LIDAR AND RADAR
TASK 2: RESOLUTION COMPARISON

Aman Dubey(35066)

January 7, 2022

Abstract

In the first task we calculated the dimensions of our Vehicle that is Length, Breadth and Height. Now we need to focus on the quality of sensors which we are using. In order to do so, We have used Velodyne and Blickfield sensors dataset[1]. We observed Resolution of our Vehicle in both the sensors point cloud and predicted which one is better or denser.

Keywords
1)ADAS(Advanced driver-assistance systems)
2)LIDAR(Light Detection and Ranging)
3)ICP(Iterative Closest Point Algorithm)

1 Introduction

In real time, environment sensors acts like organs of the vehicle, Which provides Continuous data to the system. So we are highly relied on Sensors to make our System highly safe and critical. If anyone sensor provides corrupt or insufficient data, then it may lead to accident in real time environment. In ADAS(Advanced driver-assistance systems) we need to be very precise in deciding our sensors, Which not only provide accurate data but provide sufficiently large information. If we have a larger dataset, then Computation would be minimized and our system will make fewer assumptions. In this project, we are going to compare two LIDAR sensors(Blickfield and Velodyne) on the basis of their resolution. We will check for the Vehicle in the Point cloud of Both the sensors and evaluate which one provides denser point cloud.

2 Methods

The approach we took is Quite lucid, Here we have used the bounding box data to calculate corner points and after that we have implemented a 3D algorithm to calculate the number of point lying inside Cuboid(Bounding Box). Following are two flow charts illustrating the whole algorithm of Blickfield(Fig1) and Velodyne(Fig2) respectively. Please note that the Flow chart for both Velodyne and Blickfield is same except an additional step performed to calibrate both sensor system in common coordinate system, For doing it we have used Iterative Closest Point(ICP)[2][3][4] algorithm.

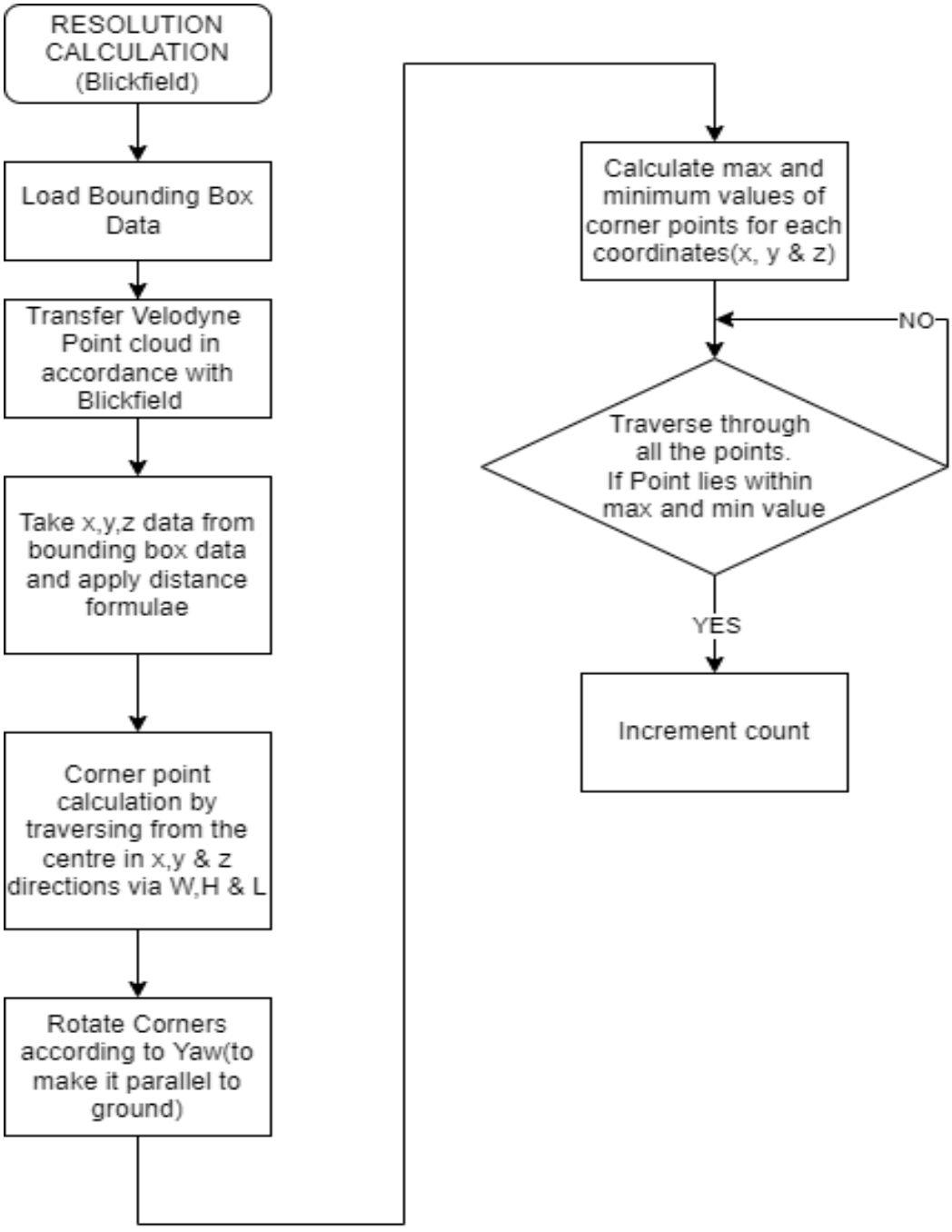

Figure 1: Number of Points inside Vehicle Bounding Box(BLICKFIELD)

