F110-AVP Early Progress Report

Zirui Zang
zzang@seas.upenn.edu

September 15, 2020

1 Abstract

Autonomous Valet Parking will increase efficiency and lower cost for parking facilities in densely populated cities, if it brings people the same smooth valet parking experience. Most proposed AVP concepts use cameras, ultra-sonic sensors and other sensors that only offer partial or indirect measurements of the environment. This report shows an AVP system that uses Lidars which give reliable and true measurements independent of lighting conditions as primary sensors. The system is built upon ZeroMQ, an open-source asynchronous messaging platform. Pointpillars model is used to infer bounding box on point cloud data at 10Hz. The odometry information from the VESC of the F110 cars are fused with detection to give a 50Hz output of localization data, on which an occupancy map is constructed. A tracking and control pipeline is created to localize and navigate multiple agents. This project shows stable detection and tracking in a demonstration of paralleling parking with two F110 cars.

2 Project Background

Autonomous vehicle technology is improving and expanding, yet there will still be a long time before every vehicle on the road becomes 'smart' enough. In the meantime, opportunities appear for applying the latest technology to normal vehicles if they can be controlled remotely. Parking garages, for example, could increase parking density and cut staff costs by implementing a central-controlled autonomous valet parking (AVP) service. By using a set of stationary sensors (lidars and cameras) and a central planner, AVP can make valet parking service more efficient and affordable. This could help further utilizing the precious parking resources in crowded metropolises around the world, especially in Asia.

Unlike solving the complex problem of autonomous driving on open roads, parking lots are a closed environment we can control. On the other hand, inside parking lots, we also need to detect and track vehicles with higher precision so that the cars can be parked with higher density. Aside from the detection and pose-estimation problem, we can seek to achieve better coordination between cars than self-parking drivers and thus save time and energy, by using a centralized control system.

Along with the development of autonomous vehicles, people are constantly studying the idea of AVP [1]. Various methods have been tried, including camera-based vision [2] [3], laser scanners, ultrasonic, in-ground detectors, etc. However, limited in detection accuracy, these proposed AVP systems either need expensive sensor infrastructures or cannot perform robustly. In recent years, Lidars have been widely used in autonomous driving experiments and have proven to be a suitable sensor in vehicle position and orientation detection. [4] [5] [6] With Lidars installed at optimized locations, we can achieve reliable detection while avoid wasting on other limited sensors.

In this project, we are developing an autonomous valet parking system by using F1/10 rally cars to represent real car scenarios. The dynamic system of F1/10 rally cars is close to real electric cars so we can use them to simulate the detection, tracking, control, planning and other problems that will appear in real life applications. We approach this project using Ouster Lidars as main source of data and will use depth cameras as auxiliary sensors if necessary in the future.

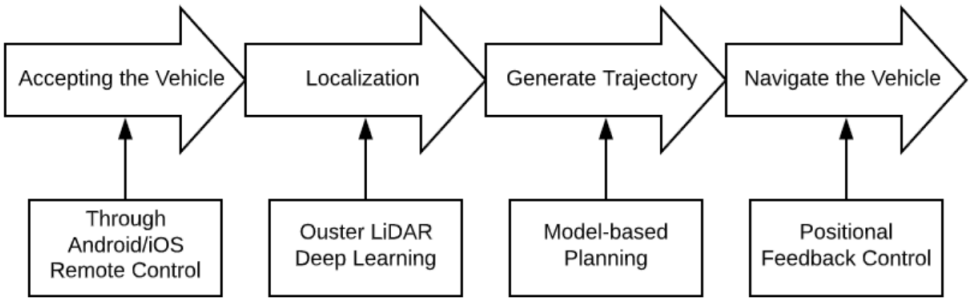

Figure 1: Illustration of Purposed AVP Pipeline

Figure 2: Experiment Setup and Painting the Shell

3 Experiment Setup

The sensor we are using is an Ouster OS1 32-line LiDAR that is looking at the car from a third-person view, as shown in figure 2. Ouster LiDAR is designed for long range detection in autonomous driving and other applications with high accuracy. There are many beam shapes for selection in their product line, wide and narrow, above and below the horizon. The model in our project is with a narrow (16 degrees) and below the horizon beam. With such a beam shape, we are able to cover our playground with sufficient scan lines for robust detection. Although the detection distance in our mini-scale experiment is much shorter than its intended use case, we were able to get point clouds with high signal to noise ratio.

