FP.1 Match 3D objects

matchBoundingBoxes function was implemented in SFND_3D_Object_Tracking-master> camFusion_Student.cpp lines 300 – 352, you can find the code below. I have used the concept of outer-looping all the matched key-points as suggested. Then finding the corresponding train and query point from the current and previous frames. Then looped over the bounding boxes from current and previous frame to verify in which Bounding box the match point belongs. Then counted the corresponding box pair in order to find out the best match with maximum number of key-point correspondence.

```
void matchBoundingBoxes(std::vector<cv::DMatch> &matches, std::map<int, int> &bbBestMatches, DataFrame &prevFrame,
DataFrame &currFrame)
{

    const int size_p = prevFrame.boundingBoxes.size();
    const int size_c = currFrame.boundingBoxes.size();
    //int count[size_p][size_c] = {};//initialize a null matrix with all values "0" of bounding boxes size prev x curr
    cv::Mat count = cv::Mat::zeros(size_p, size_c, CV_32S);
    for (auto matchpair : matches)
    {
        //take one matched keypoint at a time find the corresponsing point in current and prev frame
        //once done check to which bounding box in prev and curr frame the point belong too
        //once found store the value and increment the count
        //cv::KeyPoint kpOuterPrev = kptsPrev.at(it1->queryIdx);
        cv::KeyPoint prevkp1 = prevFrame.keypoints.at(matchpair.queryIdx);
        auto prevkp = prevkp1.pt;//previous frame take keypint

        cv::KeyPoint currkp1 = currFrame.keypoints.at(matchpair.trainIdx);
        auto currkp = currkp1.pt;//current frame take keypint

        for (size_t prevbb = 0; prevbb < size_p; prevbb++)//loop through all the prev frame bb
        {
            if (prevFrame.boundingBoxes[prevbb].roi.contains(prevkp))//check if the "previous frame take keypint" belongs to this
box
            {//if it does
                for (size_t currbb = 0; currbb < size_c; currbb++)//loop thrpugh all the curr frame bb
                {
                    if (currFrame.boundingBoxes[currbb].roi.contains(currkp))//check if the "current frame take keypint" belongs to
this box
                    {//if it does
                        //count[prevbb][currbb] = count[prevbb][currbb] + 1;//do a +1 if match is found
                        count.at<int>(prevbb, currbb) = count.at<int>(prevbb, currbb) + 1;
                    }
                }
            }
        }
    }
    //for each prev bb find and compare the max count of corresponding curr bb.
    //the curr bb with max no. of matches (max count) is the bbestmatch

        for (size_t i = 0; i < size_p; i++)//loop through prev bounding box
        {
            int id = -1;//initialize id as the matrix starts from 0 x 0 we do not want to take 0 as the initializing value
            int maxvalue = 0;//initialize max value
```

```
        for (size_t j = 0; j < size_c; j++)//loop through all curr bounding boxes to see which prev + curr bb pair has maximum
count
        {
          if (count.at<int>(i,j) > maxvalue)
          {
            maxvalue = count.at<int>(i,j);//input value for comparison
            id = j;//id
          }

        }
        bbBestMatches[i] = id;//once found for 1 prev bounding box; input the matched pair in bbBestMatches
        //bbBestMatches.insert({i, id});
      }
}
```

FP.2 Compute lidar-based TTC

computeTTCLidar function was implemented in SFND_3D_Object_Tracking-master> camFusion_Student.cpp lines 240-297, you can find the code below. Here I have used the concept of TTC determination using median method as taught in the course. I chose the median method to avoid(mitigate) the impact of outliers. Another function sorttheLidarpoints was used in order to help with sorting of the lidar points in forward direction (requirement for using median method). Finally, below formula was used to calculate TTC with lidar sensor.

*TTC = ((dt * minXcurr) / (minXprev - minXcurr));*

```
void sorttheLidarpoints(std::vector<LidarPoint> &lidarPoints)
{
  //sorting the lidarpoints only for value of x in ascending order
  std::sort(lidarPoints.begin(), lidarPoints.end(), [](LidarPoint a, LidarPoint b) {
    return a.x < b.x;
    });
}

void computeTTCLidar(std::vector<LidarPoint> &lidarPointsPrev,
                     std::vector<LidarPoint> &lidarPointsCurr, double frameRate, double &TTC)
{
  double dt = 1 / frameRate; //calculating time between 2 measurements
  //double lanewidth = 4.0;//assuming width of ego lane (already done in the main file)
  //double minXprev = 1e9, minXcurr = 1e9; //min distance to closet lidra point in prev and curr frames

  //remeber we only comparing x axis along the length in direction of the vehicle (ego lane)

  sorttheLidarpoints(lidarPointsPrev);//need to create a diff function to address the x component
  sorttheLidarpoints(lidarPointsCurr);

  auto index1 = (lidarPointsPrev.size()/2);//taking the median size prev
  auto index11 = ((lidarPointsPrev.size() - 1)/2);
  auto index2 = (lidarPointsCurr.size()/2);//taking the median size curr
  auto index22 = ((lidarPointsCurr.size() - 1)/2);


  double minXprev = lidarPointsPrev[index1].x;//odd
  double minXcurr = lidarPointsCurr[index2].x;//odd
```

```
    //calculate median values
    /*if (lidarPointsPrev.size() % 2 != 0)//check for even number of points
    {
       double minXprev = lidarPointsPrev[index1].x;//odd
    }
    else
    {
       double minXprev = (lidarPointsPrev[index1].x + lidarPointsPrev[index11].x) / 2.0;//even
    }


    if (lidarPointsCurr.size() % 2 != 0)//check for even number of points
    {
       double minXcurr = lidarPointsCurr[index2].x;//odd
    }
    else
    {
       double minXcurr = (lidarPointsCurr[index2].x + lidarPointsCurr[index22].x) / 2.0;//even
    }*/



    TTC = ((dt * minXcurr) / (minXprev - minXcurr));
    //TTC = -dt / (1 - medianDistRatio);

    cout << TTC << " time for lidar compute" <<endl;

}
```

FP.3 : Associate Keypoint Correspondences with Bounding Boxes

*clusterKptMatchesWithROI function was implemented in SFND_3D_Object_Tracking-master> camFusion_Student.cpp lines 134-174, you can find the code below. This task is required in order to find all key-point matches that belong to each 3D object. As suggested, I did an outer-loop of key-point matches and found associated key-points from each curr and prev key-points. Then checked if the current key-point belong to the BB Region of interest (BB we are calling the function for). Finally, in order to make the code robust and reduce (mitigate) outliers impact- as recommends and taught in the course I calculated mean distance and removed all the point which were beyond 2*mean distance.*

```
void clusterKptMatchesWithROI(BoundingBox &boundingBox, std::vector<cv::KeyPoint> &kptsPrev,
std::vector<cv::KeyPoint> &kptsCurr, std::vector<cv::DMatch> &kptMatches)
{
    std::vector<cv::DMatch> kptroi;

    for (auto it1 = kptMatches.begin(); it1 != kptMatches.end(); ++it1)
    {
        cv::KeyPoint kpcurr = kptsCurr.at(it1->trainIdx);
        auto kpcurr1 = kpcurr.pt;
        cv::KeyPoint kpprev = kptsPrev.at(it1->queryIdx);
        auto kpprev1 = kpprev.pt;
        if (boundingBox.roi.contains(kpcurr1))
```

```
        {
            kptroi.push_back(*it1);//all keypoints in the bounding boxes ROI- pushed back in kptroi
        }

    }

    //now removing outliers - whihc is done by calculating mean of all the matches
    double dist = 0.0;
    int m = 0;
    for (auto it2 = kptroi.begin(); it2 != kptroi.end(); ++it2)
    {
        cv::KeyPoint kpcurr2 = kptsCurr.at(it2->trainIdx);
        cv::KeyPoint kpprev2 = kptsPrev.at(it2->queryIdx);
        dist = cv::norm(kpcurr2.pt - kpprev2.pt) + dist;
        m++;
    }
    double mean = dist / m;

    //discarding the matches beyond 2* mean
    double threshold = mean * 2, dist1 = 0;
    for (auto it3 = kptroi.begin(); it3 != kptroi.end(); ++it3)
    {
        cv::KeyPoint kpcurr3 = kptsCurr.at(it3->trainIdx);
        cv::KeyPoint kpprev3 = kptsPrev.at(it3->queryIdx);
        dist1 = cv::norm(kpcurr3.pt - kpprev3.pt);
        if (dist1 < threshold)
            boundingBox.kptMatches.push_back(*it3);
    }

}
```

FP.4 Compute Camera-based TTC

computeTTCCamera function was implemented in SFND_3D_Object_Tracking-master> camFusion_Student.cpp lines 178-238, you can find the code below. In order to calculate TTC with camera the key-points belonging to each bounding box. Here as suggested I used the concept taught in the course and calculated TTC camera and used median method in order to mitigate impact of outliers. For determining TTC camera, used the below formula:

TTC = -dT / (1 - medianDistRatio);

```
// 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)

{

    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;
```

```
    }

    // compute camera-based TTC from distance ratios

    //double meanDistRatio = std::accumulate(distRatios.begin(), distRatios.end(), 0.0) / distRatios.size();

    //double dT = 1 / frameRate;

    //TTC = -dT / (1 - meanDistRatio);

    double medianDistRatio;

    std::sort(distRatios.begin(), distRatios.end());

    auto index = (distRatios.size() / 2);

    auto index1 = ((distRatios.size() - 1) / 2);

    if (distRatios.size() % 2 != 0)//check for even case

        medianDistRatio = distRatios[index];

    medianDistRatio = (distRatios[index1] + distRatios[index]) / 2;

    double dT = 1 / frameRate;

    TTC = -dT / (1 - medianDistRatio);

    cout << TTC << " time for camera compute"<< endl ;

}
```

FP.5 Performance evaluation 1

This section I looked at various TTC lidar based results. All the measurements were in the range of 8 -16s which was more or less similar to the TTC computation by using camera. This is because I tried to mitigate the impact of outliers by using the median approach. Task-005 document has more details regarding this observation.

FP.6 Performance evaluation 2

This section I ran various detector/descriptor combinations and compared the ttc computation results for lidar and camera. The document Task-006 provides the details and results of various combinations. Just to summarize- with FAST detector got the best results for both ttc lidar and ttc camera. However, with detector type like HARRIS and ORB didn’t get reliable value for ttc camera computations, also values for ttc lidar looked not that reasonable.