Project3: Track an Object in 3D Space

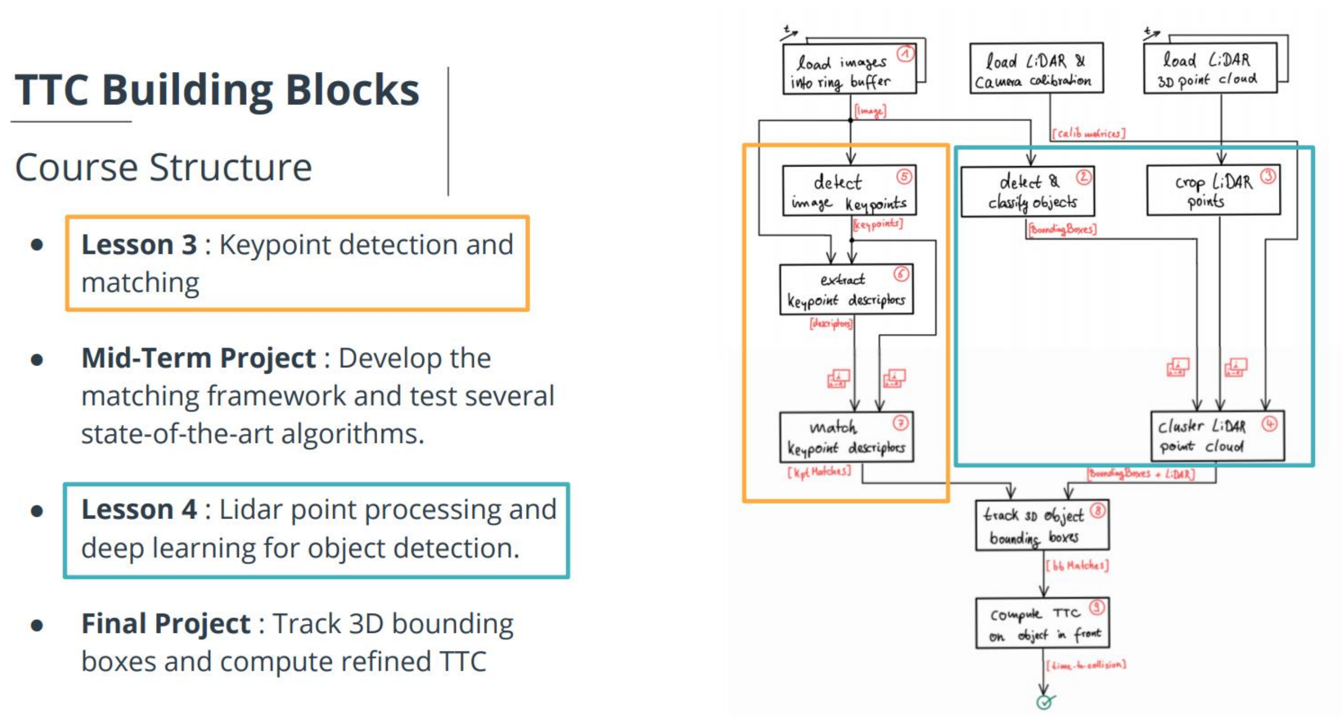
Write up

1st submit: October 20th, Kenta Kumazaki

1. Background

By completing all the lessons, I learned keypoint detectors, descriptors, and methods to match them between successive images. Also, I know how to detect objects in an image using the YOLO deep-learning framework. And finally, I know how to associate regions in a camera image with Lidar points in 3D space.
What I have learned in the lessons are contained in the following repository.
https://github.com/kkumazaki/Sensor-Fusion_Camera_Lessons.git

The program schematic shows what I already have accomplished and what's still missing.



2. Goal

In this final project, you will implement the missing parts in the schematic. To do this, you will complete four major tasks:

(1) I developed a way to match 3D objects over time by using keypoint correspondences.

(2) I computed the TTC based on Lidar measurements.

(3) I proceeded to do the same using the camera, which requires to first associate keypoint matches to regions of interest and then to compute the TTC based on those matches.

(4) I conducted various tests with the framework. My goal is to identify the most suitable detector/descriptor combination for TTC estimation and also to search for problems that can lead to faulty measurements by the camera or Lidar sensor.

*: In the last course of this Nanodegree, I will learn about the Kalman filter, which is a great way to combine the two independent TTC measurements into an improved version which is much more reliable than a single sensor alone can be.

3. Submission

(1) GitHub

https://github.com/kkumazaki/Sensor-Fusion_Project3_Track-an-Object-in-3D-Space.git

(2) Directory

I cloned the basic repository from Udacity https://github.com/udacity/SFND_3D_Object_Tracking.git and added/modified the following files.

- **Writeup_of_project3.pdf**: This file
- **README.md**: Read me file of this repository
- **src**
  - **FinalProject_Camera.cpp**: Main script to set the initial conditions and run the functions.
  - **camFusion_Student.cpp**: Script used to create the functions of Track 3D Object Bounding Boxes and Compute TTC on Object in front..
  - **matching2D_Student.cpp**: Script made in Project 2. (detectors, descriptors and matchers)
- **result**
  - **Project3_result.xlsx**: The resulting list of calculating TTC.
  - **detector_***_descriptor_***.txt**: The result of calculation with each combination of detectors/descriptors.

4. Reflection

(1) Match 3D Objects

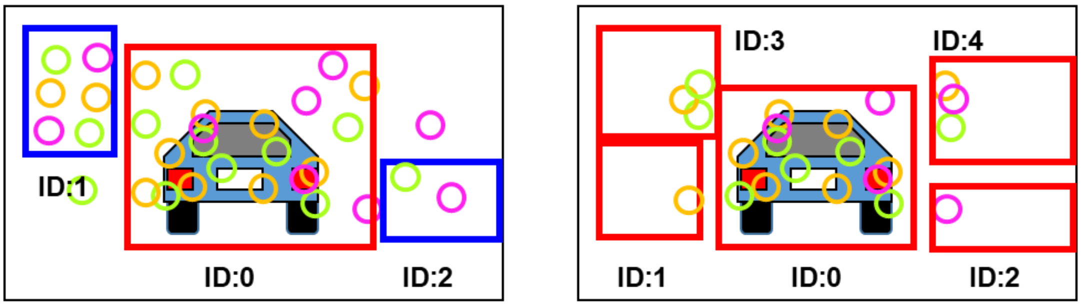
Task FP.1

In this task, I implemented 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.

The task is complete once the code is functional and returns the specified output, where each bounding box is assigned the match candidate with the highest number of occurrences.