We are developing an autonomous valet parking system by using F1/10 rally cars to represent real car scenarios. The playground is set to be a 1.5x3 meters long rectangle. The detection of position and orientation is done with a deep learning network called PointPillars that was published in 2018. We then create an occupancy grid by binning the point cloud and identify the car based on the detection. The suitable parking position is then found in the occupancy grid and trajectory is generated to navigate the car to the desired parking spot. Then the car is controlled to park into spot with positional feedback from the tracking system.

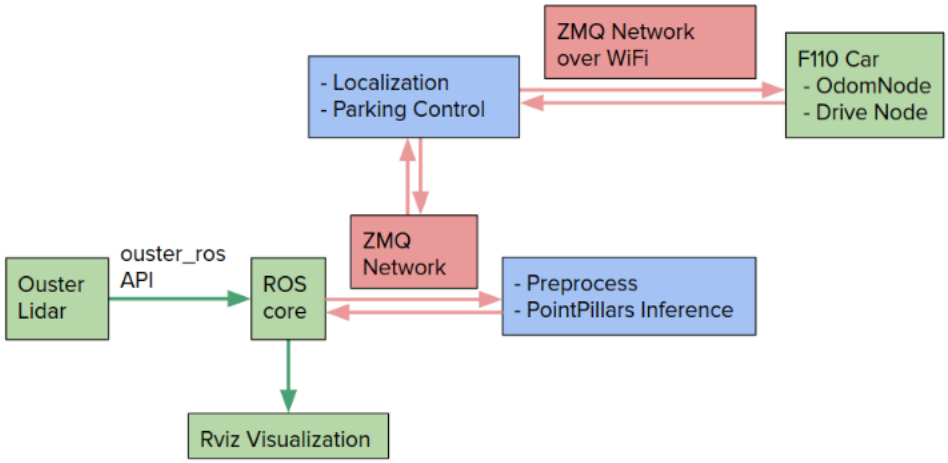

Figure 3: System Structure

4 System Structure

4.1 ZeroMQ

The tracking system consists of three parts that are detection, localization and control. To communicate between parts written in C++ and python 3 and between the controlling computer and the Jetson TX2 onboard the F110 car, a concurrency framework named ZeroMQ (ZMQ) is used. ZMQ uses TCP to communicate between processes and is simple to implement. There are many topological structures for selection inside the ZMQ library. For this project, we are using an asynchronized publisher-subscriber structure, similar to ROS.

Starting from the left in figure 3, raw point cloud is fetched at 10Hz frame rate from the Lidar using the Ouster ROS API. The data are pre-processed by cropping out useless points and rotating to the world frame before sent on the ZeroMQ network. Both the detection part and a visualization node will subscribe to the pre-processed point cloud. The visualization node also collects other results such as bounding boxes and occupancy grid images and displays them in Rviz for us to see.

The localization part of the system takes in the detection results and refine the detection of the F110 car by tracking the car's movements. Each detection may contain several bounding boxes even after thresholding the confidence scores and we have to rely on tracking to know which bound box belongs to the current car we are interested in. Details on the tracking are discussed in the code implementation section.

The update frequency of the point cloud is 10Hz, but we will need more than that to better track the car. The odometry information calculated from the VESC motor controlled on the F110 car is utilized. The linear and angular speed is published to ZMQ and included by the localization into the detection results using a simple Karman filter. This bumps up the detection frequency to more than 50Hz and also helps in keeping track of the car when bounding boxes glitches due to poor point cloud coverage. With an effective update rate of around 50Hz, we are able to track fast run cars.

With robust detection and localization results, a 25x50 pixels occupancy grid is created for path planning. Any points that are a certain height from the ground in the point cloud are binning as obstacles. On this map, pixels that are covered by the bounding rectangles and their connected neighbors are then recognized as belonging to the car.

