Project Specification

Track an Object in 3D Space

FP.0 Final Report
Provide a Write up / README that includes all the rubric points and how you addressed each one. You can submit your write up as markdown or PDF.

→ This Document

FP.1 Match 3D Objects
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.

FP.3 Associate Keypoint Correspondences with Bounding Boxes
Prepare the TTC computation based on camera measurements by associating keypoint correspondences to the bounding boxes which enclose them. All matches which satisfy this condition must be added to a vector in the respective bounding box.

→Write up

Both these rubric points are addressed in the following function from camFusion_Student.cpp
Line No 203 `int  matchBoundingBoxes(boost::circular_buffer<DataFrame> *dataBuffer)`
This function takes in the pointer to the dataBuffer containing prev and current dataFrame. Will iterate through the keypoint matches from current frame , extract the keypoints from the match and put them in respective bounding boxes for both current and previous frame.
Simultaneously it keeps a track of corresponding match counts in a 2D array for each bounding box from previous frame to all bounding boxes in current frame. Later each bounding box from prev frame is associated with a box from current frame where the keypoint match counts are maximum.

```
 int prev_to_curr[(dataBuffer->end()-2)->boundingBoxes.size()][(dataBuffer->end()-1)->boundingBoxes.size()];
```

FP.2 Compute Lidar-based TTC
Compute the time-to-collision in second for all matched 3D objects using only Lidar measurements from the matched bounding boxes between current and previous frame

→ Write Up
This rubric points are addressed in the following two function from camFusion_Student.cpp

```
Line No 194 float computeTTCLidar(double sensorFrameRate, BoundingBox
*prevBB,BoundingBox *currBB)

Line No 171 float IQR_median(std::vector<LidarPoint> &lidarPoints)
```

The IQR_median function uses interquartile range (IQR) method to reomve some of the the outliers in the lidarPoints data set. The median of the resulting data set is used to find the distance to the preceding vehicle.
`Below` motion model Figure 1 and Figure 2 is used to calculate the lidar based ttc. This approach gives sufficiently `reliable` results as discussed in the performance section below.

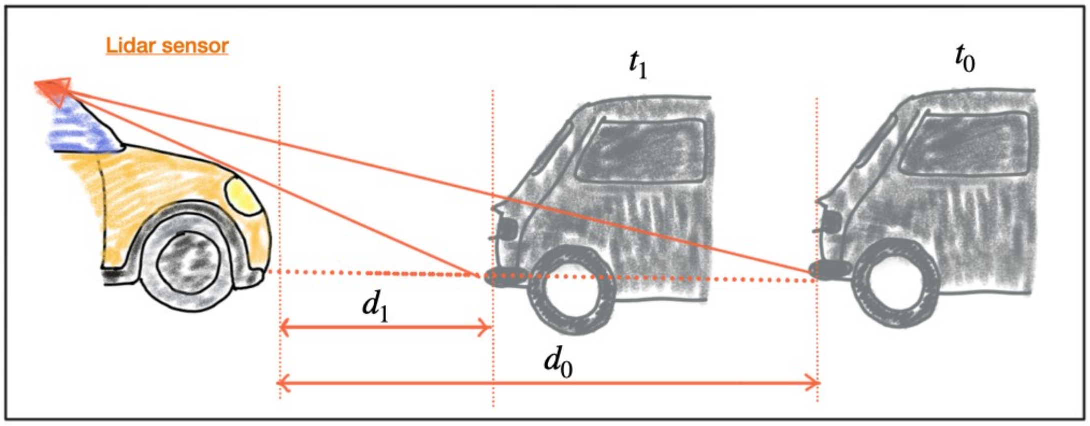


Figure 1

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

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

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

Figure 2

FP.4 Compute Camera-based TTC
Compute the time-to-collision in second for all matched 3D objects using only keypoint correspondences from the matched bounding boxes between current and previous frame.

→ Write Up
This rubric point is addressed in the following function from camFusion_Student.cpp
```
Line No 146 float computeTTCCamera(double
sensorFrameRate,boost::circular_buffer<DataFrame> *dataBuffer,BoundingBox
*prevBB,BoundingBox *currBB)

Line No 151 float IQR_median(vector<double> &distRatios)
```

Camera-based ttc algorithm used here is from the Udacity Lesson 3 “Estimating TTC with Camera”. The bounding box matches where we have lidar points are selected for ttc estimate which happens to be our ego lane.
The ratios of all the relative distances between keypoints from prev frame to current frame are calculated. The resulting dataset is passed to the IQR_median function which uses interquartile range (IQR) method to reomve some of the the outliers in the distances data set. The median of the resulting data set is returned back to **computeTTCCamera to** find the distance to the preceding vehicle. Following Figure 3 and Figure 4 camera based motion model is used to calculate the TTC.

