

# Section 1 : Compute Lidar Point-Cloud from Range Image

## Visualize range image channels (ID\_S1\_EX1)

In this section we visualize a first image showing:

- Range image
- Intensity image
- Intensity and Range one in top of the other (vertically stacked)



This image represents the cropped area in front of the vehicle (+- 45°). The full area can be seen in the image below:



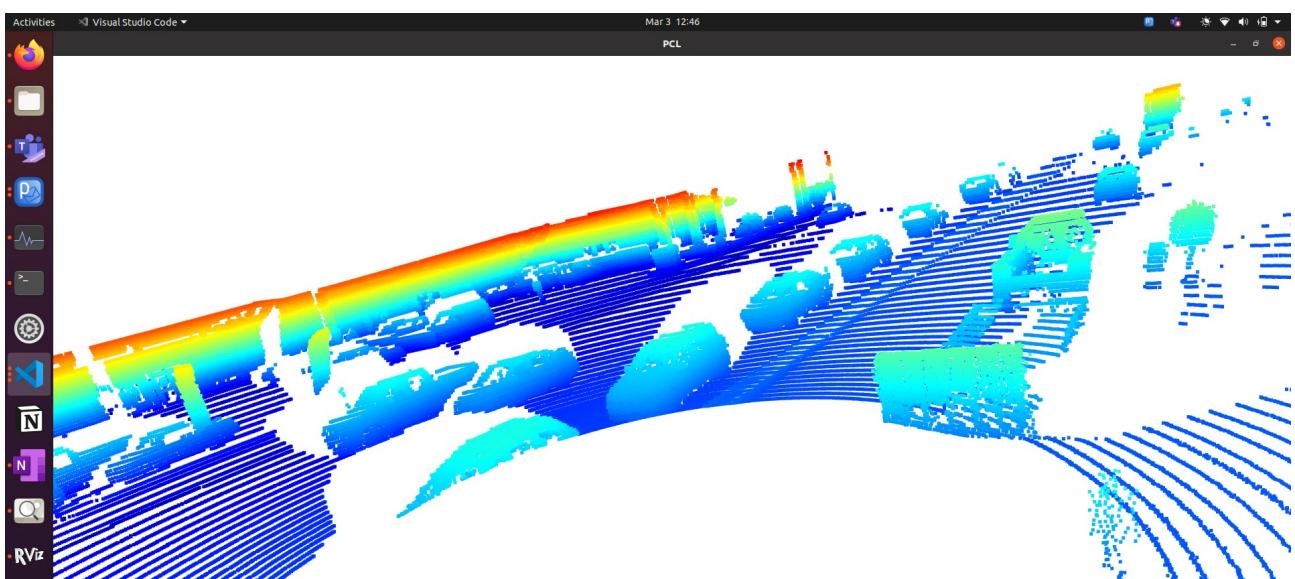
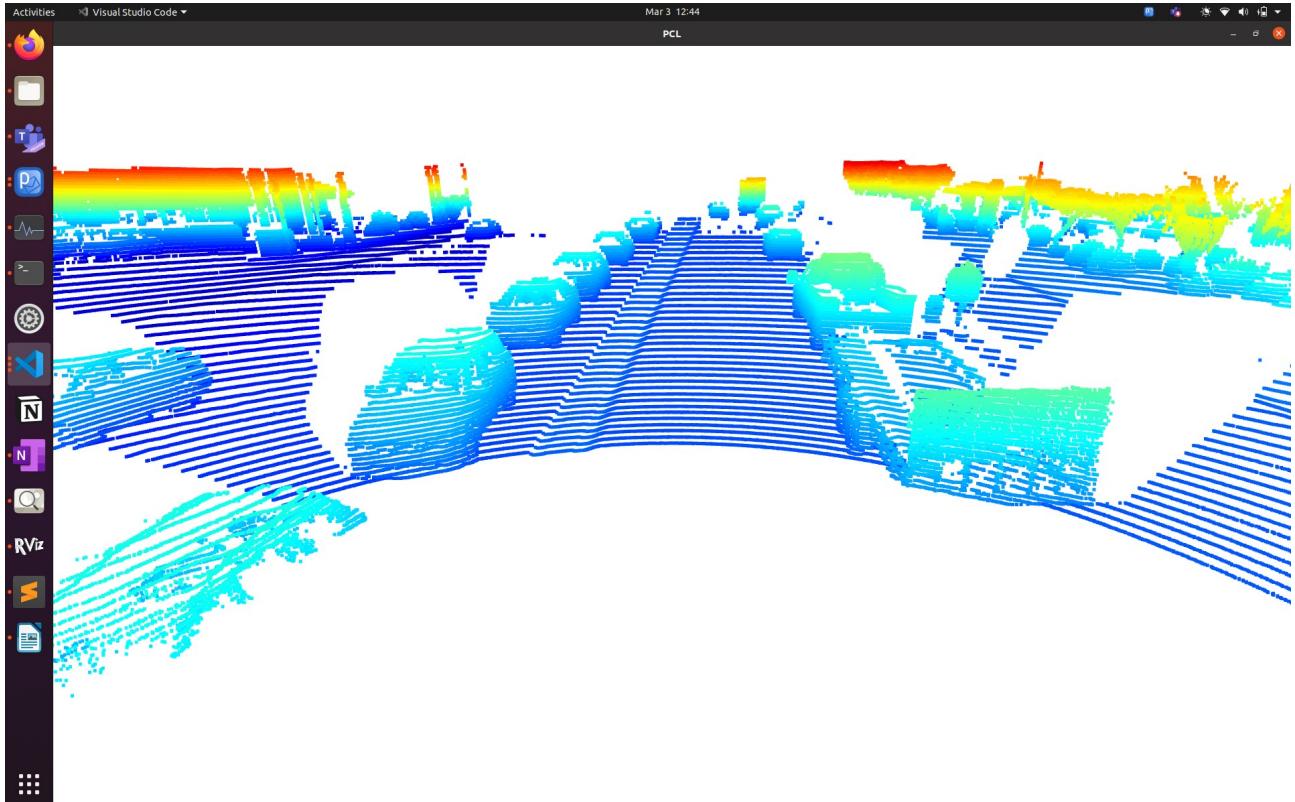
The code for adapting the range and intensity inputs into the desired output is:

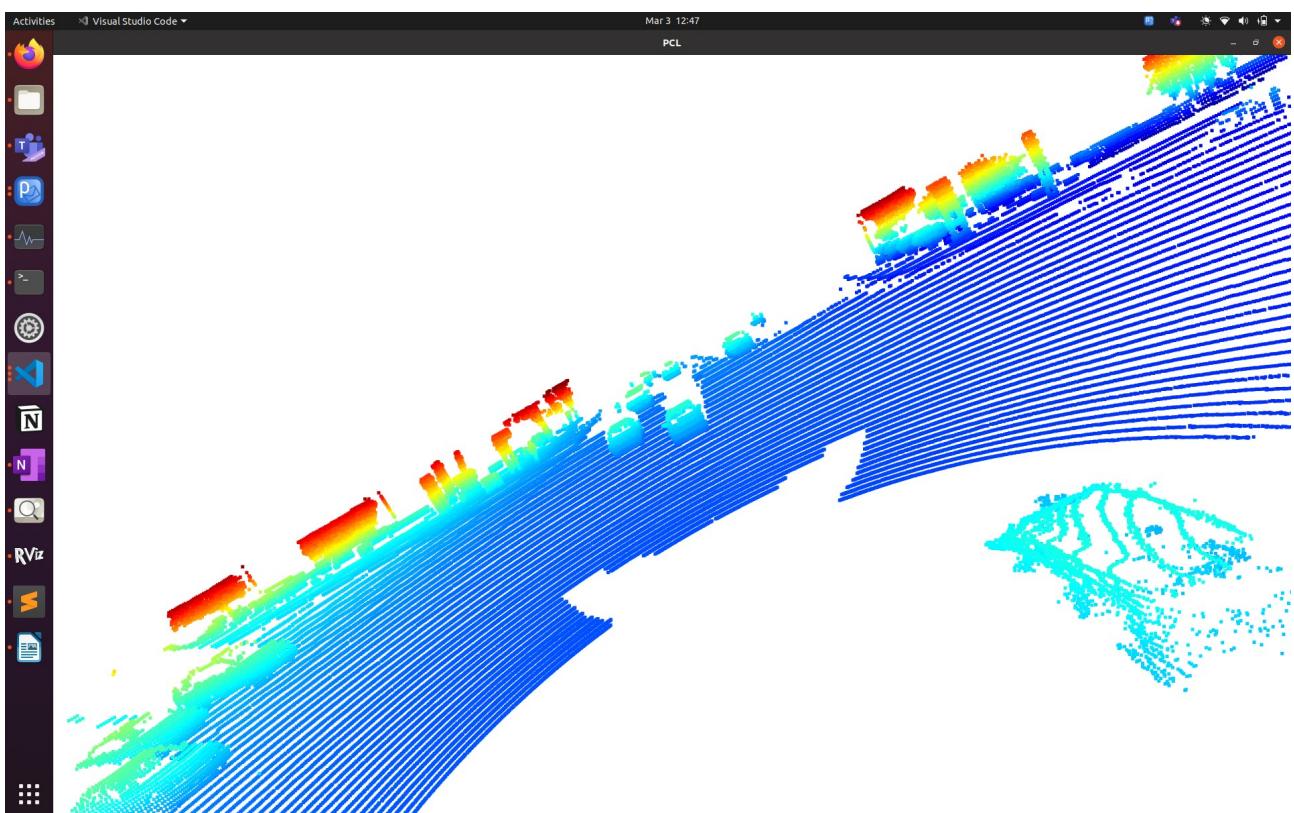
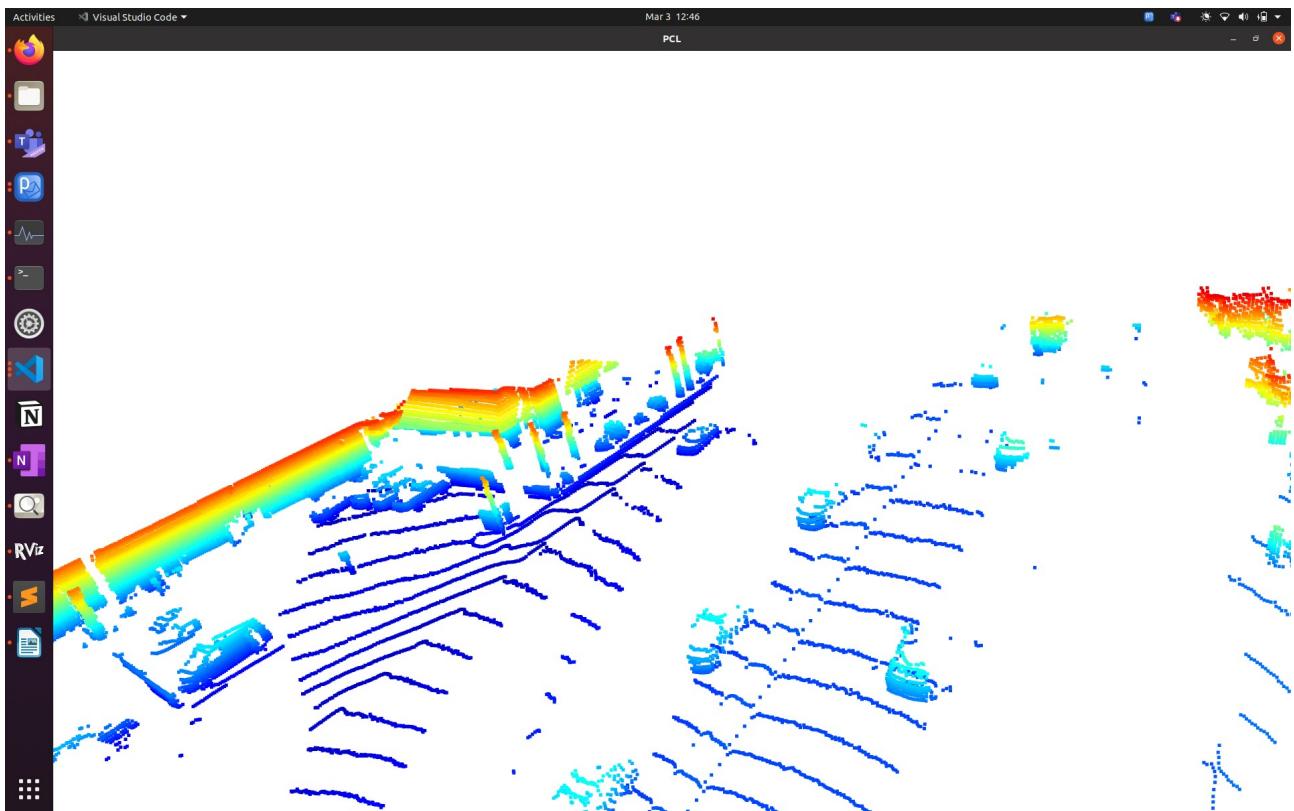
```
ri_range = ri_range * 255 / (np.amax(ri_range) - np.amin(ri_range))
```