The output image of a simple example is as following:

prev Image | curr Image



Matched keypoint count in prev Bounding Box

Iteration

ID	Count
0	245
1	73
2	34

Matched keypoint count in curr Bounding Box

ID	Count
0	231
1	2
2	5
3	0
4	7

Sum=245

Matched keypoint count in both Bounding Box

		curr BB ID				
		0	1	2	3	4
prev BB ID	0	231	2	5	0	7
	1	1	6	2	61	3
	2	3	0	29	1	1

Max Matched Bounding Box (output of this section)

prev BB ID	curr BB ID
0	0
1	3
2	2

The code of the method "matchBoundingBoxes" is shown below. (camFusion_Student.cpp)

```cpp
void matchBoundingBoxes(std::vector<cv::DMatch> &matches, std::map<int, int> &bbBestMatches, DataFrame &prevFrame, DataFrame &currFrame)
{
    // debug view
    bool debView = false;
    //bool debView = true;

    // variables
    int prevCount[prevFrame.boundingBoxes.size()][2]; //[0]: boxID, [1]: matched count
    int currCount[currFrame.boundingBoxes.size()][2]; //[0]: boxID, [1]: matched count
    int bothCount[prevFrame.boundingBoxes.size()][currFrame.boundingBoxes.size()]; //[0]: prev matched count, [1]: curr matched count

    int i, j;

    // initialize
     for (i = 0; i < prevFrame.boundingBoxes.size(); i++){
         prevCount[i][0] = -1;
         prevCount[i][1] = 0;
    }

    for (j = 0; j < currFrame.boundingBoxes.size(); j++){
         currCount[j][0] = -1;
         currCount[j][1] = 0;
    }

    for (i = 0; i < prevFrame.boundingBoxes.size(); i++){
        for (j = 0; j < currFrame.boundingBoxes.size(); j++){
         bothCount[i][j] = 0;
        }
    }

    // pixel coordinates
    cv::Point ptMatchesPrev;
    cv::Point ptMatchesCurr;
```

The double for loops create <u>"bothCount[i][j]"</u>, which is the matched keypoint count in both bounding boxes. **(A)**

(other counting matrices: prevCount and currCount are used for only debug)

```cpp
    // (2) For each prevBB, find the currBB with the most matched point
    for (auto it1 = matches.begin(); it1 != matches.end(); ++it1){
        // Index of each matched keypoints
        int prev_idx = it1->queryIdx;
        int curr_idx = it1->trainIdx;

        // x, y position of each matched keypoints
        ptMatchesPrev.x = prevFrame.keypoints[prev_idx].pt.x;
        ptMatchesPrev.y = prevFrame.keypoints[prev_idx].pt.y;
        ptMatchesCurr.x =  currFrame.keypoints[curr_idx].pt.x;
        ptMatchesCurr.y =  currFrame.keypoints[curr_idx].pt.y;

        // Count matched keypoints in each prevFrame and currFrame with the same index.
        i = 0;
        for (auto it2 = prevFrame.boundingBoxes.begin(); it2 != prevFrame.boundingBoxes.end(); ++it2){
            // check wether point is within the bounding box
            if (it2->roi.contains(ptMatchesPrev)){
                prevCount[i][0] = it2->boxID; // Input bounding box id
                prevCount[i][1] += 1; // Count up
                j = 0;
                for (auto it3 = currFrame.boundingBoxes.begin(); it3 != currFrame.boundingBoxes.end(); ++it3){
                    // check wether point is within the bounding box
                    if (it3->roi.contains(ptMatchesCurr)){
                        currCount[j][0] = it3->boxID; // Input bounding box id
                        currCount[j][1] += 1; // Count up

                        bothCount[i][j] += 1; // Count up
                    }
                    j++;
                }
            }
            i++;
        }
    }
```

As written in the output image in the previous page, I calculated the max matched curr bounding box for each prev bounding box and insert to the output map "bbBestMatches".

```cpp
    for (i = 0; i < prevFrame.boundingBoxes.size(); i++){
        // Initialize the variables
        int bothCountMax = 0;
        int bothID = -1;

        // For each prev bounding box, calculate the most matched curr bounding box.
        for (j = 0; j < currFrame.boundingBoxes.size(); j++){
            if (bothCount[i][j] > bothCountMax){
                bothCountMax = bothCount[i][j];
                bothID = j;
            }
        }
        bbBestMatches.insert(std::pair<int, int>(i, bothID));
        if (debView){
        cout << "bbBestMatches: prev= " << i << ", curr= " << bothID << endl;
        }
    }
```

A

The real output of this method is shown below.

```
#1 : LOAD IMAGE INTO BUFFER done
#2 : DETECT & CLASSIFY OBJECTS done
#3 : CROP LIDAR POINTS done
#4 : CLUSTER LIDAR POINT CLOUD done
#5 : DETECT KEYPOINTS done
#6 : EXTRACT DESCRIPTORS done
#1 : LOAD IMAGE INTO BUFFER done
#2 : DETECT & CLASSIFY OBJECTS done
#3 : CROP LIDAR POINTS done
#4 : CLUSTER LIDAR POINT CLOUD done
#5 : DETECT KEYPOINTS done
#6 : EXTRACT DESCRIPTORS done
#7 : MATCH KEYPOINT DESCRIPTORS done
bbBestMatches: prev= 0, curr= 0
bbBestMatches: prev= 1, curr= 1
bbBestMatches: prev= 2, curr= 2
bbBestMatches: prev= 3, curr= 3
bbBestMatches: prev= 4, curr= 8
bbBestMatches: prev= 5, curr= 5
-----------------------------------------
prevCount[bbID]0, prevCount[count]220
prevCount[bbID]1, prevCount[count]183
prevCount[bbID]2, prevCount[count]36
prevCount[bbID]3, prevCount[count]168
prevCount[bbID]4, prevCount[count]6
prevCount[bbID]5, prevCount[count]75
-----------------------------------------
currCount[bbID]0, currCount[count]293
currCount[bbID]1, currCount[count]158
currCount[bbID]2, currCount[count]35
currCount[bbID]3, currCount[count]220
currCount[bbID]4, currCount[count]61
currCount[bbID]5, currCount[count]83
currCount[bbID]6, currCount[count]9
currCount[bbID]7, currCount[count]2
currCount[bbID]8, currCount[count]17
currCount[bbID]9, currCount[count]4
currCount[bbID]10, currCount[count]43
-----------------------------------------
#8 : TRACK 3D OBJECT BOUNDING BOXES done
Step#9: compute TTC
```

(2) Compute Lidar-based TTC

Task FP.2 : Compute Lidar-based TTC

In this part of the final project, my task is to compute the time-to-collision for all matched 3D objects based on Lidar measurements alone. I referred to the "Lesson 3: Engineering a Collision Detection System" of this course to revisit the theory behind TTC estimation show as below.