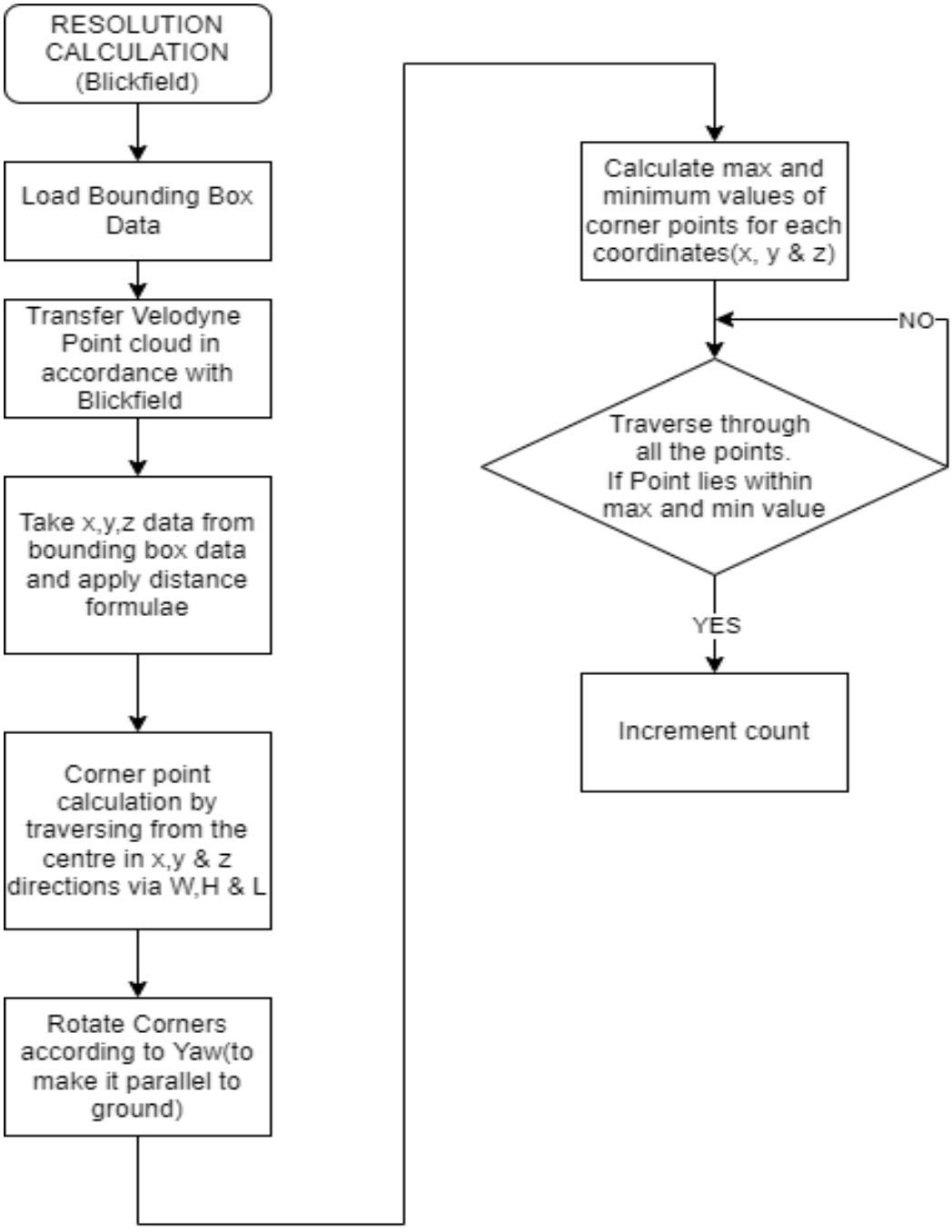

Figure 2: Number of Points inside Vehicle Bounding Box(VELODYNE)

Now we will look into Major blocks of the Flow chart, explaining their implementation as well.

Firstly we will look into distance calculation, for distance calculation(Fig 3) we used the center coordinate provided by bounding box data and applied distance formulae(Eq 1)..

$$Distance = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2} \tag{1}$$

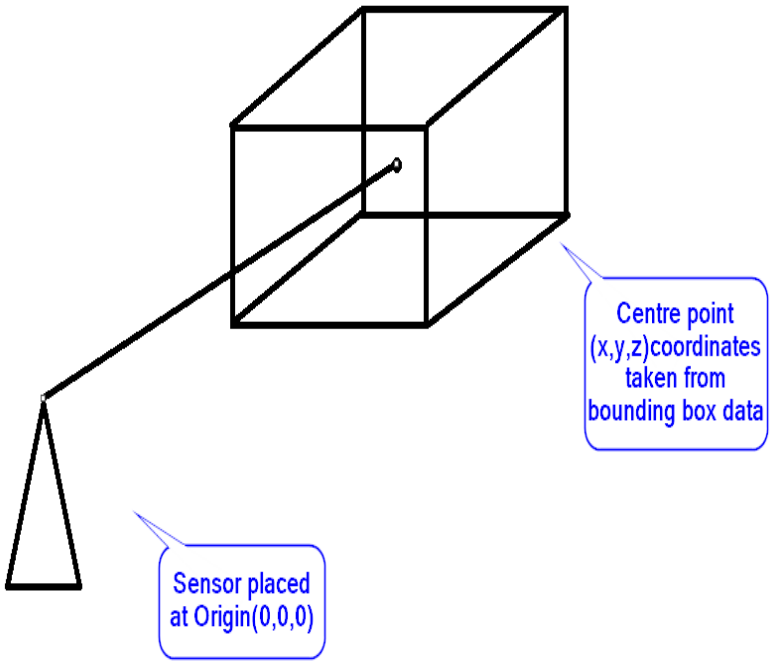

Figure 3: Distance calculation

Now take a look at the implemented code for distance calculation(see Fig 4)

Secondly, We will look into the approach we took for calculating corner points from the available Bounding Box data. As we have Bounding box data as follows

x,y,z,w,l,h,Yaw

-x,y,z represents coordinate of centre point.

-w,l,h represents width, length and height respectively.

-Yaw represents represents rotation around Z axis.

```
In [8]: ind = 20
blick = np.loadtxt(root + "Blickfeld/point_cloud/%06d.csv" % ind)
velo = np.loadtxt(root + "Velodyne/point_cloud/%06d.csv" % ind)
velo = transfer_points(velo.T[0:3], rotationmat).T

m=len(blick)
print("total number of points in blickfield = ",m)

v=len(velo)
print("total number of points in velo = ",v)

bb = np.loadtxt(root + "/Blickfeld/bounding_box/%06d.csv" % ind)
"before transferring bb into make_bounding box it represents bounding box internal values such as x,y,z,w,h etc"
"So we will take the middle point and calculate the distance from the centre point to LIDAR"
dist=((bb[0]**2)+(bb[1]**2)+(bb[2]**2))**(0.5)
print("distance = ",dist)
```