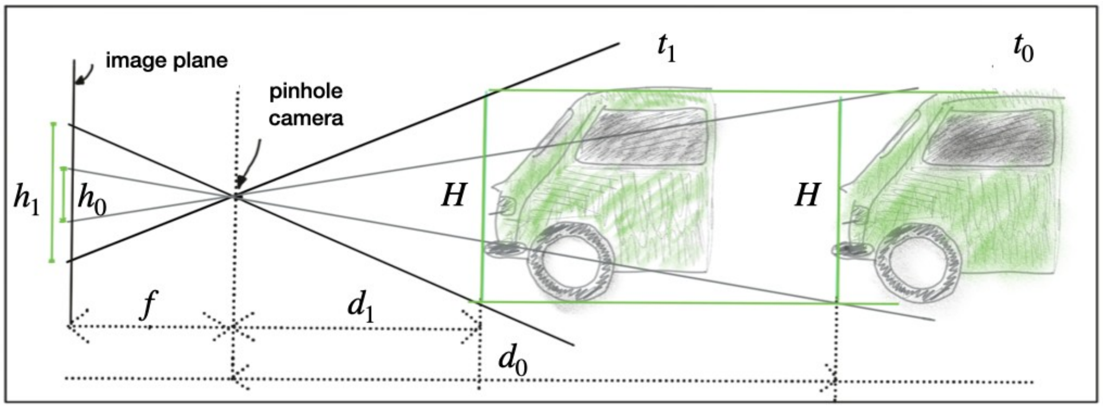


Figure 3

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

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

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

compute time to contact / collision

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

Figure 4

FP.5 Performance Evaluation 1
Find examples where the TTC estimate of the Lidar sensor does not seem plausible. Describe your observations and provide a sound argumentation why you think this happened.
FP.6 Performance Evaluation 2

Run several detector / descriptor combinations and look 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.

→ Write Up

The above algorithm is run over all the combinations of the matcher , keypoint detector and keypoint descriptor algorithms in the main function.

```
const string DetectorTypes[]=  {"SHITOMASI ","HARRIS    ", "FAST      ", "BRISK    ", "ORB       ", "AKAZE    ", "SIFT     "};
const string DescreptorTypes[]={"BRISK     ","BRIEF    ", "ORB       ", "FREAK    ", "AKAZE    ", "SIFT     "};
const string MatcherTypes[]={"MAT_BF     ","MAT_FLANN    "};
const string SelectorTypes[]={"SEL_NN     ","SEL_KNN    "};
```

The resulting data and analysis images can be found in the /Data folder. The screen output will show following information – selected combination of detector – descriptor – matcher , total processing time , camera TTC , Lidar TTC , difference and abs difference between two TTC.

```
MacherTypes SelectorTypes DetectorType descriptorType FrameNo Processing Time(ms)
TTC_Lidar(s) TTC_Camera (s) Difference (s) Abs Difference (s)
```

Analysis -

FAST Keypoint Detector -

Below Figure 5 shows the FAST keypoint detector with all the combination for descriptor and matchers over the first 50 frames. As we can see the results are very repeatable. The processing time here is quite large may be because we have a very large set of keypoints.
Figure 6 shows a zoom in the area where we have the processing time around < 50 ms. As we can see the FAST BRIEF MAT-BF SEL-KNN combination gives reliable lidar and camera based ttc estimates. The difference being around 2 seconds , but it can be as high as 5 seconds which could be high considering the expected ttc around 12 seconds.