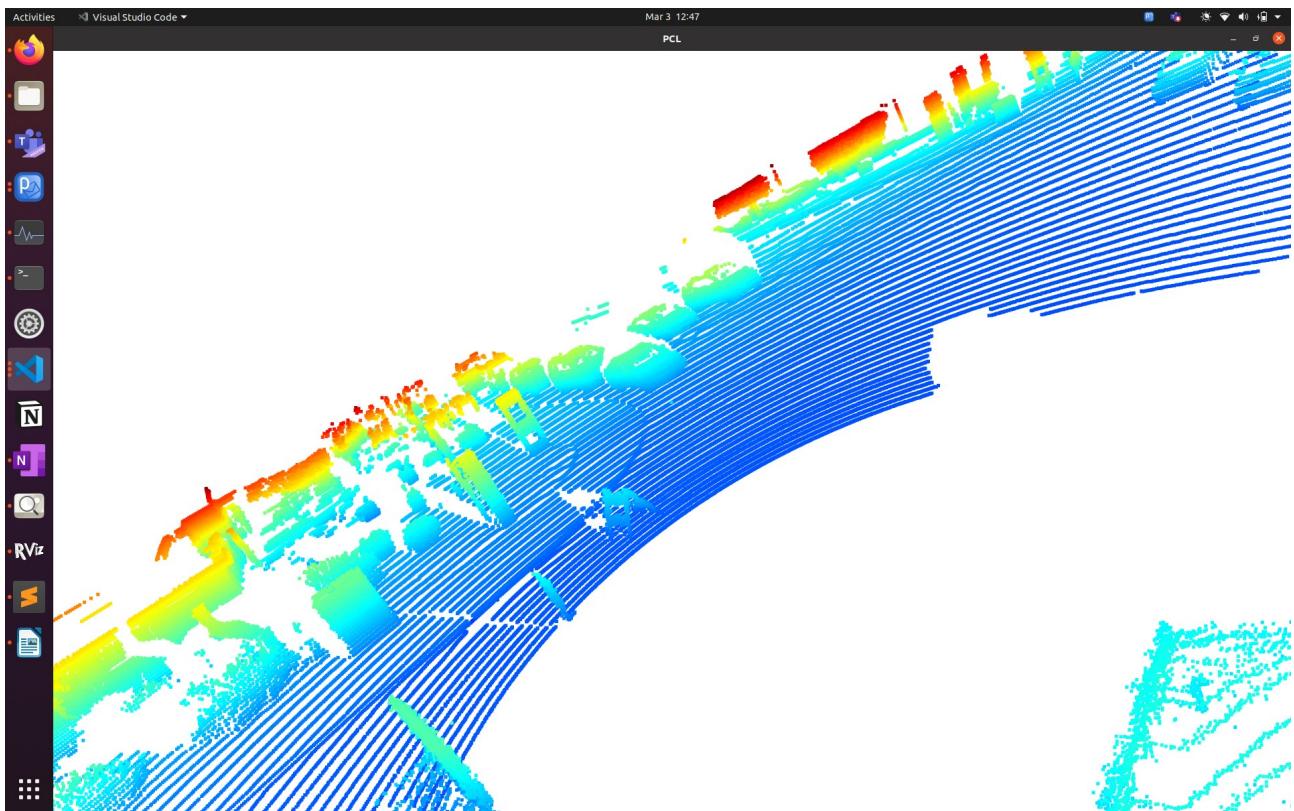
```
ri_intensity = np.amax(ri_intensity)/2 * ri_intensity * 255 / (np.amax(ri_intensity) - np.amin(ri_intensity))
```

