]) / 2.0 : prevXPoints[medIndex]; // compute median
prevXPoint remove outlier influence

    std::sort(currXPoints.begin(), currXPoints.end());
    medIndex = floor(currXPoints.size() / 2.0);
    minXCurr = currXPoints.size() % 2 == 0 ? (currXPoints[medIndex - 1] +
currXPoints[medIndex]) / 2.0 : currXPoints[medIndex]; // compute median
currXPoint to remove outlier influence
*/

    // Calculate the minXPrev and minXCurr using the mean calculation of
the prevXPoints and currXPoints
    double meanDist=0;
    for(int i=0; i<currXPoints.size();i++)
    {
        meanDist+=currXPoints[i];
    }
    minXCurr = meanDist / currXPoints.size();
    meanDist=0;
    for(int i=0; i<prevXPoints.size();i++)
    {
        meanDist+=prevXPoints[i];
    }
    minXPrev = meanDist/prevXPoints.size();

    // compute TTC from the median minX measurements
    TTC = minXCurr * dT / (minXPrev - minXCurr);
}
```

TASK FP.3 Associate Keypoint Correspondences with Bounding Boxes

The keypoints correspondences with a certain bounding box is implemented by looping through the keypoints matches and check if the corresponding matched current keypoint is enclosed by the bounding box region of interest then add it to an initial list of matches (kptMatchesRoi). After preparing this initial list of matches, calculate the mean distance between the current and previous keypoints associated with the prepared list of matches. Then filter outlier matches based on this mean distance by removing all the distances below a predefined threshold (0.75) of the mean distance, then add the keypoint matches correspondences to the "kptMatches" property of the respective bounding box.

The implementation can be shown as follows:

```
void clusterKptMatchesWithROI(BoundingBox &boundingBox,
std::vector<cv::KeyPoint> &kptsPrev, std::vector<cv::KeyPoint> &kptsCurr,
std::vector<cv::DMatch> &kptMatches)
{
   std::vector<cv::DMatch> kptMatchesRoi;
   for(auto it=kptMatches.begin(); it!=kptMatches.end(); ++it)
   {
       // get current keypoint
       cv::KeyPoint kpCurr = kptsCurr.at(it->trainIdx);

       // check if the current keypoint is enclosed by the bounding box
       if(boundingBox.roi.contains(kpCurr.pt))
               kptMatchesRoi.push_back(*it);
   }

   // Calculate the mean distance among all the kptsMatches inside the
bounding box
   double meanDist= 0;
   double threshold = 0.75;
   for (auto it1 = kptMatchesRoi.begin(); it1 != kptMatchesRoi.end();
++it1)
   {
       // get current keypoint and its matched partner in the prev. frame
       cv::KeyPoint kpCurr = kptsCurr.at(it1->trainIdx);
       cv::KeyPoint kpPrev = kptsPrev.at(it1->queryIdx);

       meanDist += cv::norm(kpCurr.pt-kpPrev.pt);
   }

   meanDist = meanDist / kptMatchesRoi.size();
```

```
   // Filter outlier matches based on the distance by removing all the
distances below a predefined threshold (0.75)
   for (auto it1 = kptMatchesRoi.begin(); it1 != kptMatchesRoi.end();
++it1)
   {
       // get current keypoint and its matched partner in the prev. frame
       cv::KeyPoint kpCurr = kptsCurr.at(it1->trainIdx);
       cv::KeyPoint kpPrev = kptsPrev.at(it1->queryIdx);

       double distance =   cv::norm(kpCurr.pt-kpPrev.pt);
       if(distance < meanDist * 0.75)
       {
           boundingBox.keypoints.push_back(kpCurr);
           boundingBox.kptMatches.push_back(*it1);
       }
   }

