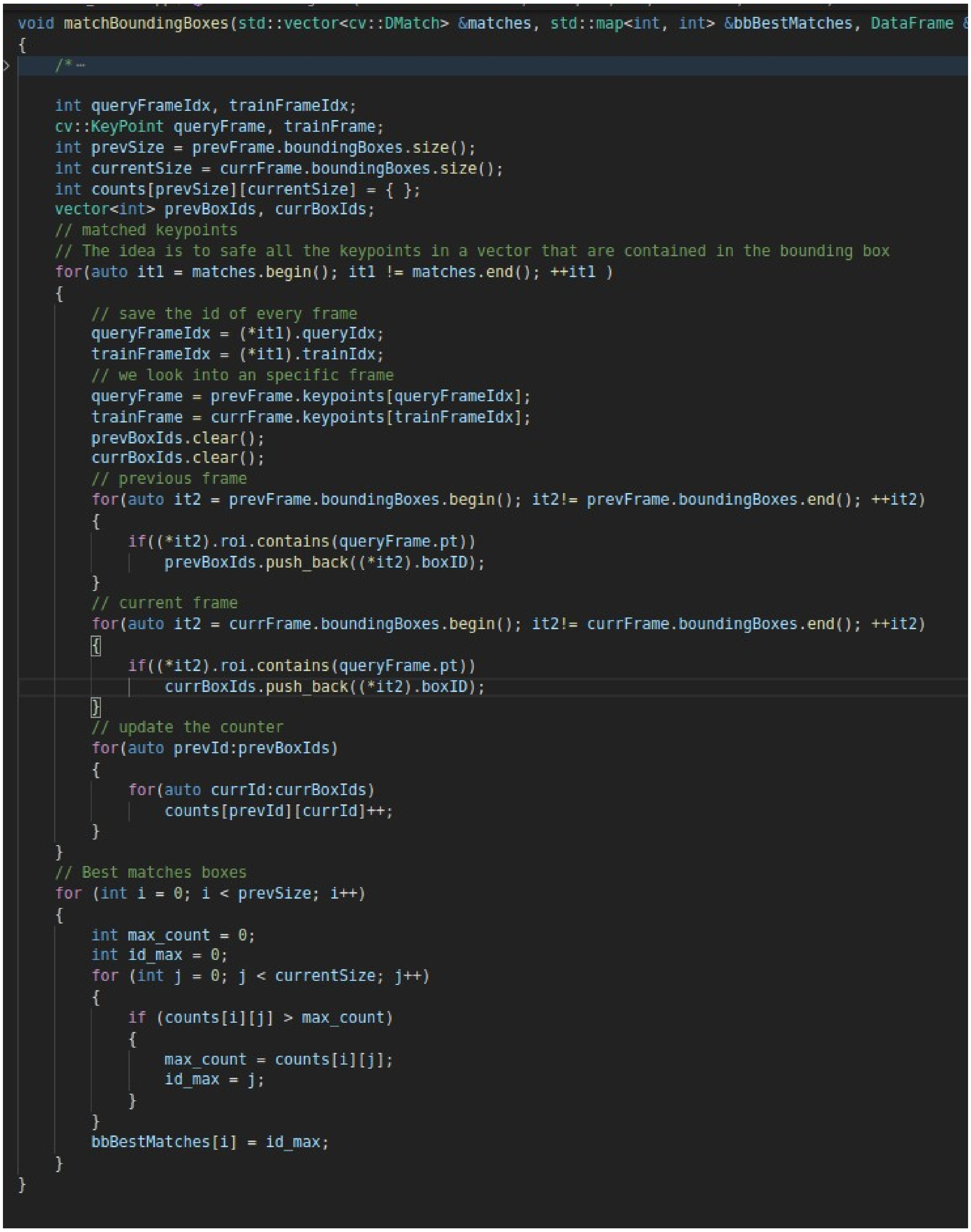
Track an Object in 3D space

Carlos E. Sanoja

FP.1 : Match 3D Objects

In this task, please implement the method "matchBoundingBoxes", which takes as input both the previous and the current data frames and provides as output the ids of the matched regions of interest (i.e. the boxID property)“. Matches must be the ones with the highest number of keypoint correspondences.

Solution: track pairs of bounding box IDs. I then counted the keypoint correspondences per box pair to determine the best matches between frames. *In the code could be found comments* ***(Fig 1)***



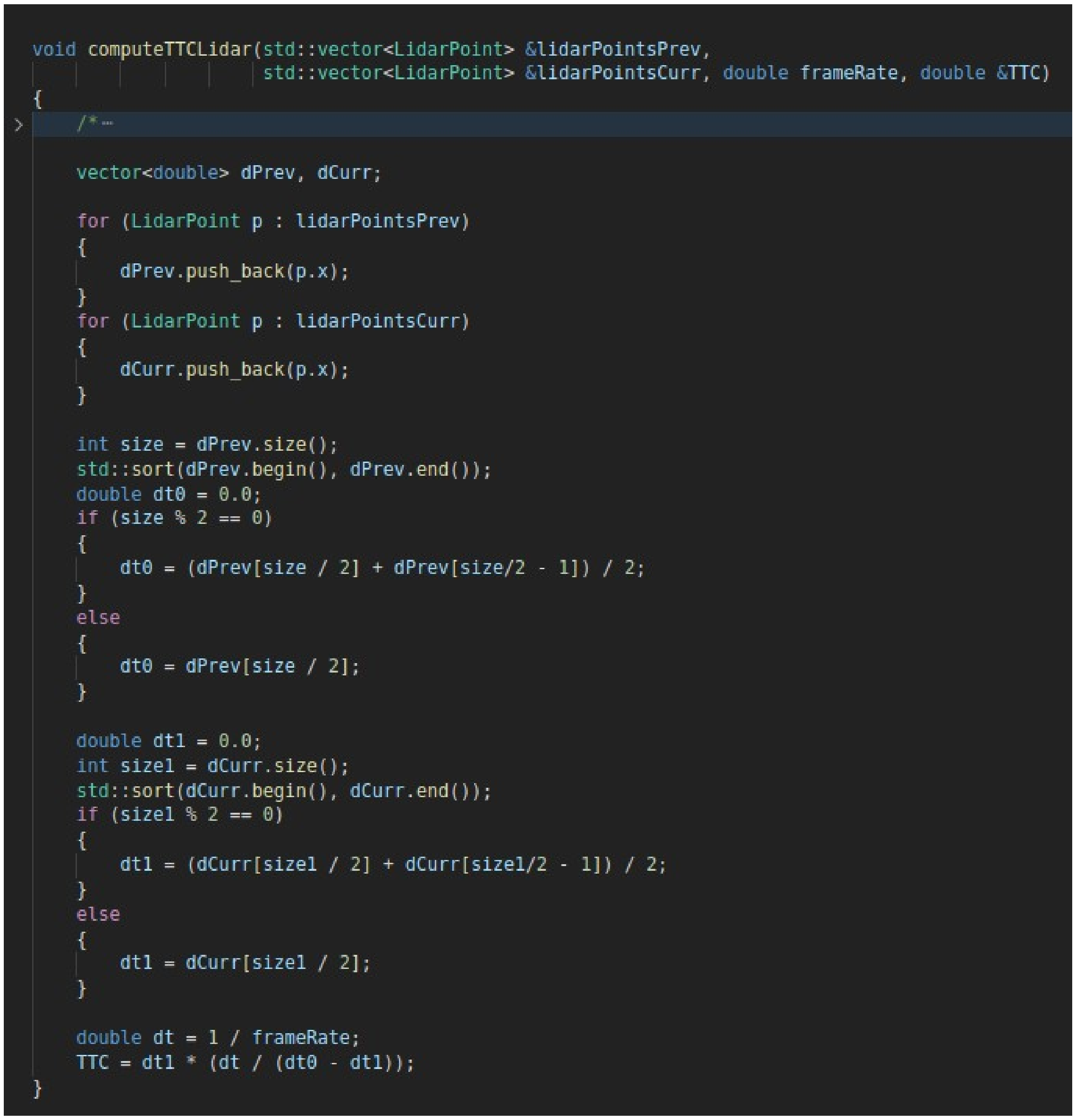
```
void matchBoundingBoxes(std::vector<cv::DMatch> &matches, std::map<int, int> &bbBestMatches, DataFrame &
{
    /*...

    int queryFrameIdx, trainFrameIdx;
    cv::KeyPoint queryFrame, trainFrame;
    int prevSize = prevFrame.boundingBoxes.size();
    int currentSize = currFrame.boundingBoxes.size();
    int counts[prevSize][currentSize] = { };
    vector<int> prevBoxIds, currBoxIds;
    // matched keypoints
    // The idea is to safe all the keypoints in a vector that are contained in the bounding box
    for(auto it1 = matches.begin(); it1 != matches.end(); ++it1 )
    {
        // save the id of every frame
        queryFrameIdx = (*it1).queryIdx;
        trainFrameIdx = (*it1).trainIdx;
        // we look into an specific frame
        queryFrame = prevFrame.keypoints[queryFrameIdx];
        trainFrame = currFrame.keypoints[trainFrameIdx];
        prevBoxIds.clear();
        currBoxIds.clear();
        // previous frame
        for(auto it2 = prevFrame.boundingBoxes.begin(); it2!= prevFrame.boundingBoxes.end(); ++it2)
        {
            if((*it2).roi.contains(queryFrame.pt))
                prevBoxIds.push_back((*it2).boxID);
        }
        // current frame
        for(auto it2 = currFrame.boundingBoxes.begin(); it2!= currFrame.boundingBoxes.end(); ++it2)
        {
            if((*it2).roi.contains(queryFrame.pt))
                currBoxIds.push_back((*it2).boxID);
        }
        // update the counter
        for(auto prevId:prevBoxIds)
        {
            for(auto currId:currBoxIds)
                counts[prevId][currId]++;
        }
    }
    // Best matches boxes
    for (int i = 0; i < prevSize; i++)
    {
        int max_count = 0;
        int id_max = 0;
        for (int j = 0; j < currentSize; j++)
        {
            if (counts[i][j] > max_count)
            {
                max_count = counts[i][j];
                id_max = j;
            }
        }
        bbBestMatches[i] = id_max;
    }
}
```

Fig 1 -. Match 3D objects

FP.2 : Compute Lidar-based TTC

In this part of the final project, your task is to compute the time-to-collision for all matched 3D objects based on Lidar measurements alone. Please take a look at the "Lesson 3: Engineering a Collision Detection System" of this course to revisit the theory behind TTC estimation. Also, please implement the estimation in a way that makes it robust against outliers which might be way too close and thus lead to faulty estimates of the TTC. Please return your TCC to the main function at the end of computeTTCLidar.

Solution: To deal with the outlier Lidar points in a statistically robust way to avoid serious estimation errors, here only the Lidar points within the ego lane are considered, and then the average distance to obtain a stable output is obtained. (**Fig 2**)



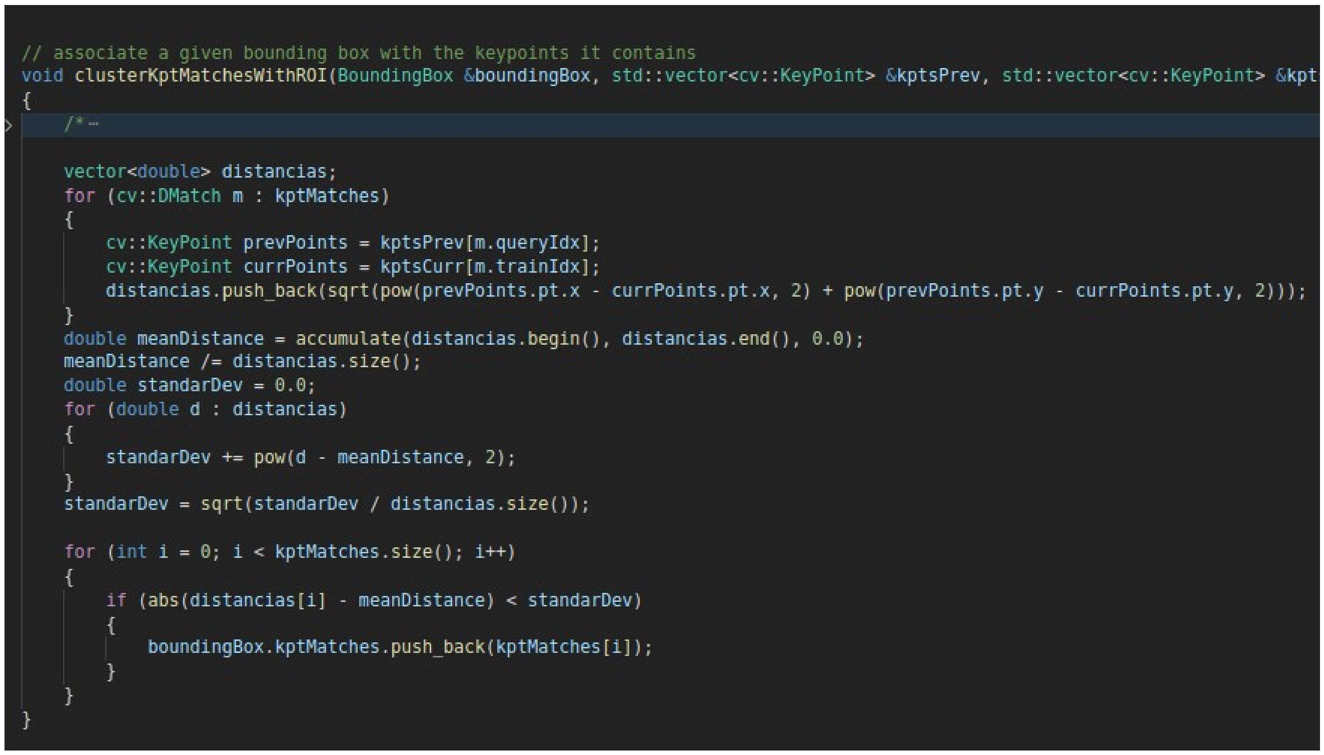
```
void computeTTCLidar(std::vector<LidarPoint> &lidarPointsPrev,
                     std::vector<LidarPoint> &lidarPointsCurr, double frameRate, double &TTC)
{
    /*…

    vector<double> dPrev, dCurr;

    for (LidarPoint p : lidarPointsPrev)
    {
        dPrev.push_back(p.x);
    }
    for (LidarPoint p : lidarPointsCurr)
    {
        dCurr.push_back(p.x);
    }

    int size = dPrev.size();
    std::sort(dPrev.begin(), dPrev.end());
    double dt0 = 0.0;
    if (size % 2 == 0)
    {
        dt0 = (dPrev[size / 2] + dPrev[size/2 - 1]) / 2;
    }
    else
    {
        dt0 = dPrev[size / 2];
    }

    double dt1 = 0.0;
    int size1 = dCurr.size();
    std::sort(dCurr.begin(), dCurr.end());
    if (size1 % 2 == 0)
    {
        dt1 = (dCurr[size1 / 2] + dCurr[size1/2 - 1]) / 2;
    }
    else
    {
        dt1 = dCurr[size1 / 2];
    }

    double dt = 1 / frameRate;
    TTC = dt1 * (dt / (dt0 - dt1));
}
```

Fig 2 -. LiDAR TTC

FP.3 : Associate Keypoint Correspondences with Bounding Boxes

Before a TTC estimate can be computed in the next exercise, you need to find all keypoint matches that belong to each 3D object. You can do this by simply checking whether the corresponding keypoints are within the region of interest in the camera image. All matches which satisfy this condition should be added to a vector. The problem you will find is that there will be outliers among your matches. To eliminate those, I recommend that you compute a robust mean of all the euclidean distances between keypoint matches and then remove those that are too far away from the mean.

Solution: For each bounding box it is necessary to call this function, then scroll through all pairs of key points on the image. Then, when a key point is found within the region of interest, I associate that pair to the bounding box.



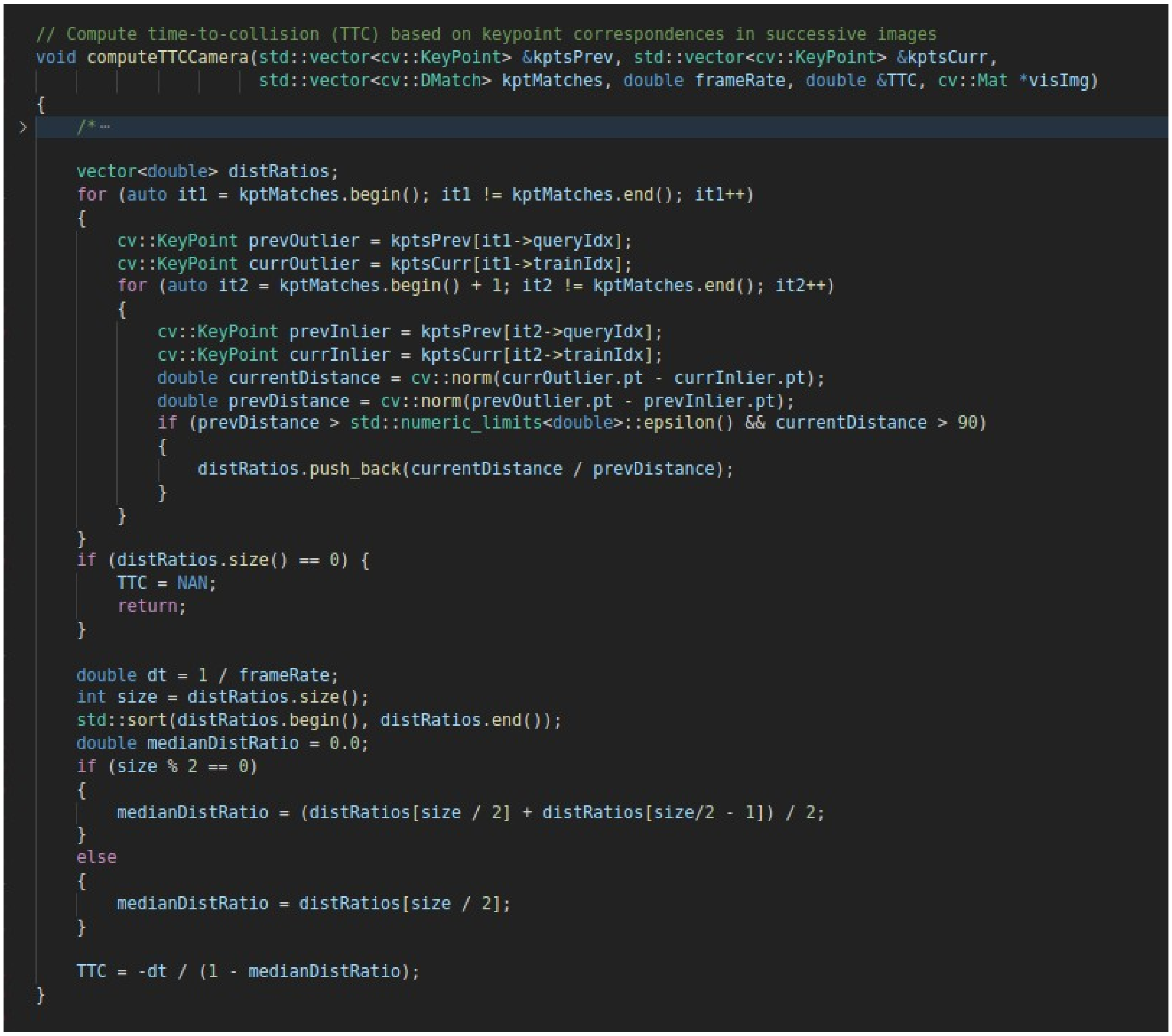
```
// associate a given bounding box with the keypoints it contains
void clusterKptMatchesWithROI(BoundingBox &boundingBox, std::vector<cv::KeyPoint> &kptsPrev, std::vector<cv::KeyPoint> &kpt
{
    /*…

    vector<double> distancias;
    for (cv::DMatch m : kptMatches)
    {
        cv::KeyPoint prevPoints = kptsPrev[m.queryIdx];
        cv::KeyPoint currPoints = kptsCurr[m.trainIdx];
        distancias.push_back(sqrt(pow(prevPoints.pt.x - currPoints.pt.x, 2) + pow(prevPoints.pt.y - currPoints.pt.y, 2)));
    }
    double meanDistance = accumulate(distancias.begin(), distancias.end(), 0.0);
    meanDistance /= distancias.size();
    double standarDev = 0.0;
    for (double d : distancias)
    {
        standarDev += pow(d - meanDistance, 2);
    }
    standarDev = sqrt(standarDev / distancias.size());

    for (int i = 0; i < kptMatches.size(); i++)
    {
        if (abs(distancias[i] - meanDistance) < standarDev)
        {
            boundingBox.kptMatches.push_back(kptMatches[i]);
        }
    }
}
```

