# SDCND : Sensor Fusion and Tracking
This is the project for the second course in the [Udacity Self-Driving Car Engineer Nanodegree Program](https://www.udacity.com/course/c-plus-plus-nanodegree--nd213) : Sensor Fusion and Tracking.

In this project, you'll fuse measurements from LiDAR and camera and track vehicles over time. You will be using real-world data from the Waymo Open Dataset, detect objects in 3D point clouds and apply an extended Kalman filter for sensor fusion and tracking.

The project consists of two major parts:
1. **Object detection**: In this part, a deep-learning approach is used to detect vehicles in LiDAR data based on a birds-eye view perspective of the 3D point-cloud. Also, a series of performance measures is used to evaluate the performance of the detection approach.
2. **Object tracking** : In this part, an extended Kalman filter is used to track vehicles over time, based on the lidar detections fused with camera detections. Data association and track management are implemented as well.

The following diagram contains an outline of the data flow and of the individual steps that make up the algorithm.

<img src="img/img_title_2_new.png"/>

Also, the project code contains various tasks, which are detailed step-by-step in the code. More information on the algorithm and on the tasks can be found in the Udacity classroom.

## Project File Structure

```
 project<br>
┣   dataset --> contains the Waymo Open Dataset sequences <br>
┃<br>
┣   misc<br>
┃ ┣ evaluation.py --> plot functions for tracking visualization and RMSE calculation<br>
┃ ┣ helpers.py --> misc. helper functions, e.g. for loading / saving binary files<br>
┃ ┗ objdet_tools.py --> object detection functions without student tasks<br>
┃ ┗ params.py --> parameter file for the tracking part<br>
┃ <br>
┣   results --> binary files with pre-computed intermediate results<br>
┃ <br>
┣   student <br>
┃ ┣ association.py --> data association logic for assigning measurements to tracks incl. student tasks <br>
┃ ┣ filter.py --> extended Kalman filter implementation incl. student tasks <br>
┃ ┣ measurements.py --> sensor and measurement classes for camera and lidar incl. student tasks <br>
┃ ┣ objdet_detect.py --> model-based object detection incl. student tasks <br>
┃ ┣ objdet_eval.py --> performance assessment for object detection incl. student tasks <br>
┃ ┣ objdet_pcl.py --> point-cloud functions, e.g. for birds-eye view incl. student tasks <br>
┃ ┗ trackmanagement.py --> track and track management classes incl. student tasks  <br>
┃ <br>
┣   tools --> external tools<br>
```

```
┃ ┣ objdet_models --> models for object detection<br>
┃ ┃ <br>
┃ ┃ ┣ darknet<br>
┃ ┃ ┃ ┣ config<br>
┃ ┃ ┃ ┣ models --> darknet / yolo model class and tools<br>
┃ ┃ ┃ ┣ pretrained --> copy pre-trained model file here<br>
┃ ┃ ┃ ┃ ┗ complex_yolov4_mse_loss.pth<br>
┃ ┃ ┃ ┣ utils --> various helper functions<br>
┃ ┃ <br>
┃ ┃ ┗ resnet<br>
┃ ┃ ┃ ┣ models --> fpn_resnet model class and tools<br>
┃ ┃ ┃ ┣ pretrained --> copy pre-trained model file here <br>
┃ ┃ ┃ ┃ ┗ fpn_resnet_18_epoch_300.pth <br>
┃ ┃ ┃ ┣ utils --> various helper functions<br>
┃ ┃ <br>
┃ ┗ waymo_reader --> functions for light-weight loading of Waymo sequences<br>
┃ <br>
┣ basic_loop.py<br>
┣ loop_over_dataset.py<br>
```

## Installation Instructions for Running Locally
### Cloning the Project
In order to create a local copy of the project, please click on "Code" and then "Download ZIP". Alternatively, you may of-course use GitHub Desktop or Git Bash for this purpose.

### Python
The project has been written using Python 3.7. Please make sure that your local installation is equal or above this version.

### Package Requirements
All dependencies required for the project have been listed in the file `requirements.txt`. You may either install them one-by-one using pip or you can use the following command to install them all at once:
`pip3 install -r requirements.txt`

### Waymo Open Dataset Reader
The Waymo Open Dataset Reader is a very convenient toolbox that allows you to access sequences from the Waymo Open Dataset without the need of installing all of the heavy-weight dependencies that come along with the official toolbox. The installation instructions can be found in `tools/waymo_reader/README.md`.

### Waymo Open Dataset Files
This project makes use of three different sequences to illustrate the concepts of object detection and tracking. These are:
- Sequence 1 : `training_segment-1005081002024129653_5313_150_5333_150_with_camera_labels.tfrecord`
- Sequence 2 : `training_segment-10072231702153043603_5725_000_5745_000_with_camera_labels.tfrecord`
- Sequence 3 : `training_segment-10963653239323173269_1924_000_1944_000_with_camera_labels.tfrecord`

To download these files, you will have to register with Waymo Open Dataset first: [Open Dataset – Waymo](https://waymo.com/open/terms), if you have not already, making sure to note "Udacity" as your institution.

Once you have done so, please [click here](https://console.cloud.google.com/storage/browser/waymo_open_dataset_v_1_2_0_individual_files) to access the Google Cloud Container that holds all the sequences. Once you have been cleared for access by Waymo (which might take up to 48 hours), you can download the individual sequences.

The sequences listed above can be found in the folder "training". Please download them and put the `tfrecord`-files into the `dataset` folder of this project.

### Pre-Trained Models
The object detection methods used in this project use pre-trained models which have been provided by the original authors. They can be downloaded [here](https://drive.google.com/file/d/1Pqx7sShlqKSGmvshTYbNDcUEYyZwfn3A/view?usp=sharing) (darknet) and [here](https://drive.google.com/file/d/1RcEfUIF1pzDZco8PJkZ10OL-wLL2usEj/view?usp=sharing) (fpn_resnet). Once downloaded, please copy the model files into the paths `/tools/objdet_models/darknet/pretrained` and `/tools/objdet_models/fpn_resnet/pretrained` respectively.

### Using Pre-Computed Results

In the main file `loop_over_dataset.py`, you can choose which steps of the algorithm should be executed. If you want to call a specific function, you simply need to add the corresponding string literal to one of the following lists:

- `exec_data` : controls the execution of steps related to sensor data.
  - `pcl_from_rangeimage` transforms the Waymo Open Data range image into a 3D point-cloud
  - `load_image` returns the image of the front camera

- `exec_detection` : controls which steps of model-based 3D object detection are performed
  - `bev_from_pcl` transforms the point-cloud into a fixed-size birds-eye view perspective
  - `detect_objects` executes the actual detection and returns a set of objects (only vehicles)
  - `validate_object_labels` decides which ground-truth labels should be considered (e.g. based on difficulty or visibility)
  - `measure_detection_performance` contains methods to evaluate detection performance for a single frame

In case you do not include a specific step into the list, pre-computed binary files will be loaded instead. This enables you to run the algorithm and look at the results even without having implemented anything yet. The pre-computed results for the mid-term project need to be loaded using [this](https://drive.google.com/drive/folders/1-s46dKSrtx8rrNwnObGbly2nO3i4D7r7?usp=sharing) link. Please use the folder `darknet` first. Unzip the file within and put its content into the folder `results`.

- `exec_tracking` : controls the execution of the object tracking algorithm

- `exec_visualization` : controls the visualization of results

  - `show_range_image` displays two LiDAR range image channels (range and intensity)
  - `show_labels_in_image` projects ground-truth boxes into the front camera image
  - `show_objects_and_labels_in_bev` projects detected objects and label boxes into the birds-eye view
  - `show_objects_in_bev_labels_in_camera` displays a stacked view with labels inside the camera image on top and the birds-eye view with detected objects on the bottom
  - `show_tracks` displays the tracking results
  - `show_detection_performance` displays the performance evaluation based on all detected
  - `make_tracking_movie` renders an output movie of the object tracking results

Even without solving any of the tasks, the project code can be executed.

The final project uses pre-computed lidar detections in order for all students to have the same input data. If you use the workspace, the data is prepared there already. Otherwise, [download the pre-computed lidar detections](https://drive.google.com/drive/folders/1IkqFGYTF6Fh_d8J3UjQOSNJ2V42UDZpO?usp=sharing) (~1 GB), unzip them and put them in the folder `results`.

## External Dependencies
Parts of this project are based on the following repositories:
- [Simple Waymo Open Dataset Reader](https://github.com/gdlg/simple-waymo-open-dataset-reader)
- [Super Fast and Accurate 3D Object Detection based on 3D LiDAR Point Clouds](https://github.com/maudzung/SFA3D)
- [Complex-YOLO: Real-time 3D Object Detection on Point Clouds](https://github.com/maudzung/Complex-YOLOv4-Pytorch)

## License
[License](LICENSE.md)

This is a template submission for the midterm course : 3D Object Detection (Midterm).

3D Object detection

We have used the Waymo dataset real-world data and used 3d point cloud for lidar based object detection.

- Configuring the ranges channel to 8 bit and view the range /intensity image (ID_S1_EX1)
- Use the Open3D library to display the lidar point cloud on a 3d viewer and identifying 6 images from point cloud.(ID_S1_EX2)
- Create Birds Eye View perspective (BEV) of the point cloud,assign lidar intensity values to BEV,normalize the heightmap of each BEV (ID_S2_EX1,ID_S2_EX2,ID_S2_EX3)
- In addition to YOLO, use the repository [https://github.com/maudzung/SFA3D.git]and add parameters ,instantiate fpn resnet model(ID_S3_EX1)
- Convert BEV coordinates into pixel coordinates and convert model output to bounding box format (ID_S3_EX2)
- Compute intersection over union, assign detected objects to label if IOU exceeds threshold (ID_S4_EX1)
- Compute false positives and false negatives, precision and recall(ID_S4_EX2,ID_S4_EX3)

The project can be run by running

python loop_over_dataset.py

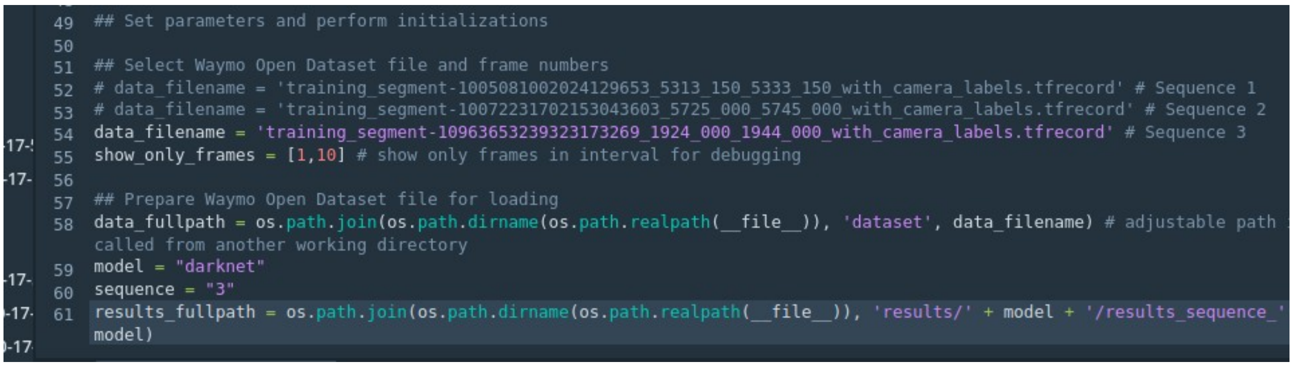
All training/inference is done on udacity workspace.

Step-1: Compute Lidar point cloud from Range Image

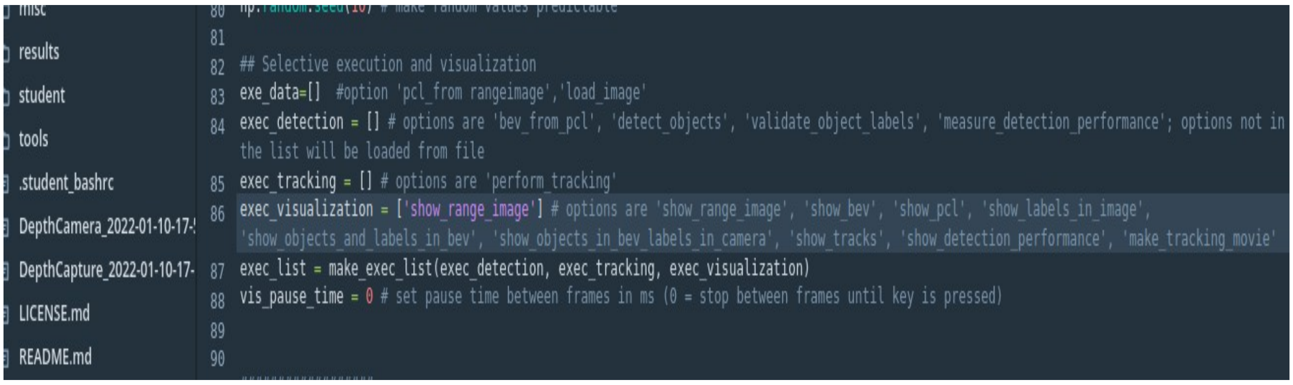
In this we are first previewing the range image and convert range and intensity channels to 8 bit format. After that, we use the openCV library to stack the range and intensity channel vertically to visualize the image.