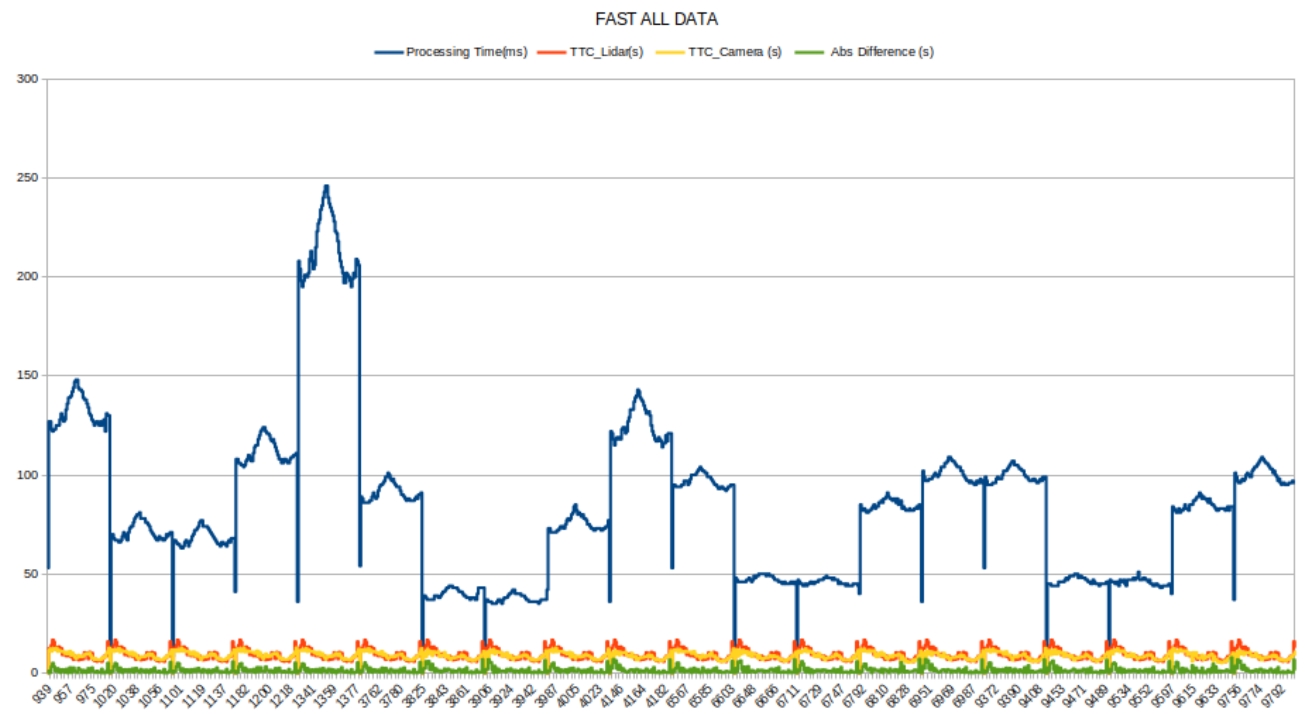


Figure 5

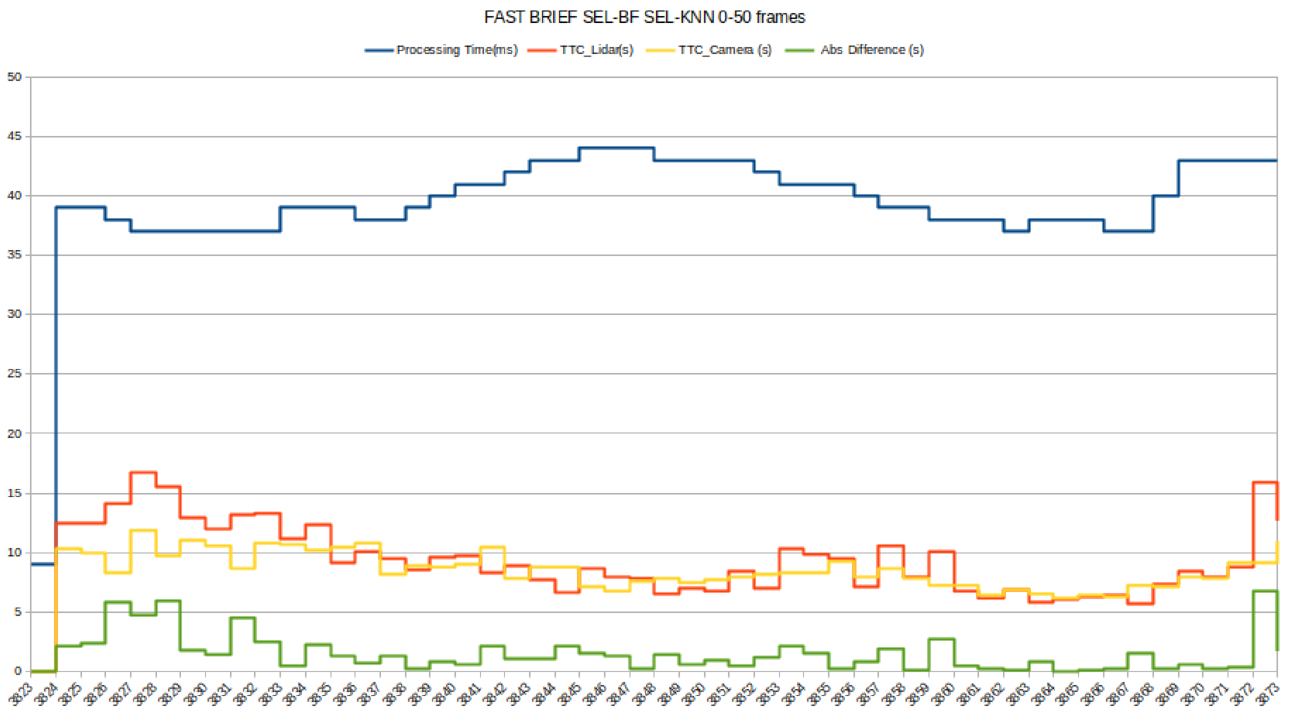


Figure 6

Other Good observations-

FAST-BRIEF (Figure 6)	<50ms	time diff <6 sec
AKAZE-BRIEF (Figure 7)	<40 ms	time diff < 6 sec
SHITOMASI-ORB (Figure 8)	<25ms	time diff <6 sec
SHITOMASI-BRIEF (Figure 9)	<20ms	time diff <6 sec

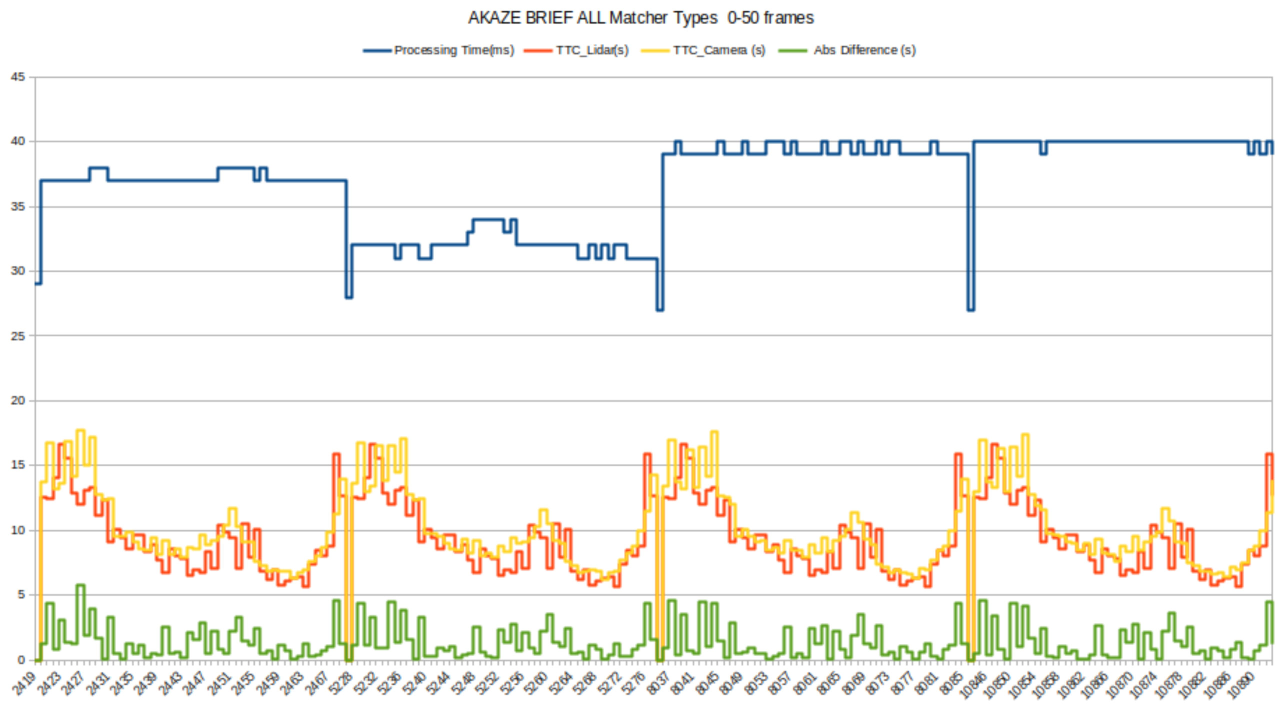


Figure 7

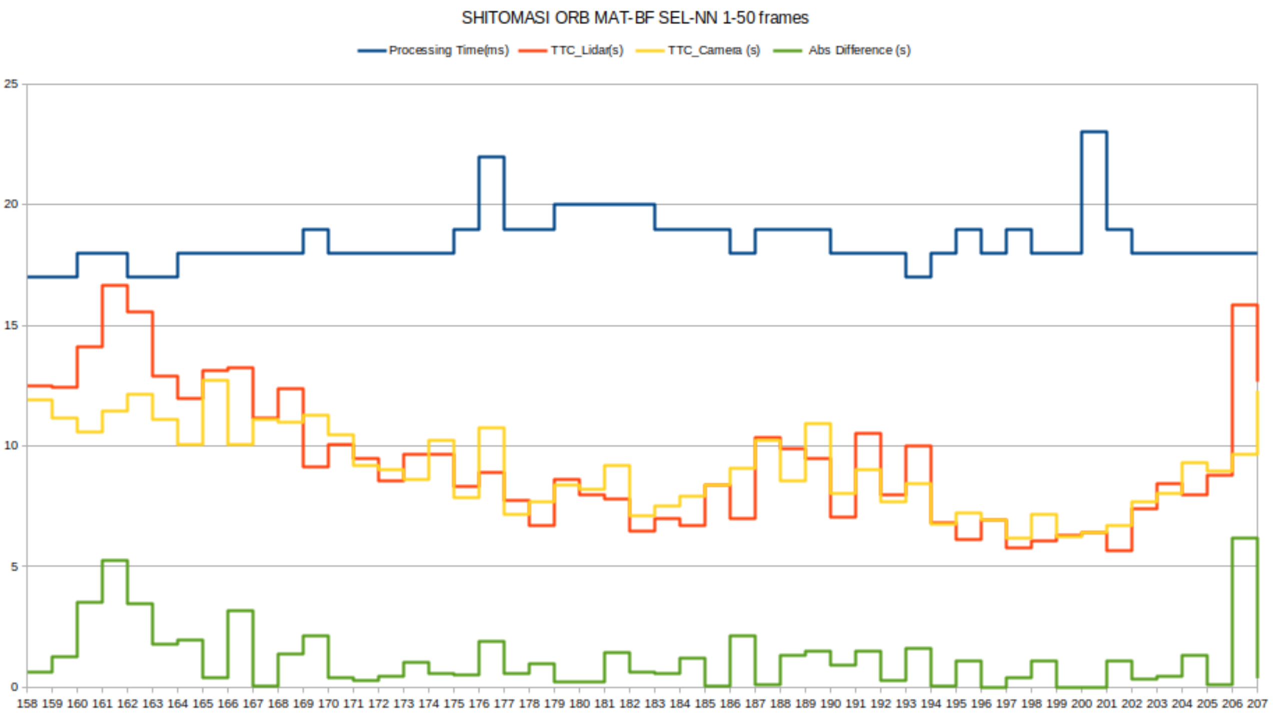


Figure 8

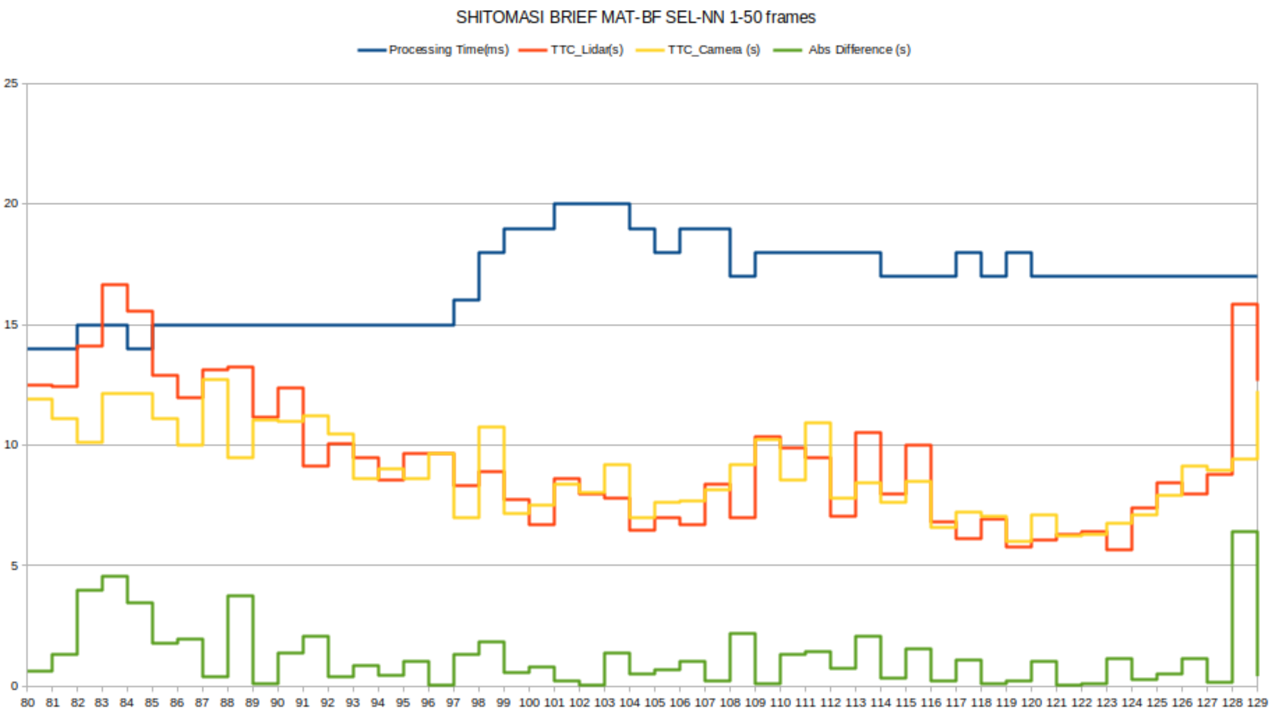


Figure 9

Unreliable Estimates -

Figure 10 11 shows the datapoints where camera-based TTC , Lidar-based TTC or both estimates are unreliable.

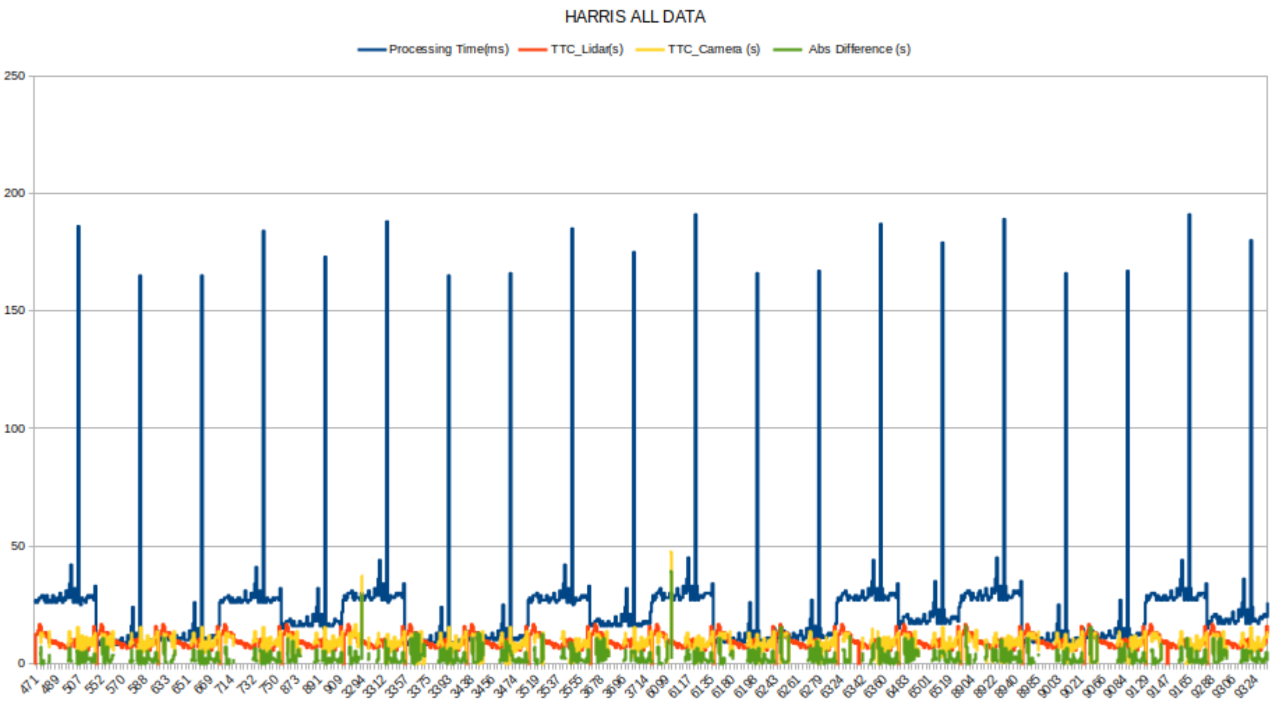


Figure 10

As we can see with Fig 10 HARRIS_ALL data there are several NAN datapoints corresponding to no keypoint matches. We conclude that HARRIS keypoint detector is algorithm not suitable for this application.

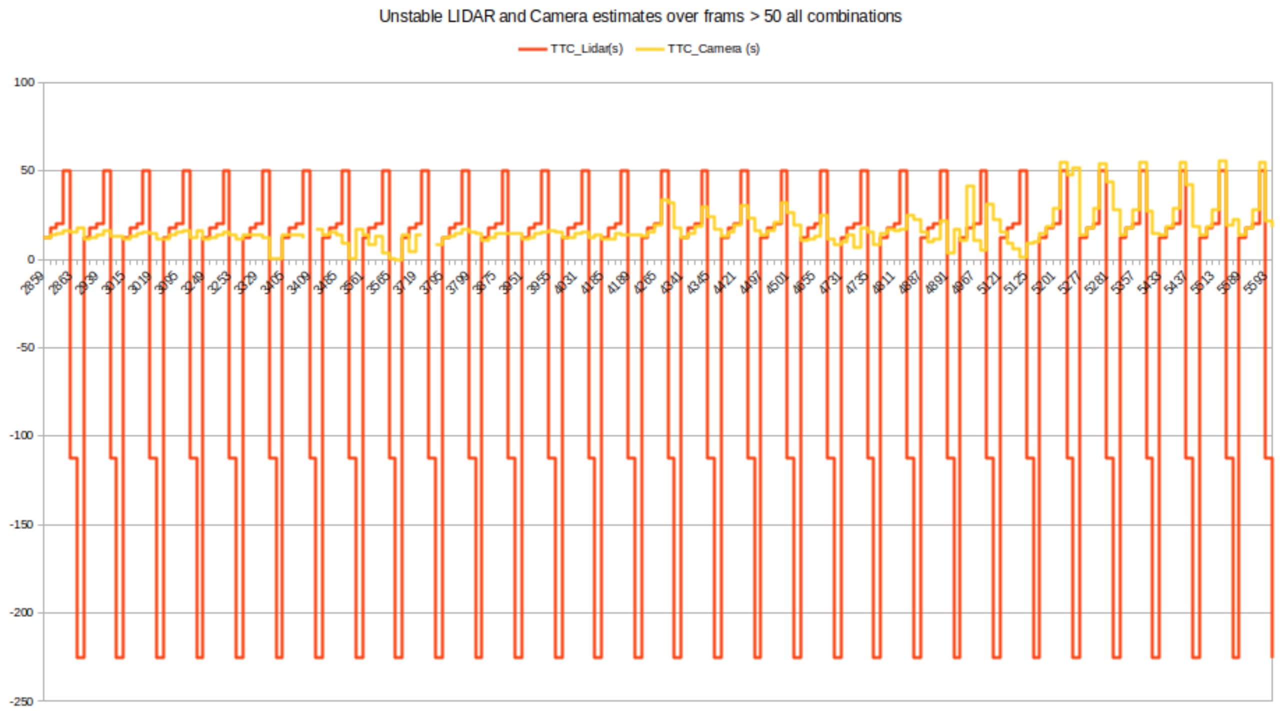


Figure 11
Data was plotted for few the frame nos > 50 for all combinations . As we can see the estimates start getting unreliable around frame nos > 50. This is true for both Lidar-based TTC and Camera-based TTC and majority of descriptor – detector combinations. Figure 11 and 12 show zoomed view of the plot over some randomly selected algorithms.

Lidar-based TTC -
The motion model used here d1 * (1.0 / sensorFrameRate) / (d0 – d1) will give negative results if the measured d1 > d0. These results are inaccurate and the motion model needs to be refined to consider this situation.
Also we see very large numbers of the TTC estimates this could be because of smaller (d0-d1) which could be accurate if the vehicles are not moving.
A closer look at the images > 50 , it seems like the preceding vehicle is too close or not moving. The estimates here seem unreliable , there could be several reasons we don’t know if the velocity estimates are accurate ? Are we already in collision? We may need to look into other sensor inputs to know for sure.
Camera-based TTC -
Similar argument applies to camera based TTC. We see that the results start becoming very unreliable after frame > 50. Figure 13 shows some randomly selected unreliable camera TTC estimates.

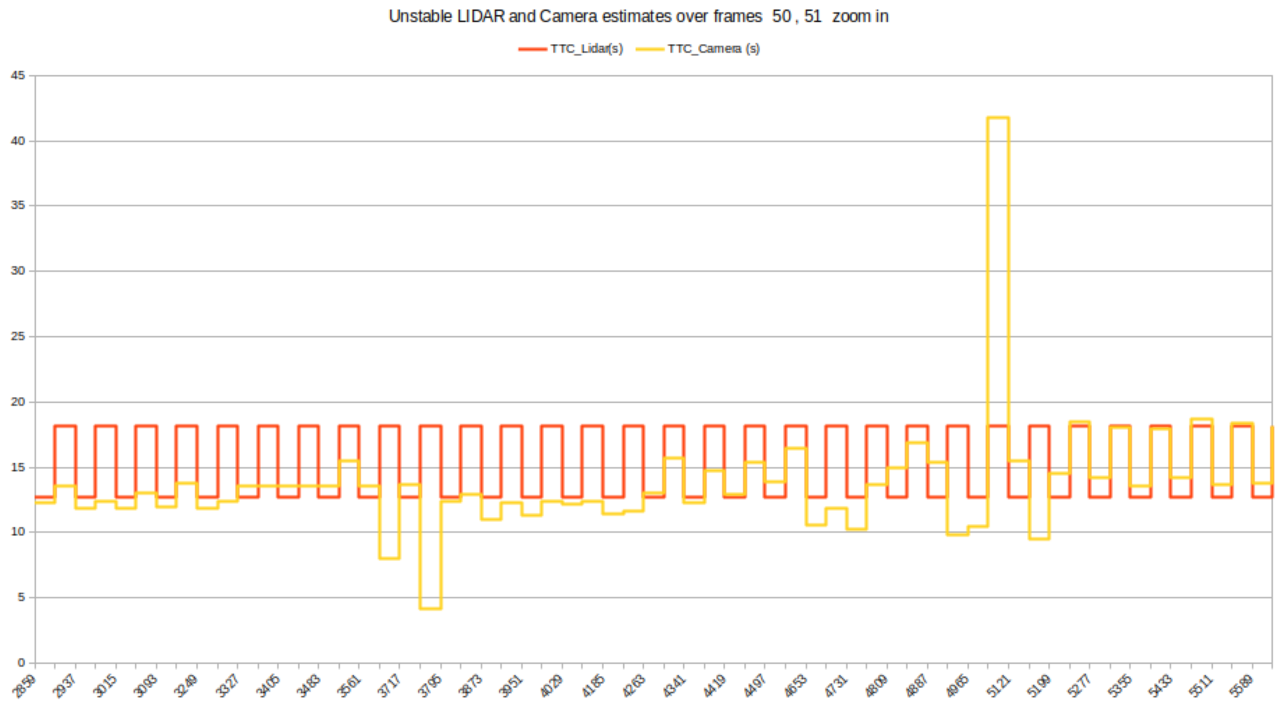
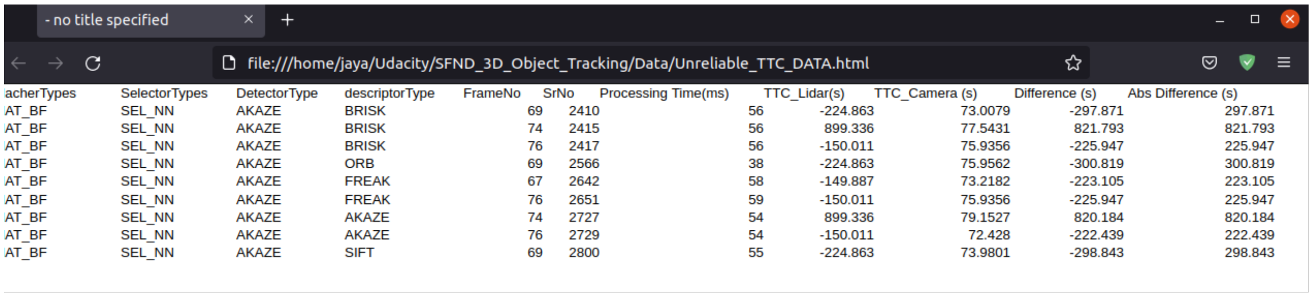


