Project 1: Understanding Multimodal Driving Data

Deep Learning for Autonomous Driving
Team ID 18
Ivan Alberico, Nicola Loi
March 01, 2021 - March 25, 2021

Problem 1 - Bird's eye view

To visualize the point cloud from a top view, only the the x and y coordinates of the point cloud must be considered, together with the reflectance intensity of each point. These information are extracted from *data.p* file, where the position of the points is expressed in the *Velodyne* frame. To simplify the creation of the image, it is convenient to work with positive pixel coordinates only, and in order to do so, a translation in the *Velodyne* frame is performed. This is achieved using the negative values of the minimum x or y of the entire point cloud as offsets for the translation, in case they are negative. If the minimum value along either the x or y axis of a coordinate is non-negative, no translation is performed in that specific coordinate (null offset):

$$\begin{aligned} x'_{i,velo} &= x_{i,velo} + x_{offset} \\ y'_{i,velo} &= y_{i,velo} + y_{offset} \end{aligned} \qquad \text{with} \qquad \begin{cases} x_{offset} = -\,min(x_{velo}) & \text{if } min(x_{velo}) < 0 \\ x_{offset} = 0 & \text{otherwise} \end{cases} \qquad \begin{cases} y_{offset} = -\,min(y_{velo}) & \text{if } min(y_{velo}) < 0 \\ y_{offset} = 0 & \text{otherwise} \end{cases} \tag{1}$$

Now that x' and y' coordinates of the point cloud are all positive, they are converted in pixel coordinates (respectively row and column coordinates) through the proportion 0.2 meter/pixel, rounding up the result to the next integer:

$$row_i = \left\lceil \frac{x'}{0.2} \right\rceil \qquad\qquad column_i = \left\lceil \frac{y'}{0.2} \right\rceil \tag{2}$$

Points belonging to the same pixel (same row and column values) are grouped together in the same bin. The maximum intensity between the points of a bin is chosen to represent the corresponding pixel. To visualize the result, the first step is the creation of a complete black image, with dimension equal to the maximum values of the pixel coordinates obtained from the previous conversion (Formula 2). Starting from the top-left corner of the image, the color channels of each pixel are set to be 255 multiplied by the corresponding maximum intensity, since the latter is a value between 0 and 1 while the pixel intensity is defined between 0 and 255. At the end, Figure 1 is obtained (rotated by 90° for visualization purposes).

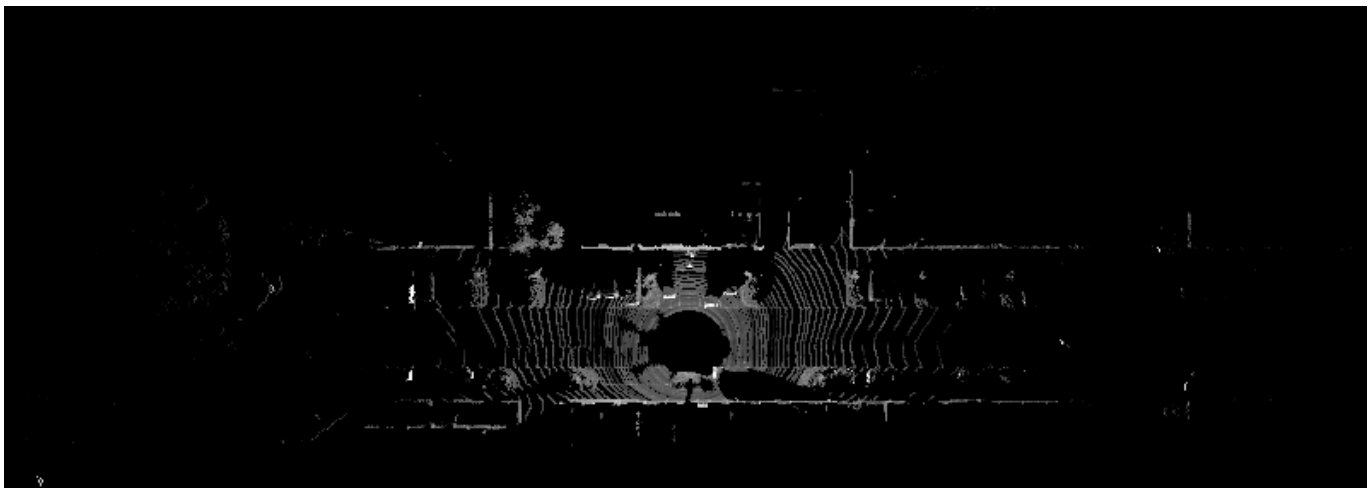


Figure 1: Bird's eye view of data.p. x positive axis pointing to the right, y positive axes pointing upwards.

Problem 2 - Visualization

The main goal of the second problem is to visualize the LiDAR point cloud of the scene onto the images of *Cam2*. The projected points should be colored according to their semantic labels and each vehicle should be delimited by a 3D bounding box.

Figure 2: Original *Cam2* image taken from *data.p* file

2.1

What is needed for this task are the coordinates of the points in the point cloud, the transformation matrix from the *Velodyne* to the *Cam0* frame, the projection matrix from the *Cam0* to the *Cam2* frame, the semantic labels of each point and the BGR color map for each label. All these data are extracted from the *data.p* file.