4.2 Pointpillars

Inside the detection part of the system, the points passed to Pointpillars for detection. As shown in figure 4, Pointpillars network first binds the input point cloud into a fine grid in the x-y plane, which gives 'pillars' of points. Points are then encoded by the arithmetic center and real center positions of the pillars they locate

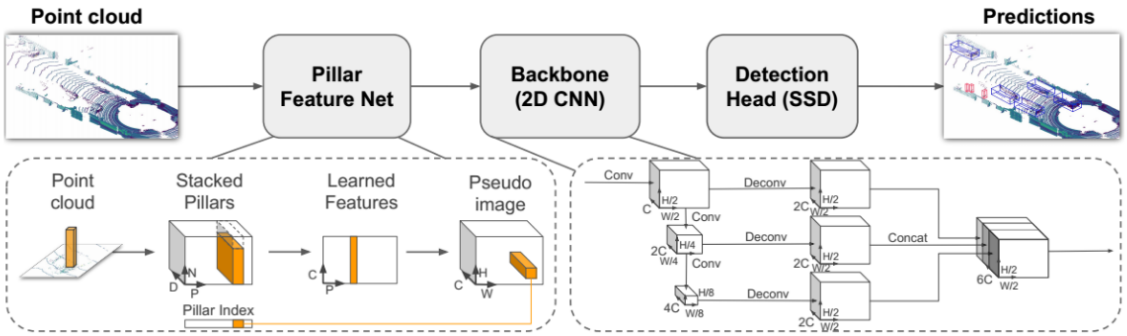

Figure 4: Detailed Structure of the Pointpillars

in. This encoded positional information is combined with the original x, y, z and reflectance to form a "pillar feature". All pillars on the x-y plane that have enough points are put together to form a pseudo-image. By converting the point cloud to a 2D image of pillar features, the network avoids doing 3D convolutions. The inference speed is increased by 2 to 3 times compared to previous in architectures that use 3D convolutions. In our project the inference time is around 100Hz with GTX1080.

Then the pillar features are fed into a 2D backbone that contains CNNs that concentrate and up-sample the pseudo-image to create a hierarchy of features. These features are finally fed into a SSD detection head, which outputs the detection bounding box, orientation and confidence scores. The detection results are published onto ZMQ network. We modified the PillarsPillars codes published on github and trained it on the KITTI dataset for 3 days with a GTX1070.

5 Code Implementation

In this section, we will go through each code file and explain the its functions in the system. Codes to run on the host computer are `ouster.launch`, `lidar_zmq_node.py`, `visualization_node.py`, `detection.py`, `localization.py`. To calibrate the point cloud for point pillars detection, `f110_viewer.py` and `lidar_save_node` are created for convenience. On the Jetson, there are `odom_zmq_node.py` and `driver_node.py`.

5.1 on Host

`ouster.launch` is the launch file for the whole Ouster Lidar ROS driver. It will set up a connection with the lidar and publish the raw point cloud onto ROS.

`lidar_zmq_node.py` is a ROS node that receives raw point cloud and preprocess it before republishing it onto ROS and ZMQ. The preprocess includes rotating and centering the point cloud, trimming the unwanted points and filtering out low intensity points.

`visualization_node.py` is a ROS node that listens to visualization info on ZMQ from various sources and published visualization markers onto ROS to show in `rviz`. These items include bounding boxes, arrows and occupancy grid image.

`detection.py` receives point cloud from ZMQ and run pointpillars inference on it. The results are post-processed and bounding boxes are published to ZMQ. Because pointpillars is trained on KIKKI dataset, in which all data are collected from a car's point of view on street. The point cloud of F110 car need to be similar as an object vehicle in the street. A centered point cloud at (0, 0) won't give good results. The calibration process will be discussed below.

`localization.py` receives point cloud from ZMQ and bin it into the occupancy grid. It receives both

bounding boxes from point cloud inferences and odometry information from the car and fuses them to generate a position and orientation estimation of the cars. Cars are recognized in the occupancy grid from other obstacles. This information is then published onto ZMQ for visualization and navigation.

There is a vehicle orientation calibration process involved in `localization.py`. The orientation inference from pointpillars fluctuates badly between an angle and 180 opposite to that angle. The detection has a larger probability to be correct if we count for a majority in some accumulated high confidence detections. Hence a calibration period (5 sec) is implemented when the car is newly detected on the grid. After the this time, the orientation of the car will be locked and fluctuation are corrected by adding a 180 degree. However, as was later found out, having the cars align to the y axis would produce better bounding box detection but much worst orientation regress. Therefore, at this stage, the orientation of the vehicle needs to be pre-defined and tracked as the vehicle moves.

