Chapter 1

Udacity Camera Final Project Report

1.1 MP.1 Match 3D Objects

The code on next page shows how I solved the task. First of all I iterate through the boundingBoxes of the previous Data frame and create for each boundingBox an separate std::map. Then I iterate through the boundingBoxes of the current Data frames. The second inner loop, iterates over all keypoint matches of both images.

Once you are in the innermost loop the comparison begins. First I check if the keypoint of prevFrame is in the region of interest. If so, then I do the same with the keypoint of the currFrame. If both keypoints are in the ROI of each frame, then I count the currBoxID. So I know which boxIDs of the current frame matches the previous frame BoxID.

To find out which boxID appears most often I use the function std::max_element, it returns me the BoxID and the number of matches with prevFrame. Finally I have the matching boxIDs from both frames

```cpp
for (auto prevBox : prevFrame.boundingBoxes)
{
  std::map<int, int> m;
  for (auto currBox : currFrame.boundingBoxes)
  {
    //loop over all matched keypoints
    for (auto match : matches)
    {
      auto prevKeyPoint = prevFrame.keypoints[match.queryIdx].pt;
      //check if keypoint within prevFrame BoundingBox
      if (prevBox.roi.contains(prevKeyPoint))
      {
        auto currKeyPoint = currFrame.keypoints[match.trainIdx].pt;
        //Yes? Then check if also in currentFrame BoundingBox
        if (currBox.roi.contains(currKeyPoint))
        {
         // If the keypoint in both BoundingBoxes,
         // count the currBox Id with an map
         // not yet available?
         if(m.count(currBox.boxID) == 0)
         {
           // create e.g. m[boxID 7,1]
           m[currBox.boxID] = 1;
         }
         else
         {
           //if the keypoint appears more often in the both boundingboxes
           //then count up the corresponding BoundingBox Id
           // create e.g. m[boxID 7,2], m[boxID 7,3] ...
           m[currBox.boxID]++;
         }
        }
      }
     } // eof iterating all matches
    } // eof iterating all current bounding boxes

  auto max = std::max_element(m.begin(), m.end(),
     [](const std::pair<int, int>& p1, const std::pair<int, int>& p2)
     {return p1.second < p2.second; });

  bbBestMatches[prevBox.boxID] = max->first;
}
```

1.2 MP.2 Compute Lidar-based TTC

To calculate the Lidar-based TTC I used the following equation from previous lessons.

$$TTC = d1 * (1.0/frameRate)/(d0 - d1)$$

To reduce the impact of the outlier mentioned in the video I have sorted in booth LidarPointClouds the points from smallest to largest. After that I calculated the median value. See the following code.

```
bool wayToSort(LidarPoint a, LidarPoint b) { return a.x < b.x; }

void computeTTCLidar(std::vector<LidarPoint> &lidarPointsPrev,
                     std::vector<LidarPoint> &lidarPointsCurr,
                     double frameRate, double &TTC)
{
// sort prevFrame Lidar Points
std::sort(lidarPointsPrev.begin(), lidarPointsPrev.end(), wayToSort);
// sort currFrame Lidar Points
std::sort(lidarPointsCurr.begin(), lidarPointsCurr.end(), wayToSort);

// crop median value for calculation
// because of the median value, the few runaways do not cause problems
double d0 = lidarPointsPrev[lidarPointsPrev.size()/2].x;
double d1 = lidarPointsCurr[lidarPointsCurr.size()/2].x;

TTC = d1 * (1.0 / frameRate) / (d0 - d1);
cout << "TTC: " << TTC << endl;
}
```

1.3 MP.3 Associate Keypoint Correspondences with Bounding Boxes

The next step is to find keypoints correspondences to the bounding boxes which enclose them. This is done with the following code. I search keypoints within a bounding box in the current frame and then calculate the euclidean distance with the OpenCV function cv::norm.

After that I calculate a mean value of the euclidean distance between keypoint matches and re-iterate them to remove those that are too far away from the mean.

```
std::vector<double> distance;
for (auto kptMatch :kptMatches)
{
   auto kptCurrPoint = kptsCurr[kptMatch.trainIdx].pt;
   if (boundingBox.roi.contains(kptCurrPoint))
   {
      auto kptPrevPoint = kptsPrev[kptMatch.queryIdx].pt;
      //calculate euclidean distance with cv::nomr
      distance.push_back(cv::norm(kptCurrPoint - kptPrevPoint));
   }
}
int xnum =  distance.size();
double xMean = std::accumulate(distance.begin(), distance.end(), 0.0) / xnum;
double scaledDistance = xMean * 1.3;

for (auto kptMatch :kptMatches)
{
   auto kptCurrPoint = kptsCurr[kptMatch.trainIdx].pt;
   if (boundingBox.roi.contains(kptCurrPoint))
   {
     auto kptPrevPoint = kptsPrev[kptMatch.queryIdx].pt;
     //calculate euclidean distance
     double tempDist = cv::norm(kptCurrPoint - kptPrevPoint));
     if(temp < scaledDistance)
     {
       boundingBox.keypoints.push_back(kptCurrPoint);
       boundingBox.kptMatches.push_back(kptMatch);
     }
   }
}
```

1.4 FP.4 Compute Camera-based TTC

The code for this task is strongly based on the TTC camera lesson. The TTC was calculated by iterating through all points in the kptMatches vector. And each of those points to all other points in the same vector using an inner loop with kptMachtes start + 1. To remove outlier influence I calculated the median value.

```
 vector<double> distRatios;
 double minDist = 100.0;
for (auto out = kptMatches.begin(); out != kptMatches.end() - 1; out++){
   cv::KeyPoint kpOutCurr = kptsCurr.at(out->trainIdx);
   cv::KeyPoint kpOutPrev = kptsPrev.at(out->queryIdx);

   for (auto in = kptMatches.begin()+1; in != kptMatches.end(); in++)
   {
     cv::KeyPoint kpInCurr = kptsCurr.at(in->trainIdx);
     cv::KeyPoint kpInPrev = kptsPrev.at(in->queryIdx);

    // compute distances and distance ratios
     double distCurr{ cv::norm(kpOutCurr.pt - kpInCurr.pt) };
     double distPrev{ cv::norm(kpOutPrev.pt - kpInPrev.pt) };

     if (distPrev > std::numeric_limits<double>::epsilon()
         && distCurr >= minDist)
     {
       // avoid division by zero
       double distRatio = distCurr / distPrev;
       distRatios.push_back(distRatio);
     }
   }
}
// Only continue if the vector of distRatios is not empty
if (distRatios.size() == 0){
   TTC = std::numeric_limits<double>::quiet_NaN();
   return;
}
 //calculate median as in Lidar part
 std::sort(distRatios.begin(), distRatios.end());
 double medianDistRatio = distRatios[distRatios.size() / 2];

 // calculate TTC
 TTC = (-1.0 / frameRate) / (1 - medianDistRatio);
```

1.5 FP.5 Performance Evaluation 1

If we check my result table later in this report, we can see, that we have sometimes outliers in the Lidar TTC estimation. This could be occured when the number of points detected by my detector and descriptor combination detected only very few points. Having a small number of points to base measurements off would explain poor accuracy in timing estimation.

Another point for a bad Lidar TTC could be outliers even we worked with the median value.

1.6 FP.6 Performance Evaluation 2

Certain detector/descriptor combinations, especially the Harris and ORB detectors, produced very unreliable camera TTC estimates. In the midterm project, I choosed the top 3 detector/descriptor. So here, we use them one by one for Camera TTC estimate. See the table on the next pages.

As we can see in the first table, sometimes we have very bad camera TTC estimation, compared with LiDAR based TTC estimation. There are several reasons that could leads to inaccurate camera-based TTC estimation. One of the reason are Key-points mismatching. For example, if in one picture the tail light would match with the roof in the other picture.