In order to project the points on the image from the *Velodyne* frame, they first need to be transformed into the *Cam0* reference frame. To do that, each point is first transformed in its homogeneous representation by adding a 1 to each array of coordinates (obtaining a 1×4 array for each point), and then the whole point cloud matrix is premultiplied with the matrix $T_{cam0,velo}$. The points in the *Cam0* frame are then filtered by excluding those behind the camera, i.e. the ones that have a negative z coordinate (since in frame *Cam0* the z axis is the one facing forward).

Once the points have been filtered out, the projected points on *Cam2* image are obtained by premultipling the filtered point cloud matrix in frame *Cam0* by the projection matrix P_{rect20}, and then dividing the result by the z coordinate of those points, according to the following formula:

$$\begin{pmatrix} x_{i,img}^{(2)} \\ y_{i,img}^{(2)} \\ 1 \end{pmatrix} = \frac{1}{z_i^{(0)}} P_{rect20} \begin{pmatrix} x_i^{(0)} \\ y_i^{(0)} \\ z_i^{(0)} \\ 1 \end{pmatrix} \tag{3}$$

After this projection onto the image plane of *Cam2*, another filtering step is performed in order to remove all the points that lie outside the image, discarding the ones that do not respect the inequalities $0 < x_{img}^{(2)} < img.shape[0]$ (width) and $0 < y_{img}^{(2)} < img.shape[1]$ (height). With this being done, at each point is assigned its corresponding BGR color, cross-referencing its semantic labels with the color map dictionary. To display the points on the image the function *cv2.circle()* from *OpenCV* library is used. The final result is shown in Figure 3.

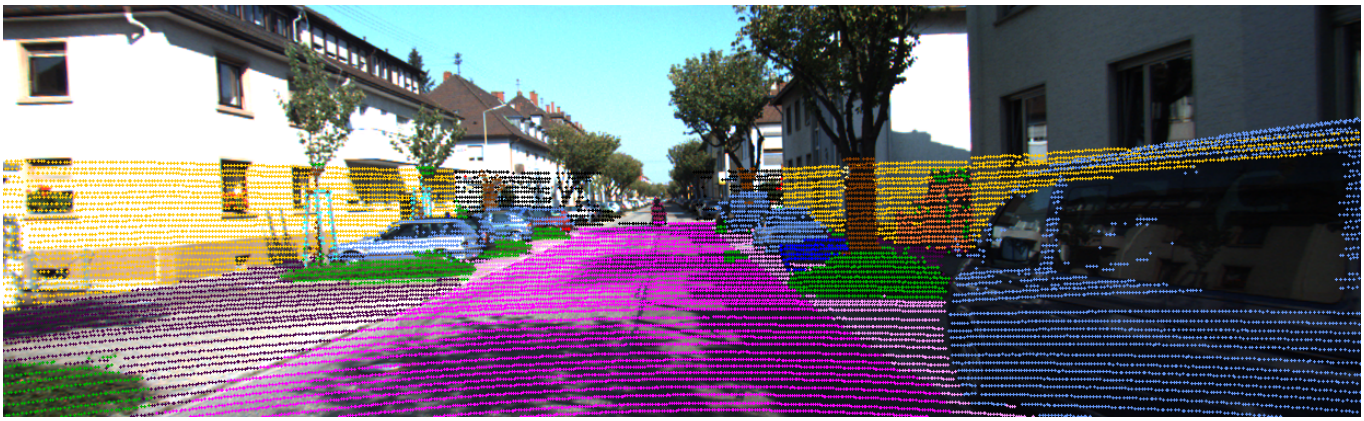


Figure 3: LiDAR point cloud projected onto the *Cam2* image, with each point being assigned to a color according to its label

2.2

The objective of the second task is to project onto the *Cam2* images a 3D bounding box around each detected vehicle. The additional data needed for this part are the dimensions of each bounding box, the x, y, z coordinates of the center of the respective bottom face and the angle of rotation around the y axis. Each bounding box is essentially a rectangular box described by the 8 vertices delimiting its shape. These data are with respect to the *Cam0* frame; to construct a bonding box in the *Cam2* image, some intermediate steps are required.

Firstly, the 8 vertices of each bounding box are computed through the given width, length and height dimensions, centering them around the origin of the *Cam0* frame. After that, each bounding box is rotated around the y axis, according to the *rotation_y* value. Once rotated, the 8 vertices are translated according to the given center coordinates of the bounding boxes. Finally, the vertices are brought in their homogeneous representation and premultiplied by the projection matrix P_{rect20}, in order to project them onto the image plane of *Cam2* (Formula 3). Then, the lines of the bounding boxes are displayed through the function *cv2.line()*. For each bounding box there are in total 12 lines connecting all the adjacent vertices.