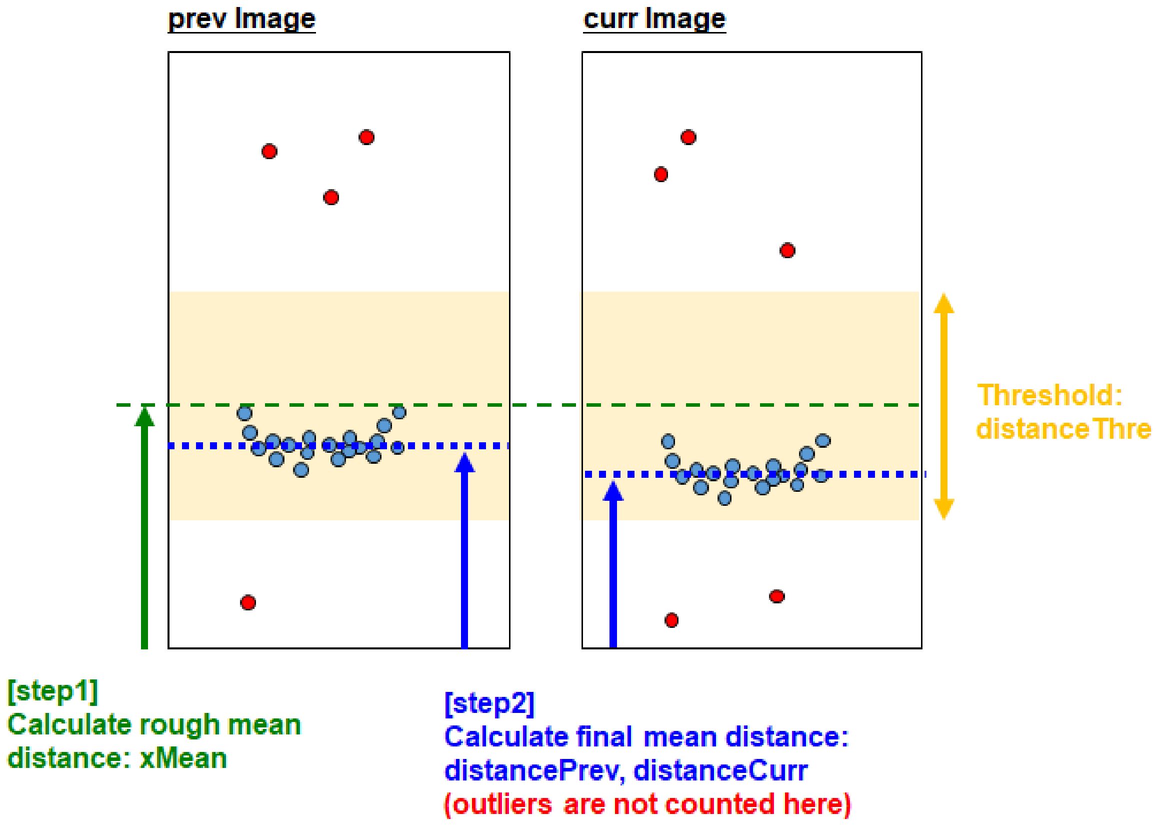
$$(1) \quad d(t+\Delta t) = d(t) - v_0 \cdot \Delta t$$

$$(2) \quad v_0 = \frac{d(t) - d(t+\Delta t)}{\Delta t} = \frac{d_0 - d_1}{\Delta t}$$

$$(3) \quad TTC = \frac{d_1}{v_0} = \frac{d_1 \cdot \Delta t}{d_0 - d_1}$$

Also, I implemented 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. Then I return my TCC to the main function at the end of the method "computeTTCLidar". The task is complete once the code is functional and returns the specified output. Also, the code is able to deal with outlier Lidar points in a statistically robust way to avoid severe estimation errors.

The output image of my code is shown below:



My code is show below. At first, I calculate the rough mean distance of the prev Lidar Points: xMean. **(A)**

By using rough mean and threshold, I remove the outliers during my final mean distance calculation. **(B)**

```cpp
void computeTTCLidar(std::vector<LidarPoint> &lidarPointsPrev,
                     std::vector<LidarPoint> &lidarPointsCurr, double frameRate, double &TTC)
{
    // debug view
    //bool debView = false;
    bool debView = true;

    double maxTTC = 50;

    // (1) Calculate the mean of the Lidar points for the robust calculation.
    // This calculation is needed only for either Prev or Curr because vehicle doesn't move so quickly.
    // (The Lidar points can be disperse, anyway )
    float xMean = 0;
```

```cpp
    for (auto it1 = lidarPointsPrev.begin(); it1 != lidarPointsPrev.end(); ++it1){
        xMean += it1->x; // world position in m with x facing forward from sensor
    }
    xMean = xMean/(float)lidarPointsPrev.size();

    if (debView){
        cout << "xMean: " << xMean << endl;
    }
```

A

```cpp
    // (2) Robust calculation of the vehicle end position.
    // Calculate the average x distance of Lidar points near from the xMean.
    float distancePrev = 0;
    float distanceCurr = 0;
    float distanceThre = 2.0;//Calculate the mean of distance only by Lidar points within threshold.
```

```cpp
    for (auto it1 = lidarPointsPrev.begin(); it1 != lidarPointsPrev.end(); ++it1){
        // world coordinates
        if ((it1->x < (xMean + distanceThre)) && (it1->x > (xMean - distanceThre))){
            distancePrev += it1->x; // world position in m with x facing forward from sensor
        }
    }
    distancePrev = distancePrev/(float)lidarPointsPrev.size();

    for (auto it1 = lidarPointsCurr.begin(); it1 != lidarPointsCurr.end(); ++it1){
        // world coordinates
        if ((it1->x < (xMean + distanceThre)) && (it1->x > (xMean - distanceThre))){
            distanceCurr += it1->x; // world position in m with x facing forward from sensor
        }
    }
    distanceCurr = distanceCurr/(float)lidarPointsCurr.size();
```

B

Finally, I calculate the Lidar TTC according to the equation.

```cpp
    // (3) Calculate TTC
    double deltaT = 1/frameRate; // [Hz]-->[second]
    TTC = ( (double)distanceCurr * deltaT )/( (double)distancePrev - (double)distanceCurr);

    if ((TTC < 0) || (TTC > maxTTC)){
        TTC = maxTTC;
    }

    if (debView){
        cout << "Lidar TCC: " << TTC << endl;
    }
}
```

(3) Compute Camera-based TTC

Task FP.3 : Associate Keypoint Correspondences with Bounding Boxes

Before a TTC estimation of Camera, I need to find all keypoint matches that belong to each 3D object.

I 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.

There are outliers among my matches, so I should calculate a robust mean of all the euclidean distances between keypoint matches and then remove those that are too far away from the mean.

My code is shown below. First of all, I calculate the “distanceMean”. **(A)**

```
// associate a given bounding box with the keypoints it contains
void clusterKptMatchesWithROI(BoundingBox &boundingBox, std::vector<cv::KeyPoint> &kptsPrev, std::vector<cv::KeyPoint> &kptsCurr, std::vector<cv::DMatch> &kptMatches)
{
    // debug view
    bool debView = false;
    //bool debView = true;

    // pixel coordinates
    cv::Point ptMatchesPrev;
    cv::Point ptMatchesCurr;

    // Distance mean to remove the outliers
    float distanceMean = 0;

    // Count matched keypoints in each prevFrame and currFrame with the same index.
    for (auto it1 = kptMatches.begin(); it1 != kptMatches.end(); ++it1){
        // Index of each matched keypoints
        int prev_idx = it1->queryIdx;
        int curr_idx = it1->trainIdx;

        // x, y position of each matched keypoints
        ptMatchesPrev.x = kptsPrev[prev_idx].pt.x;
        ptMatchesPrev.y = kptsPrev[prev_idx].pt.y;
        ptMatchesCurr.x = kptsCurr[curr_idx].pt.x;
        ptMatchesCurr.y = kptsCurr[curr_idx].pt.y;

        if ((boundingBox.roi.contains(ptMatchesPrev)) && (boundingBox.roi.contains(ptMatchesCurr))){
            boundingBox.keypoints.push_back(kptsCurr[curr_idx]);
            boundingBox.kptMatches.push_back(*it1);
            //boundingBox.kptMatches.push_back(kptMatches[it1]); // error
            //cout << "prev_idx: " << prev_idx << ", curr_idx: " << curr_idx << endl;
            distanceMean += (float) it1->distance;
        }
    }
    distanceMean =  distanceMean / (float) boundingBox.kptMatches.size();
```