- Convert "range" channel to 8 bit
- Convert "intensity" channel to 8 bit
- Crop range image to +/- 90 degrees left and right of forward facing x axis
- Stack up range and intensity channels vertically in openCV

The changes are made in 'loop_over_dataset.py'

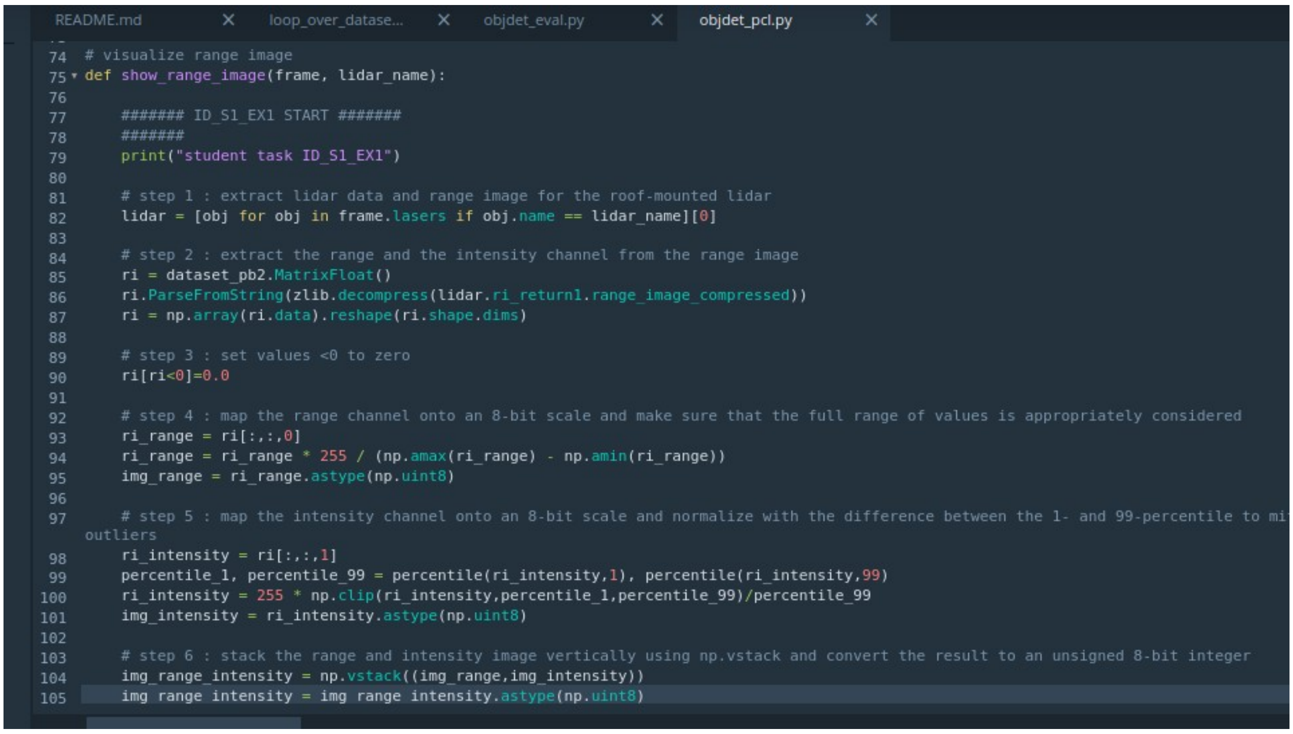


```
## Set parameters and perform initializations

## Select Waymo Open Dataset file and frame numbers
# data_filename = 'training_segment-1005081002024129653_5313_150_5333_150_with_camera_labels.tfrecord' # Sequence 1
# data_filename = 'training_segment-10072231702153043603_5725_000_5745_000_with_camera_labels.tfrecord' # Sequence 2
data_filename = 'training_segment-10963653239323173269_1924_000_1944_000_with_camera_labels.tfrecord' # Sequence 3
show_only_frames = [1,10] # show only frames in interval for debugging

## Prepare Waymo Open Dataset file for loading
data_fullpath = os.path.join(os.path.dirname(os.path.realpath(__file__)), 'dataset', data_filename) # adjustable path
called from another working directory
model = "darknet"
sequence = "3"
results_fullpath = os.path.join(os.path.dirname(os.path.realpath(__file__)), 'results/' + model + '/results_sequence_'
model)
```



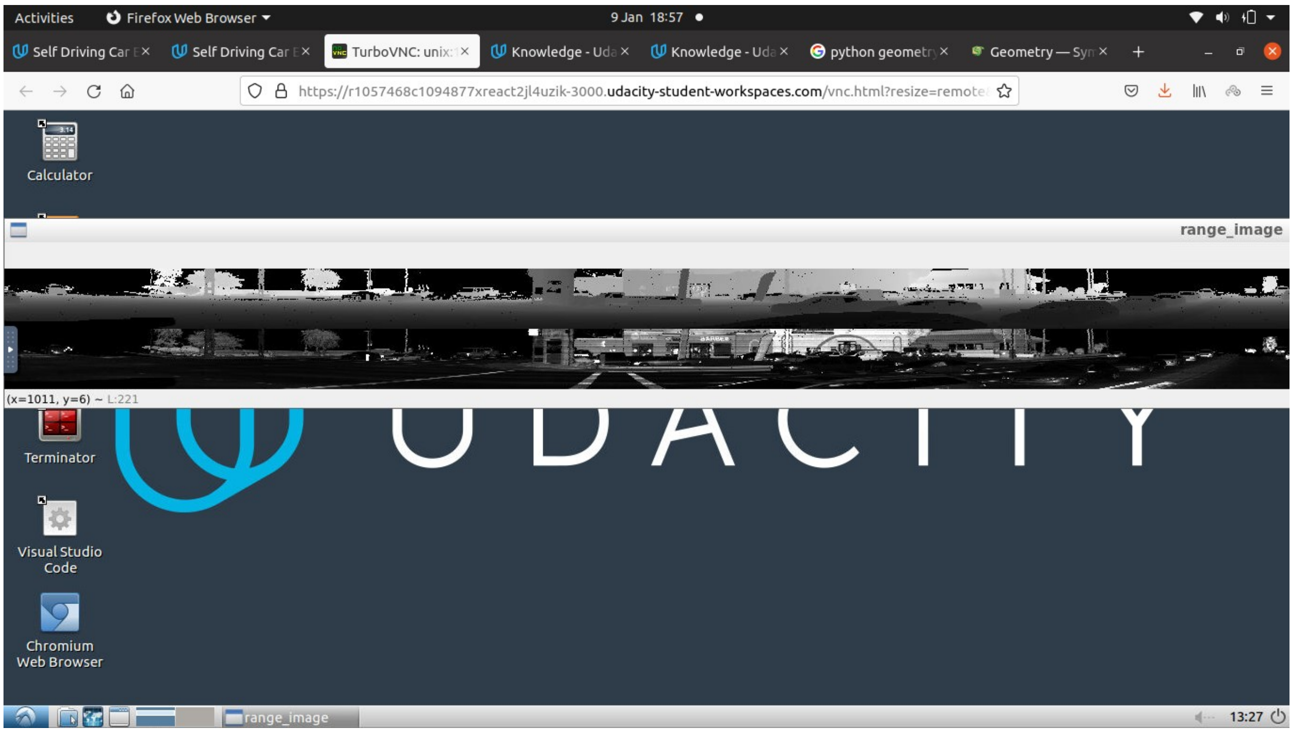
```python
np.random.seed(10) # make random values predictable

## Selective execution and visualization
exe_data=[]  #option 'pcl_from_rangeimage','load_image'
exec_detection = [] # options are 'bev_from_pcl', 'detect_objects', 'validate_object_labels', 'measure_detection_performance'; options not in the list will be loaded from file
exec_tracking = [] # options are 'perform_tracking'
exec_visualization = ['show_range_image'] # options are 'show_range_image', 'show_bev', 'show_pcl', 'show_labels_in_image', 'show_objects_and_labels_in_bev', 'show_objects_in_bev_labels_in_camera', 'show_tracks', 'show_detection_performance', 'make_tracking_movie'
exec_list = make_exec_list(exec_detection, exec_tracking, exec_visualization)
vis_pause_time = 0 # set pause time between frames in ms (0 = stop between frames until key is pressed)
```

The changes are made in "objdet_pcl.py"



```python
# visualize range image
def show_range_image(frame, lidar_name):

    ####### ID_S1_EX1 START #######
    #######
    print("student task ID_S1_EX1")

    # step 1 : extract lidar data and range image for the roof-mounted lidar
    lidar = [obj for obj in frame.lasers if obj.name == lidar_name][0]

    # step 2 : extract the range and the intensity channel from the range image
    ri = dataset_pb2.MatrixFloat()
    ri.ParseFromString(zlib.decompress(lidar.ri_return1.range_image_compressed))
    ri = np.array(ri.data).reshape(ri.shape.dims)

    # step 3 : set values <0 to zero
    ri[ri<0]=0.0

    # step 4 : map the range channel onto an 8-bit scale and make sure that the full range of values is appropriately considered
    ri_range = ri[:,:,0]
    ri_range = ri_range * 255 / (np.amax(ri_range) - np.amin(ri_range))
    img_range = ri_range.astype(np.uint8)

    # step 5 : map the intensity channel onto an 8-bit scale and normalize with the difference between the 1- and 99-percentile to mi
outliers
    ri_intensity = ri[:,:,1]
    percentile_1, percentile_99 = percentile(ri_intensity,1), percentile(ri_intensity,99)
    ri_intensity = 255 * np.clip(ri_intensity,percentile_1,percentile_99)/percentile_99
    img_intensity = ri_intensity.astype(np.uint8)

    # step 6 : stack the range and intensity image vertically using np.vstack and convert the result to an unsigned 8-bit integer
    img_range_intensity = np.vstack((img_range,img_intensity))
    img_range_intensity = img_range_intensity.astype(np.uint8)
```

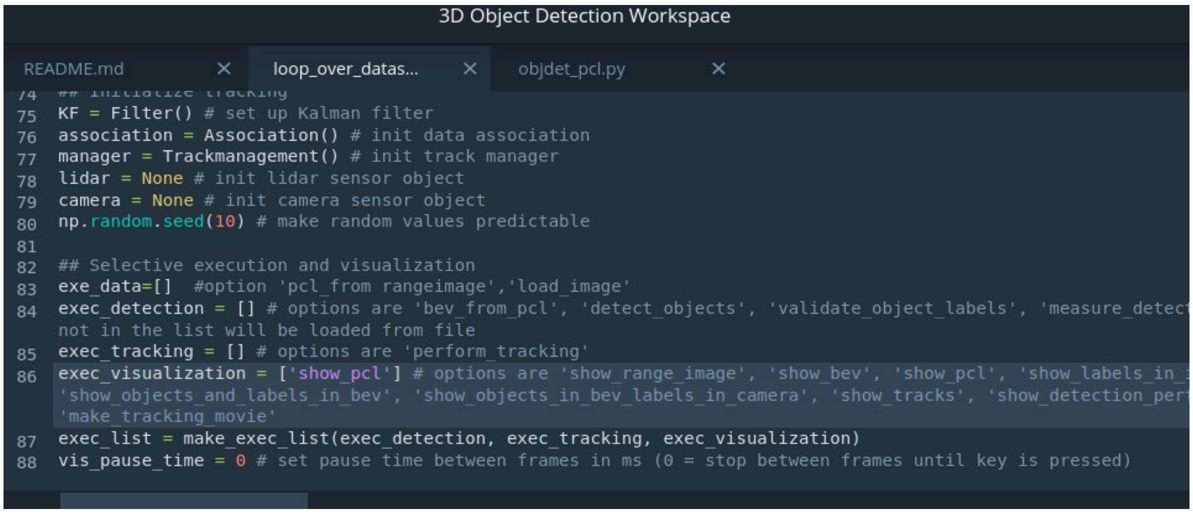
The range image sample:



For the next part, we use the Open3D library to display the lidar point cloud on a 3D viewer and identify 10 images from point cloud

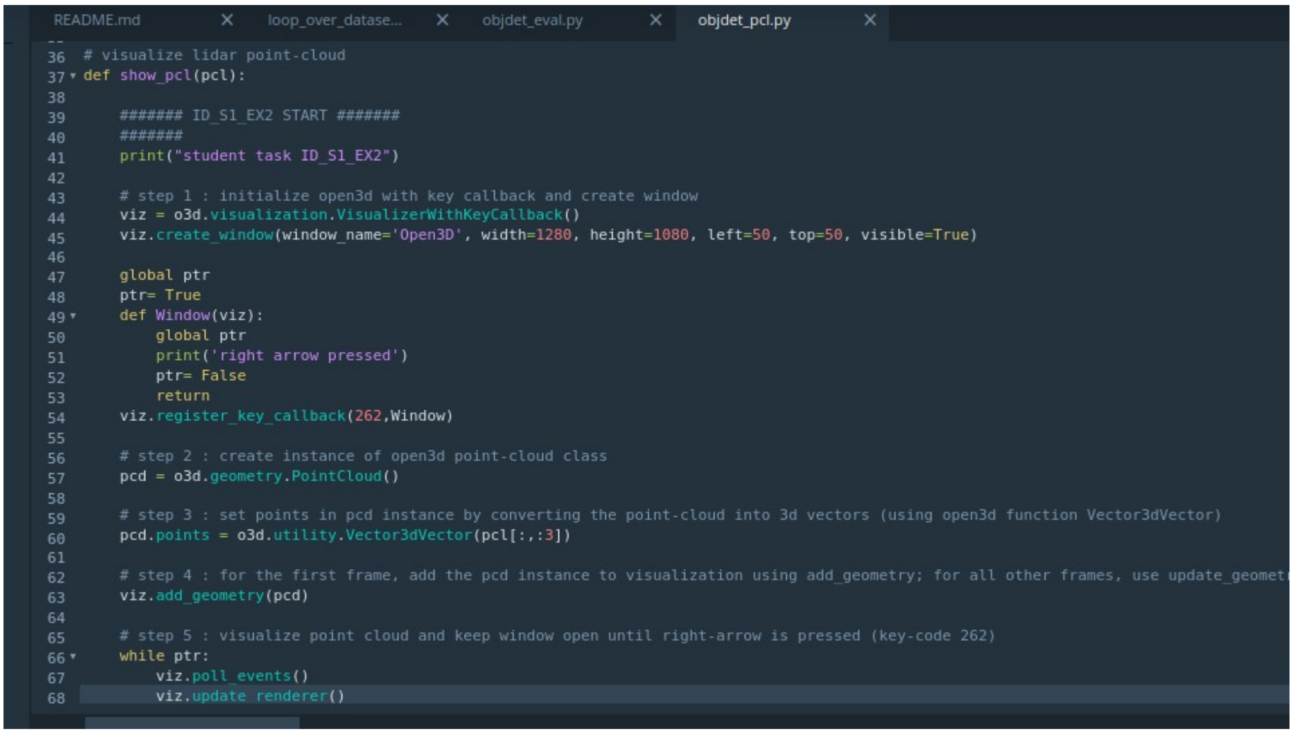
- Visualize the point cloud in Open3D
- 10 examples from point cloud with varying degrees of visibility

The changes are made in 'loop_over_dataset.py'