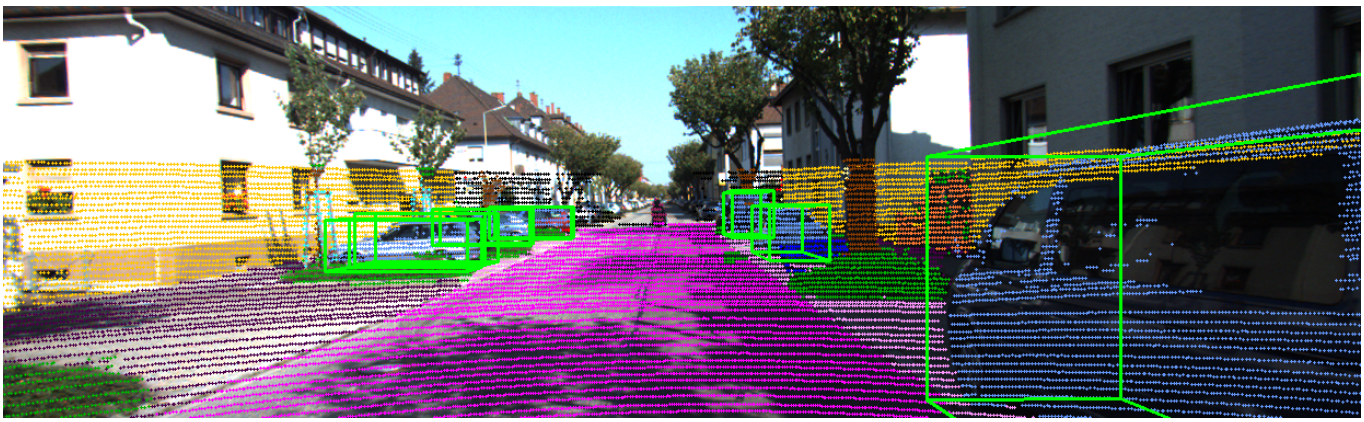


Figure 4: 3D bounding boxes projected onto the *Cam2* image around each vehicle of the scene

2.3

The goal of the last task of Problem 2 is to modify the *3dvis.py* file in order to visualize the point cloud in a 3D scene. In the *update()* function, it is necessary to assign to each point of the *Velodyne* point cloud the corresponding color according to the semantic label, and these values are then passed to the *set_data()* function. The color array is normalized (all the values are divided by 255), since the parameter *face_color* of the *set_data()* function only takes values between 0 and 1. Moreover, the first and third columns of the array are swapped, since the visualization requires colors in the RGB format and not BGR as provided by the color map. After that, the 3D points are correctly visualized with the corresponding color.

For the bounding boxes, a procedure similar to the one described in Problem 2.2 is applied. The difference here is that it is not necessary to filter any point, since it is required to display all the points of the point cloud. Once having computed the 3D coordinates of the vertices of each bounding box, it is necessary to go from the *Cam0* to the *Velodyne* reference frame. Since the data only provides the $T_{cam0,velo}$, the inverse homogeneous transformation $T_{velo,cam0}$ is computed in the following way, given that R and t are the rotation matrix and the translation vector of $T_{cam0,velo}$:

$$T_{velo,cam0} = \begin{pmatrix} R^T & -R^T t \\ 0 & 1 \end{pmatrix} \tag{4}$$

By premultipling the coordinates of the 3D vertices of the bounding boxes with the $T_{velo,cam0}$ matrix, they are now expressed in the *Velodyne* frame, are displayed in the visualization thanks to the provided *update_boxes()* function. The result obtained is shown in Figure 5.