Figure 12


- no title specified
file:///home/jaya/Udacity/SFND_3D_Object_Tracking/Data/Unreliable_TTC_DATA.html

acherTypes	SelectorTypes	DetectorType	descriptorType	FrameNo	SrNo	Processing Time(ms)	TTC_Lidar(s)	TTC_Camera (s)	Difference (s)	Abs Difference (s)
AT_BF	SEL_NN	AKAZE	BRISK	69	2410	56	-224.863	73.0079	-297.871	297.871
AT_BF	SEL_NN	AKAZE	BRISK	74	2415	56	899.336	77.5431	821.793	821.793
AT_BF	SEL_NN	AKAZE	BRISK	76	2417	56	-150.011	75.9356	-225.947	225.947
AT_BF	SEL_NN	AKAZE	ORB	69	2566	38	-224.863	75.9562	-300.819	300.819
AT_BF	SEL_NN	AKAZE	FREAK	67	2642	58	-149.887	73.2182	-223.105	223.105
AT_BF	SEL_NN	AKAZE	FREAK	76	2651	59	-150.011	75.9356	-225.947	225.947
AT_BF	SEL_NN	AKAZE	AKAZE	74	2727	54	899.336	79.1527	820.184	820.184
AT_BF	SEL_NN	AKAZE	AKAZE	76	2729	54	-150.011	72.428	-222.439	222.439
AT_BF	SEL_NN	AKAZE	SIFT	69	2800	55	-224.863	73.9801	-298.843	298.843

Figure 13

Code Explanation -

```
class helper{
    ObjectDetection2D objDetector_;
    lidar Lidar_;
    double sensorFrameRate;
    int imgStepWidth = 1;
    std::vector<string>cameraFilesNames;
    std::vector<string>LidarFilesNames;

public:
    helper(std::string dataPath, int imgcount);
    int readImage(int imgIndex, boost::circular_buffer<DataFrame> *dataBuffer);

    int detectKeypoints(boost::circular_buffer<DataFrame> *dataBuffer,int detType);
    int descKeypoints_helper(boost::circular_buffer<DataFrame> *dataBuffer,  int
descType);
```

```
    int matchDescriptors_helper( boost::circular_buffer<DataFrame>  *dataBuffer,
int descType, int matcherType, int selectorType);
    int matchBoundingBoxes_helper(boost::circular_buffer<DataFrame> *dataBuffer);
    int estimateTTC(boost::circular_buffer<DataFrame> *dataBuffer, bool bVis);
};
```

This class holds the helper functions , which in turn call the functions from ObjectDetection2D and matching2D_student and camFusion_Student files.

helper(std::string dataPath, int imgcount) constructor takes in the path to the image files and the image count. It populates the two vectors with the camera and lidar file names also creates and initializes two objects for ObjectDetection2D and Lidar class.
The ObjectDetection2D class holds all the yolo parameters , functions and process the images for yolo object detection
The Lidar class holds all the parameters and functions for Lidar point projections into camera images.

int **readImage**(int imgIndex, boost::circular_buffer<DataFrame> *dataBuffer); This function reads both camera and lidar files in the circular buffer dataFrame. Also calls the yolo object detection on the camera file . The processed data is stored in dataFrame.

int **detectKeypoints**(boost::circular_buffer<DataFrame> *dataBuffer,int detType);
helper function to call the Keypoint detector functions from matching2D_student.cpp

```
    int descKeypoints_helper(boost::circular_buffer<DataFrame> *dataBuffer,  int
descType);
```

helper function to call the descKeypoints functions from matching2D_student.cpp

```
    int matchDescriptors_helper( boost::circular_buffer<DataFrame>  *dataBuffer,
int descType, int matcherType, int selectorType);
```

helper function to call the Keypoint matcher functions from matching2D_student.cpp

The

int **matchBoundingBoxes_helper**(boost::circular_buffer<DataFrame> *dataBuffer);

```
int estimateTTC(boost::circular_buffer<DataFrame> *dataBuffer, bool bVis);

The first 4 rubric points are addressed in these two functions and the explanation
can be found above.

estimateTTC function calls both the cameraTTC and LidarTTC and the values are
stored back in the dataFrame for display.
```