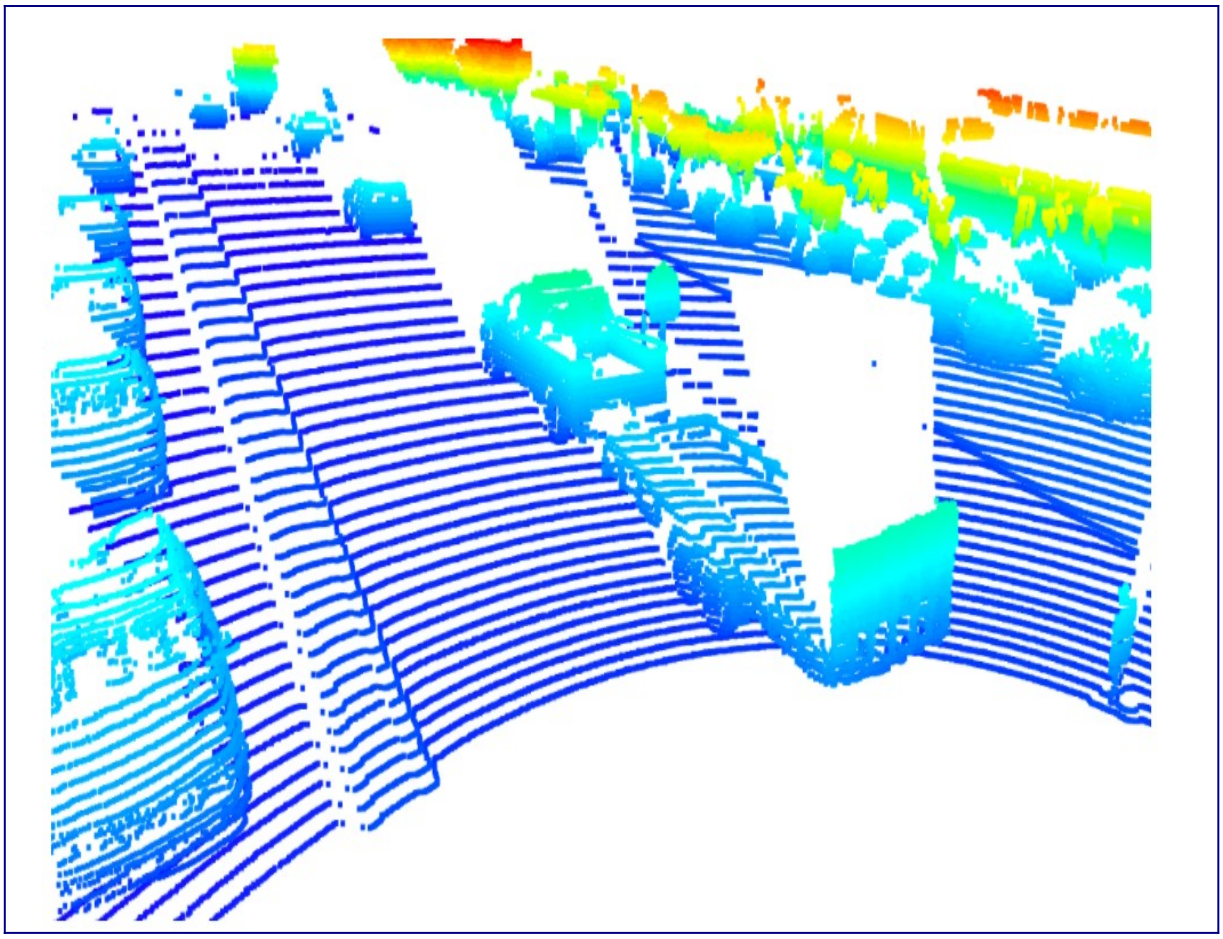
```
KF = Filter() # set up Kalman filter
association = Association() # init data association
manager = Trackmanagement() # init track manager
lidar = None # init lidar sensor object
camera = None # init camera sensor object
np.random.seed(10) # make random values predictable

## Selective execution and visualization
exe_data=[]  #option 'pcl_from rangeimage','load_image'
exec_detection = [] # options are 'bev_from_pcl', 'detect_objects', 'validate_object_labels', 'measure_detec
not in the list will be loaded from file
exec_tracking = [] # options are 'perform_tracking'
exec_visualization = ['show_pcl'] # options are 'show_range_image', 'show_bev', 'show_pcl', 'show_labels_in_
'show_objects_and_labels_in_bev', 'show_objects_in_bev_labels_in_camera', 'show_tracks', 'show_detection_per
'make_tracking_movie'
exec_list = make_exec_list(exec_detection, exec_tracking, exec_visualization)
vis_pause_time = 0 # set pause time between frames in ms (0 = stop between frames until key is pressed)
```

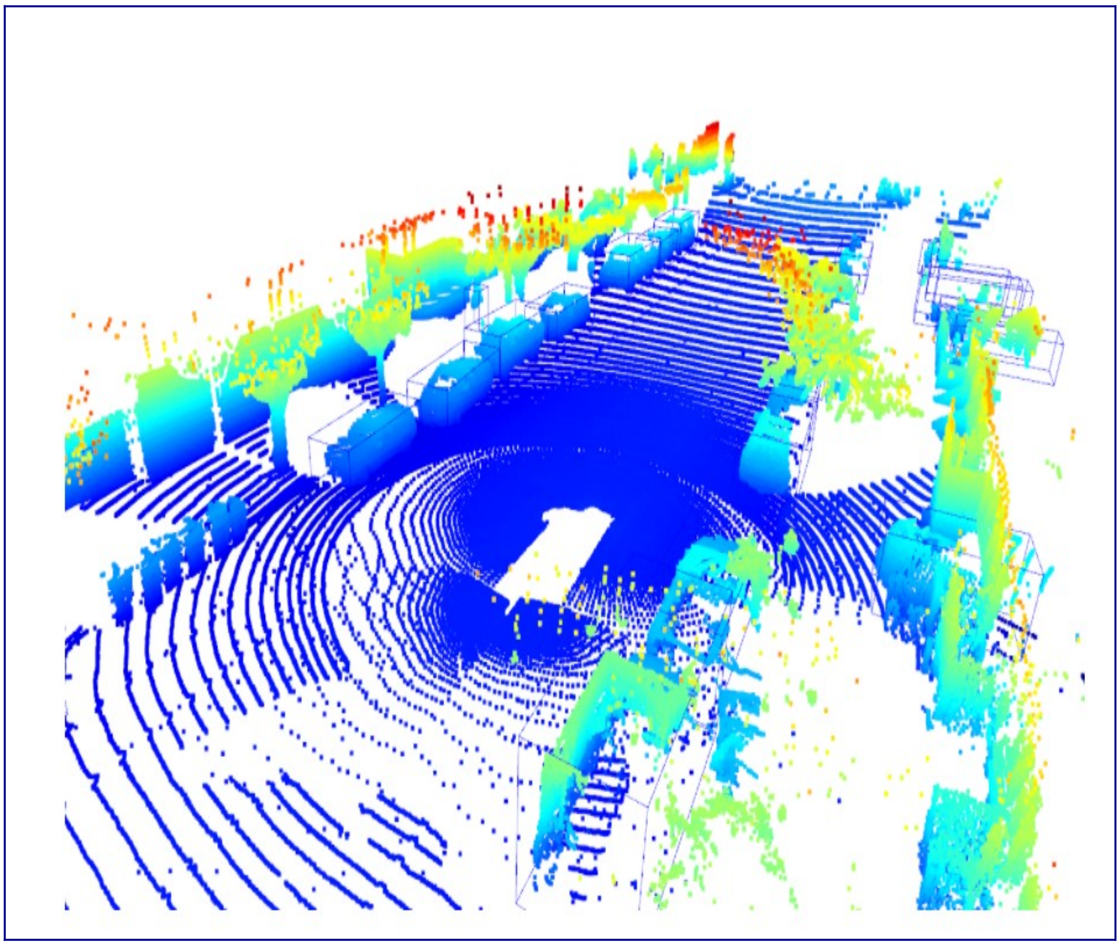
The changes are made in "objdet_pcl.py"

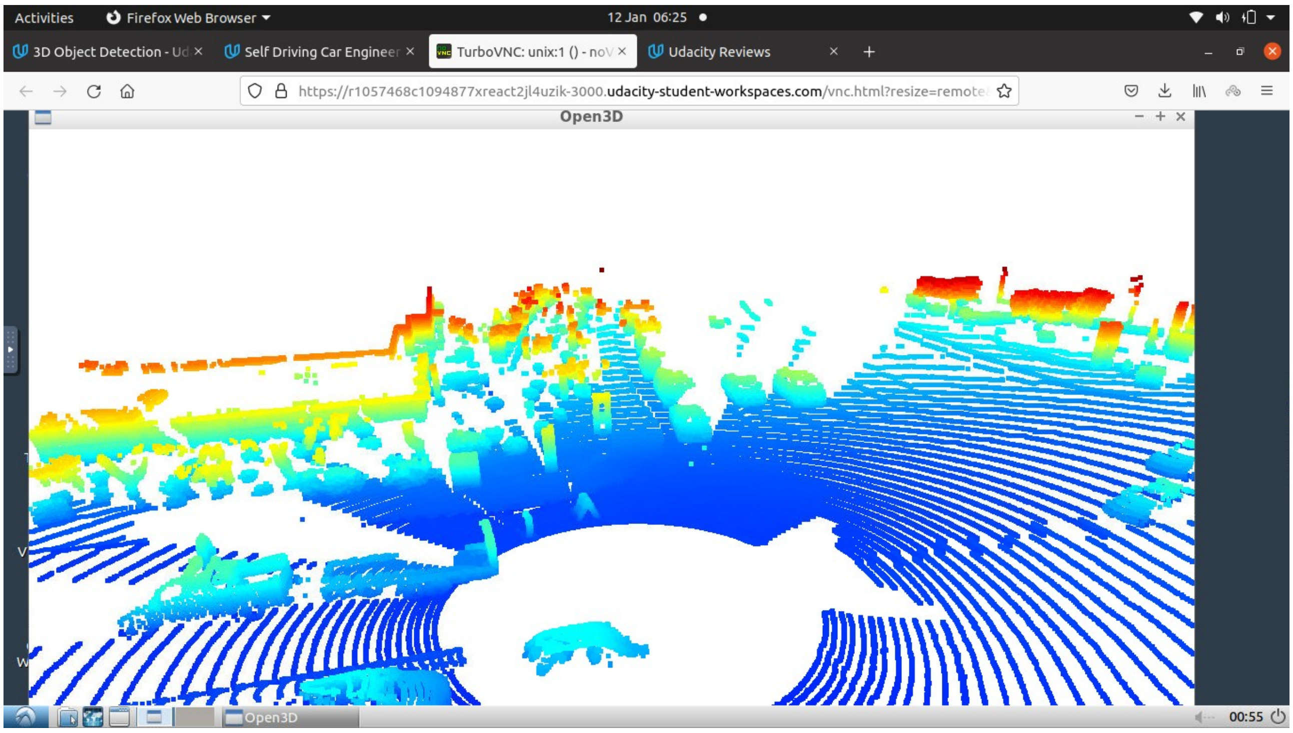


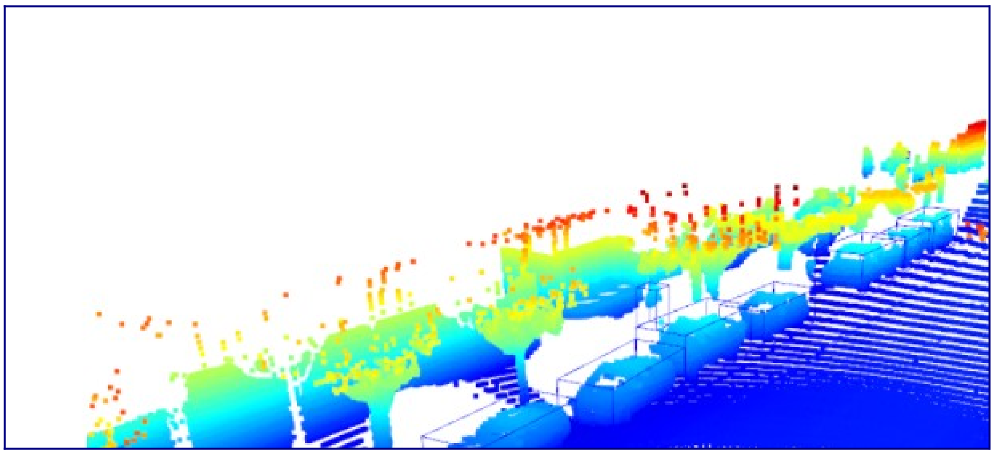
```python
# visualize lidar point-cloud
def show_pcl(pcl):

    ####### ID_S1_EX2 START #######
    #######
    print("student task ID_S1_EX2")

    # step 1 : initialize open3d with key callback and create window
    viz = o3d.visualization.VisualizerWithKeyCallback()
    viz.create_window(window_name='Open3D', width=1280, height=1080, left=50, top=50, visible=True)

    global ptr
    ptr= True
    def Window(viz):
        global ptr
        print('right arrow pressed')
        ptr= False
        return
    viz.register_key_callback(262,Window)

    # step 2 : create instance of open3d point-cloud class
    pcd = o3d.geometry.PointCloud()

    # step 3 : set points in pcd instance by converting the point-cloud into 3d vectors (using open3d function Vector3dVector)
    pcd.points = o3d.utility.Vector3dVector(pcl[:,:3])

    # step 4 : for the first frame, add the pcd instance to visualization using add_geometry; for all other frames, use update_geomet
    viz.add_geometry(pcd)

    # step 5 : visualize point cloud and keep window open until right-arrow is pressed (key-code 262)
    while ptr:
        viz.poll_events()
        viz.update_renderer()
```

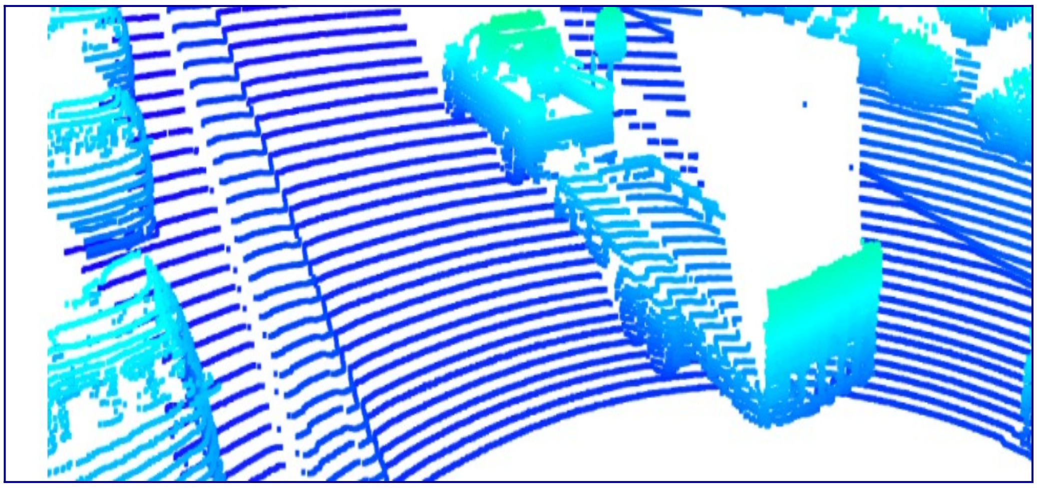
Point cloud images

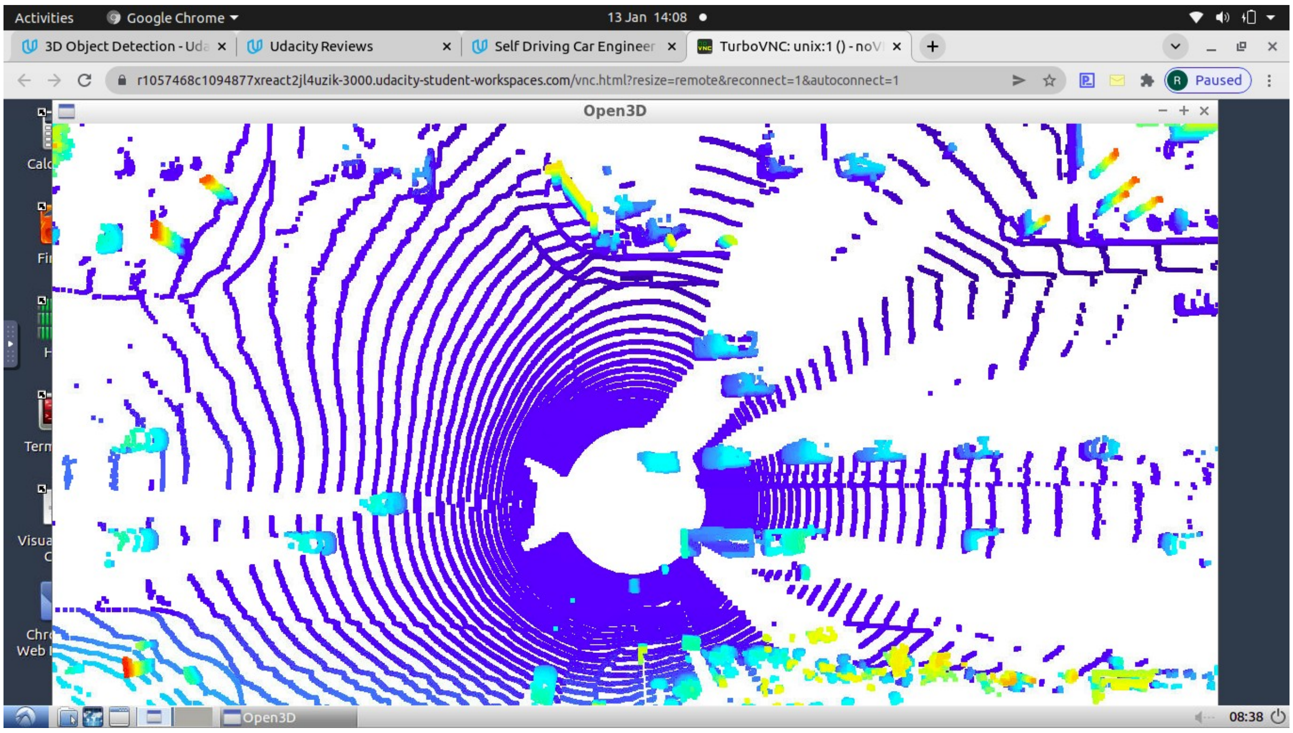


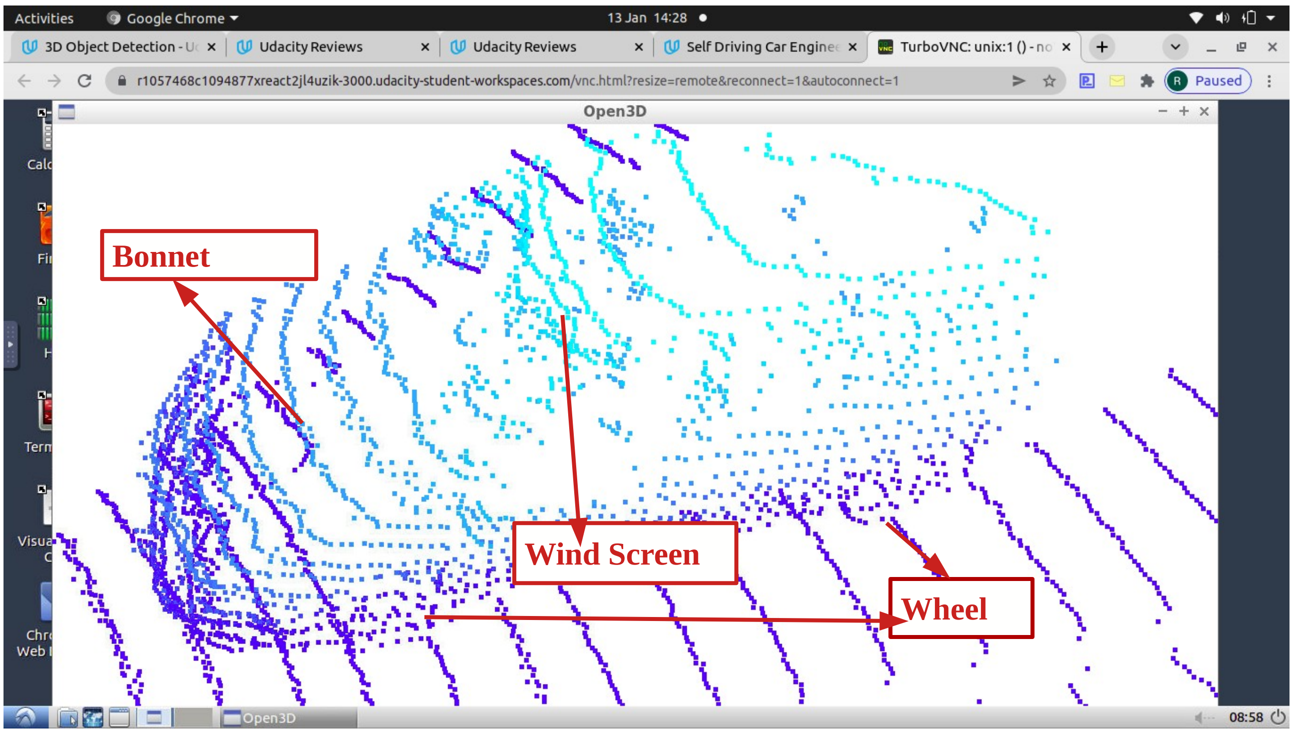


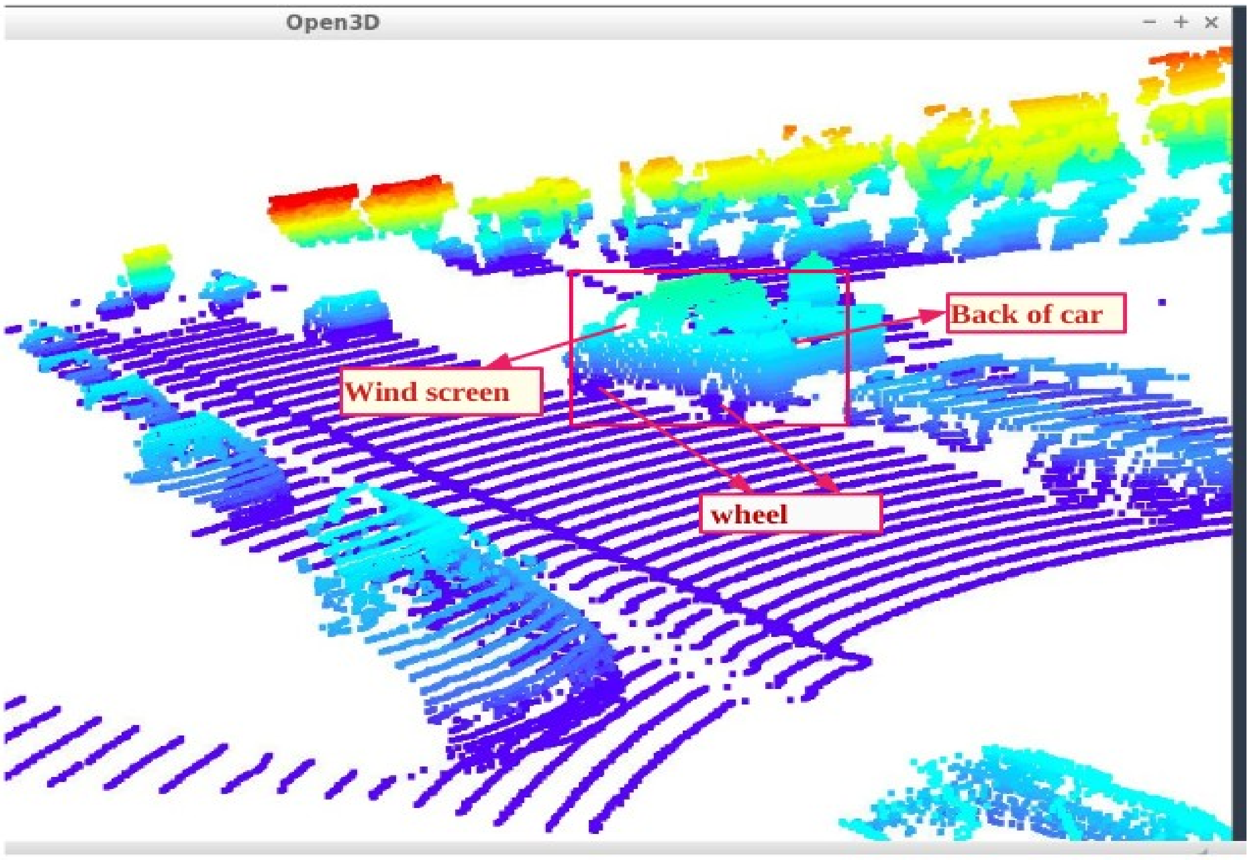

Activities
Firefox Web Browser
12 Jan 06:25
3D Object Detection - Ud
Self Driving Car Engineer
TurboVNC: unix:1 () - noV
Udacity Reviews
https://r1057468c1094877xreact2jl4uzik-3000.udacity-student-workspaces.com/vnc.html?resize=remote
Open3D
Open3D
00:55

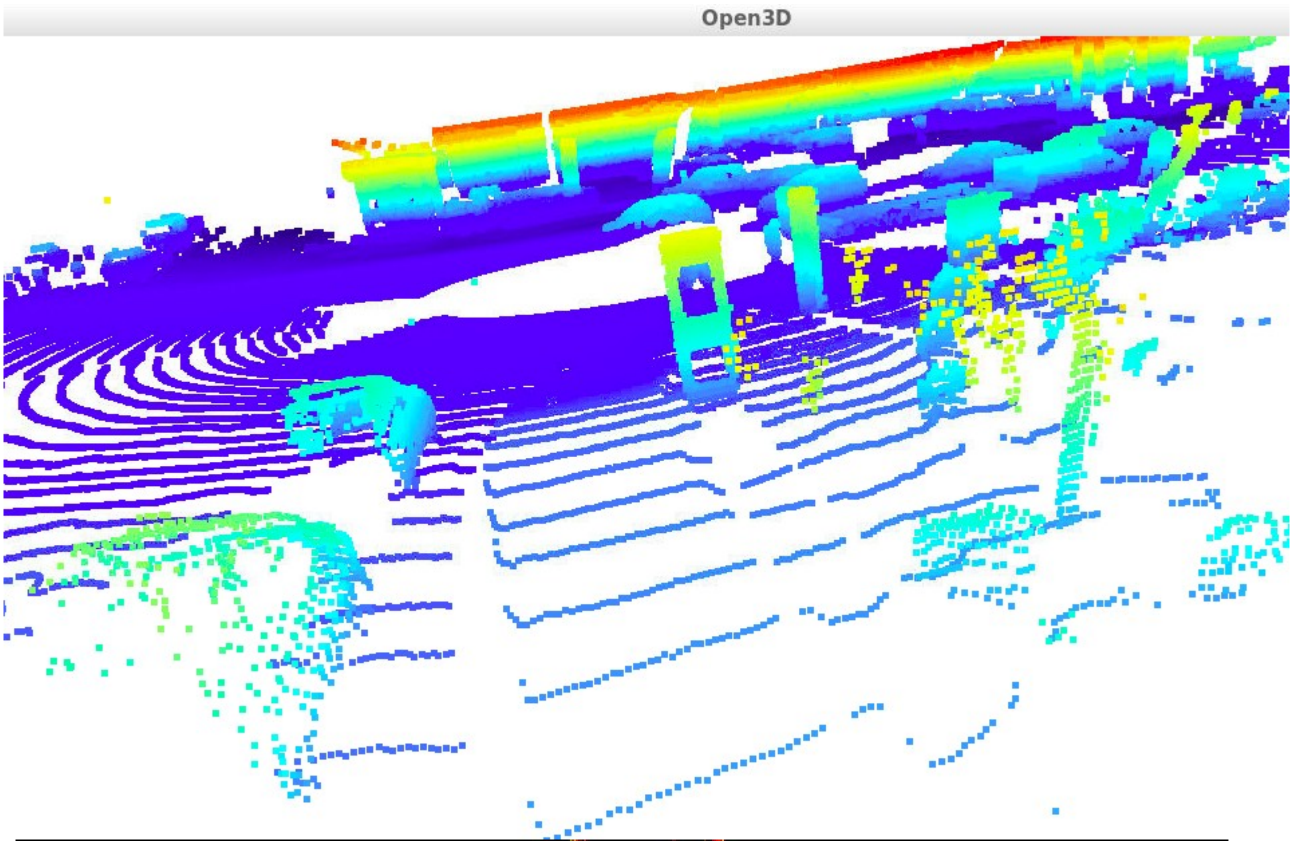


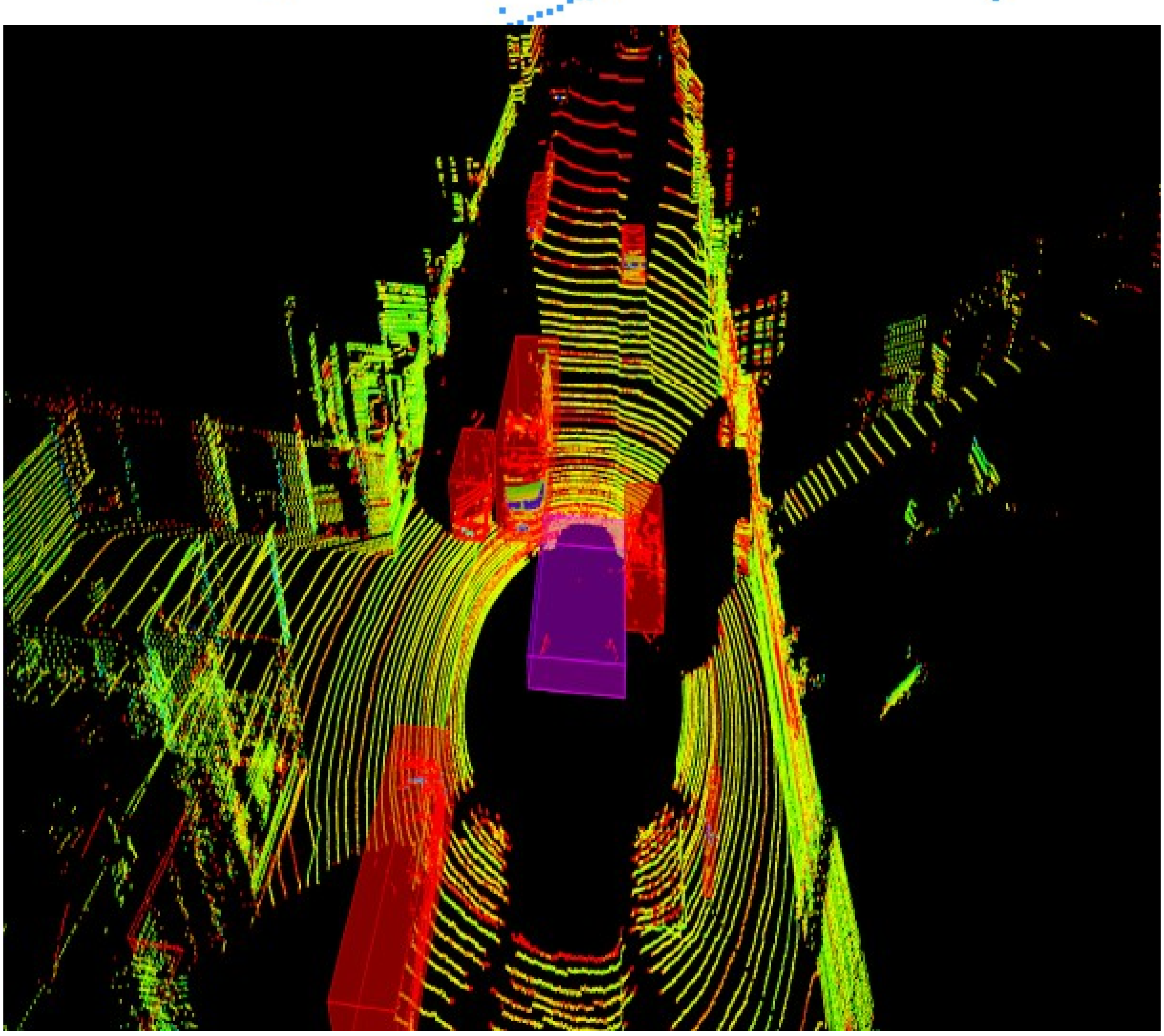



Activities
Google Chrome
13 Jan 14:08
3D Object Detection - Uda
Udacity Reviews
Self Driving Car Engineer
TurboVNC: unix:1 () - noV
r1057468c1094877xreact2jl4uzik-3000.udacity-student-workspaces.com/vnc.html?resize=remote&reconnect=1&autoconnect=1
Paused
Open3D
Calc
Fi
Term
Visua
Chr
Web
Open3D
08:38


Activities
Google Chrome
13 Jan 14:28
3D Object Detection - U
Udacity Reviews
Udacity Reviews
Self Driving Car Enginee
TurboVNC: unix:1 () - no
r1057468c1094877xreact2jl4uzik-3000.udacity-student-workspaces.com/vnc.html?resize=remote&reconnect=1&autoconnect=1
Paused
Open3D
Bonnet
Wind Screen
Wheel
08:58


Open3D
Back of car
Wind screen
wheel


Open3D



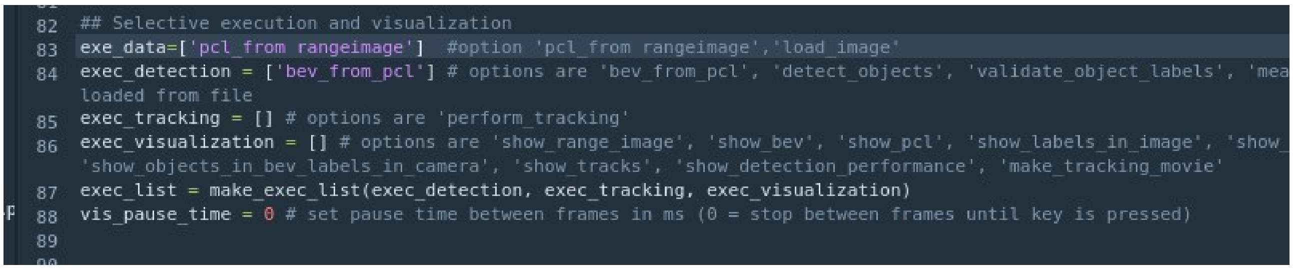
Stable features include the tail lights, the rear bumper majorly. In some cases the additional features include the headover lights, car front lights, rear window shields. These are identified through the intensity channels . The chassis of the car is the most prominent identifiable feature from the lidar perspective. The images are analysed with different settings and the rear lights are the major stable components, also the bounding boxes are correctly assigned to the cars (used from Step-3).

Step-2: Creaate BEV from Lidar PCL

In this case, we are:

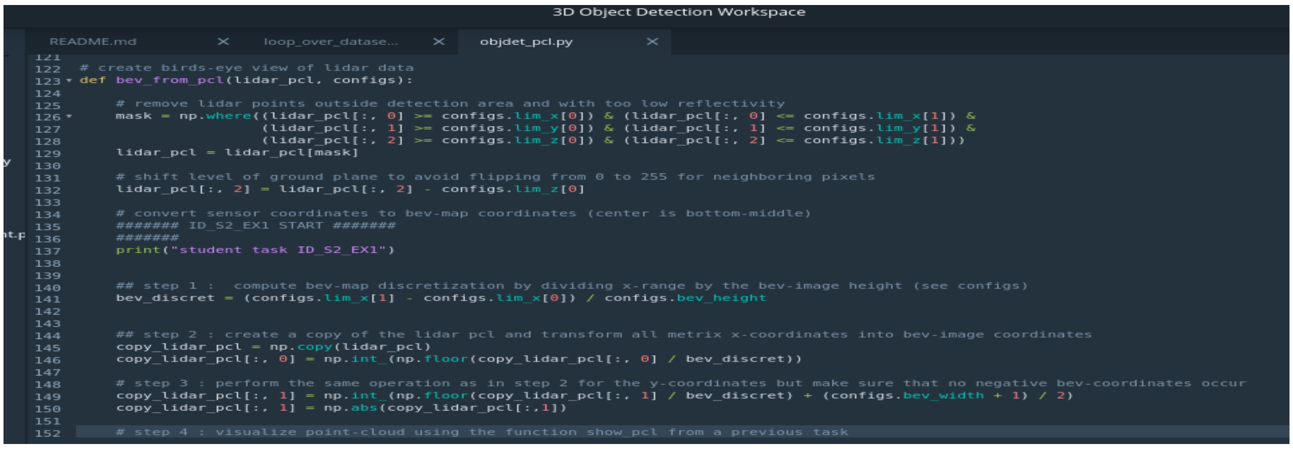
- Converting the coordinates to pixel values
- Assigning lidar intensity values to the birds eye view BEV mapping
- Using sorted and pruned point cloud lidar from the previous task
- Normalizing the height map in the BEV
- Compute and map the intensity values

The changes are in the 'loop_over_dataset.py'



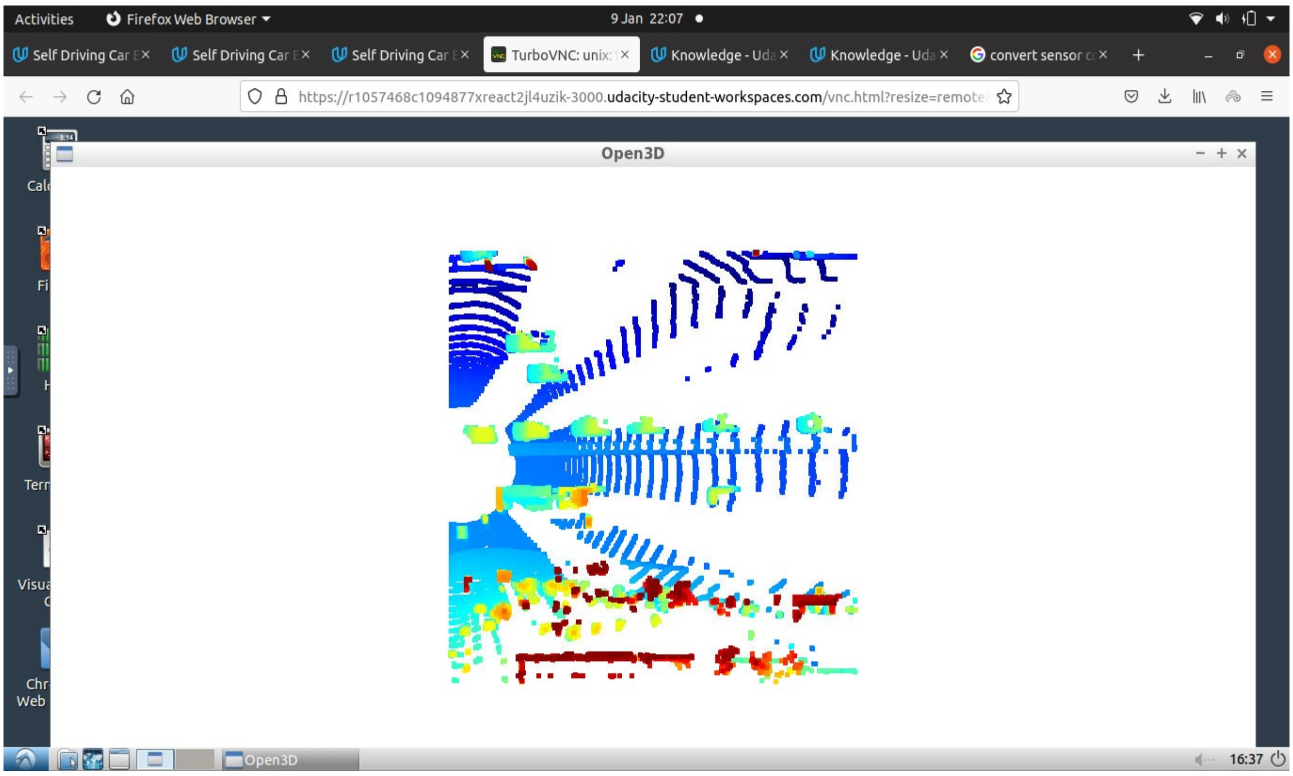
```python
## Selective execution and visualization
exe_data=['pcl_from_rangeimage']  #option 'pcl_from_rangeimage','load_image'
exec_detection = ['bev_from_pcl'] # options are 'bev_from_pcl', 'detect_objects', 'validate_object_labels', 'mea
loaded from file
exec_tracking = [] # options are 'perform_tracking'
exec_visualization = [] # options are 'show_range_image', 'show_bev', 'show_pcl', 'show_labels_in_image', 'show_
'show_objects_in_bev_labels_in_camera', 'show_tracks', 'show_detection_performance', 'make_tracking_movie'
exec_list = make_exec_list(exec_detection, exec_tracking, exec_visualization)
vis_pause_time = 0 # set pause time between frames in ms (0 = stop between frames until key is pressed)
```

The changes are also in the "objdet_pcl.py"

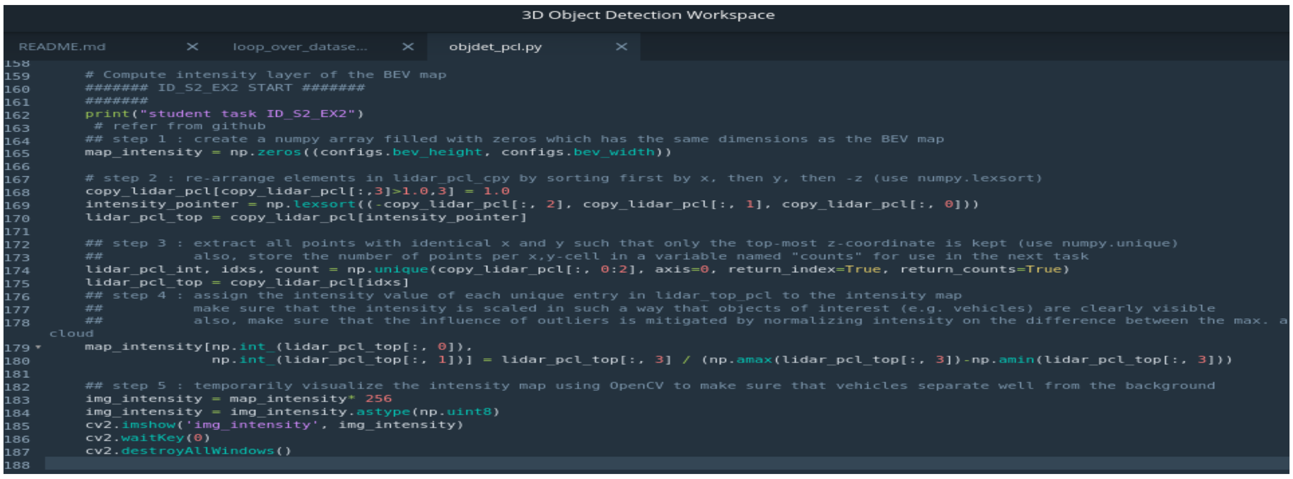


```python
# create birds-eye view of lidar data
def bev_from_pcl(lidar_pcl, configs):

    # remove lidar points outside detection area and with too low reflectivity
    mask = np.where((lidar_pcl[:, 0] >= configs.lim_x[0]) & (lidar_pcl[:, 0] <= configs.lim_x[1]) &
                    (lidar_pcl[:, 1] >= configs.lim_y[0]) & (lidar_pcl[:, 1] <= configs.lim_y[1]) &
                    (lidar_pcl[:, 2] >= configs.lim_z[0]) & (lidar_pcl[:, 2] <= configs.lim_z[1]))
    lidar_pcl = lidar_pcl[mask]

    # shift level of ground plane to avoid flipping from 0 to 255 for neighboring pixels
    lidar_pcl[:, 2] = lidar_pcl[:, 2] - configs.lim_z[0]

    # convert sensor coordinates to bev-map coordinates (center is bottom-middle)
    ####### ID_S2_EX1 START #######
    #######
    print("student task ID_S2_EX1")


    ## step 1 :  compute bev-map discretization by dividing x-range by the bev-image height (see configs)
    bev_discret = (configs.lim_x[1] - configs.lim_x[0]) / configs.bev_height


    ## step 2 : create a copy of the lidar pcl and transform all metrix x-coordinates into bev-image coordinates
    copy_lidar_pcl = np.copy(lidar_pcl)
    copy_lidar_pcl[:, 0] = np.int_(np.floor(copy_lidar_pcl[:, 0] / bev_discret))

    # step 3 : perform the same operation as in step 2 for the y-coordinates but make sure that no negative bev-coordinates occur
    copy_lidar_pcl[:, 1] = np.int_(np.floor(copy_lidar_pcl[:, 1] / bev_discret) + (configs.bev_width + 1) / 2)
    copy_lidar_pcl[:, 1] = np.abs(copy_lidar_pcl[:,1])

    # step 4 : visualize point-cloud using the function show_pcl from a previous task
```

A sample preview of the BEV:



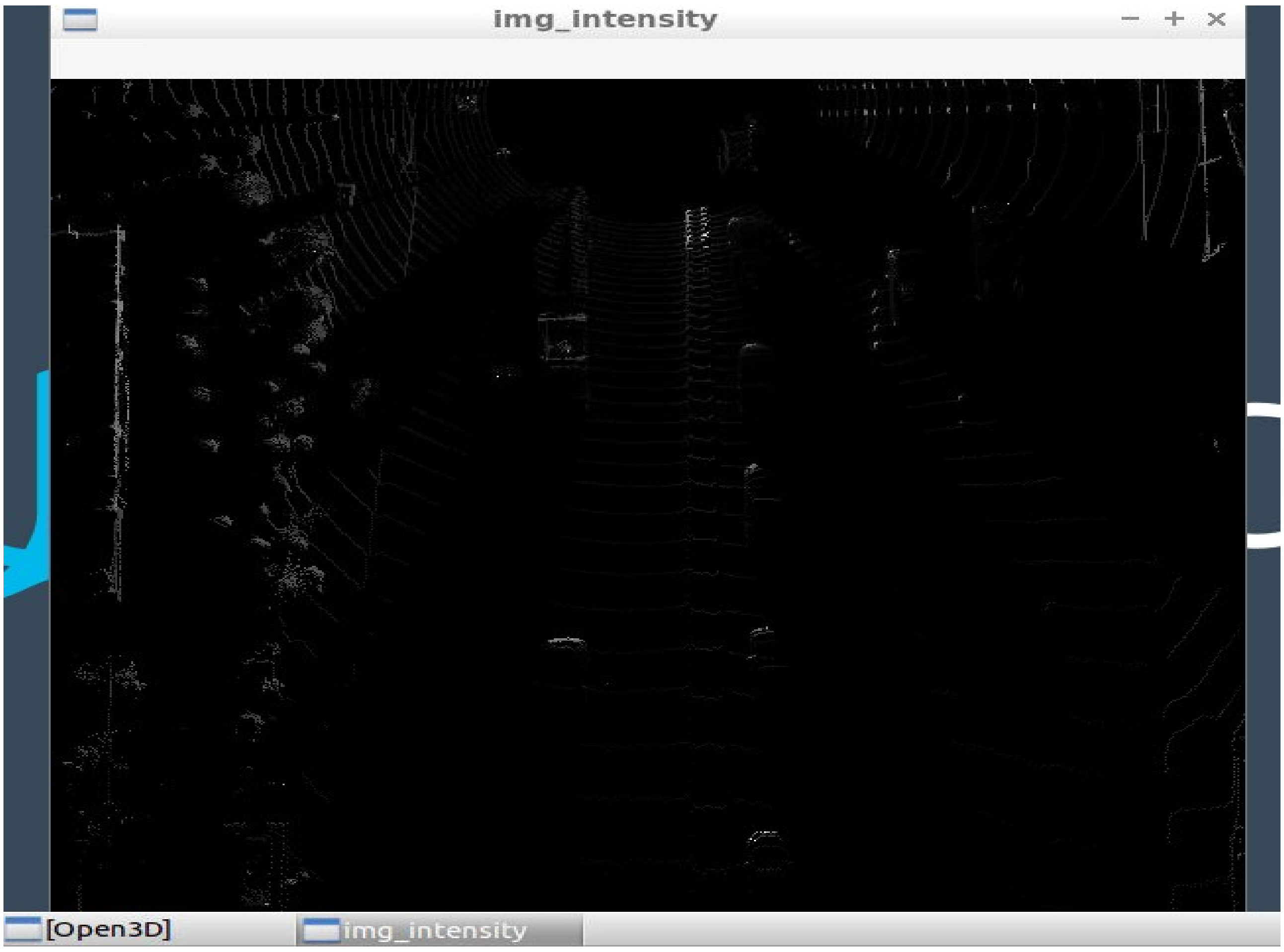
A preview of the intensity layer:



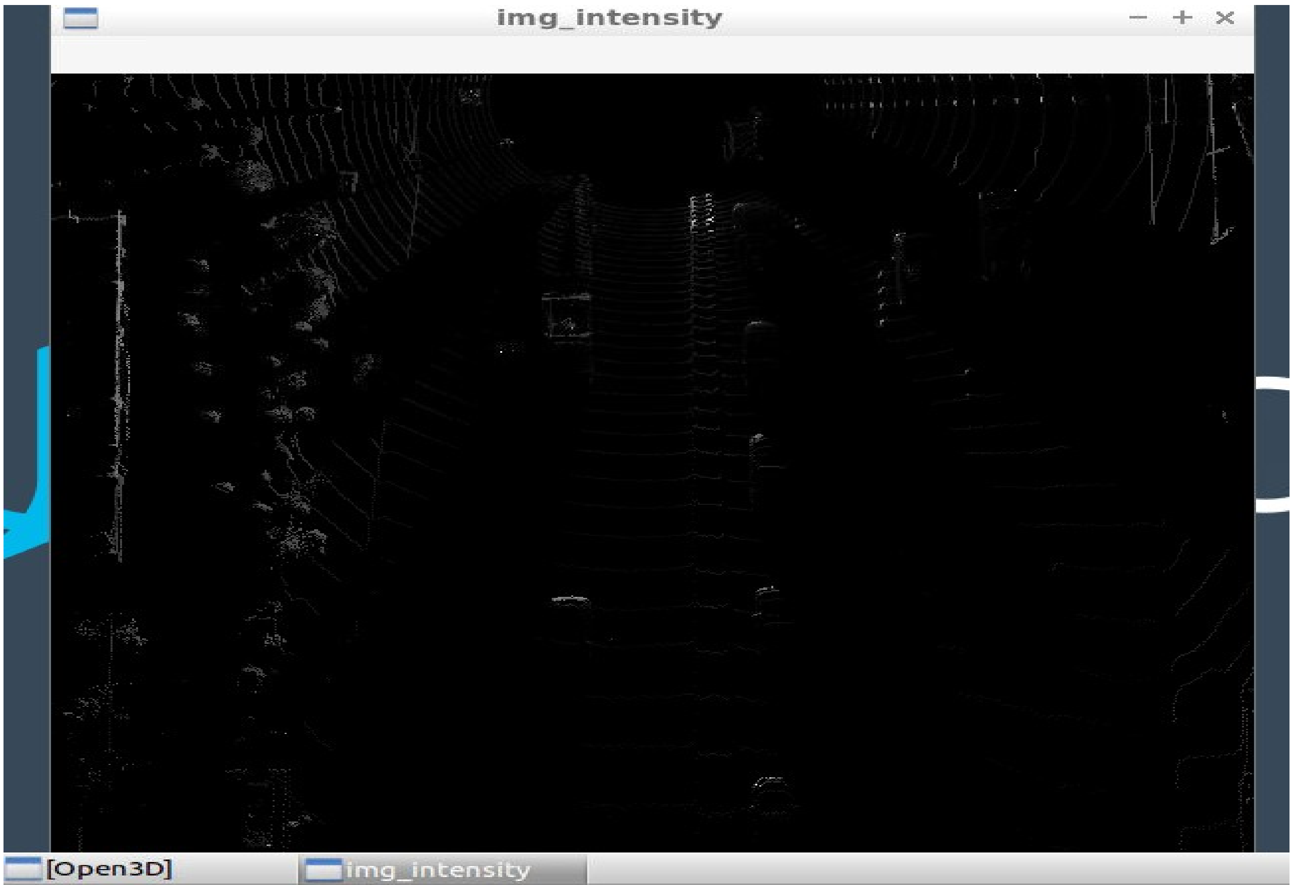
```
    # Compute intensity layer of the BEV map
    ####### ID_S2_EX2 START #######
    #######
    print("student task ID_S2_EX2")
     # refer from github
    ## step 1 : create a numpy array filled with zeros which has the same dimensions as the BEV map
    map_intensity = np.zeros((configs.bev_height, configs.bev_width))

    # step 2 : re-arrange elements in lidar_pcl_cpy by sorting first by x, then y, then -z (use numpy.lexsort)
    copy_lidar_pcl[copy_lidar_pcl[:,3]>1.0,3] = 1.0
    intensity_pointer = np.lexsort((-copy_lidar_pcl[:, 2], copy_lidar_pcl[:, 1], copy_lidar_pcl[:, 0]))
    lidar_pcl_top = copy_lidar_pcl[intensity_pointer]

    ## step 3 : extract all points with identical x and y such that only the top-most z-coordinate is kept (use numpy.unique)
    ##          also, store the number of points per x,y-cell in a variable named "counts" for use in the next task
    lidar_pcl_int, idxs, count = np.unique(copy_lidar_pcl[:, 0:2], axis=0, return_index=True, return_counts=True)
    lidar_pcl_top = copy_lidar_pcl[idxs]
    ## step 4 : assign the intensity value of each unique entry in lidar_top_pcl to the intensity map
    ##          make sure that the intensity is scaled in such a way that objects of interest (e.g. vehicles) are clearly visible
    ##          also, make sure that the influence of outliers is mitigated by normalizing intensity on the difference between the max. a
cloud
    map_intensity[np.int_(lidar_pcl_top[:, 0]),
                  np.int_(lidar_pcl_top[:, 1])] = lidar_pcl_top[:, 3] / (np.amax(lidar_pcl_top[:, 3])-np.amin(lidar_pcl_top[:, 3]))

    ## step 5 : temporarily visualize the intensity map using OpenCV to make sure that vehicles separate well from the background
    img_intensity = map_intensity* 256
    img_intensity = img_intensity.astype(np.uint8)
    cv2.imshow('img_intensity', img_intensity)
    cv2.waitKey(0)
    cv2.destroyAllWindows()
```

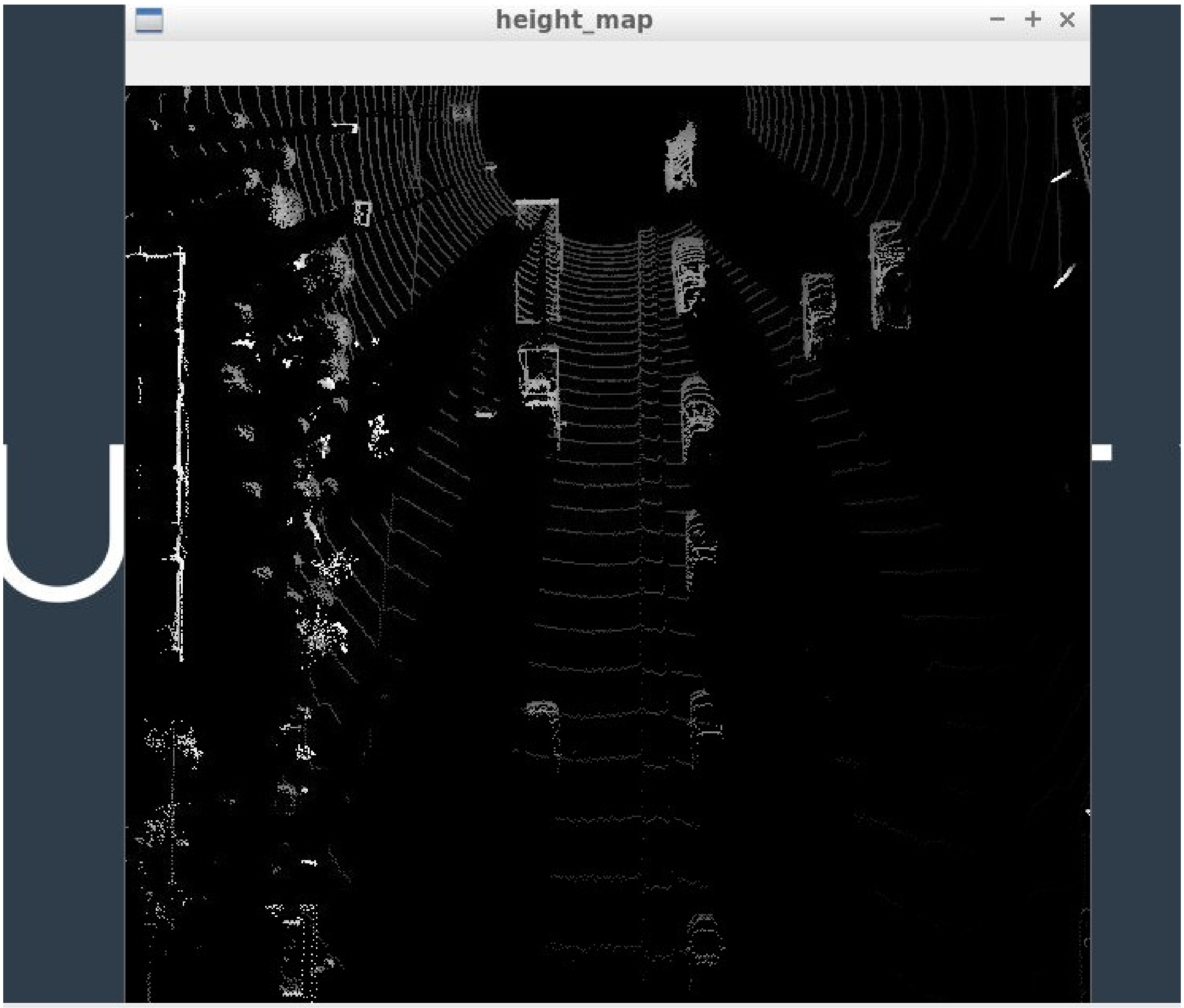
The 'lidar_pcl_top' is used in this case, shown in the Figure:

The corresponding intensity channel:



The corresponding normalized height channel:



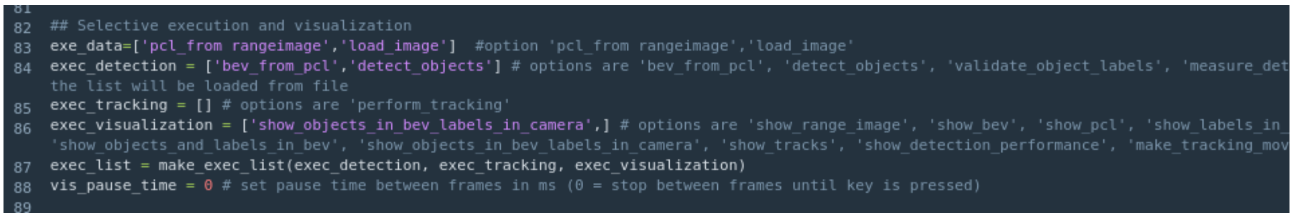


Step-3: Model Based Object Detection in BEV Image

Here, particularly the test.py file and extracting the relevant configurations from 'parse_test_configs()' and added them in the 'load_configs_model' config structure by looking into this repo [https://github.com/maudzung/SFA3D.git].

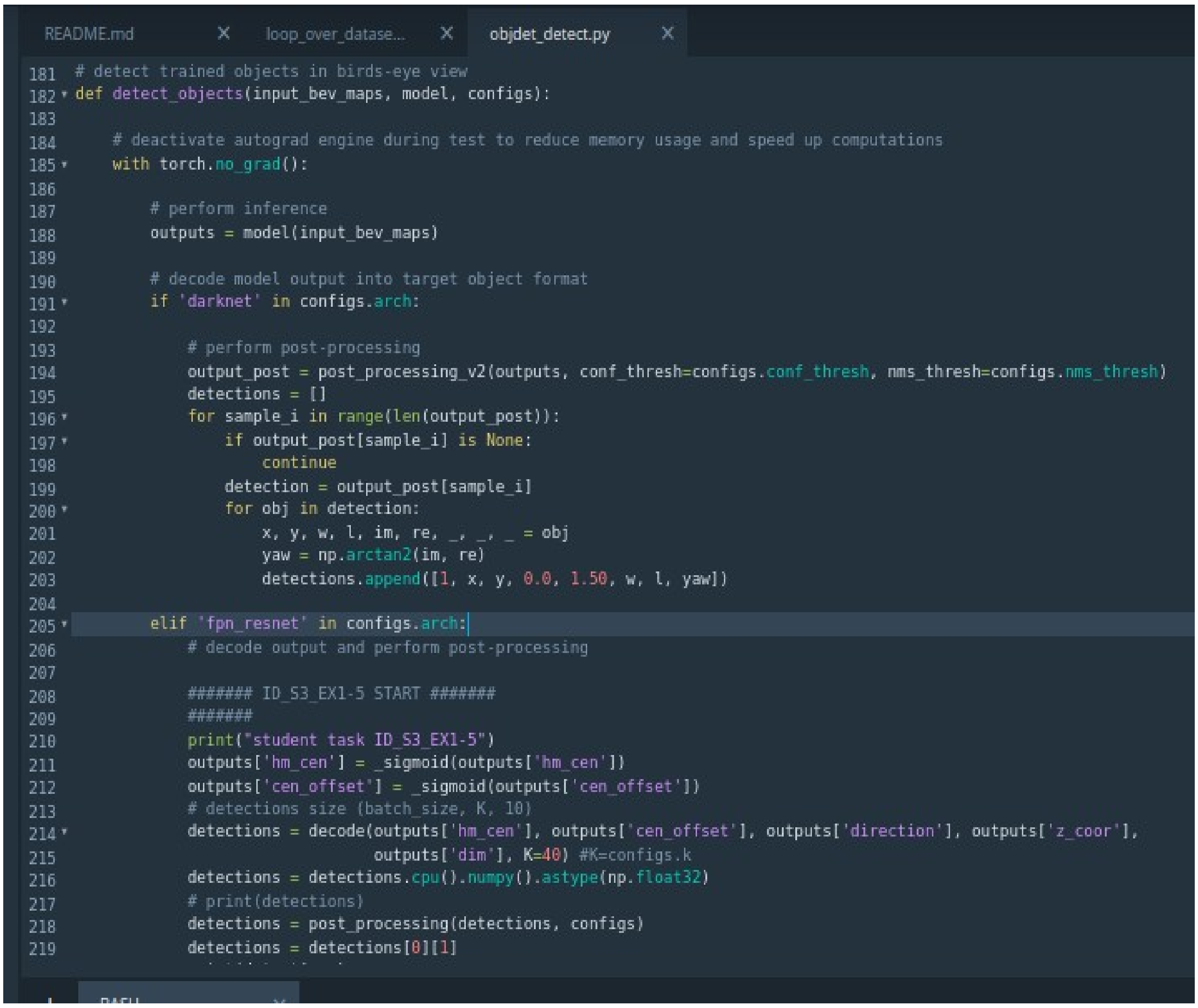
- Instantiating the fpn resnet model from the cloned repository configs
- Extracting 3d bounding boxes from the responses
- Transforming the pixel to vehicle coordinates
- Model output tuned to the bounding box format [class-id, x, y, z, h, w, l, yaw]

The changes are in "loop_over_dataset.py"



```
## Selective execution and visualization
exe_data=['pcl_from rangeimage','load_image']  #option 'pcl_from rangeimage','load_image'
exec_detection = ['bev_from_pcl','detect_objects'] # options are 'bev_from_pcl', 'detect_objects', 'validate_object_labels', 'measure_det
the list will be loaded from file
exec_tracking = [] # options are 'perform_tracking'
exec_visualization = ['show_objects_in_bev_labels_in_camera',] # options are 'show_range_image', 'show_bev', 'show_pcl', 'show_labels_in_
'show_objects_and_labels_in_bev', 'show_objects_in_bev_labels_in_camera', 'show_tracks', 'show_detection_performance', 'make_tracking_mov
exec_list = make_exec_list(exec_detection, exec_tracking, exec_visualization)
vis_pause_time = 0 # set pause time between frames in ms (0 = stop between frames until key is pressed)
```

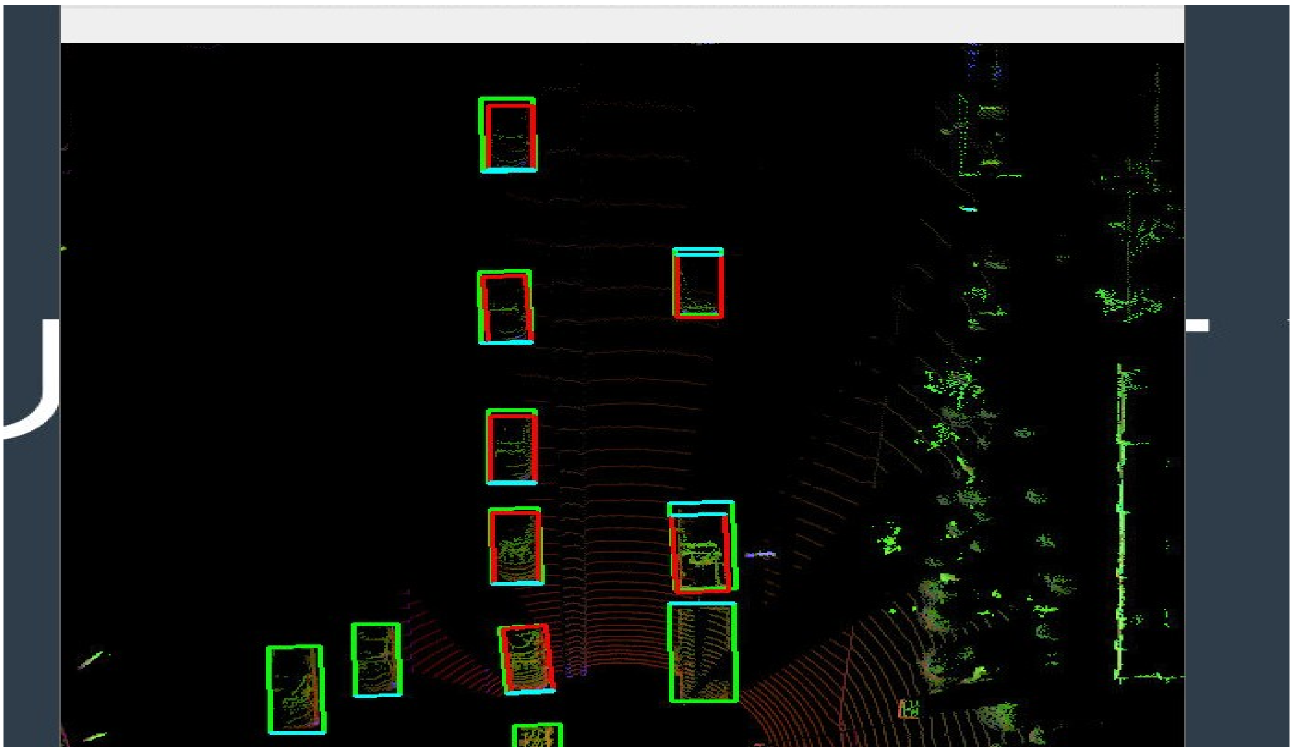
The changes for the detection are inside the "objdet_detect.py" file:

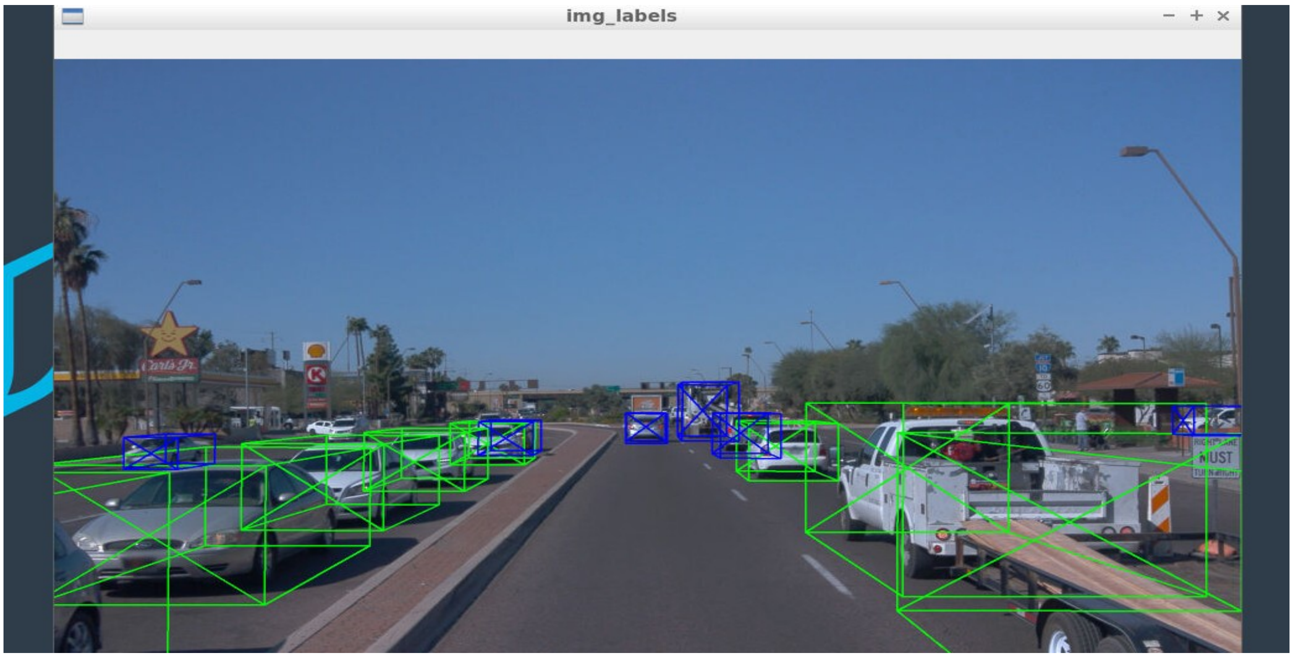


```
# detect trained objects in birds-eye view
def detect_objects(input_bev_maps, model, configs):

    # deactivate autograd engine during test to reduce memory usage and speed up computations
    with torch.no_grad():

        # perform inference
        outputs = model(input_bev_maps)

        # decode model output into target object format
        if 'darknet' in configs.arch:

            # perform post-processing
            output_post = post_processing_v2(outputs, conf_thresh=configs.conf_thresh, nms_thresh=configs.nms_thresh)
            detections = []
            for sample_i in range(len(output_post)):
                if output_post[sample_i] is None:
                    continue
                detection = output_post[sample_i]
                for obj in detection:
                    x, y, w, l, im, re, _, _, _ = obj
                    yaw = np.arctan2(im, re)
                    detections.append([1, x, y, 0.0, 1.50, w, l, yaw])

        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 size (batch_size, K, 10)
            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]
```

As the model input is a three-channel BEV map, the detected objects will be returned with coordinates and properties in the BEV coordinate space. Thus, before the detections can move along in the processing pipeline, they need to be converted into metric coordinates in vehicle space.

A sample preview of the bounding box images:





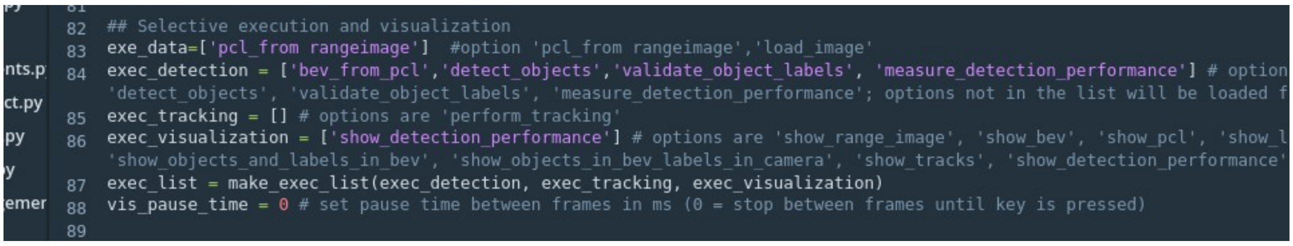
Step-4: Performance detection for 3D Object Detection

In this step, the performance is computed by getting the IOU between labels and detections to get the false positive and false negative values.The task is to compute the geometric overlap between the bounding boxes of labels and the detected objects:

- Assigning a detected object to a label if IOU exceeds threshold

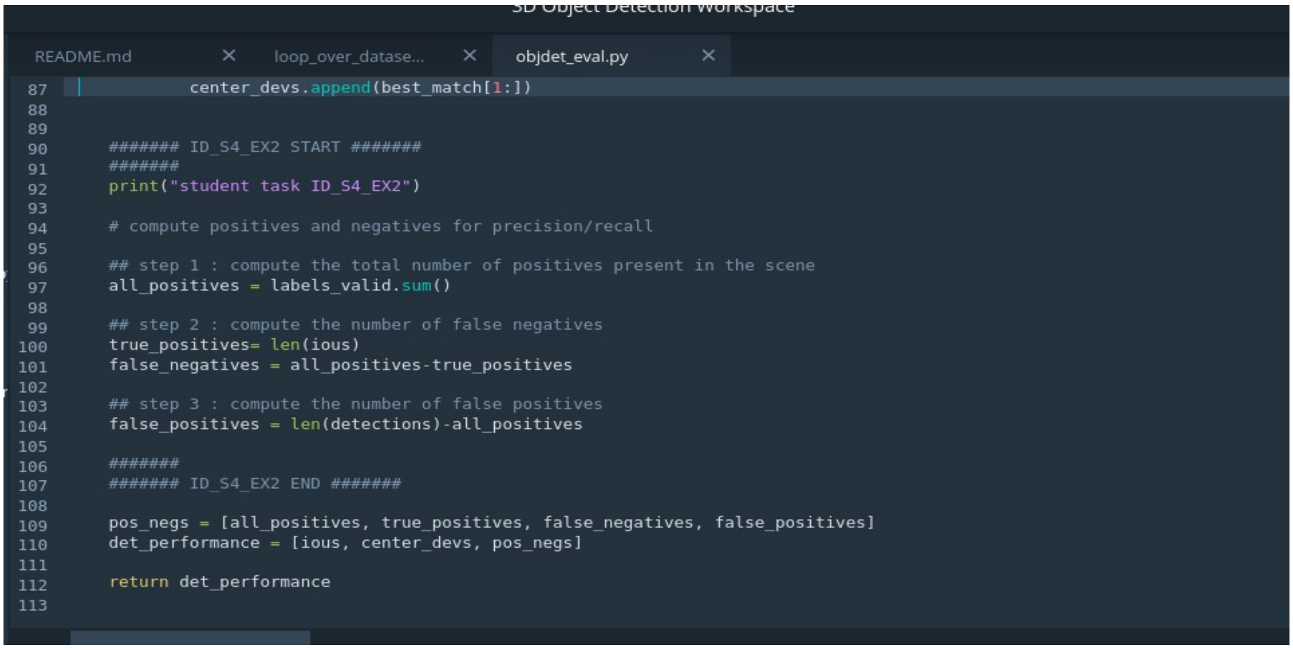
- Computing the degree of geometric overlap
- For multiple matches objects/detections pair with maximum IOU are kept
- Computing the false negative and false positive values
- Computing precision and recall over the false positive and false negative values

The changes in the code are:



```
## Selective execution and visualization
exe_data=['pcl_from rangeimage']  #option 'pcl_from rangeimage','load_image'
exec_detection = ['bev_from_pcl','detect_objects','validate_object_labels', 'measure_detection_performance'] # option
'detect_objects', 'validate_object_labels', 'measure_detection_performance'; options not in the list will be loaded f
exec_tracking = [] # options are 'perform_tracking'
exec_visualization = ['show_detection_performance'] # options are 'show_range_image', 'show_bev', 'show_pcl', 'show_l
'show_objects_and_labels_in_bev', 'show_objects_in_bev_labels_in_camera', 'show_tracks', 'show_detection_performance'
exec_list = make_exec_list(exec_detection, exec_tracking, exec_visualization)
vis_pause_time = 0 # set pause time between frames in ms (0 = stop between frames until key is pressed)
```

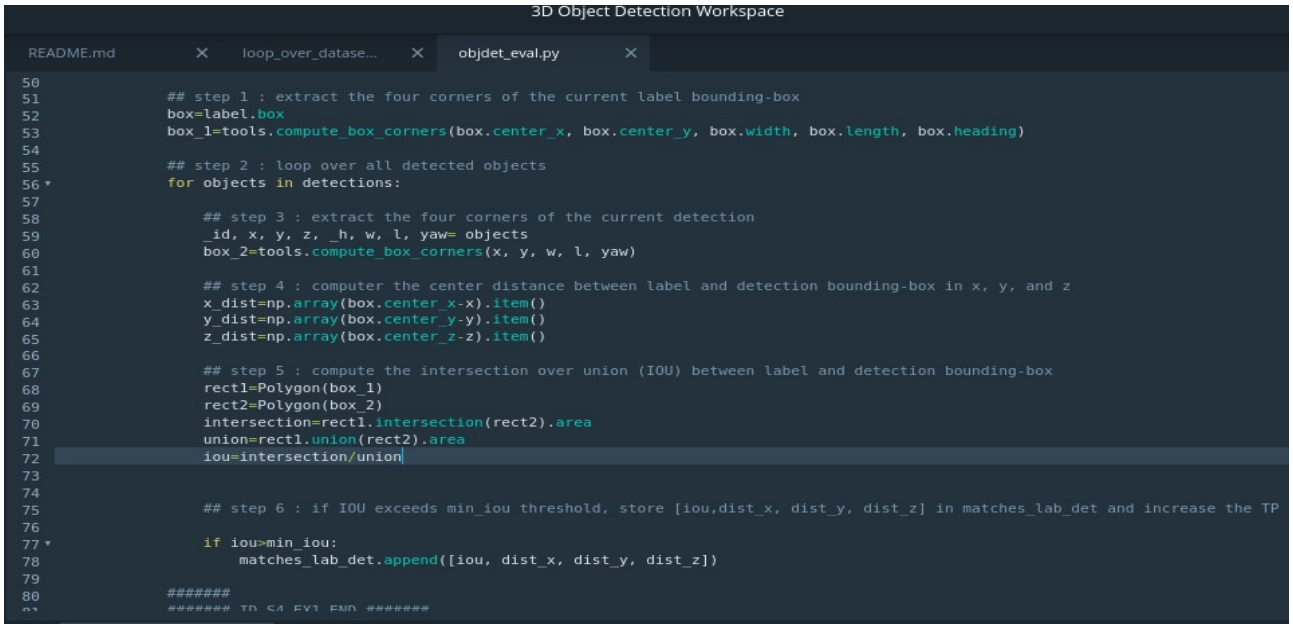
The changes for "objdet_eval.py" where the precision and recall are calculated as functions of false positives and negatives:



3D Object Detection Workspace

README.md | loop_over_datase... | objdet_eval.py

```
            center_devs.append(best_match[1:])


    ####### ID_S4_EX2 START #######
    #######
    print("student task ID_S4_EX2")

    # compute positives and negatives for precision/recall

    ## step 1 : compute the total number of positives present in the scene
    all_positives = labels_valid.sum()

    ## step 2 : compute the number of false negatives
    true_positives= len(ious)
    false_negatives = all_positives-true_positives

    ## step 3 : compute the number of false positives
    false_positives = len(detections)-all_positives

    #######
    ####### ID_S4_EX2 END #######

    pos_negs = [all_positives, true_positives, false_negatives, false_positives]
    det_performance = [ious, center_devs, pos_negs]

    return det_performance
```

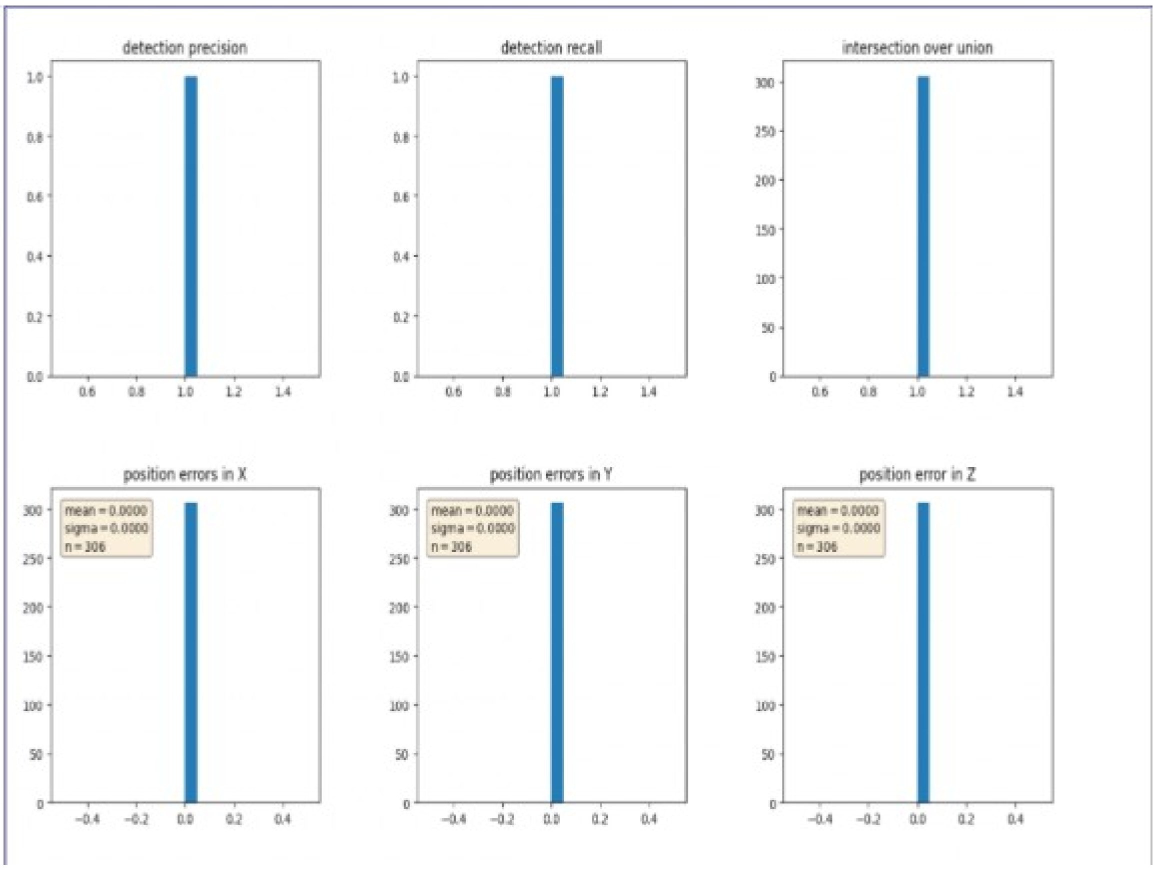


3D Object Detection Workspace

README.md | loop_over_datase... | objdet_eval.py

```

            ## step 1 : extract the four corners of the current label bounding-box
            box=label.box
            box_1=tools.compute_box_corners(box.center_x, box.center_y, box.width, box.length, box.heading)

            ## step 2 : loop over all detected objects
            for objects in detections:

                ## step 3 : extract the four corners of the current detection
                _id, x, y, z, _h, w, l, yaw= objects
                box_2=tools.compute_box_corners(x, y, w, l, yaw)

                ## step 4 : computer the center distance between label and detection bounding-box in x, y, and z
                x_dist=np.array(box.center_x-x).item()
                y_dist=np.array(box.center_y-y).item()
                z_dist=np.array(box.center_z-z).item()

                ## step 5 : compute the intersection over union (IOU) between label and detection bounding-box
                rect1=Polygon(box_1)
                rect2=Polygon(box_2)
                intersection=rect1.intersection(rect2).area
                union=rect1.union(rect2).area
                iou=intersection/union


                ## step 6 : if IOU exceeds min_iou threshold, store [iou,dist_x, dist_y, dist_z] in matches_lab_det and increase the TP

                if iou>min_iou:
                    matches_lab_det.append([iou, dist_x, dist_y, dist_z])

            #######
            ####### ID_S4_EX1 END #######
```

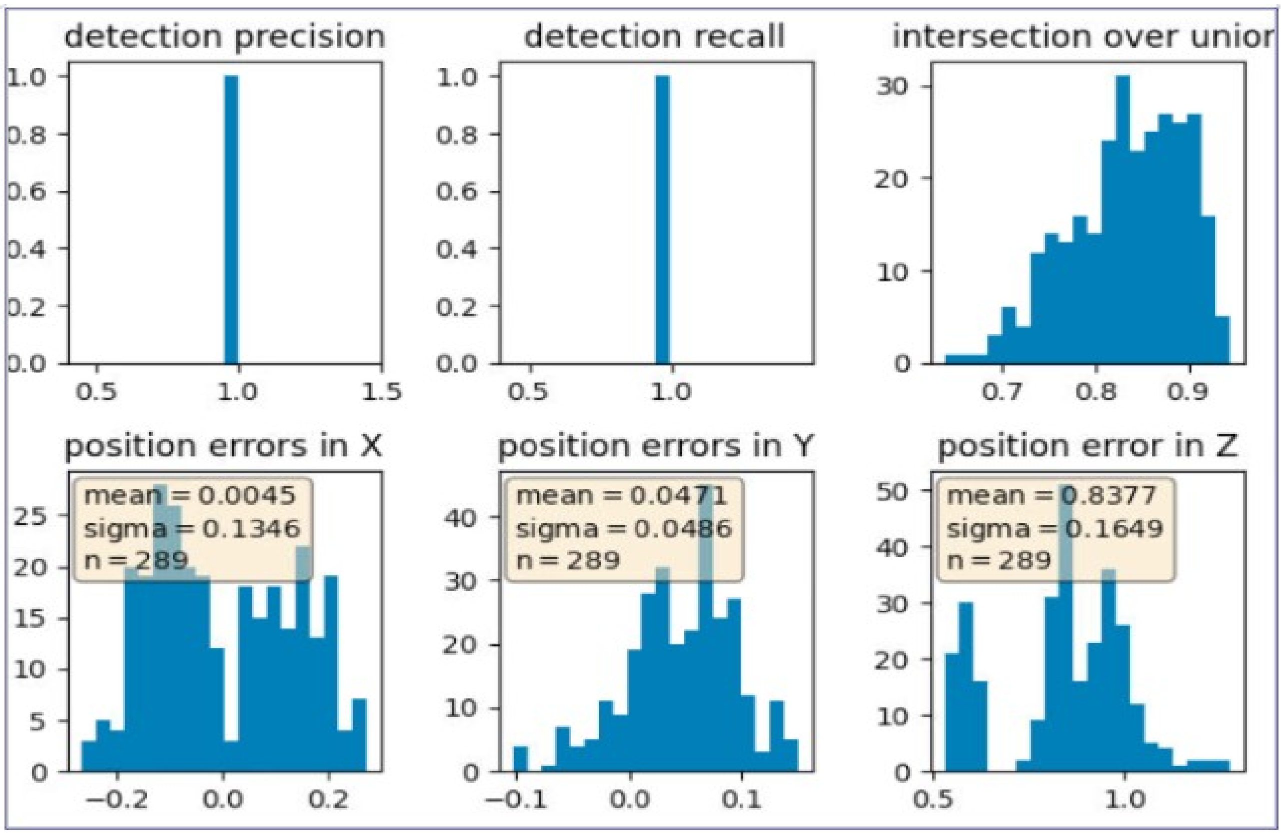
The precision recall curve is plotted showing similar results of precision =0.986 and recall=0.81372



In the next step, we set the

configs_det.use_labels_as_objects=True

which results in precision and recall values as 1.This is shown in the following image:



Summary of Lidar based 3D Object Detection

From the project, it is understandable that for a stabilized tracking, lidar should be used . The conversion of range data to point cloud through spatial volumes, or points (or CNN networks) are important for further analysis. The usage of resnet/darknet and YOLO to convert these high dimensional point cloud representations to object detections through bounding boxes is essential for 3D object detection. Evaluating the performance with help of maximal IOU mapping ,mAP, and representing the precision/recall of the bounding boxes are essential to understand the effectiveness of Lidar based detection.