A vehicle tracking loop is also implemented in `localization.py`. To filter the errors in detection bounding boxes, the loop keeps a list (namely 'matching_row' in the code) of detected vehicles. In each loop, new detections will be compared to members on the 'matching_row' and based on distances between successive detections, bounding boxes that are very close will be determined as tracking the same object.

The initial positions and IP addresses of the vehicles are pre-coded in another list called 'control_row'. When new detections arrive, members in the in 'matching_row' will be compared to the 'control_row'. If matched, 'control_row' will be updated. When new odometry information arrives, both 'control_row' and matched 'matching_row' member will be updated.

5.2 on Host Point Cloud Calibration

`lidar_save_node.py` simply subscribes to the preprocessed point cloud on ROS and then save it at a .npy file.

`f110_viewer,py` as shown in figure 5 is a python GUI that is developed by modifying the KIKKI dataset viewer tool. Below shows the interface where we can load the point cloud from a saved .npy file, manipulate it by adjusting the translation shift in xyz directions and the scaling parameter. Then we can load the trained model to test on this point cloud by clicking the "Load and Inference Network". It is found that if the car is parallel to the x axis, the position detection can be better but the front direction detection can be worst and if the car is perpendicular to the x axis, the result is opposite.

5.3 on Vehicle

`odom_zmq_node.py` is the ROS node that listens to the odometry information on the vehicle and publishes it onto ZMQ network.

`driver_node.py` is the ROS node that listens to the localization information from ZMQ and perform a pure pursuit on loaded waypoints. The current development doesn't contain functions to generate waypoints yet.

6 Result Demo

Figure 6 is a screen shot of the multi-car parallel parking demo. The localization code can take more than two cars, although only two cars are used in the demo. On the right you can see the point cloud, bounding boxes and arrows. The blue and white frames indicate these two vehicle are in the known to the system. At the lower left is the occupancy grid. Orange dots are obstacles and green dots are dots being recognized as vehicles.