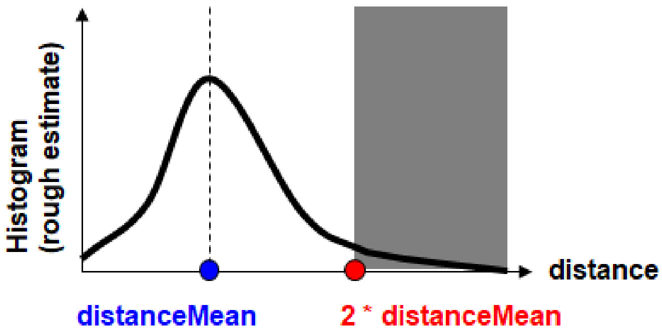
A

I erase the keypoint matches whose distance is greater than the twice amount of “distanceMean”. **(B)**

```
    // Erase kptMatches if distance is a lot more than mean.
    for (int i = boundingBox.kptMatches.size(); i > -1; i--){
    //for (int i = 0; i < boundingBox.kptMatches.size(); i++){ // When erase at vector, the indent change after that. Be careful.
        if (boundingBox.kptMatches[i].distance > (int)(distanceMean * 2.)){ // cut off more than 2 times as mean
        //if (boundingBox.kptMatches[i].distance > (int)(distanceMean + 10.)){ // 10 is too small
            if (debView){
                cout << "Delete kptMatches with distance: " << boundingBox.kptMatches[i].distance << endl;
            }
            boundingBox.kptMatches.erase(boundingBox.kptMatches.begin() + i);
        }
    }
    if (debView){
        cout << "Threshold: " << (int)(distanceMean * 2.) << endl;
        cout << "kptMatches size inside boundingBox (without outliers): " << boundingBox.kptMatches.size() << endl;
    }
}
```

B

The rough output image is as below.



Task 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. I refer Lesson 3 "compute_ttc_camera.cpp" and use the code sample there as a starting point for this task here. Camera TTC is shown below.

project object into camera

$$(1) \quad h_0 = \frac{f \cdot H}{d_0}; \quad h_1 = \frac{f \cdot H}{d_1}$$

relate projection and distance

$$(2) \quad \frac{h_1}{h_0} = \frac{\frac{f \cdot H}{d_1}}{\frac{f \cdot H}{d_0}} = \frac{d_0}{d_1} \rightarrow d_0 = d_1 \cdot \frac{h_1}{h_0}$$

substitute in constant-velocity model

$$(3) \quad d_1 = d_0 - v_0 \cdot \Delta t = d_1 \cdot \frac{h_1}{h_0} - v_0 \cdot \Delta t$$

$$\rightarrow d_1 = \frac{-v_0 \cdot \Delta t}{\left(1 - \frac{h_1}{h_0}\right)}$$

compute time to contact / collision

$$(4) \quad TTC = \frac{d_1}{v_0} = \frac{-\Delta t}{\left(1 - \frac{h_1}{h_0}\right)}$$

The following is my code.

```cpp
// 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)
{
    // debug view
    bool debView = false;
    //bool debView = true;
    if (debView){
        cout << "---------computeTTCCamera function: start---------" << endl;
        for (auto it1 = kptMatches.begin(); it1 != kptMatches.end() - 1; ++it1)
        {
            cout << "it1->trainIdx: " << it1->trainIdx << ", it1->queryIdx: " << it1->queryIdx << endl;
        }
        cout << "kptMatches size: " << kptMatches.size() << endl;
        cout << "kptsPrev size: " << kptsPrev.size() << ", kptsCurr size: " << kptsCurr.size() << endl;
        cout << "frameRate: " << frameRate << endl;
    }
    // Reference: Udacity SF, Lesson 3 "compute_ttc_camera.cpp"
    // compute distance ratios between all matched keypoints
    vector<double> distRatios; // stores the distance ratios for all keypoints between curr. and prev. frame
    int i = 0;
    for (auto it1 = kptMatches.begin(); it1 != kptMatches.end() - 1; ++it1)
    { // outer keypoint loop

        //cout << "it1->trainIdx: " << it1->trainIdx << ", it1->queryIdx: " << it1->queryIdx << endl;

        // 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);
        int j = 0;

        for (auto it2 = kptMatches.begin() + 1; it2 != kptMatches.end(); ++it2)
        { // inner keypoint loop

            //double minDist = 5.0; // min. required distance
            double minDist = 100.0; // min. required distance

            //cout << "for #1" << endl;

            // 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);

            //cout << "for #2" << endl;

            // compute distances and distance ratios
            double distCurr = cv::norm(kpOuterCurr.pt - kpInnerCurr.pt);
            double distPrev = cv::norm(kpOuterPrev.pt - kpInnerPrev.pt);
```

```cpp
            // compute distances and distance ratios
            double distCurr = cv::norm(kpOuterCurr.pt - kpInnerCurr.pt);
            double distPrev = cv::norm(kpOuterPrev.pt - kpInnerPrev.pt);

            //cout << "iteration 1: " << i << ", iteration 2: " << j << endl;
            //cout << "distCurr: " << distCurr << ", distPrev: " << distPrev << endl;

            //cout << "for #3" << endl;

            if (distPrev > std::numeric_limits<double>::epsilon() && distCurr >= minDist)
            { // avoid division by zero

                double distRatio = distCurr / distPrev;
                distRatios.push_back(distRatio);
                // Debug
                //cout << "distRatio: " << distRatio << endl;
            }
            j++;
        } // eof inner loop over all matched kpts
        i++;
    }     // eof outer loop over all matched kpts

    //cout << "distRatios.size: " << distRatios.size() << endl;

    // only continue if list of distance ratios is not empty
    if (distRatios.size() == 0)
    {
        cout << "TTC = NAN" << endl;
        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);

    // TODO: STUDENT TASK (replacement for meanDistRatio)
    vector<double> distRatiosSort;
    double temp;
    double medianDistRatio;

    copy(distRatios.begin(), distRatios.end(), back_inserter(distRatiosSort));
```

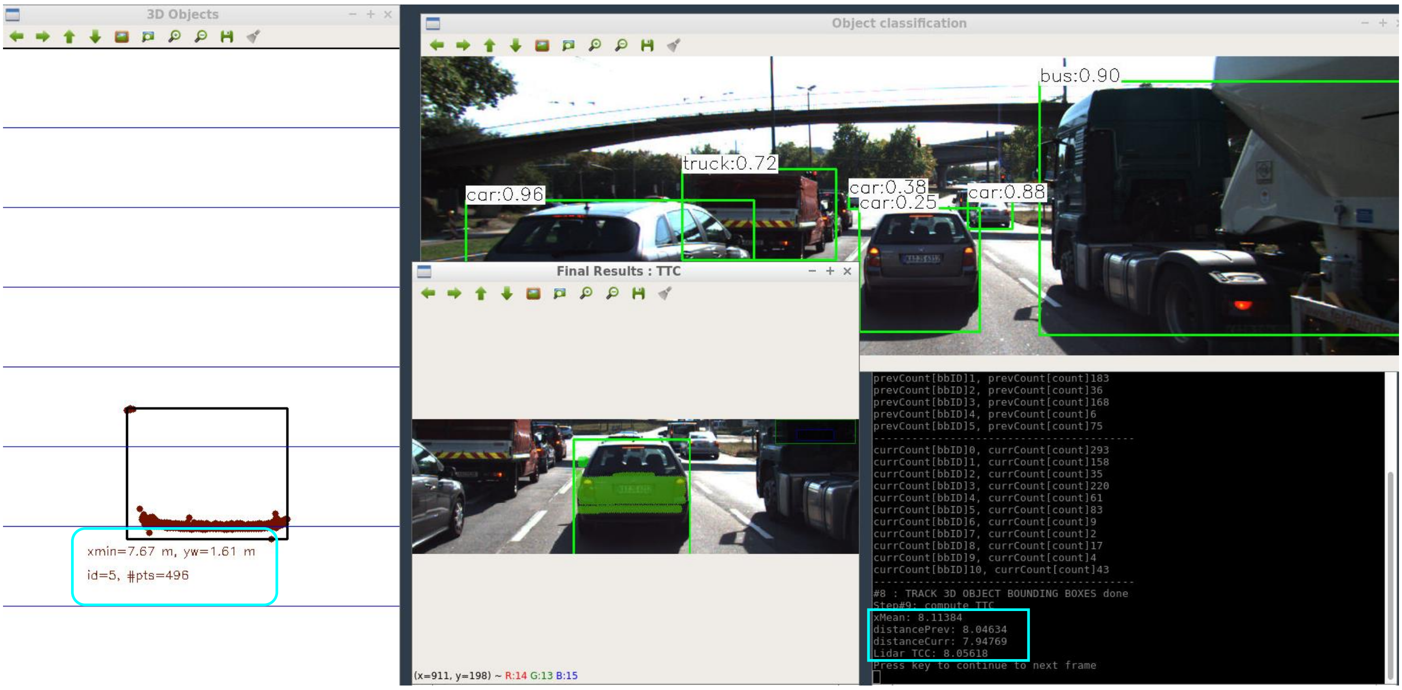
```cpp
    for (int i = 0; i < distRatiosSort.size() - 1; i++){
        for (int j = 0; j < distRatiosSort.size() - i; j++){
            if(distRatiosSort[j] > distRatiosSort[j+1]){
                temp = distRatiosSort[j];
                distRatiosSort[j] = distRatiosSort[j+1];
                distRatiosSort[j+1] = temp;
            }
        }
    }

    if (debView){
            cout << "distRatiosSort.size: " << distRatiosSort.size() << endl;
    }

    if (distRatiosSort.size() % 2 == 0){
        medianDistRatio = (distRatiosSort[distRatiosSort.size()/2 -1]+distRatiosSort[distRatiosSort.size()/2])/2;
        //cout << distRatiosSort.size()/2 -1 << std::endl;
        //cout << medianDistRatio << std::endl;
    }
    else {
        medianDistRatio = distRatiosSort[(distRatiosSort.size()-1)/2];
    }

    TTC = -dT / (1 - medianDistRatio);

    if (debView){
        cout << "Camera TTC: " << TTC << endl;
    }
}
```

(4) Performance Evaluation

Task 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 I have the impression that the Lidar-based TTC estimate is way off. Once I have found those, describe my observations and provide a sound argumentation why I think this happened.

Basically there's 1 matched object appear and Lidar TTC is calculated stably as below.



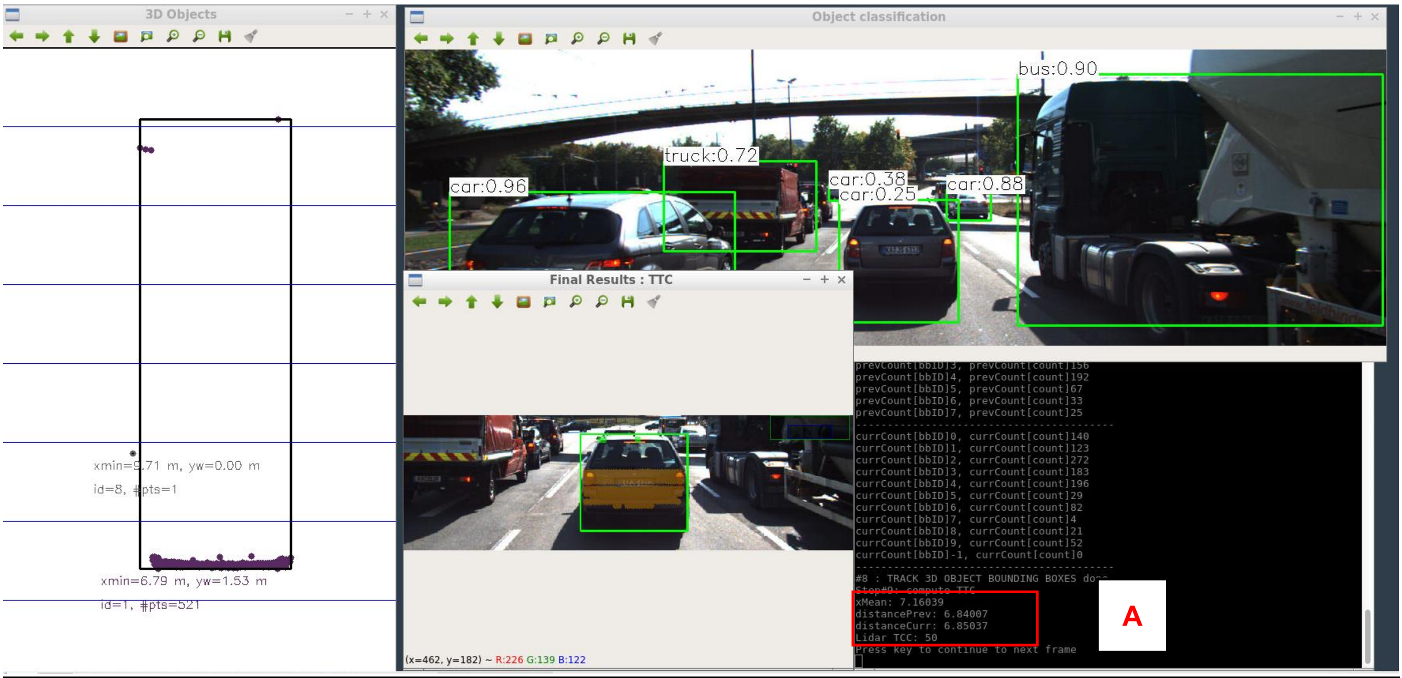
However in imageID=17, there appear 2 matched objects. **(A)**

With that unnecessary matched object, Lidar TTC becomes zero and it's not correct. **(B)**

In this kind of situation, I can reject the far different result than previous result and take the other one. **(C)**

In imageID=18, the distancePrev is smaller than distanceCurr. **(A)**

In this case TTC would be calculated as minus, but minus time doesn't make sense.

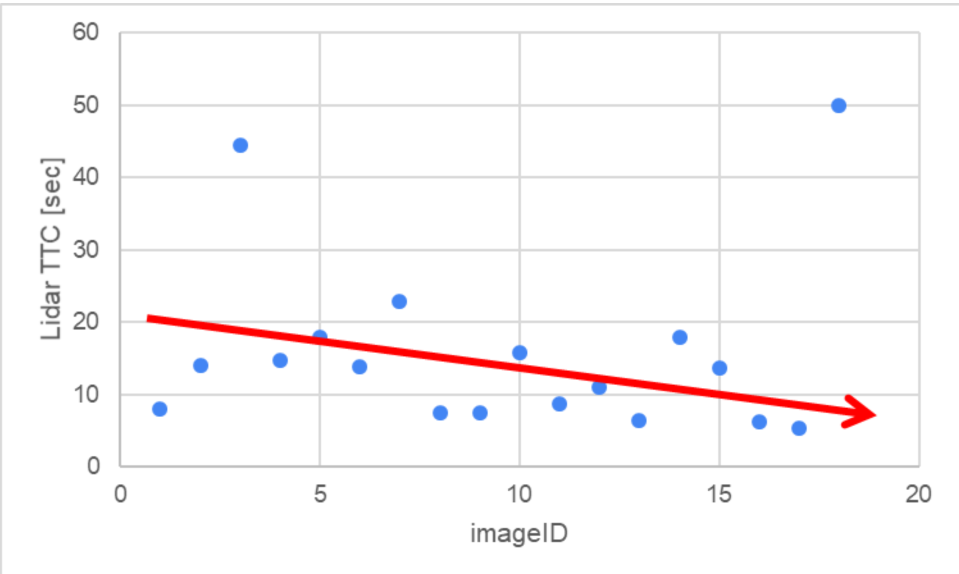


So I made a moderately large time as a threshold (maxTTC = 50), and I added the guard for minus TTC or too large TTC as below. That's why the resulting TTC is "50" in this scene. **(A)**

```
    if ((TTC < 0) || (TTC > maxTTC)){
        TTC = maxTTC;
    }
```

The resulting Lidar TTC is shown below.