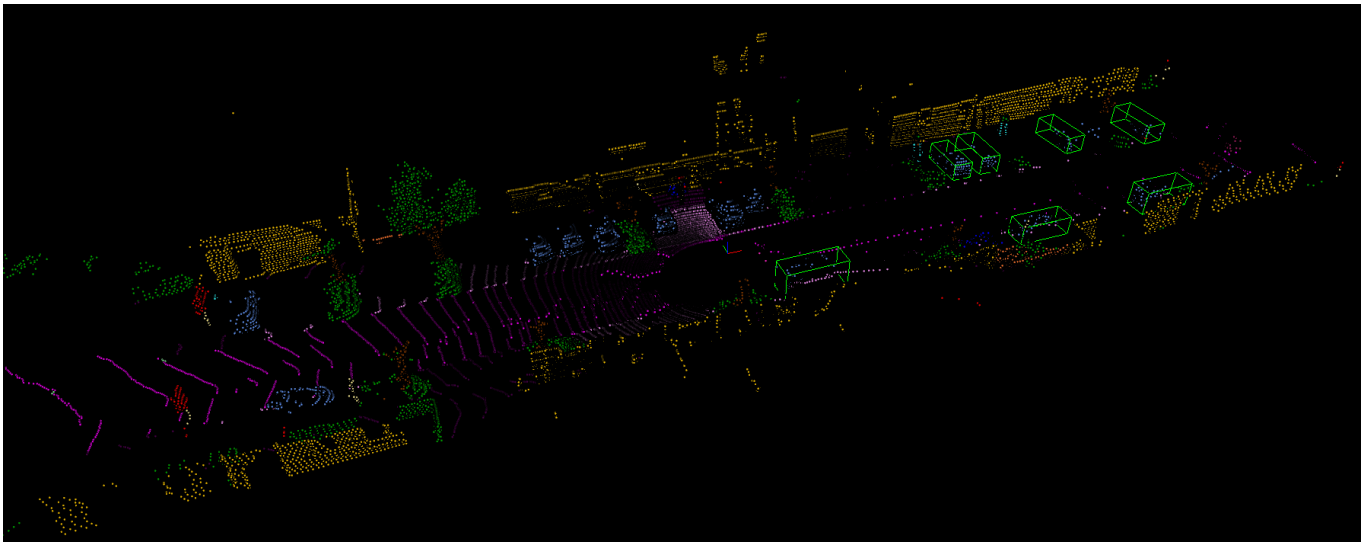


Figure 5: 3D visualization of the LiDAR point cloud in **3dvis.py**

From the visualization it is possible to see that there are two cars lying within the camera field-of-view that the network fails to identify (Figure 6). One is a blue car parked on the left side of the street and lying between the turquoise car and the red car, as it can be seen in Figure 7a , while the other is a car placed right after the Jeep, on the right side of the street, shown in Figure 7b. In the point cloud visualization, those two cars appear as sparse blue points which are not delimited by any bounding box. From a qualitative point of view, one could state that is reasonable that the network failed to identify these elements, as they are partially visible in the image and as they are partially visible in the image and they cannot be easily classified.

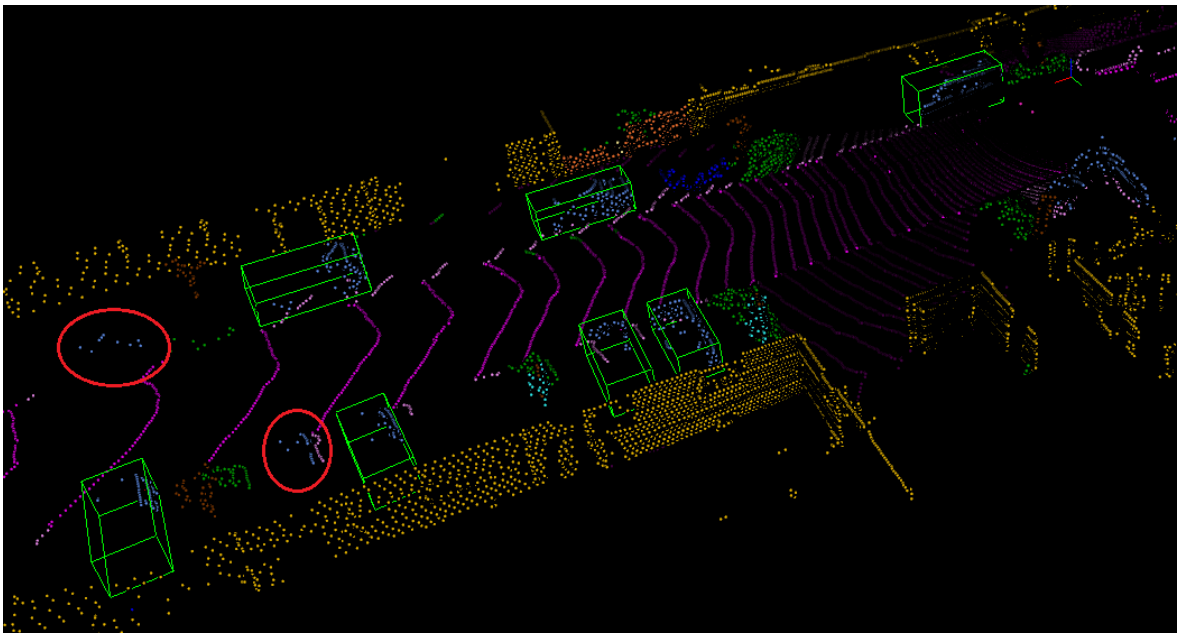


Figure 6: 3D visualization of the LiDAR point cloud in *3dvis.py*. The red circles delimit the points of the vehicles which are not identified by the network.

(a) Vehicle 1 not being identified

(b) Vehicle 2 not being identified

Figure 7: Crops of the vehicles which are not identified by the network

Problem 3 - Laser ID

The aim of the third part of the project is to classify each point of the point cloud with a laser ID based on the laser channel that has generated it. The results are then visualized directly on the projected points on the image, assigning to each point a specific color depending on its laser ID. For a better visualization, only 4 alternating colours are exploited. First, the elevation angle ε with respect to the horizontal *xy* plane of the *Velodyne* frame (as shown in Figure 8) is computed for each point:

$$\varepsilon_i = \arctan\left(\frac{z_{i,velo}}{\sqrt{x_{i,velo}^2 + y_{i,velo}^2}}\right) \tag{5}$$

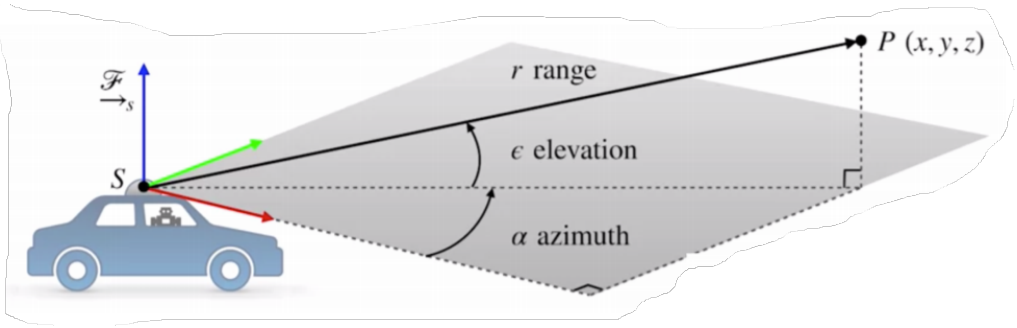


Figure 8: Reference angles representation for the point cloud with respect to the *Velodyne* frame

The steps to project the points onto the *Cam2* image frame starting from the *Velodyne* frame, passing through the *Cam0* frame, was already explained in Problem 2.1.

Two different approaches are used, which lead to comparable results. The first approach consists in calculating the FOV (field-of-view) as the difference between the maximum and the minimum ε angles of the whole point cloud, and then dividing this range by 64, which is the number of channels of the LiDAR sensor, as stated by its specifications. In this way, the FOV is divided in 64 bins, and each point is assigned to a specific bin according to the value of its ε angle. To every 4 consecutive bins, 4 different colors are associated, namely red, green, blue and magenta (Figure 9).

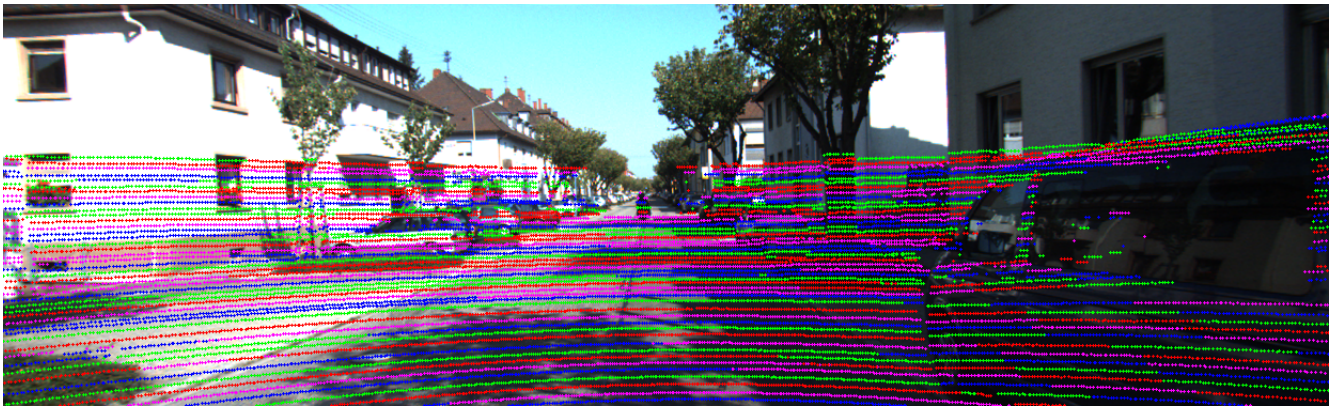


Figure 9: Laser IDs with method 1

The second method consists in assigning a laser ID to each point through clustering techniques. Different clustering methods have been tested, however after multiple tests the one that achieves the best result was DBSCAN (Density-Based Spatial Clustering of Applications with Noise) of the *scikit-learn* library. The parameters used are $eps = 0.0005$ and $min\_samples = 50$. The feature exploited for the clustering is only the ε angle. The cluster assigns to each point a numeric label, and to every 4 consecutive label, the 4 different colors are associated as for the first method. The results are shown in Figure 10.

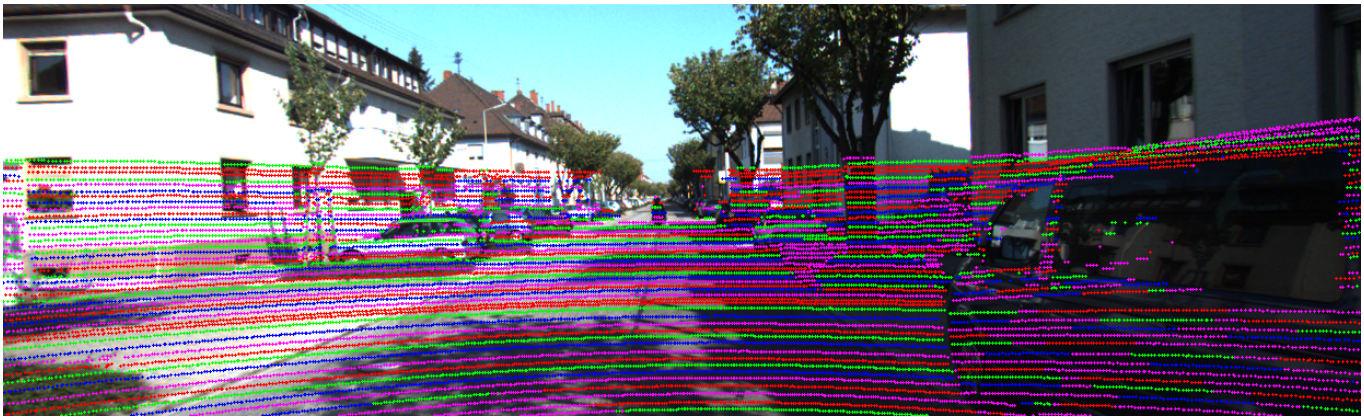


Figure 10: Laser IDs with method 2 (clustering)

The subdivision obtained with the two methods are somehow similar (but with some difference in which color is associated to which laser channel), however in the second case there are less discrepancies of color in the same channel, but the same ID is repeated for more channels in sequence, while in the first method is the opposite. Both methods have problems in the correct classification of the points projected on the car in the foreground.