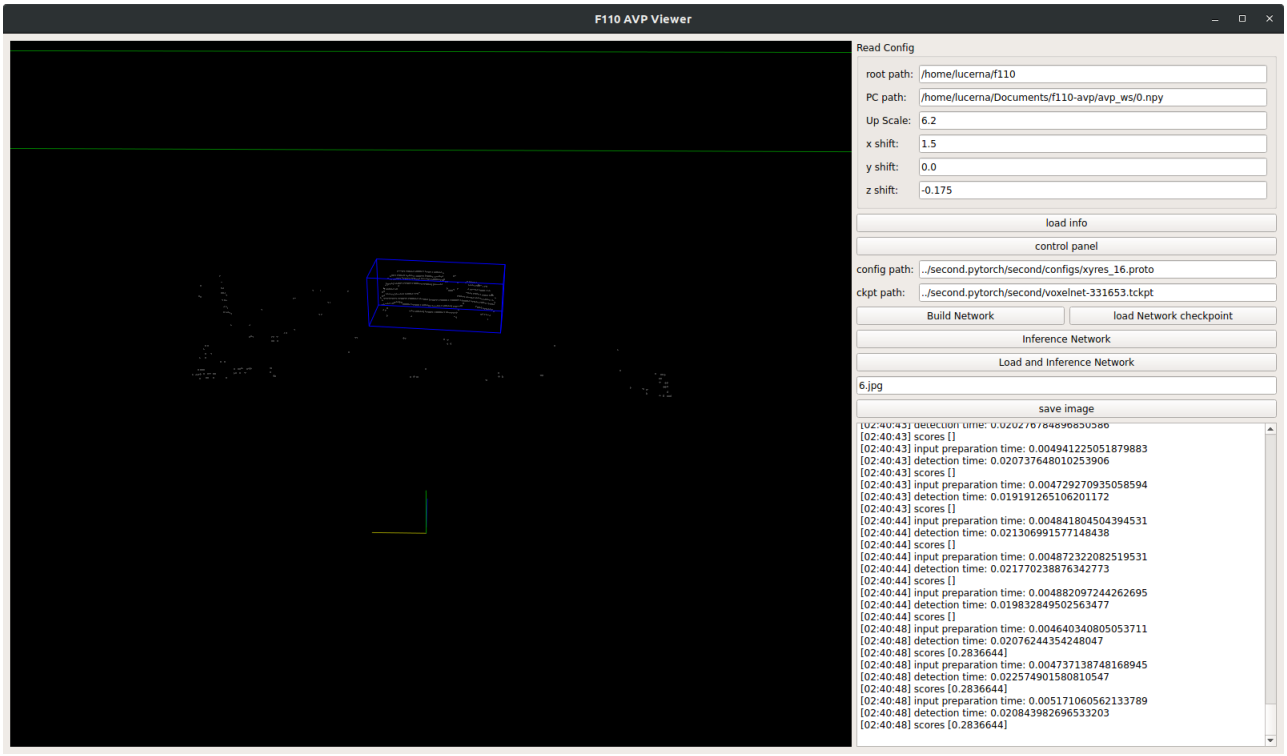

Figure 5: GUI of the f110 viewer for manual point cloud calibration

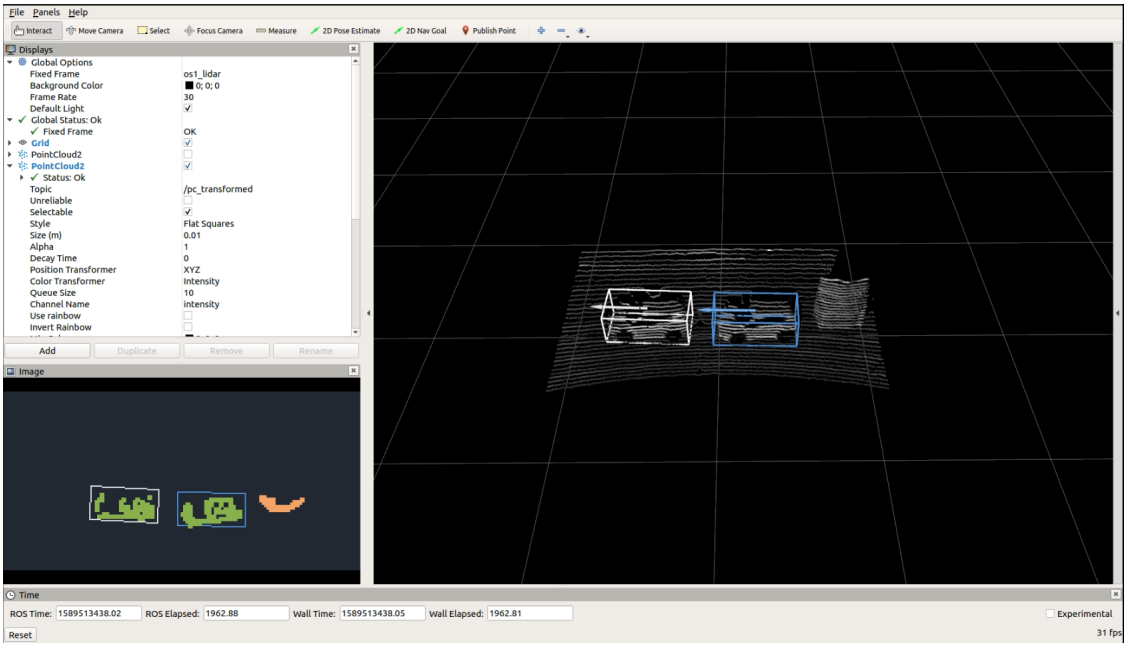

Figure 6: Screenshot of the Visualization in Rviz

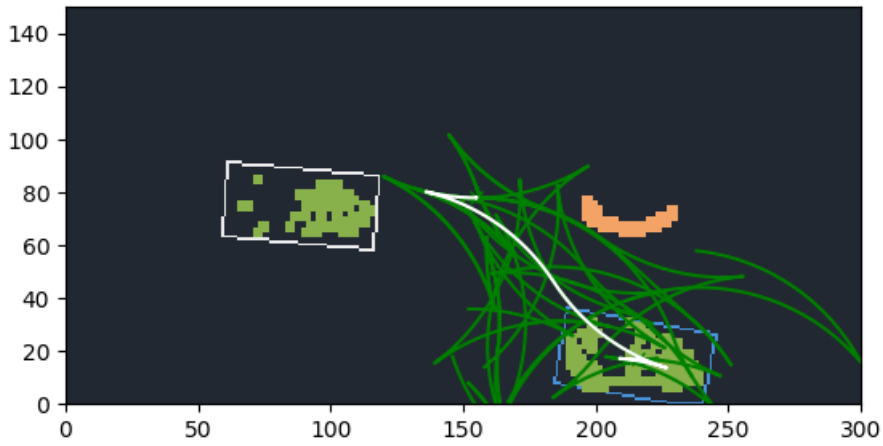

Figure 7: Preliminary RRT* with Implementation: Green curves are considered path and White is chosen.

7 Discussion

7.1 State of the Current Development

The project as reported is able to detect and track multiple F110 cars from lidar point cloud with short inference time as a result of the efficient architecture of Pointpillars model. However, since the host system is not on board, we can further increase the complexity of detection network to achieve better detection bounding boxes. The model used in this project is only trained on KIKKI dataset directly apply to F110 cars. A transfer learning with some labeled data captured with our sensor will also benefit the raw bounding box detection. More recent research works [7] [8] show that adding the RGB image along with point cloud can also greatly improve detection performance.

The currently localization code is able to incorporate odometry information sent from F110 cars to smooth out the 10Hz lidar detection signals. In practice, it is noticed that the latency of ZMQ transmission through wifi can intermittently spike and interfere with localization. This possibly comes from instability of wifi environment. There are also resent research works on using deep learning as a tracking algorithm [7] instead of a simple Kalman filter.

The current waypoints are recorded from manually tele-operating the car. There is some preliminary work on implementing RRT* with dubins path as shown in figure 7 and other parking path generating methods, but the special dynamic of the option to back up in addition to drive forward requires further study and tuning in order to function properly.

7.2 Further Plans

In addition to improving this project according to mentions issues above, this project will be developed with a vehicle detection, localization and navigation system with real cars. The planned development will be interfacing the current system with openPilot, which is a open-sourced autonomous driving code that can running on many commercial vehicle. The purpose is to test this current development on real cars in parking lots and further develop towards a complete autonomous valet parking system.

8 Conclusion

Autonomous driving, as it has been one of the hottest topics in recent years, requires near perfection in code performance in order to be accepted by general public as a safe and reliable technology. The logic behind this AVP project is to prove one of many cases where we can limit down the variations in environment and

can apply the latest technologies from AV (digital lidars, point cloud object detection, etc.) to ground and benefit people's lives. Although the demonstration in this report are not done with real cars, we expect the system to function on real cars and parking lots with limited amount of adjustments.

References

[1] H. Banzhaf, D. Nienhüser, S. Knoop, and J. M. Zöllner. The future of parking: A survey on automated valet parking with an outlook on high density parking. In *2017 IEEE Intelligent Vehicles Symposium (IV)*, pages 1827–1834, June 2017.

[2] N. Kaempchen, U. Franke, and R. Ott. Stereo vision based pose estimation of parking lots using 3d vehicle models. In *Intelligent Vehicle Symposium, 2002. IEEE*, volume 2, pages 459–464 vol.2, June 2002.

[3] P. Jeevan, F. Harchut, B. Mueller-Bessler, and B. Huhnke. Realizing autonomous valet parking with automotive grade sensors. In *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3824–3829, Oct 2010.

[4] Zetong Yang, Yanan Sun, Shu Liu, Xiaoyong Shen, and Jiaya Jia. Std: Sparse-to-dense 3d object detector for point cloud, 2019.

[5] Alex H. Lang, Sourabh Vora, Holger Caesar, Lubing Zhou, Jiong Yang, and Oscar Beijbom. Pointpillars: Fast encoders for object detection from point clouds, 2018.

[6] Shaoshuai Shi, Xiaogang Wang, and Hongsheng Li. Pointrcnn: 3d object proposal generation and detection from point cloud, 2018.

[7] Tianwei Yin, Xingyi Zhou, and Philipp Krähenbühl. Center-based 3d object detection and tracking, 2020.

[8] Sourabh Vora, Alex H. Lang, Bassam Helou, and Oscar Beijbom. Pointpainting: Sequential fusion for 3d object detection, 2019.