There are some noise and outliers as described above, but overall Lidar TTC keeps decreasing as the preceding vehicle gets closer to ego vehicle.



Task FP.6 : Performance Evaluation 2

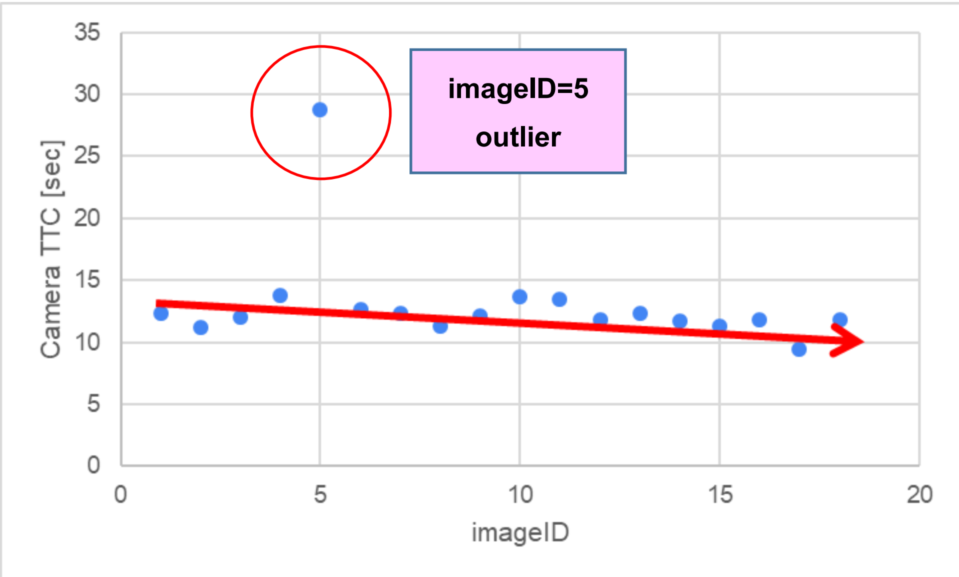
(1)Basic result and analysis of Camera TTC

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, I describe my observations again and also look into potential reasons. This is the last task in the final project.

As the basic setup, I've been using "detector: FAST" and "descriptor: ORB", which was the best combination in my Project 2: Camera Based 2D Feature Tracking.

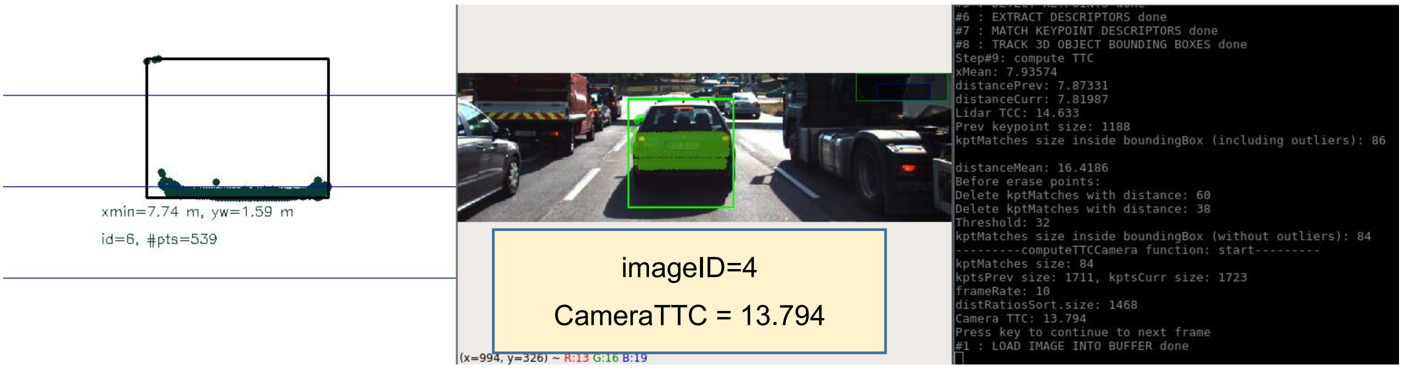
```
/* MAIN PROGRAM */
int main(int argc, const char *argv[])
{
    /* INIT VARIABLES AND DATA STRUCTURES */

   // Change the location of combinations for efficiency
    string detectorType = "FAST";  // Task MP.2 Modern fast methods: FAST, BRISK, ORB, AKAZE, SIFT // SIFT detector
    string descriptorType = "ORB"; // BRIEF, ORB, FREAK, AKAZE, SIFT, BRISK // SIFT/AKAZE descriptor is only good wi
    string matcherType = "MAT_BF";        // MAT_BF, MAT_FLANN // Basically use BF because it's assigned in Lesson i
    string descType = "DES_BINARY"; // DES_BINARY, DES_HOG // Basically use BINARY because it's faster // Change the
    string selectorType = "SEL_KNN";       // SEL_NN, SEL_KNN // Use KNN with minDescDistRatio: 0.8
```

First of all, I show the resulting Camera TTC as below.
Except for one outlier, there's less variance than Lidar TTC.

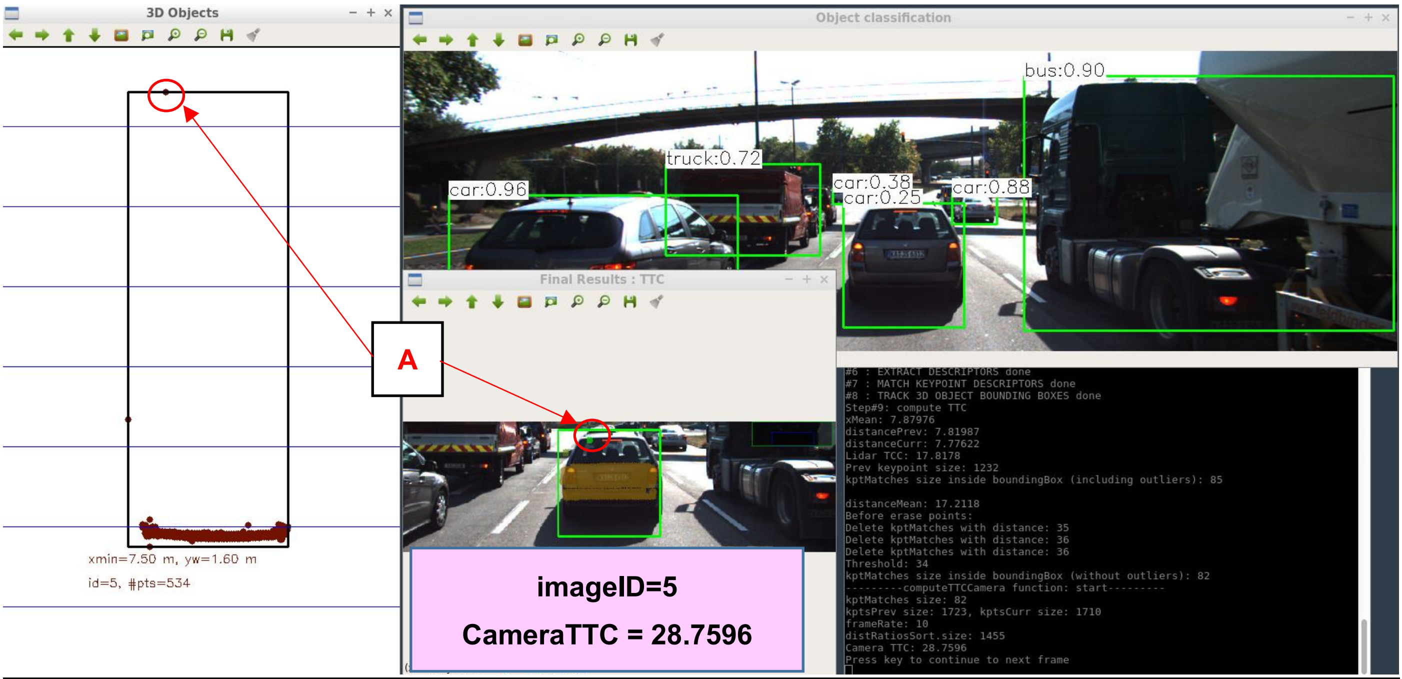


I will explain the details in the next page.

The following imageID=4 is the right before outlier occurs. Bounding box is covering only the preceding vehicle.



The following imageID=5 is the exact timing of the outlier occurs. The difference between imageID=4 is that the Lidar detects the points of the vehicle in front of the preceding vehicle **(A)** and bounding box becomes longer. I assume that the size change of bounding box changed the average distance of matched keypoints.



Once the size of bounding box keeps same, CameraTTC comes back to normal as shown below.

(2)Compare Camera TTC with different kinds of combinations

The task is complete once all detector / descriptor combinations implemented in previous chapters have been compared with regard to the TTC estimate on a frame-by-frame basis. To facilitate the comparison, a spreadsheet and graph should be used to represent the different TTCs.

From the result of my Project 2 below, I should choose the combinations that can calculate detection & description faster than 100ms (faster than 10 Hz) to execute sensor fusion stably.

I will try the following red-circled 6 combinations.

Log time of detection & description *: minDescDistRatio = 0.8

Detector	Descriptor	Matcher	Descriptor Type	Selector Type *	Image										Average	Standard deviation
					1	2	3	4	5	6	7	8	9	10		
FAST	BRIEF	BF	BINARY	KNN	13.78482	2.63205	2.589599	1.918801	1.715051	2.0994	1.860771	1.922764	1.803991	2.65278	3.2980027	3.7023798
FAST	ORB	BF	BINARY	KNN	3.91497	2.46766	2.83399	3.36009	2.379488	2.115638	2.42344	2.17239	2.14725	2.378315	2.6193231	0.589425
FAST	FREAK	BF	BINARY	KNN	50.0574	47.47236	47.129002	44.7902	46.460801	44.75397	44.31788	44.368731	44.23166	43.67391	45.725591	2.0113239
FAST	AKAZE	BF	BINARY	KNN	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!
FAST	SIFT	BF	BINARY	KNN	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!
FAST	BRISK	BF	BINARY	KNN	341.99349	334.44043	333.39725	336.80917	333.8677	335.38144	337.70074	337.07355	330.86572	333.43304	335.49625	3.0762553
BRISK	BRIEF	BF	BINARY	KNN	379.58215	375.14927	383.07944	370.36348	373.95816	375.89489	371.29072	378.383	369.48902	370.16587	374.7356	4.5695051
BRISK	ORB	BF	BINARY	KNN	398.5169	371.75964	375.5496	372.39759	372.07871	371.91483	371.24304	373.17009	371.8695	379.15854	375.76584	8.3490715
BRISK	FREAK	BF	BINARY	KNN	428.0999	417.6433	429.0905	415.5593	416.184	419.347	425.2221	408.478	413.0412	419.2958	419.19611	6.5911802
BRISK	AKAZE	BF	BINARY	KNN	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!
BRISK	SIFT	BF	BINARY	KNN	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!
BRISK	BRISK	BF	BINARY	KNN	707.415	703.477	718.42	708.213	713.271	710.001	720.133	705.321	699.717	707.737	709.3705	6.3725998
ORB	BRIEF	BF	BINARY	KNN	18.0461	11.22003	9.540118	8.499755	8.955175	8.478594	9.057613	8.442334	9.426711	8.567693	10.023412	2.9398764
ORB	ORB	BF	BINARY	KNN	24.38513	15.47827	16.14907	15.02095	15.52066	15.37132	15.01398	14.2217	15.01864	15.24786	16.142758	2.937269
ORB	FREAK	BF	BINARY	KNN	56.7768	53.8343	52.68927	49.62825	50.42939	51.10066	49.662	50.742	49.93431	51.09253	51.588951	2.2609773
ORB	AKAZE	BF	BINARY	KNN	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!
ORB	SIFT	BF	BINARY	KNN	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!
ORB	BRISK	BF	BINARY	KNN	344.4093	341.66439	347.37533	340.89892	345.59549	339.69733	345.60187	339.40693	339.70318	338.60915	342.29619	3.159592
AKAZE	BRIEF	BF	BINARY	KNN	123.07029	111.03576	112.47584	107.87008	107.17069	108.26384	108.0801	115.04378	111.16318	109.51094	111.36845	4.779381
AKAZE	ORB	BF	BINARY	KNN	118.53384	111.58489	114.82577	109.64297	112.58801	110.95995	109.88618	110.30779	110.02497	112.61345	112.09678	2.7758205
AKAZE	FREAK	BF	BINARY	KNN	152.2798	149.398	157.9001	150.7148	147.4728	145.9482	158.9215	147.145	140.1395	145.9262	149.58459	5.6875114
AKAZE	AKAZE	BF	BINARY	KNN	202.7055	188.451	194.6408	196.3547	196.0294	191.6899	191.2015	194.823	193.0745	194.249	194.32193	3.8061133
AKAZE	SIFT	BF	BINARY	KNN	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!
AKAZE	BRISK	BF	BINARY	KNN	454.75	449.233	443.869	450.253	438.771	448.095	441.489	449.89	443.54	435.508	445.5398	5.9234794
SIFT	BRIEF	BF	HOG	KNN	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!
SIFT	ORB	BF	HOG	KNN	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!
SIFT	FREAK	BF	HOG	KNN	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!
SIFT	AKAZE	BF	HOG	KNN	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!
SIFT	SIFT	BF	HOG	KNN	309.244	234.2365	256.557	240.33	247.321	234.1698	233.4812	233.8341	234.4652	233.9066	245.75454	23.586508
SIFT	BRISK	BF	HOG	KNN	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!	#REF!

At first, I show the resulting Lidar TTC below.

It's clear that Lidar TTC don't change according to the combinations of detector & descriptor.