Problem 4 - Remove Motion Distortion

This last part of the assignment consists of removing from the point cloud the distortion generated by the motion of the vehicle in which the LiDAR is mounted. For this part, information coming from the camera, the LiDAR and the GPS/IMU sensors are used. First, all the required are extracted from the *data\problem_4* folder through the functions provided in the *data_utils.py* file. The point cloud stored in the binary file is extracted through the *load_from_bin()* function, the timestamps through the function *computed_timestamps()*, the linear and angular velocity of the IMU frame through the functions *load_oxts_velocity()* and *load_oxts_angular_rate()*, while all the calibration matrices between the different frames are imported with *calib_velo2cam()* (used to import $T_{imu,velo}$ as well) and *calib_cam2cam()*.

Since in the provided point cloud the temporal order is lost (hence the information about which point is taken before which other point is unknown), the first step is to correctly reorder the points. To recover the temporal order, the points are first sorted according to the azimuth angle spanning in the horizontal plane (Figure 8) going from 0 to -2π values of α. In other words, the points are reordered starting from the one that faces the positive *Velodyne x* axis and then continue in clockwise direction (i.e. in order IV-III-II-I quadrant).

The obtained array has the points arranged in this order: from the point taken at ts_{camera} to the one taken at $ts_{velo,end}$, and then the points taken from $ts_{velo,start}$ to ts_{camera} (ts_{camera} is the timestamp of the photo being taken). To sort this array based on the chronological order of the points' scans ($ts_{velo,start} \rightarrow ts_{camera} \rightarrow ts_{velo,end}$), it is only needed to cut the array in two parts and exchange them. To do this, it is necessary to know where the array has to be cut, namely which is the index of the point taken at $ts_{velo,start}$. Firstly, the time difference between the scans of two consecutive points is computed:

$$\Delta ts_{scan} = \frac{ts_{velo,end} - ts_{velo,start}}{n_{points}} \tag{6}$$

Then the index of the point taken when the *Velodyne* is starting the scan cycle at ts_{velo_start} is calculated by:

$$index_{start} = \frac{ts_{camera} - ts_{velo,start}}{\Delta ts_{scan}} \tag{7}$$

The array is then correctly rearranged to have the points in the order $ts_{velo,start} \rightarrow ts_{camera} \rightarrow ts_{velo,end}$. From this, a new vector of dimension $n_{points} \times 4$ is created, which contains the coordinates of the ordered points and the corresponding timestamp ts_i at which they are taken:

$$ts_i = ts_{image} + (i - index_{start})\Delta ts_{scan} \tag{8}$$

With this being done, next step is to use the velocity information coming from the IMU to remove the motion distortion. Since the coordinates of the point cloud are given in the *Velodyne* frame, first it is necessary to bring the points in the IMU frame, by premultiplying them with $T_{imu,velo}$, which is computed by inverting the matrix $T_{velo,imu}$ provided in the data:

$$\begin{pmatrix} x_{i,imu} \\ y_{i,imu} \\ z_{i,imu} \\ 1 \end{pmatrix} = T_{imu,velo} \begin{pmatrix} x_{i,velo} \\ y_{i,velo} \\ z_{i,velo} \\ 1 \end{pmatrix} = \begin{pmatrix} R^T_{velo,imu} & -R^T_{velo,imu}\, t_{velo,imu} \\ 0_{3x3} & 1 \end{pmatrix} \begin{pmatrix} x_{i,velo} \\ y_{i,velo} \\ z_{i,velo} \\ 1 \end{pmatrix} \tag{9}$$

Once the points are in the IMU frame, the linear and angular velocity of the IMU sensor are used to compensate the motion distortion in the points. The linear velocity is used to compensate for the displacement while the angular velocity for the orientation. The correction is computed as follows:

$$R_{i,distortion} = \begin{pmatrix} \cos(\theta_i) & -\sin(\theta_i) & 0 \\ \sin(\theta_i) & \cos(\theta_i) & 0 \\ 0 & 0 & 1 \end{pmatrix} \qquad t_{i,distortion} = \begin{pmatrix} v_{x,imu}\,\Delta ts_i \\ v_{y,imu}\,\Delta ts_i \\ v_{z,imu}\,\Delta ts_i \end{pmatrix} = \begin{pmatrix} v_{x,imu}\,(ts_{camera} - ts_i) \\ v_{y,imu}\,(ts_{camera} - ts_i) \\ v_{z,imu}\,(ts_{camera} - ts_i) \end{pmatrix} \tag{10}$$

$$\begin{pmatrix} x'_{i,imu} \\ y'_{i,imu} \\ z'_{i,imu} \\ 1 \end{pmatrix} = \begin{pmatrix} R_{i,distortion} & t_{i,distortion} \\ 0_{3x3} & 1 \end{pmatrix} \begin{pmatrix} x_{i,imu} \\ y_{i,imu} \\ z_{i,imu} \\ 1 \end{pmatrix} \tag{11}$$

where $\theta_i = \omega_{z,imu}\,\Delta ts_i = \omega_{z,imu}\,(ts_{camera} - ts_i)$ refers to the rotation of the points around IMU z axis due to distortion and $v_{z,imu} = 0$. Only the rotation around the z axis and the displacement in the xy plane are taken into account, since the car motion is assumed to be planar in the short term period. Once the points have been corrected, they are transformed back in the *Velodyne* reference frame:

$$\begin{pmatrix} x'_{i,velo} \\ y'_{i,velo} \\ z'_{i,velo} \\ 1 \end{pmatrix} = T_{velo,imu} \begin{pmatrix} x'_{i,imu} \\ y'_{i,imu} \\ z'_{i,imu} \\ 1 \end{pmatrix} = \begin{pmatrix} R_{velo,imu} & t_{velo,imu} \\ 0_{3x3} & 1 \end{pmatrix} \begin{pmatrix} x'_{i,imu} \\ y'_{i,imu} \\ z'_{i,imu} \\ 1 \end{pmatrix} \tag{12}$$

The function *depth_color()* from *data_utils.py* is then used to assign to each corrected point an H value (in the HSV color scale) according to its distance $dist_i$ from the origin of the *Velodyne* frame.

$$dist_i = \sqrt{{x'_{i,velo}}^2 + {y'_{i,velo}}^2 + {z'_{i,velo}}^2} \tag{13}$$

Finally, the colored corrected points must be projected onto the *Cam2* image plane. The steps to project them onto the *Cam2* image frame starting from the *Velodyne* frame, passing through the *Cam0* frame, was already explained in Problem 2.1. The colors previously computed are converted in the RGB color scale and the *cv2.circle()* function is used to visualize the points, as shown in Figure 11. For comparison, the projection obtained without correcting the distortion of the point cloud is displayed in Figure 12.

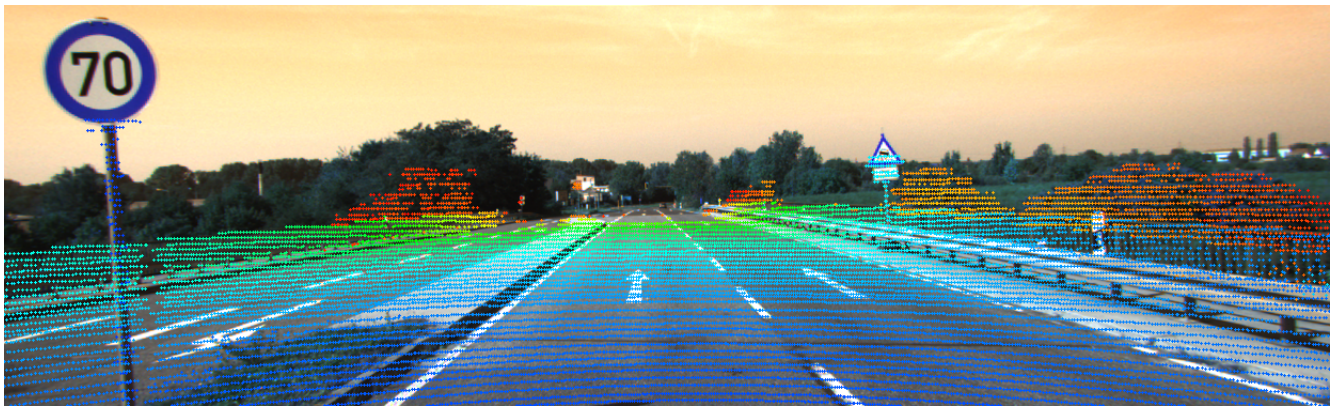


Figure 11: Point cloud projection on image '0000000037' after motion distortion has been corrected