## Visualize lidar point-cloud (ID\_S1\_EX2)

In this section we are visualizing the point cloud in a 3D environment as shown in the images below with the purpose of identifying the main and most stable vehicle features no matter the car's degree of visibility. Here are 5 images showing different cars in different visibility conditions:







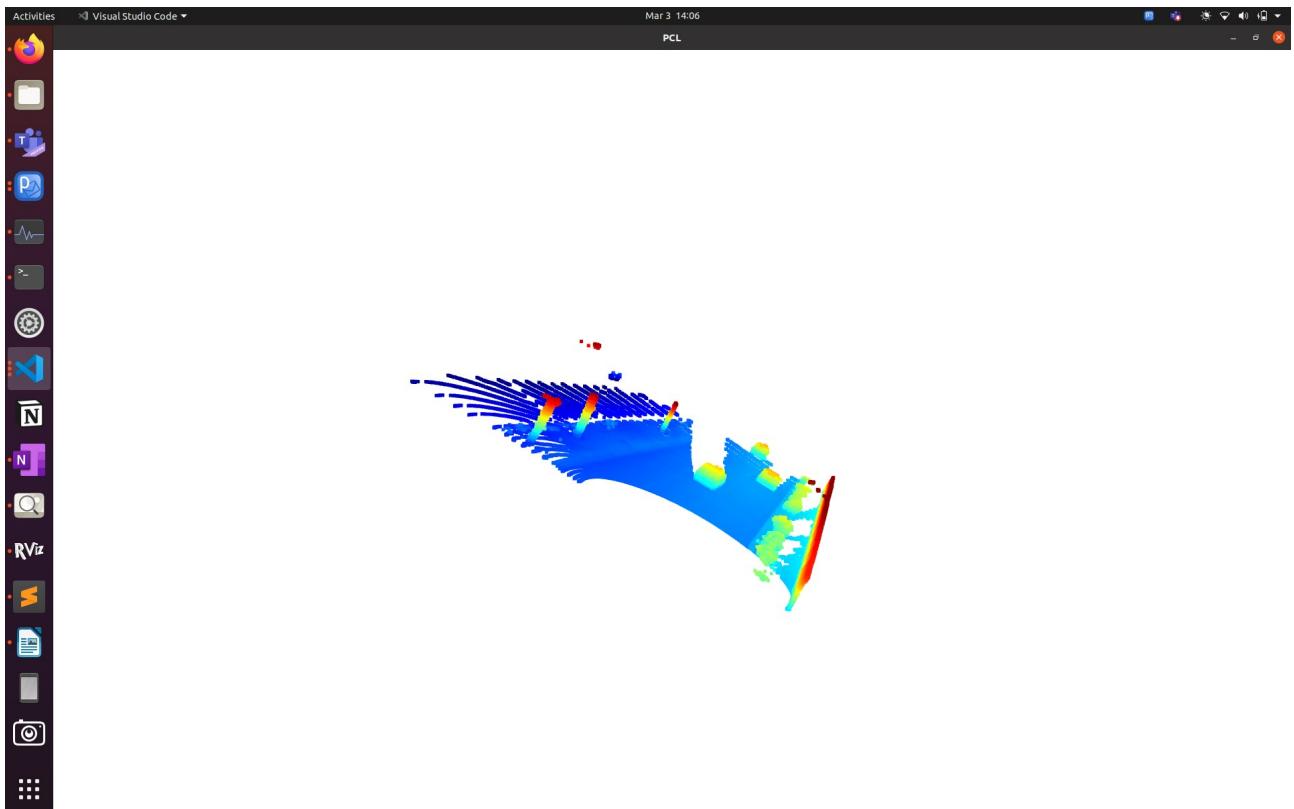
The main and most stable vehicle features are:

- Horizontal surface corresponding to the vehicle's roof.
- Void in the pcl corresponding to the windows.
- A big cluster of points corresponding to the hood of the car.
- If you visualize the color of the points depending on the intensity channel, the license of the cars (and even the front and back lights) are highlighted.