Figure 3-. Cluster ROI function

FP.4 : Compute Camera-based TTC

Once keypoint matches have been added to the bounding boxes, the next step is to compute the TTC estimate. As with Lidar, we already looked into this in the second lesson of this course, so you please revisit the respective section and use the code sample there as a starting point for this task here. Once you have your estimate of the TTC, please return it to the main function at the end of compute TTCCamera.

Solution: Following the example given in the class, the distances between the key points are related and with it a TTC is estimated. It is perceived that the calculations obtained by this function sometimes present very different values to what they should be (as can be seen in table 2 marked in red) in the performance document.



```
// 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;
    for (auto it1 = kptMatches.begin(); it1 != kptMatches.end(); it1++)
    {
        cv::KeyPoint prevOutlier = kptsPrev[it1->queryIdx];
        cv::KeyPoint currOutlier = kptsCurr[it1->trainIdx];
        for (auto it2 = kptMatches.begin() + 1; it2 != kptMatches.end(); it2++)
        {
            cv::KeyPoint prevInlier = kptsPrev[it2->queryIdx];
            cv::KeyPoint currInlier = kptsCurr[it2->trainIdx];
            double currentDistance = cv::norm(currOutlier.pt - currInlier.pt);
            double prevDistance = cv::norm(prevOutlier.pt - prevInlier.pt);
            if (prevDistance > std::numeric_limits<double>::epsilon() && currentDistance > 90)
            {
                distRatios.push_back(currentDistance / prevDistance);
            }
        }
    }
    if (distRatios.size() == 0) {
        TTC = NAN;
        return;
    }

    double dt = 1 / frameRate;
    int size = distRatios.size();
    std::sort(distRatios.begin(), distRatios.end());
    double medianDistRatio = 0.0;
    if (size % 2 == 0)
    {
        medianDistRatio = (distRatios[size / 2] + distRatios[size/2 - 1]) / 2;
    }
    else
    {
        medianDistRatio = distRatios[size / 2];
    }

    TTC = -dt / (1 - medianDistRatio);
}
```

Figure 4-. Camara TTC

FP.5 : Performance Evaluation 1

This exercise is about conducting tests with the final project code, especially with regard to the Lidar part. Look for several examples where you have the impression that the Lidar-based TTC estimate is way off. Once you have found those, describe your observations and provide a sound argumentation why you think this happened.

Solution: In table 2, the values that do not agree with the estimated values are marked in red for ALL combinations of descriptors.

FP.6 : Performance Evaluation 2

This last exercise is about running the different detector / descriptor combinations and looking at the differences in TTC estimation. Find out which methods perform best and also include several examples where camera-based TTC estimation is way off. As with Lidar, describe your observations again and also look into potential reasons. This is the last task in the final project.

Solution: To appreciate the graph correctly, the very large values were limited to +100,-100. So in table 1 you can see all the TTC values for each combination and analyze the differences and similarities of each of them.