}
```

TASK FP.4 Compute Camera-based TTC

Compute the time-to-collision using only keypoint correspondences from the matched bounding boxes between current and previous frame based on the constant velocity model. This is done by calculating the ratio of all relative distances (prev and curr) at each keypoint match and taking the median for all distance ratios to deal with outlier correspondences in a way to avoid severe estimation errors.

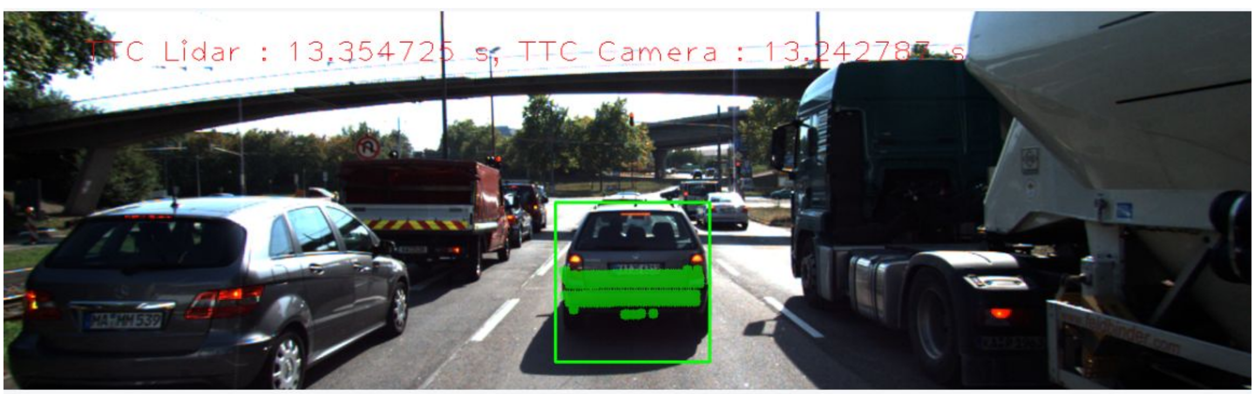
The implementation can be shown as follows:

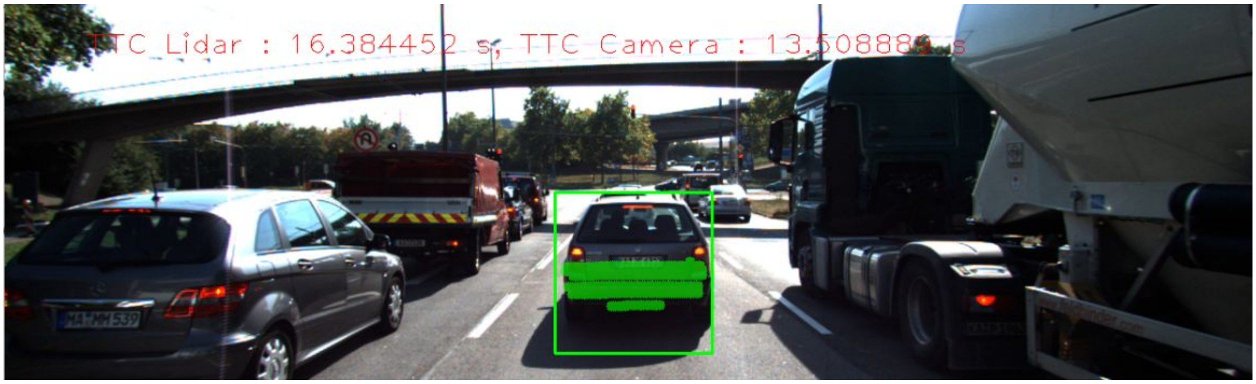
```
void computeTTCCamera(std::vector<cv::KeyPoint> &kptsPrev,
std::vector<cv::KeyPoint> &kptsCurr,
                      std::vector<cv::DMatch> kptMatches, double frameRate,
double &TTC, cv::Mat *visImg)
{
   // compute distance ratios between all matched keypoints
   vector<double> distRatios; // stores the distance ratios for all
keypoints between curr. and prev. frame
   for (auto it1 = kptMatches.begin(); it1 != kptMatches.end() - 1; ++it1)
```

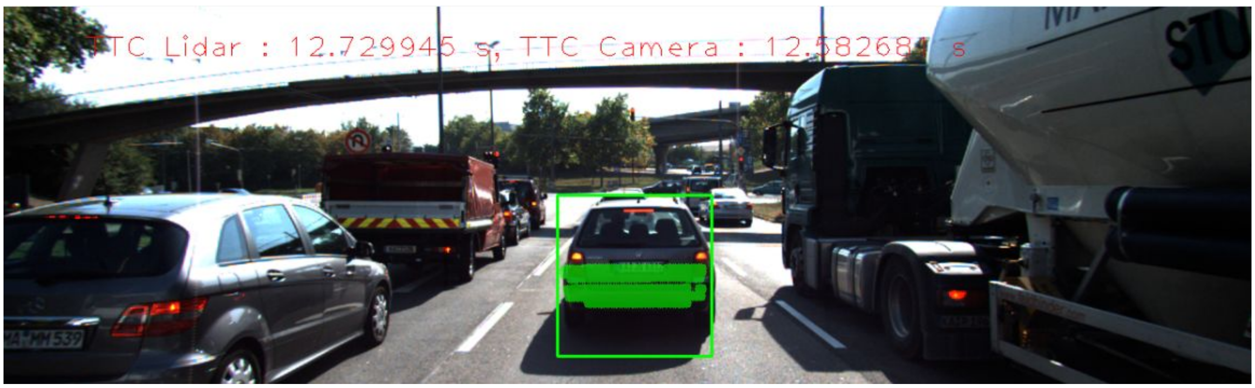
```
    { // outer kpt. loop