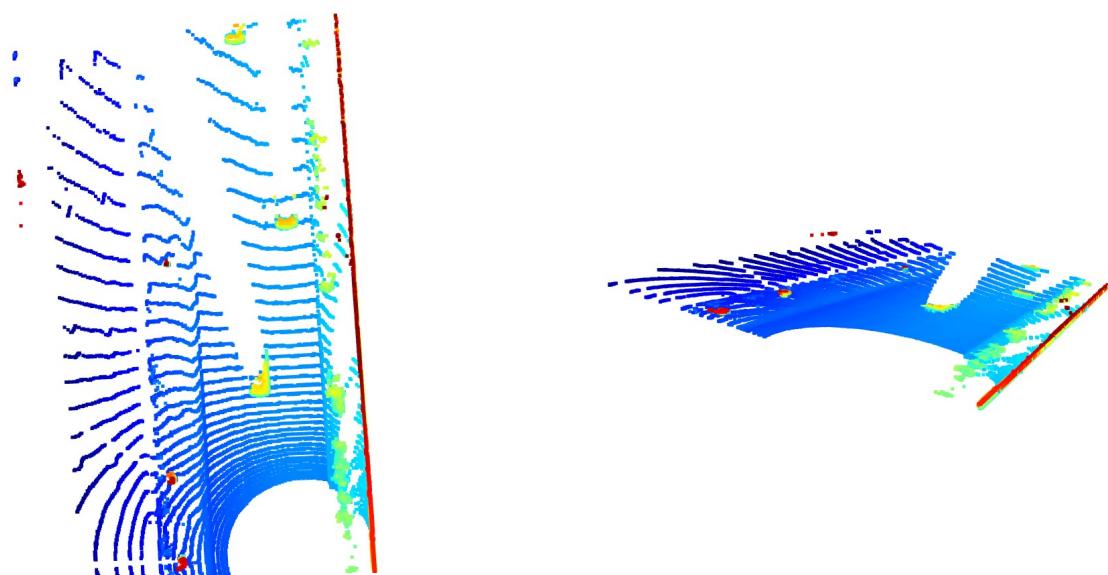
## Section 2 : Create Birds-Eye View from Lidar PCL

### Convert sensor coordinates to BEV-map coordinates (ID\_S2\_EX1)

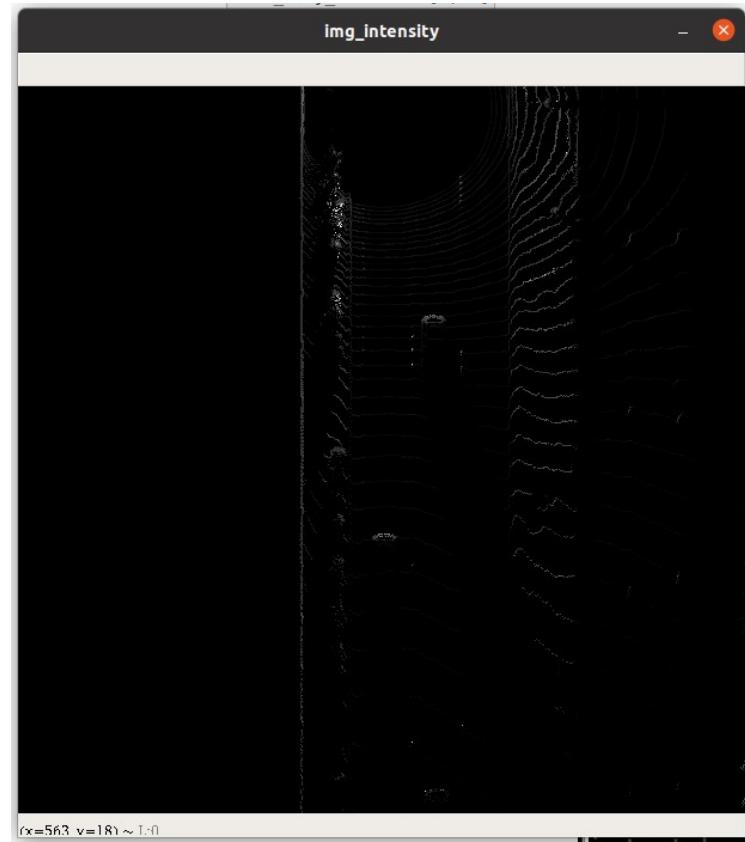
In this first image we can see the pointcloud in the same way as in the previous section:



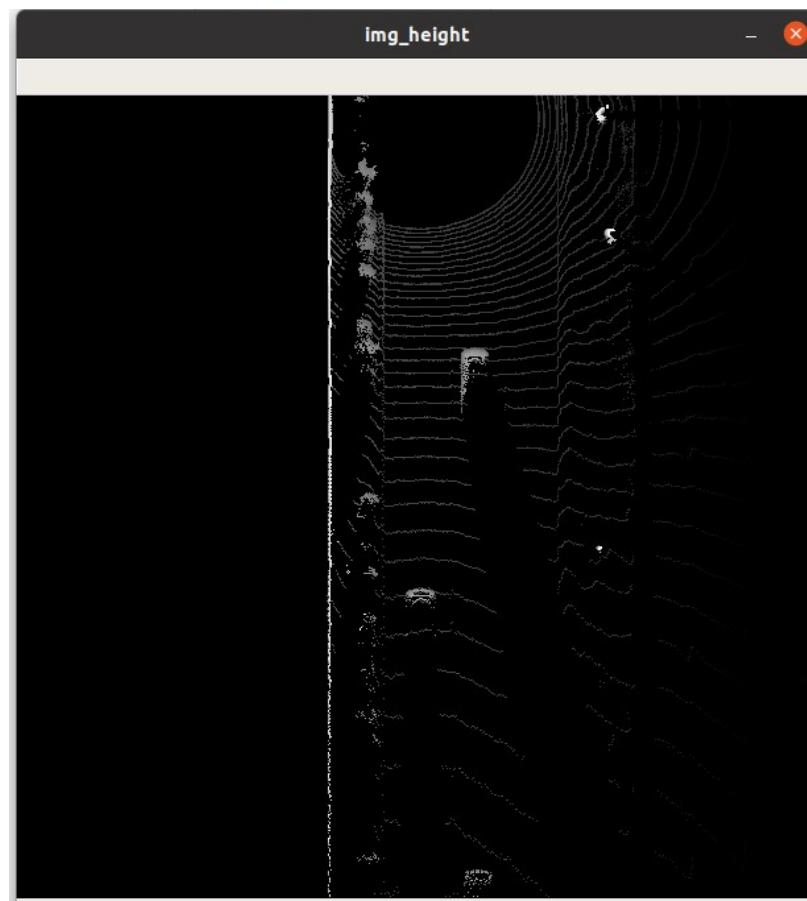
Now, we transform the coordinates of that pcl and convert the height into a different dimension, with the result shown in the 2 images below:



### Compute intensity layer of the BEV map (ID\_S2\_EX2)

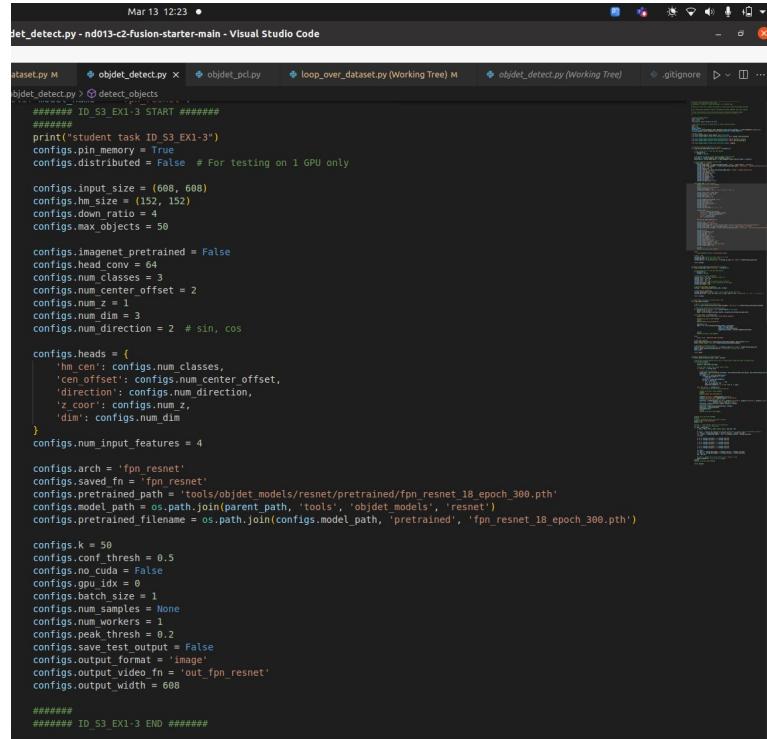


### Compute height layer of the BEV map (ID\_S2\_EX3)



# Section 3 : Model-based Object Detection in BEV Image

## Add a second model from a GitHub repo (ID\_S3\_EX1)



```
Mar 13 12:23 •
dataset.py M objdet_detect.py x objdet_pcl.py loop_over_dataset.py (Working Tree) M object_detect.py (Working Tree) .gitignore ...
objdet_detect.py > detect_objects
#####
## ID_S3_EX1-3 START #####
#####
print("student task ID_S3_EX1-3")
configs.pin_memory = True
configs.distributed = False # For testing on 1 GPU only

configs.input_size = (608, 608)
configs.hm_size = (152, 152)
configs.down_ratio = 4
configs.max_objects = 50

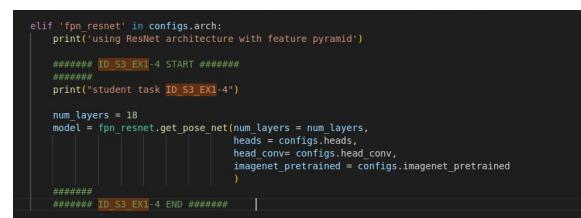
configs.imagenet_pretrained = False
configs.head_conv = 64
configs.num_classes = 3
configs.num_center_offset = 2
configs.num_z = 1
configs.num_dim = 3
configs.num_direction = 2 # sin, cos

configs.heads = {
    'hm_cen': configs.num_classes,
    'cen_offset': configs.num_center_offset,
    'direction': configs.num_direction,
    'z_coor': configs.num_z,
    'dim': configs.num_dim
}
configs.num_input_features = 4

configs.arch = 'fpn_resnet'
configs.saved_fn = 'fpn_resnet'
configs.pretrained_path = 'tools/objdet_models/resnet/pretrained/fpn_resnet_18_epoch_300.pth'
configs.model_path = os.path.join(parent_path, 'tools', 'objdet_models', 'resnet')
configs.pretrained_filename = os.path.join(configs.model_path, 'pretrained', 'fpn_resnet_18_epoch_300.pth')

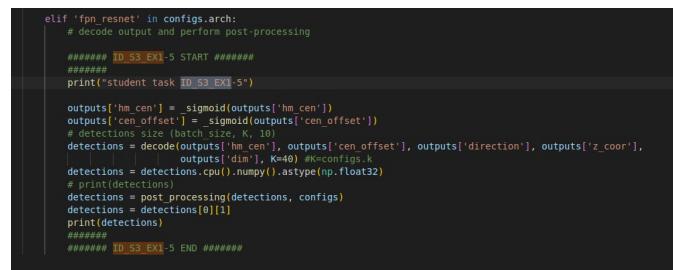
configs.K = 50
configs.conf_thresh = 0.5
configs.no_cuda = False
configs.gpu_idx = 0
configs.batch_size = 1
configs.num_samples = None
configs.num_workers = 1
configs.peak_thresh = 0.2
configs.save_test_output = False
configs.output_format = 'image'
configs.output_video_fn = 'out_fpn_resnet'
configs.output_width = 608

#####
## ID_S3_EX1-3 END #####
```



```
elif 'fpn_resnet' in configs.arch:
    print('using ResNet architecture with feature pyramid')
    #####
    ## ID_S3_EX1-4 START #####
    #####
    print("student task ID_S3_EX1-4")

    num_layers = 18
    model = fpn_resnet.get_pose_net(num_layers = num_layers,
                                    heads = configs.heads,
                                    head_conv = configs.head_conv,
                                    imagenet_pretrained = configs.imagenet_pretrained
                                    )
    #####
    ## ID_S3_EX1-4 END ##### |
```



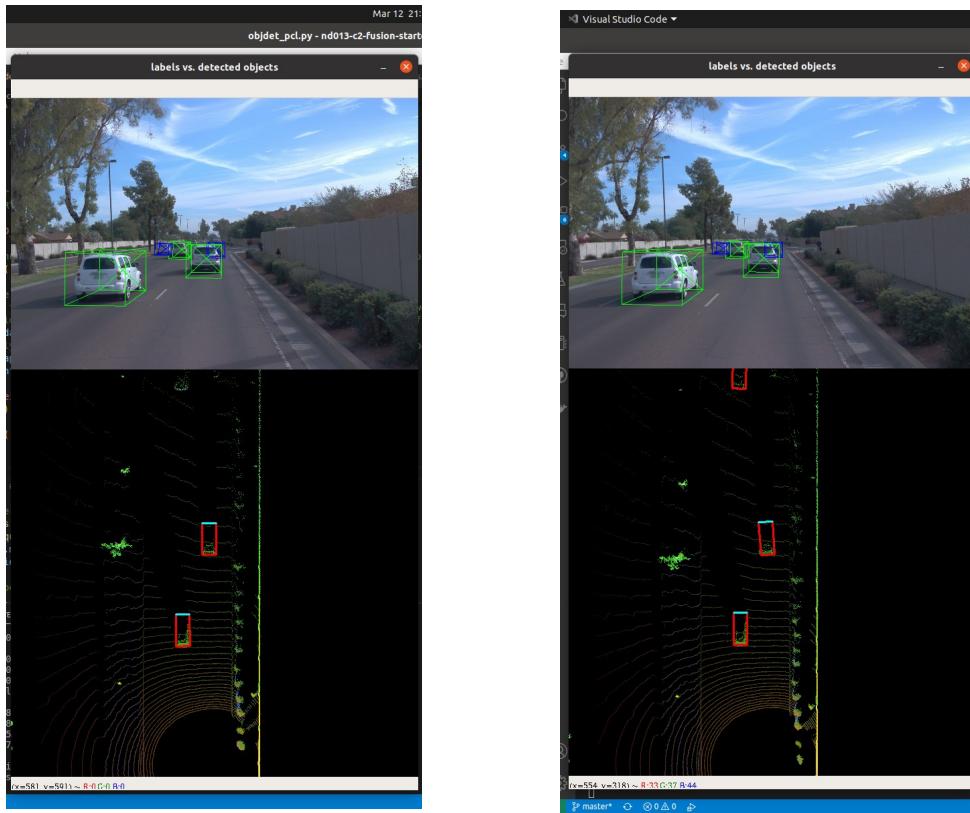
```
elif 'fpn_resnet' in configs.arch:
    # decode output and perform post-processing
    #####
    ## ID_S3_EX1-5 START #####
    #####
    print("student task ID_S3_EX1-5")

    outputs['hm_cen'] = sigmoid(outputs['hm_cen'])
    outputs['cen_offset'] = sigmoid(outputs['cen_offset'])

    detections = decode(outputs['hm_cen'], outputs['cen_offset'], outputs['direction'], outputs['z_coor'],
                         outputs['dim'], K=40) #K=configs.K
    detections = detections.cpu().numpy().astype(np.float32)
    # print(detections)
    detections = post_processing(detections, configs)
    detections = detections[0][1]
    print(detections)
    #####
    ## ID_S3_EX1-5 END #####
```