A possible solution for a better Camera-TTC could be done by adding a Kalman filter by minimalizing covariance or maybe a simpler solution to improve the results would be compare multiple frames instead of only considering 2 consecutive frames.

The second table contains all detector combinations

FAST + BRISK		FAST + BRIEF		FAST + ORB	
Lidar TTC	**Camera TTC**	**Lidar TTC**	**Camera TTC**	**Lidar TTC**	**Camera TTC**
12,51	12,30	12,51	11,17	12,51	12,20
12,61	12,34	12,61	13,00	12,61	12,93
14,09	16,61	14,09	14,82	14,09	16,62
16,68	12,88	16,68	13,66	16,68	14,08
15,9	-inf	15,9	-inf	15,9	-inf
12,67	13,03	12,67	36,79	12,67	55,79
11,98	12,04	11,98	12,75	11,98	12,38
12,12	11,40	12,12	12,76	12,12	12,18
13,02	11,86	13,02	13,92	13,02	12,87
11,17	13,34	11,17	16,21	11,17	16,23
12,80	12,94	12,80	13,59	12,80	14,13
8,95	12,11	8,95	12,98	8,95	12,98
9,96	12,77	9,96	13,14	9,96	13,55
9,59	11,60	9,59	11,70	9,59	11,29
8,57	11,40	8,57	12,60	8,57	10,71
9,51	12,25	9,51	13,00	9,51	11,91
9,54	9,29	9,54	11,25	9,54	11,94
8,39	11,85	8,39	13,79	8,39	13,78

DETECTOR TYPE	DESCRIPTOR TYPE	TOTAL KEYPOINTS	TTC LIDAR	TTC CAMERA	IMAGE NO.
HARRIS	BRISK	209	6.9526e-310	1.4493e-316	1
HARRIS	BRISK	213	6.90052e-310	3.11261e-321	16
HARRIS	ORB	181	6.90057e-310	6.90057e-310	13
FAST	BRISK	1898	12.3245	12.3	1
FAST	BRIEF	1898	12.3245	11.1776	1
FAST	ORB	1898	12.3245	12.2019	1
FAST	FREAK	1898	12.3245	11.9	1
FAST	FREAK	1840	16.3633	13.2117	3
FAST	SIFT	1898	12.3245	12.3317	1
BRISK	BRISK	3000	12.3245	13.031	1
BRISK	BRIEF	3000	12.3245	14.863	1
BRISK	ORB	3000	12.3245	27.0734	1
BRISK	FREAK	3000	12.3245	12.3601	1
BRISK	SIFT	3000	12.3245	13.7021	1
SIFT	BRIEF	1915	12.3245	12.2492	1
SIFT	FREAK	1915	12.3245	11.2639	1
SIFT	SIFT	1915	12.3245	11.3857	1
SIFT	BRISK	1915	12.3245	11.5922	1
SHITOMASI	BRISK	1759	12.3245	14.1119	1
SHITOMASI	BRIEF	1759	12.3245	13.8948	1
SHITOMASI	ORB	1759	12.3245	13.8801	1
SHITOMASI	FREAK	1759	12.3245	13.7249	1
SHITOMASI	SIFT	1759	12.3245	14.0746	1
ORB	BRISK	500	12.3245	37.4084	1
ORB	BRISK	500	12.6624	-inf	5
ORB	BRISK	500	13.6958	10.8411	6
ORB	BRIEF	500	12.3245	32.5943	1
ORB	BRIEF	500	12.6624	28.5454	5
ORB	BRIEF	500	13.6958	-48.152	6
ORB	ORB	500	12.3245	-inf	1
ORB	ORB	500	12.6624	-inf	5
ORB	ORB	500	13.6958	18.4549	6
ORB	FREAK	500	12.3245	12.2074	1
ORB	FREAK	500	13.6958	29.8927	6
ORB	FREAK	500	12.0968	19.0607	9
ORB	SIFT	500	12.3245	16.7687	1
ORB	SIFT	500	12.6624	73.2489	5
ORB	SIFT	500	13.6958	14.8713	6