Figure 4: Distance calculation Code

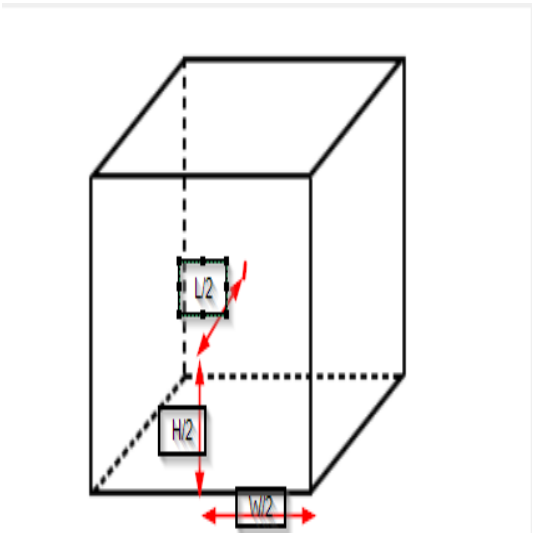

Figure 5: Corner Calculation From Centre Point Using BB Data

In Figure 5 We can see that how we traversed from centre to the corner point using dimensions of our Bounding Box. We started from centre and travel along Length(l/2) to reach the face centre, After that we travel along the Height(h/2) and then Width(w/2) along the face to reach our desired point. After that, we need to rotate the point cloud according to yaw in order to make it parallel to the ground. Code for this approach looks as follows(Fig 6):

```
In [5]: def make_boundingbox(label):
            """
                Calculates the Corners of a bounding box from the parameters.
            """
            corner = np.array([
                [+ label[3]/2, + label[4]/2, + label[5]/2],
                [+ label[3]/2, + label[4]/2, - label[5]/2],
                [+ label[3]/2, - label[4]/2, + label[5]/2],
                [+ label[3]/2, - label[4]/2, - label[5]/2],
                [- label[3]/2, + label[4]/2, + label[5]/2],
                [- label[3]/2, - label[4]/2, + label[5]/2],
                [- label[3]/2, + label[4]/2, - label[5]/2],
                [- label[3]/2, - label[4]/2, - label[5]/2],
            ])
            print(label[0])
            print(label[1])
            print(label[2])
            corner = transfer_points(corner.T, rt_matrix(yaw = label[6])).T
            corner = corner + label[0:3]
            return corner
```

Figure 6: Code for Corner Calculation From Centre Point Using BB Data

We have distance and corner points of the bounding box, Now we are going to calculate number of points inside our BB. For doing so we need to calculate maximum and minimum values of corner coordinates in x, y and z axis. We will use bb.max(axis=0) and bb.min(axis=0) to calculate maximum and minimum values, respectively(See Fig 7).

Finally, we will use this maximum and minimum values of corner points and check whether the point lies within the cuboid(Bounding Box). For checking this, we will check whether our point of concern has its values within maximum and minimum values of Bounding Box corner point. For Example, we have a point with x, y, z as coordinate. Now we are going to code(see fig 8) it as follows:
IF point lies inside Xmin $\leq x \leq Xmax, Ymin \leq y \leq Ymax, Zmax \leq z \leq Zmax$.

```
bb = make_boundingbox(bb) #Computing corners of Bounding Box and storing inside bb
print(np.array(bb))       # Printing array just to check
bbmaxarray=bb.max(axis=0) # Taking maximum values from each coloumn
bbminarray=bb.min(axis=0) # Taking minimum values from each coloumn
print(bbmaxarray[0])      ##Just Printing values to check whether all stuff is correct or not ##
print(bbmaxarray[1])
print(bbmaxarray[2])
print(bbminarray[0])
print(bbminarray[1])
print(bbminarray[2])
"now we have two arrays of max and min value(corner points)"

"We will do the same stuff for Velodyne data"
```

Figure 7: Code for max and min value calculation of corner points

```
In [9]: blick
        print(blick[0][1])
        print(blick[1][0])
        print(blick[2][2])

        n = 0       #Loop iteration variable
        count = 0   #points inside BB
        ## We are having max and min value(corner) for x,y and z axes. Now we will iterate the loop for all points in the point cloud
        ## and check for the points occuring inside our BB  Xmax <=X <=Xmin and so on.....y...z...##
        while n != m:
            if bbmaxarray[0] >= blick[n][0] >= bbminarray[0] and bbmaxarray[1] >= blick[n][1] >= bbminarray[1] and bbmaxarray[2] >= blick
              count += 1


            n += 1

        print("Number of times loop iterated for blickfield = ",n)
        print("Resolution(blickfield) =  ",count,"with Index = ",ind,"at a distance of(in m)= ",dist)
```

Figure 8: Code for calculating Resolution

3 Results and Analysis

As we know that, our objective is to determine the quality of the sensor by calculating Resolution of a vehicle for both of them. Graphs were created by storing distance and number of Points in a matrix, Code looked like as follows:

```
dfB = pd.DataFrame(dataB, columns=['distanceB','pointB'])   #Providing names to coloumns of blickfield data#

dfV = pd.DataFrame(dataV, columns=['distanceV','pointV'])   #Providing names to coloumns of Velodyne data#
```

Figure 9: Providing names to columns of Matrix

In fig9 dataB and dataV represents Matrix of Blickfiled and Velodyne respectively.

```
In [23]: #print(np.array(dataB))
         px.scatter(data_frame = dfB, x ='distanceB', y ='pointB')
         #print(np.array(dataV))
         px.scatter(data_frame = dfV, x ='distanceV', y ='pointV')
```

Figure 10: Scatter function for plotting graph

Now we are ready with our result, They are shown below via two graphs.

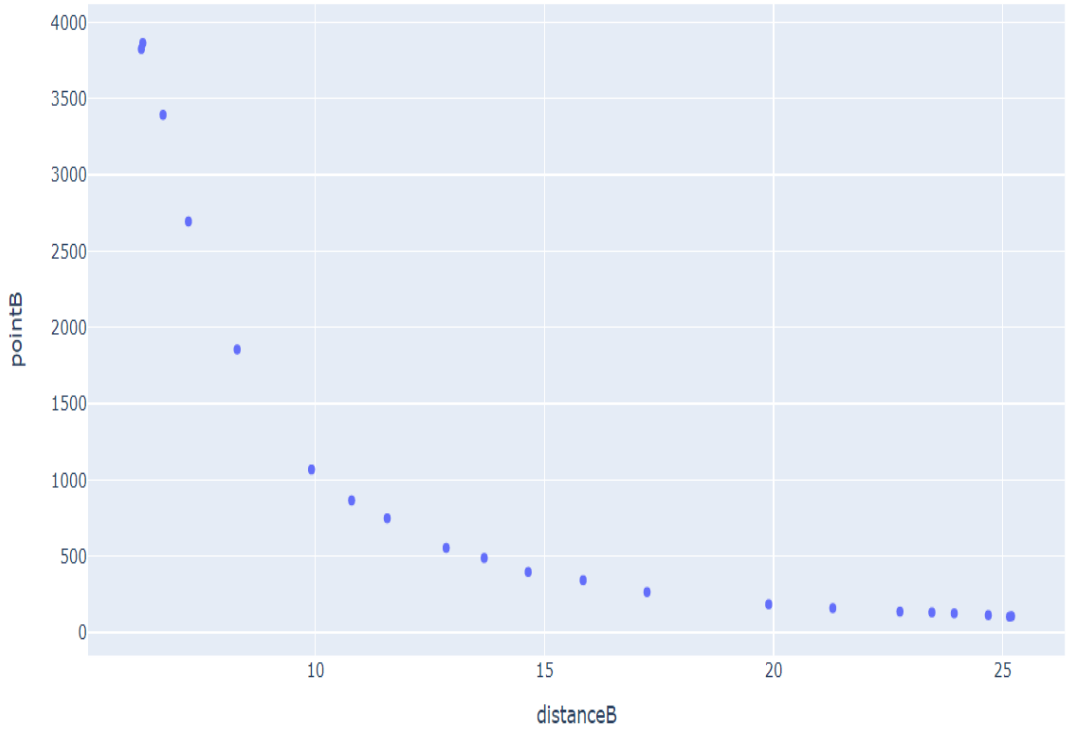

Figure 11: Distance vs Number of Points(BLICKFIELD)

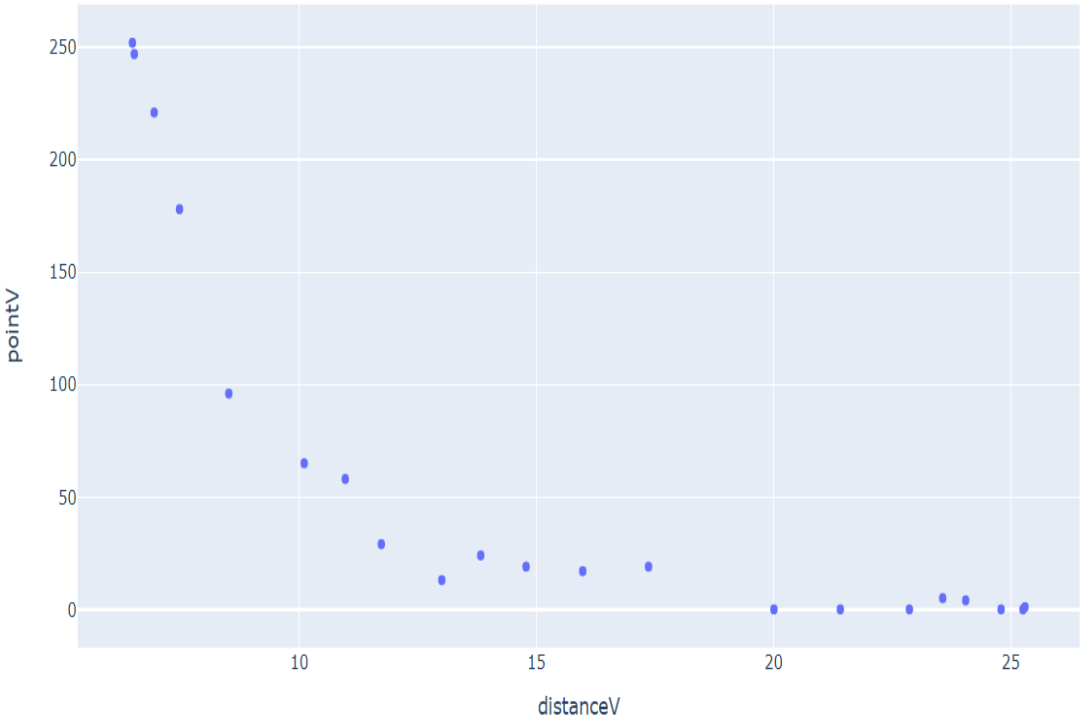

Figure 12: Distance vs Number of Points(VELODYNE)

From the above two graphs, it is clear that our resolution decreases as we move away from the sensor. But when we look at the resolution for both Sensors, it becomes very clear that Blickfield Provides much higher resolution. For example take the resolution for both the sensor at 6 m distance, Blickfield Provides the resolution around 3900 whereas Velodyne Provides around 250 points. So Blickfield approximately has 15 times resolution as compared to Velodyne. Blickfield provides a denser point cloud and the object under it is more clearly visible.

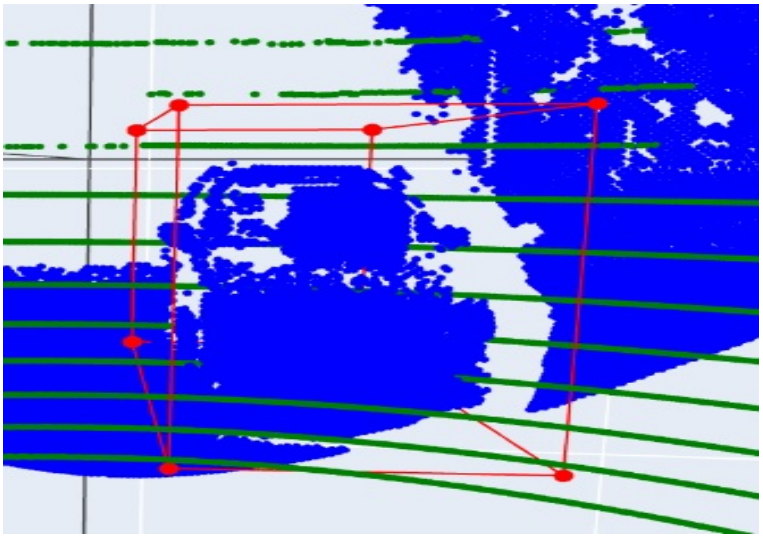

Figure 13: Bounding Box for Blickfield Sensor(Index 18

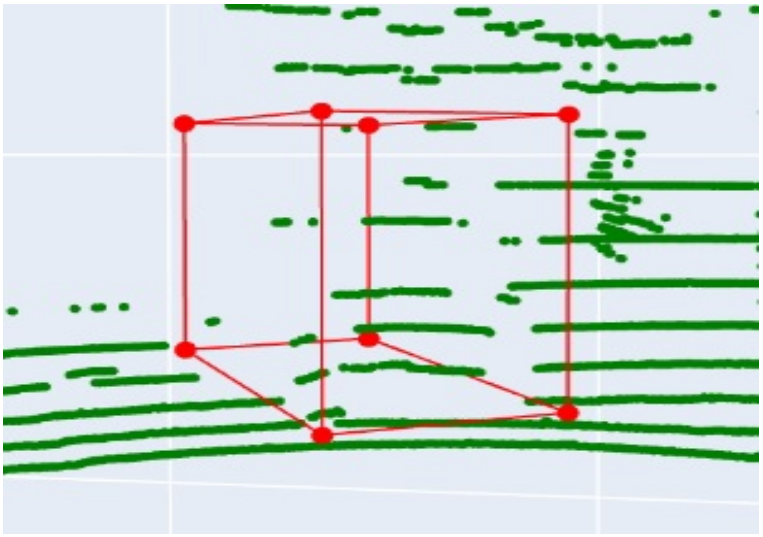

Figure 14: Bounding Box for Velodyne Sensor(Index 18)

4 Conclusion

So Finally we can conclude that our algorithm can be used to determine the resolution provided by any Lidar sensor having bounding box data and point cloud. Among the two sensor, we can clearly say that Blickfield is the winner, As Blickfiled offers far denser point cloud which will help in creating robust and highly safe ADAS systems.

Appendix: Number of Points Observed in Bickfield and Velodyne

Below are the two table showing number of Points and Distance for Blickfield and Velodyne sensors respectively.

distance (m)	Points
25.1958117	106
25.15063452	103
24.68810151	113
23.94489309	125
23.45705783	131
22.75946532	136
21.29271098	159
19.89466719	184
17.23836665	264
15.84302953	342
14.64364158	396
13.6839327	488
12.85383207	554
11.56641522	748
10.78909556	865
9.914146319	1068
8.290619536	1855
7.226611473	2692
6.67276879	3370
6.230818691	3820
6.199546458	3783

(a) BLICKFIELD

distance (m)	Points
25.30006677	1
25.25719405	0
24.79673652	0
24.04999569	4
23.5651442	5
22.86344619	0
21.40761522	0
20.00791366	0
17.36411134	19
15.9766887	17
14.7831898	19
13.82710381	24
13.00685278	13
11.73236819	29
10.97163934	58
10.10633704	65
8.516400035	96
7.477348861	178
6.943569903	221
6.524260975	247
6.484844208	252

(b) VELODYNE

Table 1: Resolution vs Distance Table for both sensors

References

[1] Felix Berens and Saravanan. Lidar dataset at rwu.

[2] Yang Chen and Gérard Medioni. Object modelling by registration of multiple range images. *Image and vision computing*, 10(3):145–155, 1992.

[3] Szymon Rusinkiewicz and Marc Levoy. Efficient variants of the icp algorithm. In *Proceedings third international conference on 3-D digital imaging and modeling*, pages 145–152. IEEE, 2001.

[4] Xinshuo Weng and Kris Kitani. Monocular 3d object detection with pseudo-lidar point cloud. In *Proceedings of the IEEE/CVF International Conference on Computer Vision Workshops*, pages 0–0, 2019.