RWU Hochschule Ravensburg-Weingarten University of Applied Sciences

Faculty for Electrical Engineering and Computer Science

PROJECT REPORT: TASK - 1

Lidar Data Comparison

Guided by:

Dr. Professor Stefan Elser

*S*ubmitted by:

Prasad Sonani-36202

Master-Electrical Engineering and Embedded Systems

Akshay Chobe-36316

Master-Mechatronics

February 20, 2024

Contents

Chapter 1

Introduction

1.1 Introduction

In this study, Velodyne Puke and Blickfeld Cube 1 lidar sensors' data is measured, the findings are compared, and a graph of the distance versus the number of points inside the bounding box is plotted for each sensor, with the distance on the X-axis and the number of points on the Y-axis. In this case, we created a graph from a dataset using Jupyter notebook for this calculation. The car moves from forward to reverse using the first 0 to 120 frame until it reaches its greatest backward position, at which point it starts driving forward once more till 120 to 239 frame.

First of all, the X, Y, and Z coordinates are known to calculate the number of points within the bounding box. These coordinates represent the eight corners of the bounding box, with (x1,y1,z1), (x2,y2,z1), (x3,z3,y2), (x4,y4,z2) for each corner upto (x8,y8,z8). For various distances, the number of points inside the bounding box are computed by considering the maximum and minimum values of the (X,Y,Z) coordinates and then applying the iteration point algorithm. Alternatively, if the bounding box is not parallel to the origin, we must perform a yaw rotation on the coordinate system to align the bounding box with the origin. The concept of yaw rotation is also employed here.

Chapter 2

Measurements

2.1 Distance Measurement

The manual calculation of the distance measurement involves selecting various points and employing the center of bounding box or center of object, which are written as xc, yc and zc. Position of our sensors is at the origin=(0,0,0). We calculate distance between center of object and origin is by using distance formula,

$$\text{distance} = \sqrt{(xc-0)^2+(yc-0)^2+(yc-0)^2}$$

```
Darr = []            # Array of distance value for all frame_id.

"""For distance from the origin"""

center=(xc,yc,zc)
origin=(0,0,0)
distance= math.sqrt( ((xc-0)**2)+((yc-0)**2)+((zc-0)**2) )
Darr.append(distance)                          # Update distance array by adding value of 'distance' of object.
```

Figure 2.1: Distance Measurement Algorithm

Each value of distance for every frame-id stored in Array of distance= Darr.

2.2 Inverse of Rotation Matrix

Bounding Box has 8 corners. Here, we are considering C1=(x1,y1,z1) and C8=(x2,y2,z2) as opposite to each other and with respect to them we are able to find coordinate of other 6 corners. Center of bounding box=(xc,yc,zc).

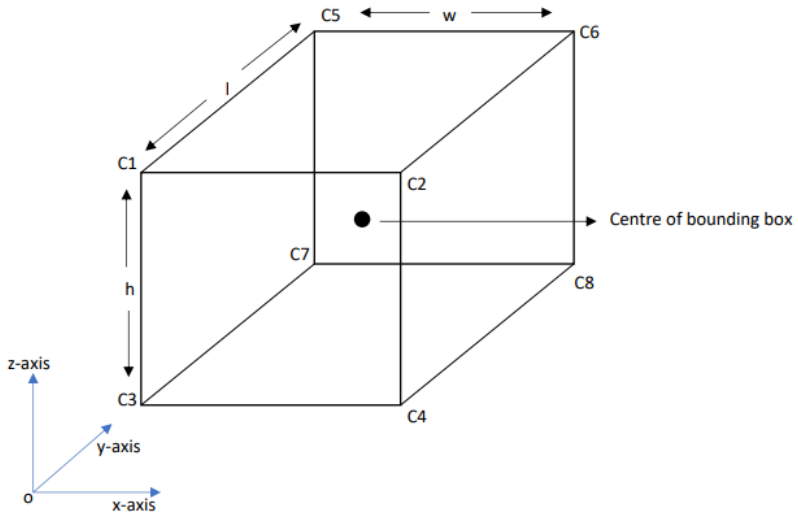

Figure 2.2: Bounding Box

c1(x1,y1,z1)	c2(x2,y2,z2)	c3(x3,y3,z3)	c4(x4,y4,z4)
x1= xc - w/2	x2= xc + w/2	x3= xc - w/2	x4= xc + w/2
y1= yc - l/2	y2= yc - l/2	y3= yc - l/2	y4= yc - l/2
z1= zc + h/2	z2= zc + h/2	z3= zc - h/2	z4= zc - h/2

c5(x5,y5,z5)	c6(x6,y6,z6)	c7(x7,y7,z7)	c8(x8,y8,z8)
x5= xc - w/2	x6= xc + w/2	x7= xc - w/2	x8= xc + w/2
y5= yc + l/2	y6= yc + l/2	y7= yc + l/2	y8= yc + l/2
z5= zc + h/2	z6= zc + h/2	z7= zc - h/2	z8= zc - h/2

so,

x1=x3=x5=x7;	x2=x4=x6=x8;
y1=y3=y5=y7;	y2=y4=y6=y8;
z1=z3=z5=z7;	z2=z4=z6=z8.

A value for yaw, expressed in degrees, depicts how the bounding box is rotated around the Z-axis. When yaw is positive, the bounding box rotates in a clockwise direction, and when yaw is negative, the rotation is counterclockwise. Here, we are using label[6] to store the value from the supplied data set. The number "6" indicates the location of the yaw value storage.

We must rotate the point cloud so that the bounding box's edges are remain same to the axis in order to accurately measure the number of points from each sensor inside the bounding box. With each point and piece of data, we require inverse rotation matrix multiplication. rot= rotation matrix and rotinverse= inverse rotation marix.

```
"""For minimal and Maximum value of BB corners in every direction"""

yaw= label[6]
Zmin= c8[2] = c3[2] = c4[2] = c7[2]  #minimum value of corner in Z direction.
Zmax= c1[2] = c2[2] = c5[2] = c6[2]  #maximum value of corner in Z direction.
Xmin= c1[0] = c3[0] = c5[0] = c7[0]  #minimum value of corner in x direction.
Xmax= c8[0] = c2[0] = c6[0] = c4[0]  #maximum value of corner in x direction.
Ymin= c1[1] = c2[1] = c3[1] = c4[1]  #minimum value of corner in y direction.
Ymax= c8[1] = c5[1] = c6[1] = c7[1]  #maximum value of corner in y direction.
```

Figure 2.3: maxima and minima

```
yaw=label[6]                          # yaw/z-axis rotation in degree.
roll=0
pitch=0

yaw = yaw * np.pi / 180               # convert degree to radian
roll = roll * np.pi / 180
pitch = pitch * np.pi / 180
c_y = np.cos(yaw)
s_y = np.sin(yaw)
c_r = np.cos(roll)
s_r = np.sin(roll)
c_p = np.cos(pitch)
s_p = np.sin(pitch)

# Rotationmatrix
rot = np.dot(np.dot(np.array([[c_y, - s_y,   0],          # rot=rotation marix(row=3,column=3)
                              [s_y,   c_y,   0],
                              [0,      0,    1]]),
                    np.array([[c_p,    0,    s_p],
                              [0,      1,    0],
                              [-s_p,   0,    c_p]])),
                    np.array([[1,      0,    0],
                              [0,    c_r, - s_r],
                              [0,    s_r,   c_r]]))

rotinvers=  np.linalg.inv(rot)
```

Figure 2.4: Inverse of rotation matrix

2.3 Intersection Algorithm

To provide an exact result, the intersection algorithm rotates the axes or moves the position of the monument. It aids in finding new coordinates without shifting an object's position by allowing rotation of points along all axes.

```
pc_blick
Barr = []            # Array for number of blickfeld sensor points inside BB=bounding box for all frame_id.
blick_TNP=len(pc_blick)
pc_blick=np.delete(pc_blick,3,axis=1)   # blick_TNP = Total number of blickfeld points in frame.

"""For counting number of BLICKFELD points/data inside BB"""

blick_TNP_I_BB = 0                    # total number of blickfeld points in BB
Rbc=[]                                # data of bleckfeld point cloud after rotation
del Rbc[:]
i = 0

for i in range(blick_TNP):
    Rbcp= np.array(np.dot(rotinvers,pc_blick[i]))        # I NUMBER OF POINT OF BLICKFELD DATA SET.
    Rbc.append(Rbcp)
    vRbc=Rbc[i]                                       # store value of i number of point from the array Rbc.
    xb=vRbc[0]                                        # i(x,y,z)=(xb,yb,zb)
    yb=vRbc[1]
    zb=vRbc[2]

    if Xmax >= xb >= Xmin and Ymax >= yb >= Ymin and Zmax >= zb >= Zmin:
        blick_TNP_I_BB += 1

    i+=1
Barr.append(blick_TNP_I_BB)             # Update blickfeld array by adding value of 'blick_TNP_I_BB' of frame.
```

Figure 2.5: Inside the bounding box, blickfeld points are counted..

The code for counting the number of blickfeld points inside each bounding box is illustrated in Figure 2.5. It stores the results into the array Barr=Array for storing the values of the number of points according to the frame-id. After being multiplied by the inverse of the rotation matrix, the data from the blickfeld point cloud is stored in the array 'Rbc'. Only points with x, y, and z coordinates between the bounding box's Xmin and Xmax, as well as y, z, and coordinates between those values, are tallied.

In Figure 2.6 The same pattern and algorithm were used to measure the velodyne points inside the frame's bounding box. Data from before rotation is contained in variable 'pc-velo', and point cloud data from the 'Rve' array has been multiplied by the rotation's inversion. Here, Varr is an array that can carry the number of velodyne points that are contained inside a box according to the frame.

Now, we have 3 array which have values of all 240 frames like,

Darr = Array of distance value for all according frame .

Barr = Array for number of blickfeld sensor points inside BB=bounding box for all frame.

Varr = Array for number of velodyne sensor points inside BB for all frame.

```
Varr = []            # Array for number of velodyne sensor points inside BB for all frame_id.
pc_velo
velo_TNP=len(pc_velo)                  # velo_TNP = Total number of Velodyne points in frame.
pc_velo=np.delete(pc_velo,3,axis=1)

"""For counting number of VELODYNE points/data inside BB"""

velo_TNP_I_BB = 0                      # total number of velodyne points in BB
Rve= []
i = 0
for i in range(velo_TNP):
   Rvep= np.array(np.dot(rotinvers,pc_velo[i]))
   Rve.append(Rvep)
   xv=Rvep[0]
   yv=Rvep[1]
   zv=Rvep[2]

   if Xmax >= xv >= Xmin and Ymax >= yv >= Ymin and Zmax >= zv >= Zmin:
      velo_TNP_I_BB += 1

   i+=1
Varr.append(velo_TNP_I_BB)              # Update velodyne array by adding value of 'velo_TNP_I_BB' of frame.
```

Figure 2.6: Inside the bounding box, velodyne points are counted

Chapter 3

Results and Conclusion

3.1 Lidar Comparison

After carefully examining all the data, it is obvious that the results from Blickfeld and Velodyne sensor are very different from one another. Our last theory is that it has to do with the Blickfeld's ability to measure distances up to 250 meters whereas the Velodyne can only measure distances up to 100 meters. Readings that are more precise than Blickfeld's are the one exception to this length restriction. The Velodyne was probably positioned very near to the vehicle when the dataset was measured from the left side view. Blickfeld just covers the front of the vehicle body, whereas Velodyne covers the remainder of the surrounding region.

```
import matplotlib.pyplot as plt

Darr [:]
Barr [:]
Varr [:]
plt.scatter(Darr,Barr,color='blue',s=20,marker="o")
plt.ylabel('No.points')
plt.xlabel('Distance (m)')
plt.title('BLICKFELD SENSOR')
plt.grid()
```

Figure 3.1: Ploting graph

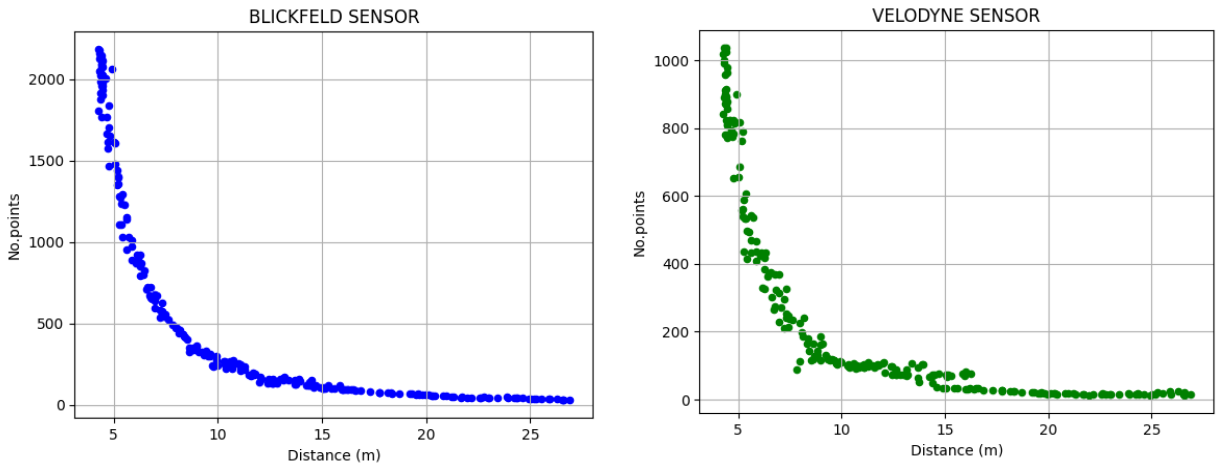

Figure 3.2: Graphs for point cloud of Blickfeld (left) and Velodyne (right) Sensor

A key piece of information was the fact that both sensors produced graphs with downward exponential tendencies. The number of points enclosed within the bounding boxes, however, unexpectedly changes as the distances are altered, as shown in Figure 3.2 for the blickfeld sensor and the velodyne sensor. For instance, at a distance of 6.99 meters, the velodyne sensor only picks up about 370 points while the blickfeld sensor detects roughly 678 points within the bounding box. This is because the blickfeld sensor has a higher resolution than the velodyne sensor. However, blickfeld scored approximately 2200 points at a distance of 4 meters, whereas velodyne scored only around 1000 points.

3.2 Conclusion

The Blickfeld cube and the Velodyne puck are both helpful tools for dimensioning in a broader sense. Even if Blickfeld data is more accurate over long distances, Velodyne lidar is still useful in some situations. Both have inherent restrictions that need to be considered. Both are imperfect. Velodyne performs better at close range than Blickfeld, and the device's tiny footprint enables it to produce the most accurate results. Additionally, Blickfeld's pricing scheme is less aggressive than Velodyne's.

The fact that the blickfeld sensor produces 3 times as many points as the velodyne sensor finally proves that it is more accurate. Additionally, it has been noted that as the distance between the car and the origin (the lidar sensor) grows, the number of points at a given distance drops, and vice versa when the distance lowers. Close examination of

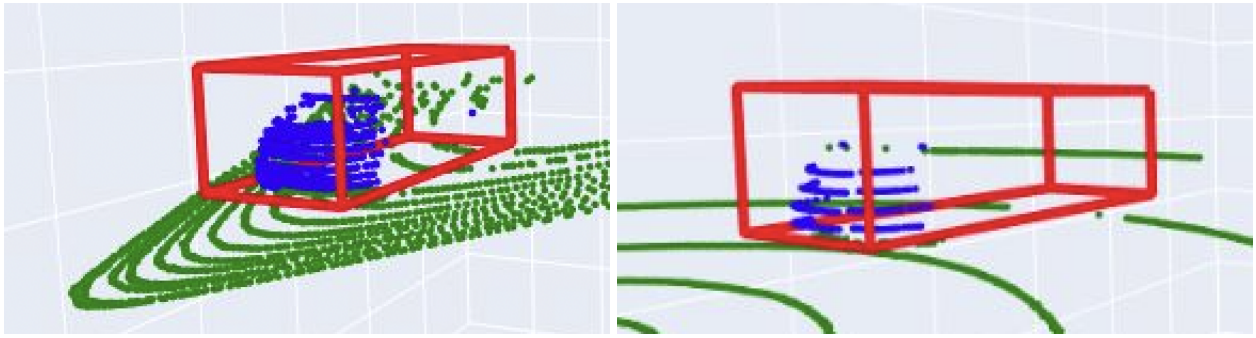

Figure 3.3: Frame ID 120: Blickfeld (left) and Velodyne (right) Sensor

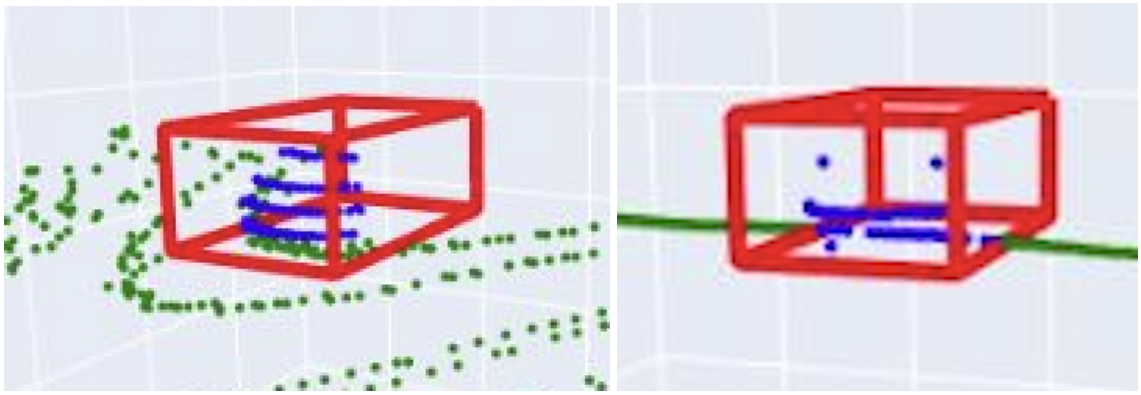

Figure 3.4: Frame ID 60: Blickfeld (left) and Velodyne (right) Sensor

both graphs reveals that the blickfled graph has a superior advantage over the velodyne graph. There is a 90 percentage chance that the graph for the blickfeld sensor is strictly exponential if a line is drawn between all of the points, given that the ratio of the number of points to the total number of points is roughly 3:1.