Lidar TTC *: minDescDistRatio = 0.8

Detector	Descriptor	Matcher	escriptorType	electorType *	Image																	
					1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
FAST	BRIEF	BF	BINARY	KNN	8.05618	13.9136	44.5655	14.633	17.8178	13.8814	22.9091	7.41929	7.4619	15.8242	8.68199	10.94	6.37853	17.9264	13.714	6.1514	5.30408	50
FAST	ORB	BF	BINARY	KNN	8.05618	13.9136	44.5655	14.633	17.8178	13.8814	22.9091	7.41929	7.4619	15.8242	8.68199	10.94	6.37853	17.9264	13.714	6.1514	5.30408	50
FAST	FREAK	BF	BINARY	KNN	8.05618	13.9136	44.5655	14.633	17.8178	13.8814	22.9091	7.41929	7.4619	15.8242	8.68199	10.94	6.37853	17.9264	13.714	6.1514	5.30408	50
FAST	AKAZE	BF	BINARY	KNN																		
FAST	SIFT	BF	BINARY	KNN																		
FAST	BRISK	BF	BINARY	KNN																		
BRISK	BRIEF	BF	BINARY	KNN																		
BRISK	ORB	BF	BINARY	KNN																		
BRISK	FREAK	BF	BINARY	KNN																		
BRISK	AKAZE	BF	BINARY	KNN																		
BRISK	SIFT	BF	BINARY	KNN																		
BRISK	BRISK	BF	BINARY	KNN																		
ORB	BRIEF	BF	BINARY	KNN	8.05618	13.9136	44.5655	14.633	17.8178	13.8814	22.9091	7.41929	7.4619	15.8242	8.68199	10.94	6.37853	17.9264	13.714	6.1514	5.30408	50
ORB	ORB	BF	BINARY	KNN	8.05618	13.9136	44.5655	14.633	17.8178	13.8814	22.9091	7.41929	7.4619	15.8242	8.68199	10.94	6.37853	17.9264	13.714	6.1514	5.30408	50
ORB	FREAK	BF	BINARY	KNN	8.05618	13.9136	44.5655	14.633	17.8178	13.8814	22.9091	7.41929	7.4619	15.8242	8.68199	10.94	6.37853	17.9264	13.714	6.1514	5.30408	50

Next, Camera TTC is shown below.

The results of detector=ORB are not stable because some of TTCs become too high or too low.

So, there are only 3 choices remaining.

Camera TTC *: minDescDistRatio = 0.8

Detector	Descriptor	Matcher	escriptorType	electorType *	Image																	
					1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
FAST	BRIEF	BF	BINARY	KNN	11.2388	11.0789	12.8872	14.0676	20.9709	12.888	11.8985	11.7219	11.7599	13.3278	13.7899	11.0836	11.6502	11.5887	11.9075	12.381	7.57979	11.0526
FAST	ORB	BF	BINARY	KNN	12.3806	11.1626	11.9712	13.794	28.7596	12.6793	12.352	11.3441	12.1119	13.6608	13.4364	11.777	12.3008	11.7213	11.3421	11.7851	9.41431	11.8307
FAST	FREAK	BF	BINARY	KNN	11.9301	15.6025	13.2	14.033	173.98	12.3086	12.2021	11.5965	12.3316	13.4105	13.0759	11.8644	11.6118	11.0205	10.8147	11.9191	8.3405	11.9006
FAST	AKAZE	BF	BINARY	KNN																		
FAST	SIFT	BF	BINARY	KNN																		
FAST	BRISK	BF	BINARY	KNN																		
BRISK	BRIEF	BF	BINARY	KNN																		
BRISK	ORB	BF	BINARY	KNN																		
BRISK	FREAK	BF	BINARY	KNN																		
BRISK	AKAZE	BF	BINARY	KNN																		
BRISK	SIFT	BF	BINARY	KNN																		
BRISK	BRISK	BF	BINARY	KNN																		
ORB	BRIEF	BF	BINARY	KNN	22.07	24.1294	92.8179	15.233	20.1612	13.3056	36.3987	326.784	2.85E+06	1[illegible].7467	19.4803	21.2644	12.8562	8.82683	6.80537	10.9804	15.0819	21.8102
ORB	ORB	BF	BINARY	KNN	17.4635	9.63158	18.2793	28.1023	30.1357	17.9796	220.536	10.4367	-inf	13.2779	8.2896	36.2236	10.4189	13.4253	9.30504	9.76932	13.4367	26.6197
ORB	FREAK	BF	BINARY	KNN	12.6861	38.7882	11.3805	11.2087	-inf	11.1989	-inf	9.38588	12.8815	-inf	7.86174	26.106	7.32054	63.442	8.35406	7.35272	11.3219	-0.20083

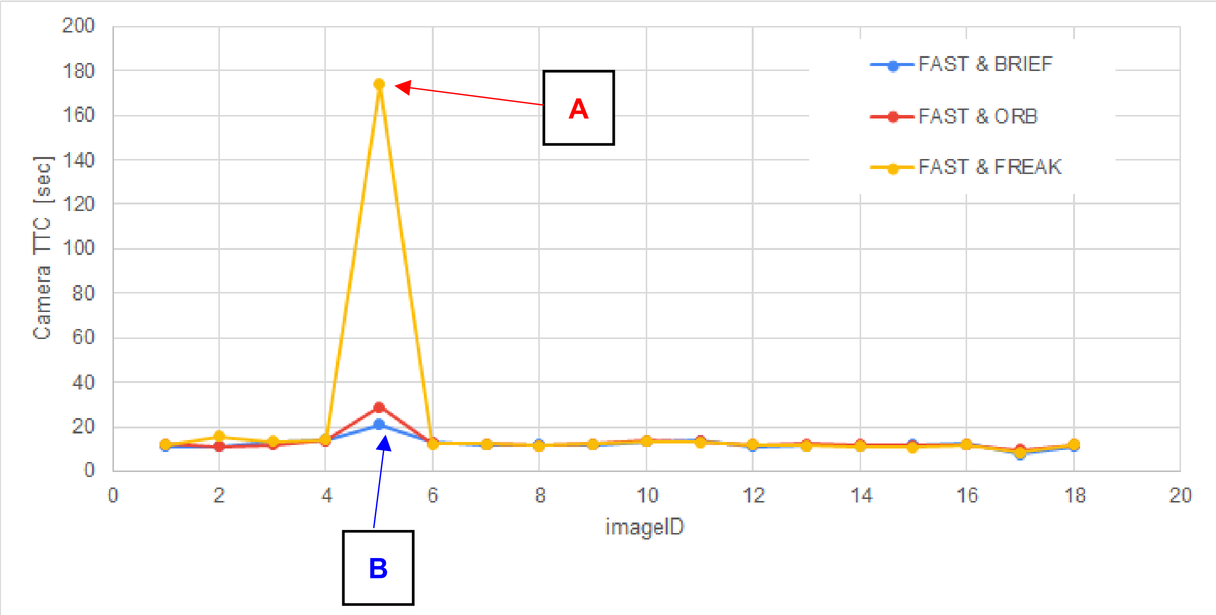
A

I saved the Excel file in "result/Project3_result.xlsx"

Finally, I will choose the best combination from 3 choices.

Detector: FAST & descriptor: FREAK has one unstable TTC, so it's not the best one. **(A)**

FAST & BRIEF and FAST & ORB are almost same, but FAST & BRIEF is a little better. **(B)**



<u>As a conclusion, "detector: FAST & descriptor: BRIEF" is the best combination in Project3.</u>