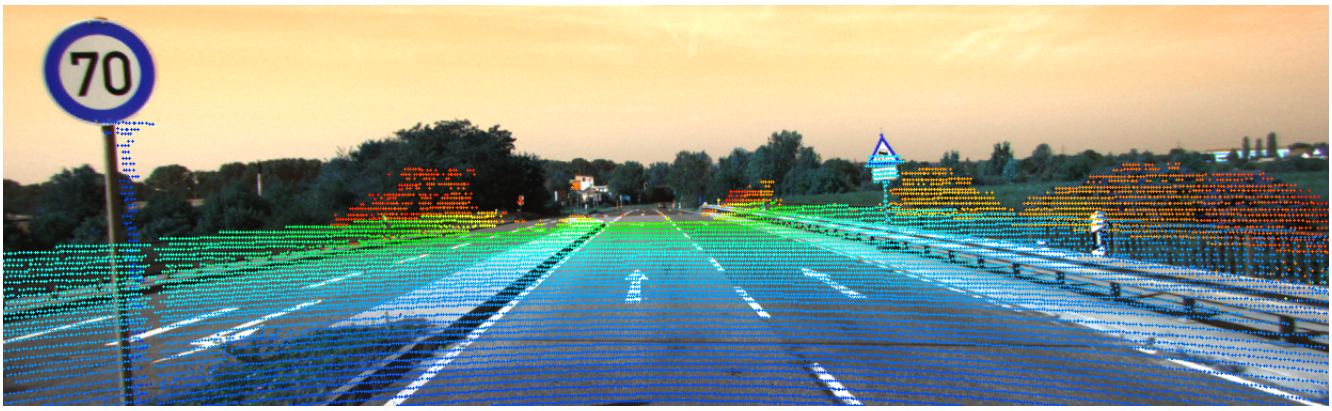


Figure 12: Point cloud projection on image ’0000000037’ without correcting motion distortion

Problem 5 - Bonus Questions

1)

The risk is higher when the eyes are closer to the sensor because the beams of light emitted by the LiDAR lose energy as they move away from the emitter. Hence, they have the maximum energy near the sensor, and so the danger is higher. This is the same reason why the LiDAR can work only up to a specific range: after a certain distance the returning light beams are too weak to be detected.

2)

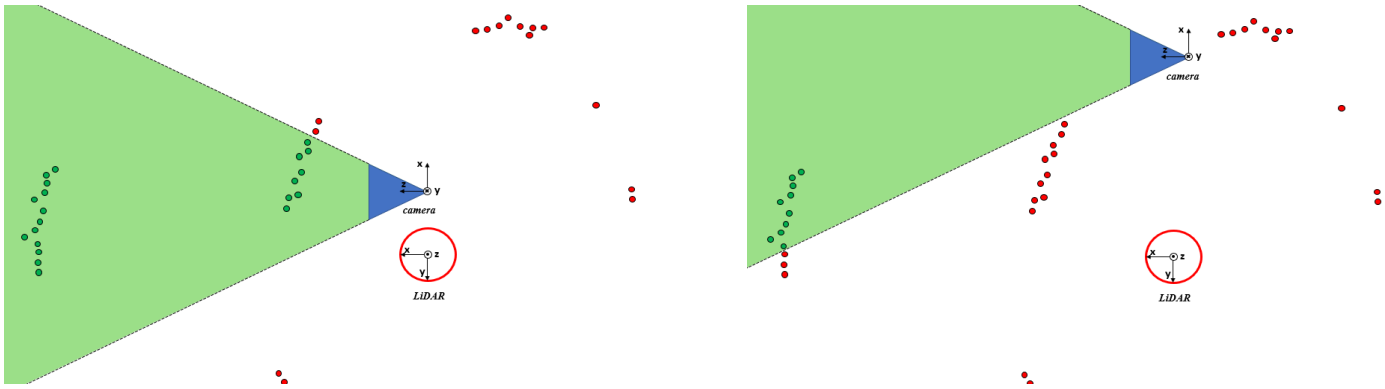
The main challenge of wet roads for both camera and LiDAR sensors regards the presence of water in the road, which becomes a water mirror. A camera works by detecting the passive light that the objects reflect; the problem with a wet road is that it reflects specularly an object located above it; this is dangerous because the system could erroneously localize the reflected object in a position where effectively it is not present. It will see the object where actually there is the mirror of water that reflected it.

A LiDAR sensor instead works by detecting the emitted active light that reflects off objects and comes back. In a normal condition, the diffuse reflection of the light that hit the road allows the sensor to receive back the light emitted, and to properly detect the road. But if it is wet, the beams of light will be reflected specularly and not diffusely, hence they will not come back to the sensor which is then incapable to correctly identify the road.

3)

Sensors could never be cocentered because they have a physical volume and occupy a physical three dimensional space: it is impossible to place two sensor in the same exact position, the best one can do is to try to place them as close as possible to each other. Since the sensors are non-cocentered, some important points of the point cloud in front of the LiDAR could be lost in the transformation from the LiDAR frame to the image planes of the cameras. The camera could see only the points that lie in the superposition of its FOV with the one of the LiDAR. So even if the two sensors are close, the camera will be blind to a large chunk of the point cloud space, as shown in Figure 13a.

Increasing the relative position of the camera with respect to the LiDAR, the superposition of their FOVs varies, and consequently the camera may see less points inside its FOV. As shown in Figure 13b, when the sensors are moved laterally from each other more and more, the blind spot of the camera for close points (such as an obstacle in front of the vehicle) augments. Instead if the sensor are moved longitudinally, i.e. the camera is placed ahead of the LiDAR, increasing their distance will increase the blind spot of the camera for points lateral to the vehicle.



(a) Sensors close to each other

(b) Sensors far away from each other

Figure 13: Superposition (in green) of the FOVs of a camera and of a LiDAR.

Another problem is occlusion: the more the camera and the sensor are far away from each other, the higher the chance that some area of the image will be free of projected points of the point cloud. This occurs when objects are relatively near to the two sensors; the LiDAR will detect them, but not what is behind them. The camera, since it is placed in another position, will see the space that has been occluded by the objects to the LiDAR.