DLAD Project 1

Understanding Multimodal Driving Data

Nicolas Hoischen

Student ID: 16-819-880

Dario Bolli

Student ID: 16-832-685

FS 2021

227-0560-00L: Deep Learning for Autonomous Driving

Eidgenössische Technische Hochschule Zürich

Due Date: 21-03-2021

1 Problem 1: Bird's eye view

To display the BEV image we need to extract the x and y coordinates as well as the reflectance value of the point cloud. So we start by extracting these parameters from the dictionary using the key 'velodyne'. Since the final result is expected to be an image, the resolution specification of 0.2m will help us map the points position to pixel positions. Indeed, if we would directly cast x and y from the point cloud into integers, each pixel in the image would represent a square of 1m x 1m in the liDAR coordinates. Therefore, we can start by dividing the x and y coordinates of each points by the resolution i.e. 0.2m.

To display the final image, we have chosen to use the python PIL library. This library uses a cartesian coordinates system with (0, 0) in the upper left corner and pixel coordinates are passed as tupple (x,y). Therefore given the lidar coordinates with x pointing in the forward direction and y in the leftward direction, we need to translate x and y to pixel coordinates. In the following part of the code we express the coordinates (x,y) from the point cloud in the image coordinates i.e. x is mapped to -x and y to -y. Moreover we also take into account the 0.2m resolution requirement before converting it to pixel indices, which are integers.

```
bev_x = (-x_velodyne/res).astype(np.int32) #bins of 0.2 m resolution
bev_y = (-y_velodyne/res).astype(np.int32)
```

The last step shifts the minimum pixel indices x and y calculated before and shift it to (0,0) since the pixel indices have to be positive integers. Thus the image is also centred.

```
bev_x -= np.amin(bev_x)
bev_y -= np.amin(bev_y)
```

Intensity values for each bins or pixel with coordinates (x,y) are computed using:

$$reflectance \times 255$$

since the reflectance value is in $[0, 1]$.
Finally, we assign the highest intensity value if we have multiple velodyne points mapped to the same bin i.e. same pixel.

Below is the result image for data.p, rotated by 90°for visualization purposes:

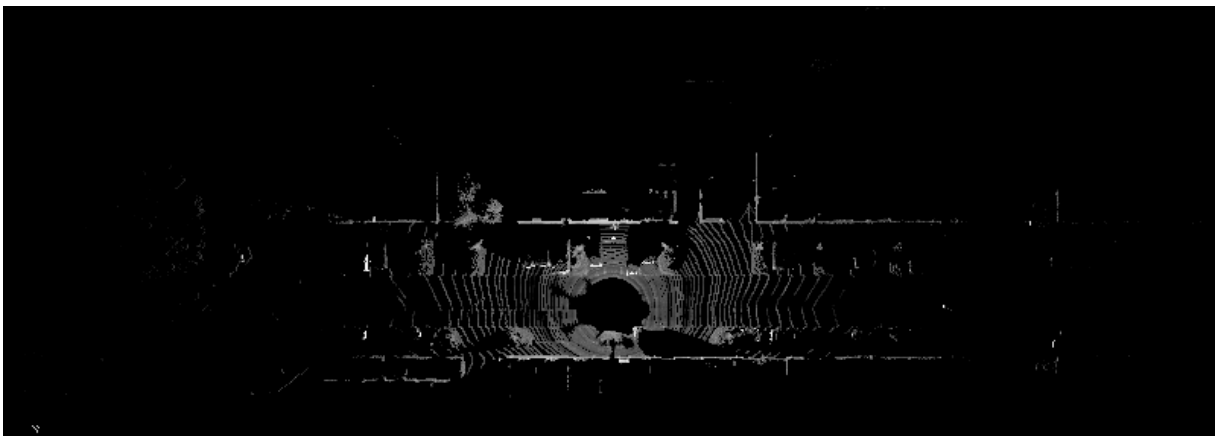

Figure 1: Bird Eye View for data.p rotated by -90 degrees

Matching Velodyne axes on the BEV :

- horizontal direction, from left to right: Velodyne **x-axis** (forward direction)
- vertical direction, from bottom to top: Velodyne **y-axis**
- pointing out of the image towards reader: velodyne **z-axis**

2 Problem 2: Visualization

2.1 Part 1

The idea of this sub-task is to project the velodyne point cloud onto the image taken by camera2 and color each point according to their respective semantic label.

So we begin by extracting all the elements necessary for the projection of the point cloud onto camera2, i.e:

- Velodyne points and their coordinates: x,y,z
- Semantic labels of these points
- Extrinsic calibration matrix: *T_cam0_velo*, from velodyne frame to camera0 frame
- Intrinsic calibration matrix: *P_rect_20*, from camera0 to camera2

The first aim is to transform the velodyne point coordinates into pixel coordinates in camera2 frame. The equation to perform this transformation is described by:

$$\begin{bmatrix} u_i \\ v_i \\ 1 \end{bmatrix} = \frac{1}{z_i} \times \text{P\_rect\_20} \times \text{T\_cam0\_velo} \times \begin{bmatrix} x_i \\ y_i \\ z_i \end{bmatrix} \tag{1}$$

where i is the index of the current point.
This transformation is handled by the function *get_cloud_pixel_coordinates()*.
However, it does not make sense to project the velodyne points which lie behind the camera onto image 2 since camera2 is facing forward. Therefore we first undertake the transformation to rectified camera 0 frame, where we will filter the points with negative z coordinate (former x-axis in velodyne frame). Homogeneous coordinates are used to respect matrices dimensions.

```
# To homogeneous
ones_array = np.ones((xyz_velodyne.shape[0],1))
xyz_velodyne = np.hstack((xyz_velodyne, ones_array))
# x,y,z as rows, point indexes as columns
xyz_velodyne = np.transpose(xyz_velodyne)
extrin_calib = np.matmul(T,xyz_velodyne)
```

Listing 1: Transformation to rectified camera0 frame in get_cloud_pixel_coordinates()

Where T represents *T_cam0_velo* and *extrin_calib* the velodyne point cloud in camera 0 frame.

Now, one can locate the indices for which the z coordinate is positive (velodyne points that match the camera view). The following part of code gets a new point cloud array (a subset of *extrin_calib*), expressed in camera 0 frame, for which the z coordinate of every point is positive (≥ 0). As we want to keep track of the respective semantic labels, the same operation is performed on the semantic label array:

```
#filter points with negative z
sem_label_fltrd = np.zeros(len(indexes))
extrin_calib_fltrd = np.zeros((4, len(indexes)))
for i in range(len(indexes)):
    extrin_calib_fltrd[:,i] = extrin_calib[:,indexes[i]]
    sem_label_fltrd[i] = sem_label[indexes[i]]
```

Listing 2: Filter velodyne points in camera0 frame, part of get_cloud_pixel_coordinates()

At this point, we are ready to compute the pixel coordinates u and v for each point. The last transformation uses *P_rect_20* to map each velodyne point from camera0 to the normalized image plane to the image coordinates u and v (of Camera 2). The matrix *P_rect_20* contains the so called intrinsic parameters to perform this operation. As described by the equation above, we also normalize the point coordinates by z (index 2 of *extrin_calib_fltrd*, which is the array containing the filtered velodyne points expressed in camera 0 coordinates).

```
#Projection of point cloud in image 2 coordinates
proj_cloud = np.matmul(P,extrin_calib_fltrd)/extrin_calib_fltrd[2,:] #
    normalization by Zc
u,v,k = proj_cloud   #k is an array of ones
u = u.astype(np.int32)
v = v.astype(np.int32)
```

Listing 3: Projection in image coordinates, part of get_cloud_pixel_coordinates()

The function finally returns the pixel coordinates u and v as well as the filtered point cloud ans the associated semantic labels, which are needed to color points.

At this stage we have the velodyne points projection into camera2 coordinates and their semantic labels. To draw and show/save images we use the cv2 library. Drawing the points and coloring is handled by the function *draw_points_cloud2image()* which uses the semantic label of each points to extract the corresponding color from the dictionary *color_map*. This function loops through all the filtered projected points and draws a colored circle at the location u_i, v_i for point i, according to the color associated to it's semantic label.

```
for i in range(velodyne_fltrd.shape[1]):
    label=sem_label_fltrd[i]
    color = color_map.get(label)
    # Draw a circle of corresponding color
    cv2.circle(img,(u[i],v[i]), 1, color, -1)
```

Listing 4: Draw projected points as colored circles, part of draw_points_cloud2image()

where img is the image recorded by camera2.
The output gives the following projection on the image taken by camera 2 for data.p:

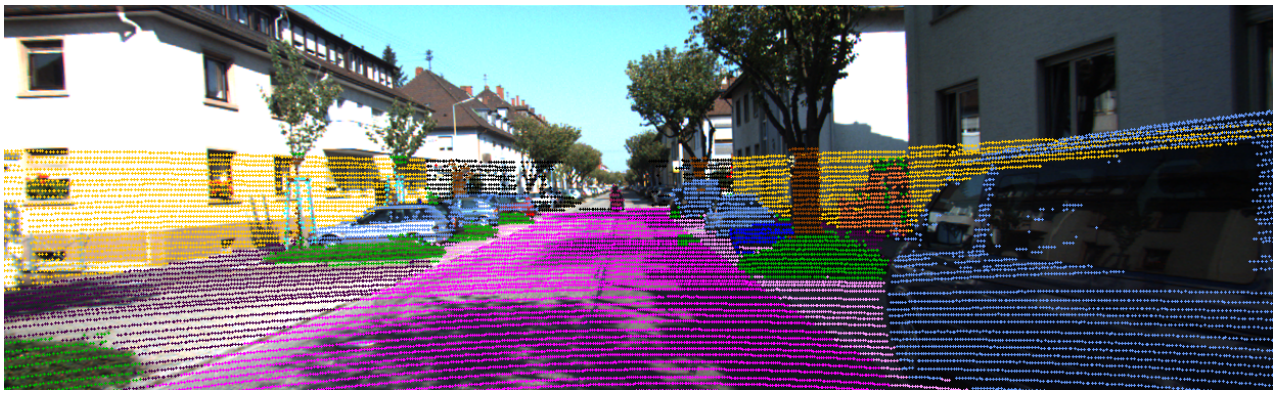

Figure 2: LiDAR point cloud projected onto Cam 2 image: **data.p**

2.2 Part 2

The aim is to project the 3D bounding boxes of all vehicles on the image taken by camera 2. To do so, one use the following sublists of *'objects'* in the dictionary data:

- dimensions
- location
- rotation y

In the code, we use the function *box_corner_coordinates()* which returns the coordinates of the eight corners necessary to draw a box on the image. The first argument of this function is the dictionary with key *'object'*, whereas the second argument is a Boolean called *box2image* to determine whether we want to draw the box in the image coordinates or in the velodyne coordinates. In this part, *box2image* is set to true since we want to draw the boxes on the image taken by camera 2.
In the function *box_corner_coordinates()* we extract the box dimensions for each car i.e. we get the

height, width and length of each box in meters.

Here we start by shaping the box in cam0 coordinates in the same order as implemented in *draw_box_image()* (order of drawing). The corner positions are defined w.r.t to the center of the bottom face of the bounding box, which for now, is assumed to be centered at the origin of the camera 0 frame.

```
x = [length/2, length/2, -length/2, -length/2, length/2, length/2, -length/2,
    -length/2]
y = [-height, -height, -height, -height, 0, 0, 0, 0]
z = [width/2, -width/2, -width/2, width/2, width/2,-width/2, -width/2, width/2]
```

Listing 5: Corner locations of one box, part of box_corner_coordinates()

So by default, this box is centered around the origin of the camera0 frame. We will translate it later to the object location. The next step is to rotate the box around the y axis using the *rotation_y* value. We can directly apply a rotation matrix around the Y-axis to the corner coordinates because they are already expressed in the Camera0 frame.
This rotation is implemented by the *y_rotation* function which is applied to each corner position (x,y,z) as described by this equation:

$$\begin{bmatrix} x_{rot} \\ y_{rot} \\ z_{rot} \end{bmatrix} = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix} \begin{bmatrix} x_{corner} \\ y_{corner} \\ z_{corner} \end{bmatrix} \tag{2}$$

where θ is equal to rotation_y.
This is implemented in the code as follows:

```
    Rot = y_rotation(objects[i][14])
    box_dim = Rot @ box_dim
```

Listing 6: Rotation of one box around the Y-axis, part of box_corner_coordinates()

Concretely we now have an array of dimension 3 x 8 (position coordinates × number of corners) for each box, where the indices represent the corner number:

$$\begin{bmatrix} x_1 & x_2 & \dots & x_7 & x_8 \\ y_1 & y_2 & \dots & y_7 & y_8 \\ z_1 & z_2 & \dots & z_7 & z_8 \end{bmatrix} \tag{3}$$

So we obtained the 8 corners position w.r.t. to the origin (0,0,0) of camera0 frame.
Thus, we still need to center this box around the object, by using the location information describing the center of the bottom face of the bounding box. The box location is extracted from the dictionary and then we simply centre our eight corner positions around this location as described in the following line of code:

```
box_dim += box_centre
```

Listing 7: Box translation to object location, part of box_corner_coordinates()

This is a simple translation of the box from the (0,0,0) origin to the (x,y,z) location of the object in Camera0 frame.

At this stage we are still in cam0 coordinates. Going to camera2 in pixel coordinates u and v is done by the function *rectification_Cam0toCam2(box_dim)*, which takes the box coordinates in the frame Camera0 and maps them to pixel in the image coordinates. This function implements the following equation using homogeneous coordinates:

$$\begin{bmatrix} u_i \\ v_i \\ 1 \end{bmatrix} = \frac{1}{z_i} \times \text{P\_rect\_20} \times \begin{bmatrix} x_i \\ y_i \\ z_i \end{bmatrix} \tag{4}$$

where i represents the corner index. (Similar to task 2.1)

Finally *box_corner_coordinates()* loops around all objects (all cars here) and stacks the eight pixel position (u,v) of each box in an array of dimension N x 8 x 2. N represents the number of boxes, 8 are the eight corner position and 2 is the dimension for (u,v).

Drawing these boxes on the camera 2 picture is delegated to the function *draw_box_images()* with the image result from part 1 (colored projected points) and the corner pixel location as arguments. Using the library cv2, we can draw a line for each vertice beween the corners (e.g. corner 0 to 1, then corner 0 to 3 etc...). Thus, the corners of each box are linked with *cv2.line()* and the function *draw_box_images*() iterates over all boxes (N boxes in the general case).

The image below depicts the result for demo.p:

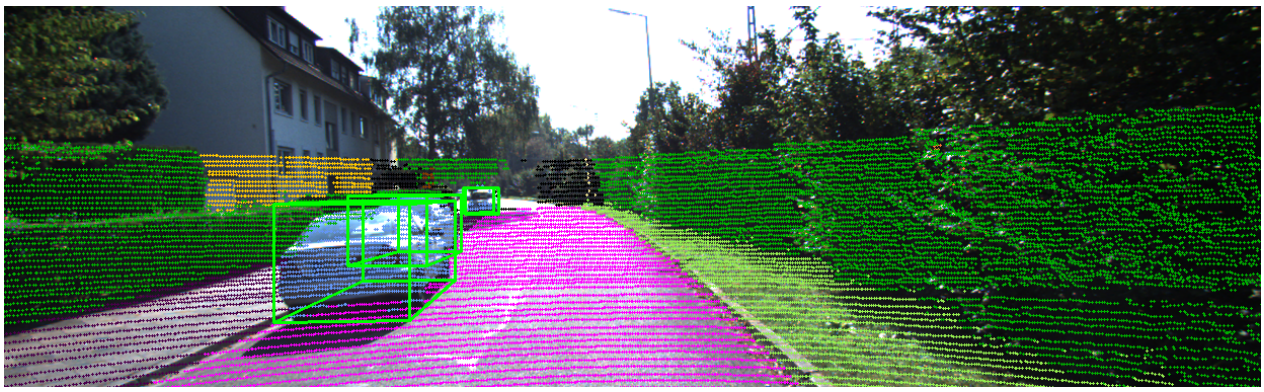

Figure 3: 3D bounding boxes onto Cam2 image for **demo.p**

And for data.p:

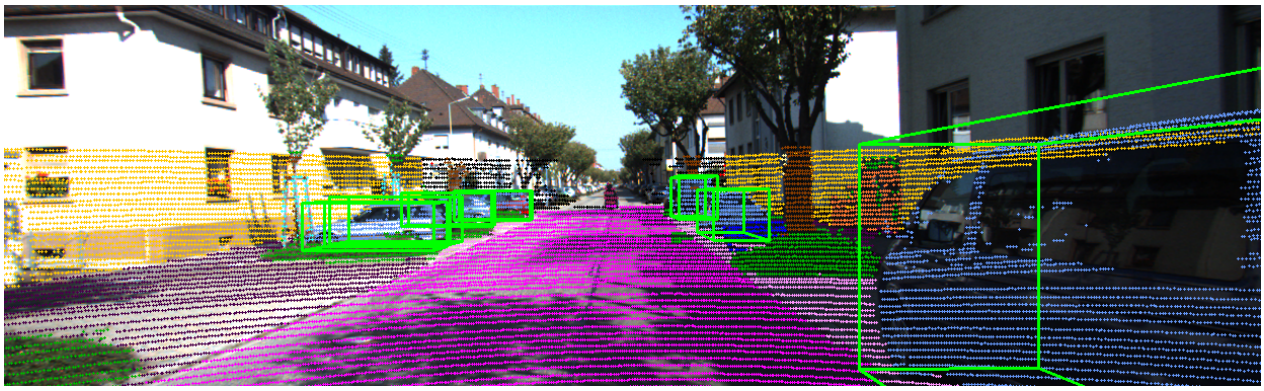

Figure 4: 3D bounding boxes onto Cam2 image for **data.p**

2.3 Part 3

The approach taken here is very similar to what has been done in part 2. Although, this time we want to draw the boxes directly in 3D in velodyne coordinates. Again, we call the function *box_corner_coordinates()* but with the argument *box2image* set as false. This function loops through all the boxes that need to be plotted.
Furthermore we want to express the box dimensions **directly w.r.t. the velodyne coordinate frame** and also w.r.t the center of the bottom face (as done before). This implies a slight change for the corner position (x,y,z) definition:

```
x = [-width/2, width/2, width/2, -width/2,-width/2, width/2, width/2, -width/2]
y = [length/2, length/2, -length/2, -length/2, length/2, length/2, -length/2, -
    length/2]
z = [height,height, height, height, 0, 0, 0, 0 ]
```

Listing 8: Corner locations of one box, in box_corner_coordinates() with *box2image* = False

Now that we have the corners position of the box we need to rotate it accordingly to the y_rotation value, in order to have the box well aligned with cars. From figure 1 in the project data we can infer that a rotation along the y axis in camera0 coordinates is equivalent to a rotation around **-z** in the velodyne coordinates (right-hand rule). Therefore the rotation around z is implemented in the function *z_rotation()* and performs the following operation for each corner:

$$\begin{bmatrix} x_{rot} \\ y_{rot} \\ z_{rot} \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_{corner} \\ y_{corner} \\ z_{corner} \end{bmatrix} \tag{5}$$

where θ is equal to $-rotation\_y$.

Concretely, the box has now the correct shape and correct rotation in the velodyne frame, but is still centred in (0,0,0) (velodyne frame). Since the location of the box (=object location center) is given in Cam0 coordinates, we first need to transform this location into velodyne coordinates. The function *coordCam0toVelo()* first calculates the inverse of the transformation matrix *T_cam0_velo* given in the data dictionary, in other words, it calculates the **transformation matrix from camera 0 to velodyne.**

```
def coordCam0toVelo(points):
    points.shape = (3,-1) # vector form
    # Inverse transformation in homogenous coordinates
    T_inv = np.linalg.inv(data['T_cam0_velo'])
    homogenous_points = np.vstack((points, np.ones((1,points.shape[1]))))
    pointsLidar = T_inv @ homogenous_points
    return pointsLidar[:3,:]
```

Listing 9: Coordinate transformation for the object location: Camera0 to Velodyne in coordCam0toVelo()

This function returns the location where the box has to be centered (at the object location) in the velodyne coordinates.
Using the translation operation as implemented in part 2, we center the box around the location that we extracted and transformed to velodyne coordinates.

The function *box_corner_coordinates* performs all of these operations for each box and returns a 3D array with shape N x 8 x 3 (number of boxes × number of corners × coordinates in velodyne frame). This array is collected in the main program and send to *update_boxes*() in **3dvis.py** which is in charge of drawing these boxes using the vispy library.

The coloring of the scene is implemented in the 3dvis.py class function *update()*:

```
def update(self, points, sem_label, color_map):
    #Normalize colors
    for i in color_map:
        color_map[i] = np.array(color_map[i])/255
        color_map[i] = color_map[i][::-1] #BGR to RGB conversion
    # declare color as numpy array, N x 3
    color = np.zeros((points.shape[0],3))
    # get color for each point
    for i in range(points.shape[0]):
        label=sem_label[i][0].astype(np.uint8)
        color[i,:] = color_map.get(label)
    # Plot point cloud with colors
    self.sem_vis.set_data(points, size=3, face_color = color)
```

Listing 10: update() function in 3dvis.py

This function first performs normalization and BGR to RGB conversion over the color_map key of the data dictionary. Afterwards, we use the semantic label of each point to get it's normalized color in RGB format, which is stored in the array "color" of dimension N points × 3 (RGB). This array of colors is passed along the points to the method set_data which plots the points and their color in 3D using vispy.

The final result is shown below:

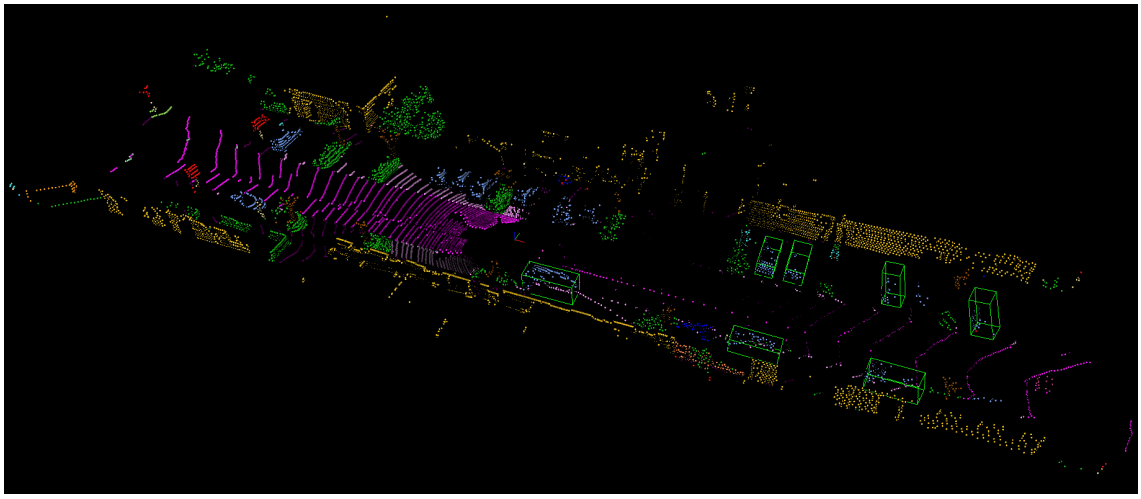

Figure 5: 3D bounding boxes using Vispy visualization image

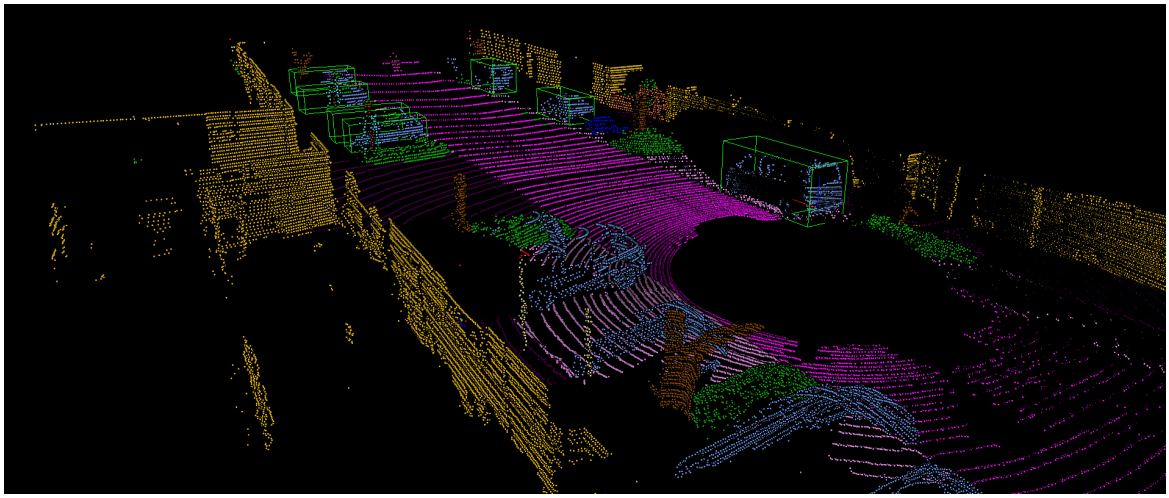

Figure 6: Other angle of view: 3D bounding boxes

3 Problem 3: ID Laser ID

We use a LIDAR scanner (Velodyne HDL64) to detect the objects surrounding the car at 360°. It has 64 lasers covering the space vertically from +2.0° to -24.9°(FOV = 26.9°), and those 64 lasers then rotates to cover the 360° horizontally.
The purpose of this third task is to identify the laser ID of each point in a given 3D point cloud. Different approaches exists such as clustering or angle classification for example.
We chose the most straightforward method, that is to divide the maximal vertical range in 64 equal intervals, and to assign each point to one of the intervals.
To do so, we divided the vertical range in 64 intervals based on the elevation angle (see below), and for each point, computed the elevation angle, and assigned the point to the closest interval.
For the maximal vertical range, we can take either the one from the datasheet, or the one from our data. We choose to take the one deduced from our data.

First, we had to find the elevation angle ε for each point of the point cloud:

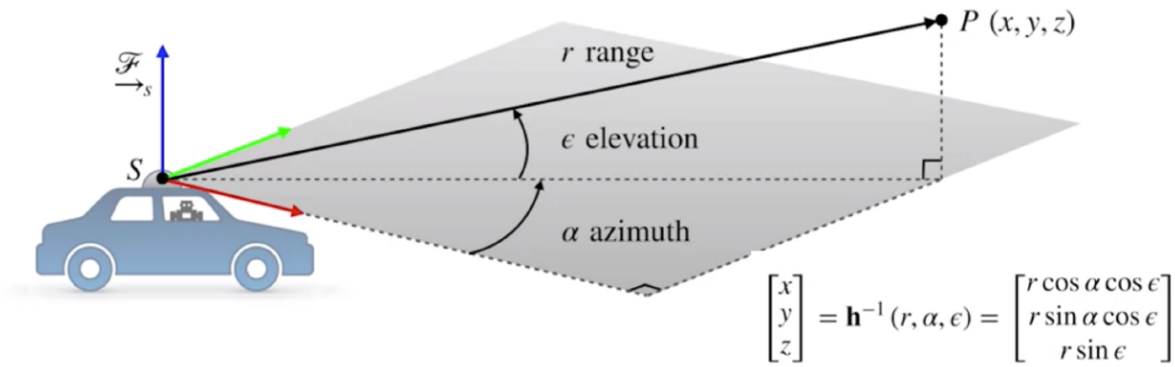

Figure 7: elevation angle

We used Pythagoras's theorem and basic trigonometry to find the ε for each point. From this information, we deduced the field of view (FOV) maximal range and divided it by 64 to get the resolution i.e. the angle difference between two consecutive lasers. Then we assigned the laser ID of the closest laser angle to our point. This is done in the *get_laser_id()* function:

```
def get_laser_id(xyz_velodyne):
    #compute elevation angle
    epsilon = np.zeros(xyz_velodyne.shape[0])
    laser_id = np.zeros(xyz_velodyne.shape[0])
    for i in range(xyz_velodyne.shape[0]):
        pythagore = math.sqrt(xyz_velodyne[i, 0]**2 + xyz_velodyne[i, 1]**2)
        z = xyz_velodyne[i, 2]
        epsilon[i] = np.arctan(z/pythagore)

    FOV = max(epsilon)-min(epsilon)
    resolution = FOV/64
    nb_bins = 65
    laser_angle = np.zeros(nb_bins)
    for i in range(nb_bins):
        laser_angle[i] = min(epsilon)+i*resolution

    for i in range(xyz_velodyne.shape[0]):
        for j in range(nb_bins-1):
            if epsilon[i]>=laser_angle[j] and epsilon[i]<laser_angle[j+1]:
                laser_id[i] = j+1
    return laser_id
```

Then, we used the same method as in exercise 2 to project the 3D points of the point cloud on the 2D image of the camera (see 2.1 for point cloud filtering and transformation to pixel coordinates). This has been implemented in the *getProjected_pointCloud()* function.
In order to assign a different color to the projected points according to their laser ID, the *laser_color()* function has been created. We first applied modulo 4 to the laser ID, and then, depending on the result, return a chosen H-value (4 different values for the 64 IDs):

```
def laser_color(val, min_d=1, max_d=64):
    """
    print Color(HSV's H value) corresponding to laser id
    """
    alter_num = 4
    H = 0
    id = (val - min_d)%alter_num
    if id == 0:
        H = 140
    elif id == 1:
        H = 120
    elif id == 2:
        H = 0
    elif id == 3:
        H = 60
```

```
    else:
        print("ERROR laser ID")
    return H
```

Listing 11: laser_color() function

This hue value (HSV format) is passed to *print_projection_plt* (from data_utils) as argument, which prints a circle (using *cv2.circle()* from opencv library) at the location of the points projected on the image, with the color defined by the H-value.

```
##get The laser ID associated to each point of the point cloud
laser_id = get_laser_id(xyz_velodyne)
##get projected filtered point cloud with associated laser id's
proj_cloud, laser_id_fltrd = getProjected_pointCloud(xyz_velodyne, laser_id)

##Draw laser ID color of the point cloud on image
img = image2.astype(np.uint8)

color=np.zeros(proj_cloud.shape[1])
for i in range(proj_cloud.shape[1]):
    label=laser_id_fltrd[i]
    color[i] = laser_color(label)
image = data_utils.print_projection_plt(proj_cloud, color, img)
```

We can see the result on **demo.p**. It looks very similar to the example provided in the instructions.

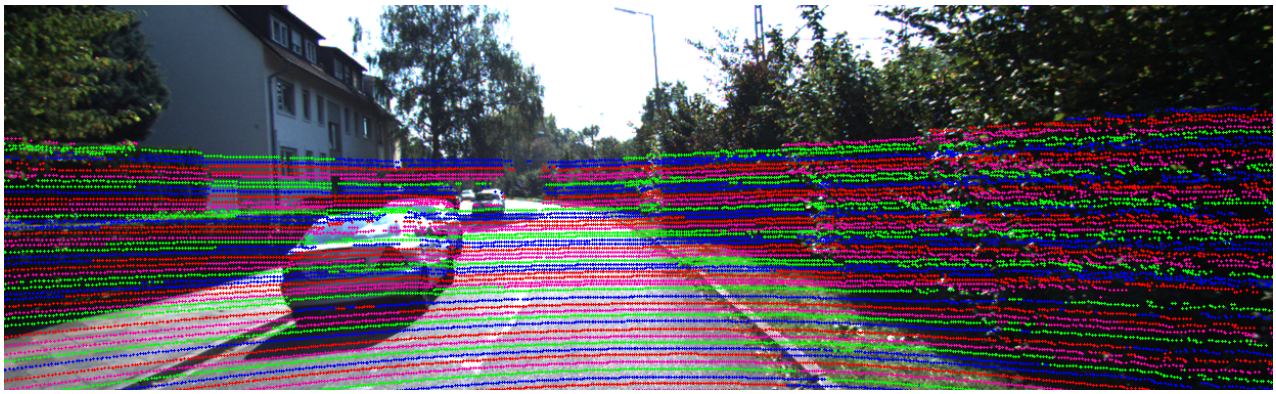

Figure 8: Laser ID's results on demo.p

And here is the result on **data.p**:

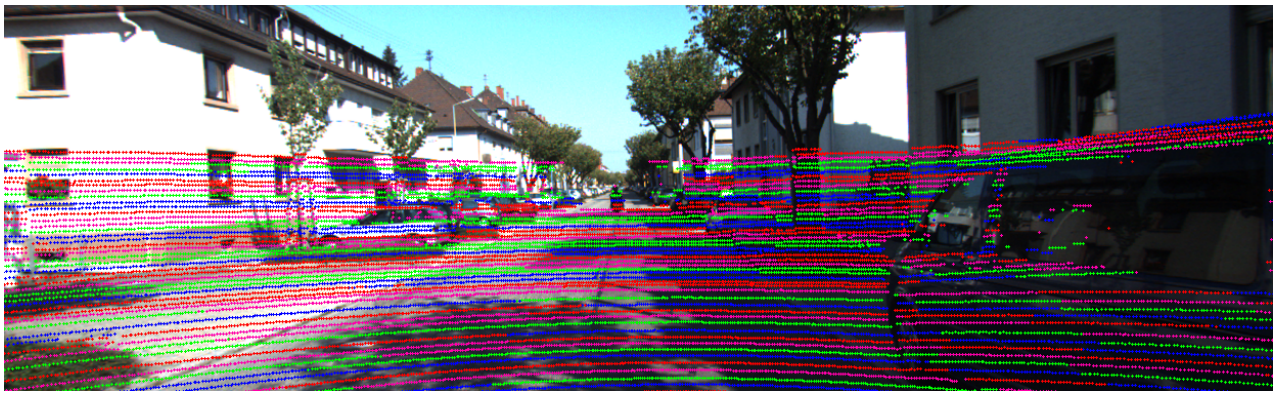

Figure 9: Laser ID's results on data.p

4 Problem 4: Remove Motion Distorsion

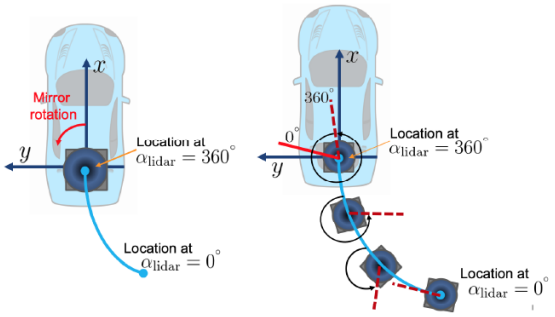

Figure 10: distortion caused by vehicle movement

As the car is moving, the Velodyne scanner also moves, but it continues to rotate and treats the points as if they were static. So the 3D point cloud obtained while the car moves, is distorted. This might generate errors as the LIDAR scanner is used for objects detection, and precise perception of the environment around the car. Indeed, some points of the velodyne were taken before the image was captured, and some points were taken after. Thus we need to recompute the real LIDAR points location as if the Velodyne had the possibility to achieve a complete scan at the moment where the corresponding image was taken on camera 2 (without the car moving).

The code we implemented works on a specific velodyne .bin file and the associated image. This can be changed in the following code part:

```
#-!!!!---------- Velodyne Bin number and image number -----------!!!!-#
binAndImage_file = 312 # Change HERE
#-!!!!------------------------------------------------------------!!!!-#
```

Listing 12: task4.py, change image file number where distortion removal is needed

For this task, we use a GPS system to have access to the position, speed, and angular velocity of the car where the /velodyne frame is attached to.
Knowing the Velodyne position (or car position in world coordinates) **at the beginning and at the end of a scan**, we can then approximate the velodyne position in world coordinates when scanning each point, by linearly interpolating between the start and the end of a scan.

Prior to interpolating, we first need to determine where the scan started, that is the first scanned point.
We know that when the camera triggers, the velodyne is always facing the front of the car, so we can compute the difference between the time at which the scan started and the time at which the picture was taken.
We chose to take timestamps from the velodyne data (*'velodyne_points/timestamps.txt'*) as the time at which the image is taken. We thought it would be more precise, as the velodyne triggers the camera exactly when it is passing the front of the car, and this time sample is stored in the velodyne timestamps. The timestamps stored with the image_02 data might have a small delay due to the transmission to the board computer.

Then, knowing the angular velocity of the velodyne (different for each scan):

$$w_{velo} = \frac{2\pi}{time\_end\_scan - time\_start\_scan} \tag{6}$$

We can compute the angle where the velodyne starts the scan (with respect to the front of the car):

$$angle\_velo\_start = w_{velo} * (time\_camera - time\_start\_scan) \tag{7}$$

Next, we need to compute the alpha angle (see Figure 7) for each point of the 3D point cloud. Again, we found it using Pythagoras's theorem and basic trigonometry.

After that, we found the closest alpha to *angle_velo_start* and labelled the corresponding point with this angle as the start point for the velodyne scan. As the order of the points is lost in the provided LIDAR point cloud, it was necessary to sort them. To do so, we sorted the indices of the point cloud based on the alpha values of the points (starting from angle_velo_start, and going clockwise).

```
def getSorted_PointCloud(point_cloud, angle_start_velo):
    """
    :param velodyne point cloud, starting scan angle of velodyne
    :returns: velodyne point cloud sorted in an ascending order from the starting
    horizontal
              angle at timestamp_start to the end of the scan at timestart_end
    """
    alpha = np.zeros(point_cloud.shape[0])
    #--- compute alpha angle i.e. azimuthal angle in the x, y plane ---#
    #----------- Sorted by ascending azimuthal angle ------------------#
    for j in range(point_cloud.shape[0]):
        hyp = np.sqrt(point_cloud[j,0]**2+point_cloud[j,1]**2)
        alpha[j] = np.arcsin(point_cloud[j,1]/hyp)
        cos = point_cloud[j,0]/hyp
        if cos < 0:
            alpha[j] = alpha[j] + np.pi
        if alpha[j] < 0:
            alpha[j] = 2*np.pi + alpha[j]
    # Ascending azimuthal angle in clockwise direction (liDAR scan direction)
    indices_sorted = np.argsort(2*np.pi-alpha) # in the clockwise direction
    alpha_sorted = np.sort(2*np.pi-alpha)
    # Find point cloud index where the velodyne scan starts #
    start_point_x = int(min(np.argwhere(alpha_sorted >= angle_start_velo)))
    indices_1 = indices_sorted[start_point_x:]
    indices_2 = indices_sorted[0:start_point_x-1]
    # Array of indices in the LiDAR scan order (velodyne start angle -> end angle)
    indices_orderOfScan = np.append(indices_1 ,indices_2)
    # Point cloud index associated to where camera2 triggers
    index_CameraTrigger = len(indices_sorted[start_point_x:])

    return point_cloud[indices_orderOfScan,:], index_CameraTrigger
```

Therefore, we now have a sorted point cloud in the scan order. This allows us to distinguish between the points which were taken **before** the image capture, and the point that were taken **after** the trigger of camera 2. One should mention that this sorting may not be the most precise, as there might be some noise in the data, resulting in different alphas for some points who should be aligned.
However, we use those alpha sorted to interpolate between time_start and time_end. The time interval (*delta_t*) between two points is of order 10^{-8}, so we can assume that even if we misclassified a point by 64 units, it is still very negligible.

Next we transform the sorted point cloud into IMU coordinates, since this is the frame where the velocity and angular velocity of the car have been recorded. To do so we implemented a function *calib_imu2velo* which works similarly to *calib_velo2cam* in data_utils.py.
calib_imu2velo returns the transformation matrices R and T from *calib_imu_to_velo.txt*, which we concatenate to get our full transformation matrix. To perform the transformation from velodyne to IMU we inverse this transformation matrix and multiply it by our homogeneous sorted point cloud.

Now that we know where the scan starts, and what is the order of the scanned points, we can remove the distortion.
First, we need to find the reference position. As we want to map the point cloud to the captured

images, we chose the reference position to be the position when the velodyne triggers the camera. We assumed the velocity and the angular velocity of the car to be **constant during an entire scan** (from time_start to time_end).
We also assume that the time difference between two consecutive points in the scan to be constant. Thus we get the time difference for each scan as:

$$delta\_t = \frac{lidar\_end - lidar\_start}{number\_points\_in\_pointCloud} \tag{8}$$

From the point scanned at time_camera, we can now interpolate **backwards** to time_start_scan and **forwards** to time_end_scan with a step of delta_t between each point.
The transformation matrix that we apply is a **concatenation** of the rotation matrix (where θ is the rotation angle applied to a point to remove motion distortion):

$$\begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{9}$$

and the translation(we assume 0 translation in z-axis):

$$\begin{bmatrix} translation\_x \\ translation\_y \\ 0 \end{bmatrix} \tag{10}$$

Where translation is the old referenced frame expressed in the new (translated) referenced frame:

$$^{B}\mathbf{P} = ^{A}\mathbf{P} + ^{B}\mathbf{O}_{A}$$

Figure 11: Translation-slide 6-Course02

We combine the rotation and the translation to apply a rigid transformation to the points:

$$^{B}\mathbf{P} = ^{B}_{A}\mathbf{R}\,^{A}\mathbf{P} + ^{B}\mathbf{O}_{A}$$

Figure 12: Rigid Transformation-slide 8-Course02

We then need to append a line of zeros terminated by a 1, to work with a homogeneous matrix.This gives us the Transformation matrix:

$$T = \begin{bmatrix} \mathbf{R} & \mathbf{t} \\ \mathbf{0}^{T} & 1 \end{bmatrix}$$

Figure 13: Transformation Matrix-slide 10-Course02

This correction is done in the *getCorrected_pointCloud* function in our code:

```
def getCorrected_pointCloud(point_cloud, delta_t, vel, ang, trigger_cam):
"""
:param point cloud, time intervall between each point in the velodyne scan,
       velocity and angular interpolation
       time where image is taken on camera2
:return corrected velodyne points without distorsion effect
"""
M_p = np.zeros((4,4,point_cloud.shape[1])) # Rigid Body Transformation Matrix
```

```
9     pointCloud_corrected = np.zeros((4, point_cloud.shape[1]))
10    homogeneous = np.array([0,0,0,1]) # to homogeneous coordinates
11
12    #------ Velodyne points taken BEFORE the image trigger on camera 2-------#
13    for j in range(0,trigger_cam):
14        #Compute translation using angular velocity and velocity
15        pos_xy_p = -vel[0:2]*(trigger_cam-j)*delta_t
16        #Compute rotation using pos and angular velocity
17        theta_p = -ang[2]*(trigger_cam-j)*delta_t
18        R = np.array([[np.cos(theta_p), -np.sin(theta_p), 0],
19                      [np.sin(theta_p), np.cos(theta_p), 0],
20                      [0,               0,      1]])
21        T = np.array([[pos_xy_p[0]],
22                      [pos_xy_p[1]],
23                      [0]])
24
25        R_T = np.hstack((R,T))
26        M_p[:,:,j] = np.vstack((R_T, homogeneous))
27        # Rigid Body transformation in homogeneous coordinates
28        homg_coordinates= np.hstack((point_cloud[:,j], 1))
29        # Remove distorsion
30        pointCloud_corrected[:,j] = np.matmul(M_p[:,:,j], homg_coordinates)
31    #--------------------------------------------------------------------------#
32
33    #------ Velodyne points taken AFTER the image trigger on camera 2---------#
34    for j in range(trigger_cam, point_cloud.shape[1]):
35        #Compute translation using angular velocity and velocity
36        pos_xy_p = vel[0:2]*(j-trigger_cam)*delta_t
37        #Compute rotation using pos and angular velocity
38        theta_p = ang[2]*(j-trigger_cam)*delta_t
39        R = np.array([[np.cos(theta_p), -np.sin(theta_p), 0],
40                      [np.sin(theta_p), np.cos(theta_p), 0],
41                      [0,               0,      1]])
42        T = np.array([[pos_xy_p[0]],
43                      [pos_xy_p[1]],
44                      [0]])
45
46        R_T = np.hstack((R,T))
47        M_p[:,:,j] = np.vstack((R_T, homogeneous))
48        # Rigid Body transformation in homogeneous coordinates
49        homg_coordinates= np.hstack((point_cloud[:,j], 1))
50        # Remove distorsion
51        pointCloud_corrected[:,j] = np.matmul(M_p[:,:,j], homg_coordinates)
52
53    return pointCloud_corrected[:3,:]
```

Important aspects of this function are described here:

- j is the index of the point to be translated in the sorted point cloud and *trigger_cam* the index where the camera has been triggered in the sorted point cloud. Δt is the timestep between two scanned points.
- line 15: Translation for points taken before the image capture, using IMU car velocity and $translation = pos\_xy\_p = -velocity \times \Delta t \times (trigger\_cam - j)$
- line 16: Rotation for points taken before the image capture, using IMU car angular velocity and $\theta_p = -angular\_velocity \times \Delta t \times (trigger\_cam - j)$
- Lines 36 and 38 work on the same principle but with opposite sign (since these velodyne point where taken after the captured image on the camera).

To conclude this section we notice that the points closer to the captured image time have smaller translation and rotation applied compared to the points at the extremities (long time before camera trigger or long time after camera trigger).
So *getCorrected_pointCloud()* returns the corrected point cloud i.e. with motion distortion removed

for the given velodyne .bin file and image.
Finally we go back to velodyne coordinates using transformation matrices R and T from *calib_imu_to_velo.txt* in homogeneous coordinates.

After that we compute the depth of each of the rectified points as $\sqrt{x\_coordinate^2 + y\_coordinate^2}$, and get the color (H-value, in HSV colorspace) associated to the depth with the depth_color function from data_utils.
We also need to express the 3D corrected points in the image-coordinates (similar procedure as implemented in 2.1).

Then, we plot circles with color according to the depth of the point, at the projected points locations using the *print_projection_plt* function. Here is the code:

```
def printCloud2image(xyz_velodyne, img):
    """
    :param point cloud and current image
    From velo -> cam0 -> img coordinates of cam2
    :return image with colored projected points on it in function of their depth
    """
    R, T = data_utils.calib_velo2cam('data/problem_4/calib_velo_to_cam.txt')
    # Get Extrinsic Transformation matrix from velo to cam (homogeneous coordinates)
    Trans_matrix = np.hstack((R,T))
    Trans_matrix = np.vstack((Trans_matrix,np.array([0, 0, 0, 1])))

    velodyne_homogeneous = np.hstack((xyz_velodyne, np.ones((xyz_velodyne.shape[0],1))))
    # indices as columns and 3D coordinates as rows
    velodyne_homogeneous = np.transpose(velodyne_homogeneous)
    # To cam0 coordinate frame
    extrin_calib = np.matmul(Trans_matrix,velodyne_homogeneous)
    # filter points with negative z in Cam0 coordinates
    indexes = np.argwhere(extrin_calib[2,:]>=0).flatten()
    extrin_calib_fltrd = extrin_calib[:, indexes]
    #compute depth estimation: sqrt(x^2 + z^2) in Cam0 frame
    depth = np.sqrt(extrin_calib_fltrd[0,:]**2 + extrin_calib_fltrd[2,:]**2)

    # Project on camera 2
    intrins_calib = data_utils.calib_cam2cam("data/problem_4/calib_cam_to_cam.txt", '02')
    proj_cloud = np.matmul(intrins_calib,extrin_calib_fltrd)/extrin_calib_fltrd[2,:] #normalization by Zc

    #color points in function of their depth and print velodyne points on cam2 image
    color=data_utils.depth_color(depth, np.amin(depth), np.amax(depth))
    img = data_utils.print_projection_plt(proj_cloud, color, img)
    return img
```

If we display the point cloud with the associated color(corresponding to the depth of the point) projected on the image, without taking into account the translation and rotation of the car, we observe some distortion.
We can see this on the bin num 37, especially near the 70 speed limitation sign.

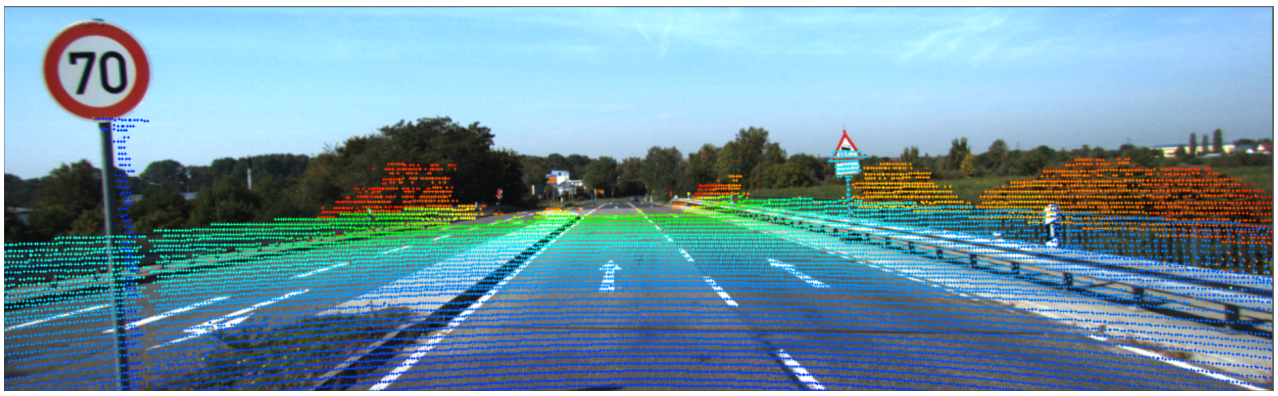

Figure 14: bin 37, image without distortion removed

We can see that after taking into account the translation and rotation by linearly interpolating between the start and the end of the scan, fixing the origin at the time when the picture is taken, we obtain quite good results (on all the bins). For the bin num. 37, we see that the mismatched LIDAR points near the 70 speed limitation sign do now match with the image.

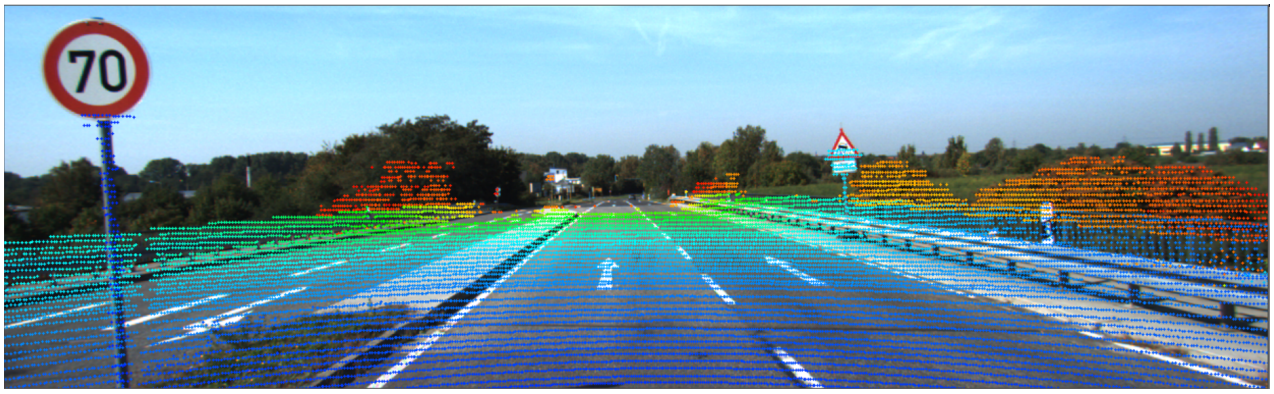

Figure 15: bin num 37, image with distortion removed

The bin number 37 undergoes mainly a translation motion. To check if the rotation is also compensated, we chose to look at bin number 312, where the car is turning left.
Here is the result of the bin 312 without motion distortion removed:

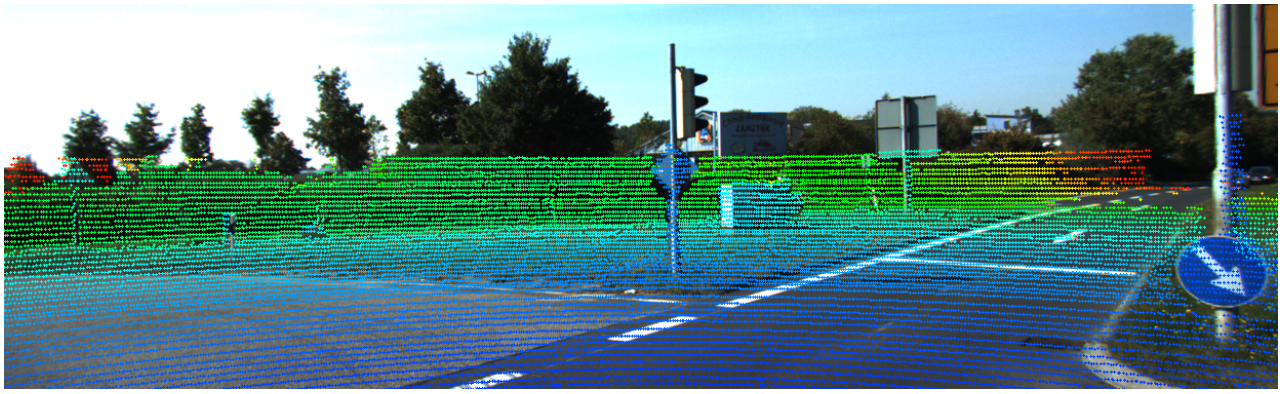

Figure 16: bin num 312, image without motion distortion removed

And here is the result of the bin 312 with the motion distortion removed: Note the correction on the traffic light on the right of the image.

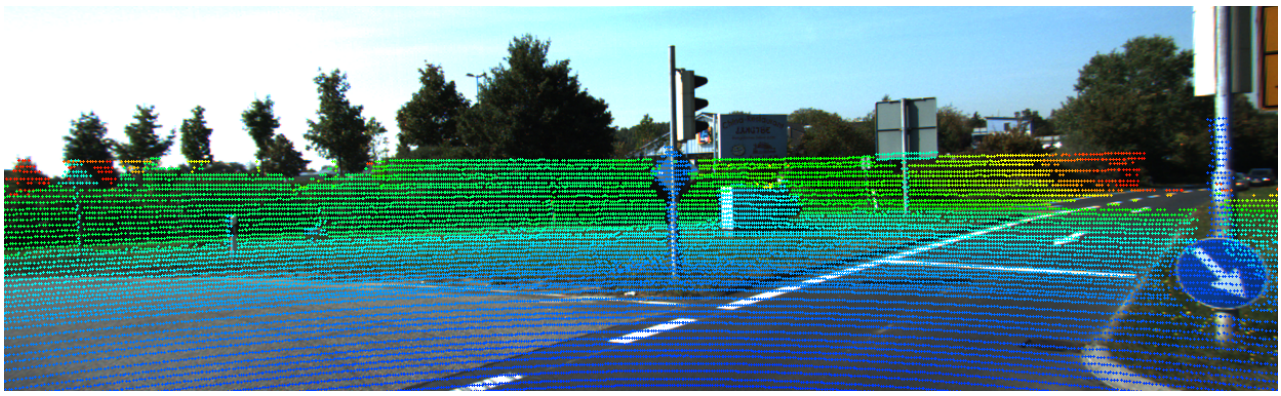

Figure 17: bin num 312, image with motion distortion removed

We can observe good results on bin this as well.
To conclude, the distortion due to rotation, and translation motion are quite well compensated.

5 Problem 5: Bonus Questions

5.1 Question 1

Eye safety: It is not safe for the human eye to look at a spinning LiDAR when it is too close. Why is the risk higher when we are closer to the sensor?

LiDAR safety for human eye depends on many parameters e.g. power, exposure time, direction, wavelength. LiDAR are class 1 Laser product, which are supposed to be eye-safe. Nonetheless, a high performance liDAR needs high pulse energy, small beam size and high pulse repetition frequency, hence the stationary beam itself could be hazardous (1). The smaller the laser beam diameter, the higher the energy density. The predominant cause of laser radiation injury are the thermal effects, which are linked to the power of the electromagnetic wave. A spinning laser, however, would constantly move so that the thermal effect due to laser pulses accumulated over a period of time would be less than with a non-spinning constant laser beam (1).

Furthermore, the electromagnetic waves are attenuated (energy decrease) with distance, and in the far field, the electromagnetic field is inversely proportional to the distance.
So if we are too close, the LIDAR lasers could still have sufficient energy to damage (burn) the retina, hence the higher risk especially at a wavelength of 903nm, which is the one used by the velodyne in this exercise. On top of that, the closer we are to the velodyne, the more lasers arrive in our retina at the same time.

To conclude this question, the risk is larger because at close range the energy of the laser is still high and has not been attenuated as it would have been at long distance.

Note: Generally, lasers can be dangerous for the human eye if their wavelength are in the range of 400-1400nm. Wavelengths longer than 1400nm are strongly absorbed in the cornea and lens of the eye, thus the energy reaching the retina is lower.
The energy also depends on the wavelength according to $E = \frac{h*c}{\lambda}$ where h is the Planck constant and c the light speed. Hence using a higher wavelength for the laser (e.g. 1500nm) appears to be safer than the wavelength of the HDL-64E S3 (903 nm) for the human eye.

5.2 Question 2

Wet roads pose challenges for both cameras and LiDAR. What are these challenges and why?

On a dry road, the asphalt of the road will diffuse light, sending light rays in all directions (causing a diffuse reflection).

On the other hand, on a wet road, the water acts as a mirror-like surface, causing a specular reflection (the light rays will reflect at the same angle to the surface normal as the incident ray, but on the opposite side of the surface normal).
The camera has to deal with objects, and their reflections. That might lead to interpretation error. The LIDAR is less affected by the wet conditions. However, we can say that the range will decrease. Indeed, the energy transmittance of the laser decreases with rain conditions and with the rain rate if we consider the laser beam to behave as an electromagnetic wave (2). Intuitively, we can think that with a higher rain rate, the laser will have more probability to be reflected on a rain drop and would be scattered in the environment causing distortion and noise when constructing the point cloud.
On top of that, for long range points(important depth), the laser will reflect away (as the reflection tends to be more specular), and thus only a small portion of the laser will come back to the velodyne. For closer point, and objects around the car, most of the laser will come back to the velodyne, thus they won't be affected.

5.3 Question 3

In this exercise, you have projected LiDAR points onto images. In the setup in Fig.1, the LiDAR sensor and the cameras are non-cocentered it can never be exactly cocentered.
What problem this may cause for the data projection between the two sensors (LiDAR and Cam2 for instance)? Do you think this problem will be more severe or less severe when the two sensors are more distant from each other?

If the LIDAR and the camera are non-cocentred, it means that the axes are not perfectly aligned and that the LIDAR referential and the camera referential are different. Meaning that the same point will have different coordinates in each referential. We can think of it as if we take two pictures with non-cocentered camera, and superposes them without any computation, they won't match. We will have a bad correspondence and the projection of the LIDAR points won't be projected to the right location. We will then observe a distorsion.
The problem is more severe if the two sensors are more distant, because the two referential will then be more distant, thus the same points will have more discrepancy in their coordinates.

References

[1] X.Zhu, D.Elgin. *Laser Safety in Design of Neear-Infrared Scanning LIDARs*. Proc. SPIE 9465, Laser Radar Technology and Applications XX; and Atmospheric Propagation XII, 946504 (19 May 2015); doi: 10.1117/12.2176998

[2] C.C.Chen. *Attenuation of Electromagnetic Radiation by Haze, Fog, Clouds, and Rain*. R-1694-PR, April 1975, A Report prepared for United States Air Force Project RAND