        // 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 = 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

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

```
    // STUDENT TASK (replacement for medianDistRatio)
    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

    double dT = 1 / frameRate;
    TTC = -dT / (1 - medDistRatio);
    // EOF STUDENT TASK
}
```

TASK FP.5 Performance evaluation 1
In general using the mean or median calculation of min distances in estimating the distance from the front vehicle based on Lidar measurements prevent a lot of estimation errors that would have arised if we just used only the min distance, but still it is observed in some sequence that the TTC based lidar goes from 13s directly to 16s and then after only one frame back to 12s and this 3s gap in my opinion is very critical when it comes to collision detection.
The following results could be seen as follows:
TASK.5.6: Image Number :2 ,TTC_Lidar ,13s, TTC_CAMERA,13s
TASK.5.6: Image Number :3 ,TTC_Lidar ,16s, TTC_CAMERA,13s
TASK.5.6: Image Number :4 ,TTC_Lidar ,14s, TTC_CAMERA,12s
TASK.5.6: Image Number :5 ,TTC_Lidar ,12s, TTC_CAMERA,12s







TASK FP.6 Performance Evaluation 2

I used the bash script that I made in the midterm project to call the ./3D_object_tracking executable with most of the combination of detectors and descriptors as follows:

```
cd build
declare -a detector=("SHITOMASI" "HARRIS" "FAST" "BRISK" "ORB")
declare -a descriptor=("BRISK" "BRIEF" "ORB" "FREAK" )
for i in "${detector[@]}"
do
  for j in "${descriptor[@]}"
  do
     ./3D_object_tracking "$i" "$j"
  done
done
```

The results of Lidar based TTC and Camera based TTC are stored into a file as follows:

```
   /* ofstream outfile;
   //outfile.open("../perfEvaluation.txt", fstream::app);
   outfile.open("../perfEvaluationTest.txt", fstream::app);
   outfile << "Detector Type: " << detectorType << " Descriptor Type: " <<
descriptorType << endl << endl;
```

```
    for(int i = 0; i< vttcLidar.size();i++)
    {
        outfile << "TASK.5.6: Image Number :" << i+1 << " ,TTC_Lidar ," <<
vttcLidar[i] << "s, TTC_CAMERA," << vttcCamera[i] << "s" << endl;
    }
    outfile << endl;
   outfile.close(); */
```

I tested this several times as follows:

- perfEvaluation.txt: contain the result of the combination of the Top3 detector/descriptor that are obtained in the midterm project (FAST/ORB, FAST/ BRIEF, ORB/BRIEF) by using the lidar based ttc based on the median calculation.
- perfEvaluationMean.txt: contain the result of the combination of the Top3 detector/descriptor that are obtained in the midterm project (FAST/ORB, FAST/ BRIEF, ORB/BRIEF) by using the lidar based ttc based on the mean calculation.
- perfEvaluationTest.txt: contian the result of the combination of the most detector/descriptor used in the project.

```
declare -a detector=("SHITOMASI" "HARRIS" "FAST" "BRISK" "ORB")
declare -a descriptor=("BRISK" "BRIEF" "ORB" "FREAK" )
```

Conclusion: It can be observed that SHITOMASI and FAST detectors are among the best results as the TTC measurements are consistent along the 18 image sequence. On the other hand HARRIS and ORB shows results that are not acceptable at all such as negative TTC values or too large TTC values and in this case the calculation is based no more on the front vehicle but either on the right vehicle or on too far object.

The perfEvaluation comparison results can be found in the SFND_3D_Object_Tracking_PerfEvaluation.xlsx sheet.