## Extract 3D bounding boxes from model response (ID\_S3\_EX2)

In this section we are working with the transformed pcl into RGB image which is fed to the model generated in the previous task. The detections predicted by that model can be seen in the images below:



# Section 4 : Performance Evaluation for Object Detection

## Compute intersection-over-union between labels and detections (ID\_S4\_EX1)

```
student > objdet_eval.py > measure_detection_performance
40     for label, valid in zip(labels, labels_valid):
41         matches_lab_det = []
42         if valid: # exclude all labels from statistics which are not considered valid
43             # compute intersection over union (iou) and distance between centers
44
45             ##### ID_S4_EX1 START #####
46             print("student task ID_S4_EX1 ")
47
48             ## step 1 : extract the four corners of the current label bounding-box
49             label_corners = tools.compute_box_corners(label.box.center_x, label.box.center_y, label.box.width, label.box.length, label.box.heading)
50
51             ## step 2 : loop over all detected objects
52             for det in detections:
53                 ## step 3 : extract the four corners of the current detection
54                 det_corners = tools.compute_box_corners(det[1], det[2], det[5], det[6], det[7])
55
56                 ## step 4 : compute the center distance between label and detection bounding-box in x, y, and z
57                 dist_x = label.box.center_x - det[1]
58                 dist_y = label.box.center_y - det[2]
59                 dist_z = label.box.center_z - det[3]
60
61                 ## step 5 : compute the intersection over union (IOU) between label and detection bounding-box
62                 label_poly = Polygon(label_corners)
63                 det_poly = Polygon(det_corners)
64                 i = label_poly.intersection(det_poly).area
65                 U = label_poly.union(det_poly).area
66                 IOU = i/U
67
68                 ## step 6 : if IOU exceeds min_iou threshold, store [iou,dist_x,dist_y,dist_z] in matches_lab_det and increase the TP count
69                 if(IOU >= min_iou):
70                     matches_lab_det.append([IOU,dist_x,dist_y,dist_z])
71                     true_positives += 1
72
73             ##### ID_S4_EX1 END #####
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
```

The screenshot shows a Visual Studio Code interface with the file `objdet_eval.py` open. The code implements a function to calculate the Intersection-over-Union (IoU) between labels and detections. It uses the `tools.compute_box_corners` function to extract the four corners of each bounding box. It then calculates the center distance between the label and detection boxes in three dimensions (x, y, z). Finally, it computes the IoU using the `Polygon` class and stores the results in the `matches_lab_det` list. The code includes comments for steps 1 through 6 and handles both valid and invalid labels.

## Compute false-negatives and false-positives (ID\_S4\_EX2)

```
student > objdet_eval.py > measure_detection_performance
70     ##### ID_S4_EX2 INIT #####
71
72     # find best match and compute metrics
73     if matches_lab_det:
74         best_match = max(matches_lab_det,key=itemgetter(1)) # retrieve entry with max iou in case of multiple candidates
75         ious.append(best_match[0])
76         center_devs.append(best_match[1:])
77
78
79     ##### ID_S4_EX2 START #####
80     print("student task ID_S4_EX2")
81
82     # compute positives and negatives for precision/recall
83     true_positives = len(ious)
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
```

The screenshot shows a Visual Studio Code interface with the file `objdet_eval.py` open. The code implements a function to calculate false-negatives and false-positives. It first finds the best match between labels and detections based on IoU. Then, it initializes lists for `ious` and `center_devs`. It prints the student task ID\_S4\_EX2. Finally, it computes the total number of positives present in the scene and the number of false negatives and false positives. The code includes comments for steps 1 through 3 and handles both valid and invalid labels.

## Compute precision and recall (ID\_S4\_EX3)

In this last task, we are computing the precision and recall of the model. In the image below is shown a frame where the model is predicting the presence of a car due to the confusion with the plants in the side of the road (false positive).



The precision and recall for the 50 first frames are:

